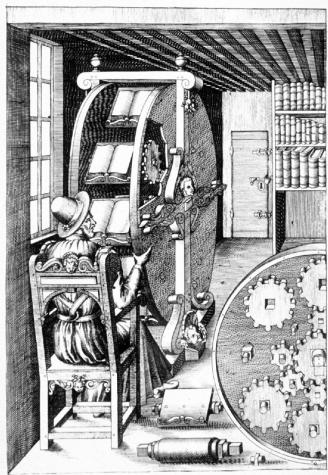


# **Structure and Interpretation of Computer Programs**



**SECOND EDITION**

Unofficial Texinfo Format  
2.andresraba5.2

Harold Abelson and  
Gerald Jay Sussman  
with Julie Sussman,  
foreword by Alan J. Perlis

©1996 by The Massachusetts Institute of Technology

Structure and Interpretation of Computer Programs,  
second edition

Harold Abelson and Gerald Jay Sussman  
with Julie Sussman, foreword by Alan J. Perlis



This work is licensed under a Creative Commons  
Attribution-NonCommercial-ShareAlike 3.0 Unported License  
(CC BY-NC-SA 3.0). Based on a work at [mitpress.mit.edu](http://mitpress.mit.edu).

The MIT Press  
Cambridge, Massachusetts  
London, England

McGraw-Hill Book Company  
New York, St. Louis, San Francisco,  
Montreal, Toronto

Unofficial Texinfo Format [2.andresraba5.2](#) (February 10, 2014),  
based on [2.neilvandyke4](#) (January 10, 2007).

日本語 : by [minghai](#) based on 2.andresraba5.2 (March 31, 2014).

# Contents

<b>Unofficial Texinfo Format</b>	<b>ix</b>
<b>Dedication</b>	<b>xii</b>
<b>Foreword</b>	<b>xiv</b>
<b>Preface to the Second Edition</b>	<b>xxiii</b>
<b>Preface to the First Edition</b>	<b>xxvi</b>
<b>Acknowledgements</b>	<b>xxxii</b>
<b>1 手続を用いた抽象化の構築</b>	<b>1</b>
1.1 The Elements of Programming . . . . .	8
1.1.1 Expressions . . . . .	10
1.1.2 Naming and the Environment . . . . .	13
1.1.3 Evaluating Combinations . . . . .	16
1.1.4 Compound Procedures . . . . .	20
1.1.5 手続適用の置換モデル . . . . .	23
1.1.6 Conditional Expressions and Predicates . . . . .	29
1.1.7 Example: Square Roots by Newton's Method . . . . .	36
1.1.8 Procedures as Black-Box Abstractions . . . . .	44
1.2 Procedures and the Processes They Generate . . . . .	52

1.2.1	Linear Recursion and Iteration . . . . .	54
1.2.2	Tree Recursion . . . . .	62
1.2.3	Orders of Growth . . . . .	71
1.2.4	Exponentiation . . . . .	75
1.2.5	Greatest Common Divisors . . . . .	81
1.2.6	Example: Testing for Primality . . . . .	85
1.3	Formulating Abstractions with Higher-Order Procedures . . . . .	97
1.3.1	引数としての手続 . . . . .	99
1.3.2	Constructing Procedures Using Lambda . . . . .	107
1.3.3	Procedures as General Methods . . . . .	115
1.3.4	Procedures as Returned Values . . . . .	124
<b>2</b>	<b>Building Abstractions with Data</b> . . . . .	<b>137</b>
2.1	Introduction to Data Abstraction . . . . .	144
2.1.1	Example: Arithmetic Operations for Rational Numbers . . . . .	146
2.1.2	Abstraction Barriers . . . . .	152
2.1.3	What Is Meant by Data? . . . . .	157
2.1.4	Extended Exercise: Interval Arithmetic . . . . .	162
2.2	Hierarchical Data and the Closure Property . . . . .	169
2.2.1	Representing Sequences . . . . .	173
2.2.2	Hierarchical Structures . . . . .	187
2.2.3	Sequences as Conventional Interfaces . . . . .	196
2.2.4	Example: A Picture Language . . . . .	218
2.3	Symbolic Data . . . . .	243
2.3.1	Quotation . . . . .	243
2.3.2	Example: Symbolic Differentiation . . . . .	249
2.3.3	Example: Symbolic Differentiation . . . . .	250
2.3.4	Example: Representing Sets . . . . .	259
2.3.5	Example: Huffman Encoding Trees . . . . .	276
2.4	Multiple Representations for Abstract Data . . . . .	290
2.4.1	Representations for Complex Numbers . . . . .	294
2.4.2	Tagged data . . . . .	300

2.4.3	Data-Directed Programming and Additivity . . . . .	307
2.5	Systems with Generic Operations . . . . .	322
2.5.1	Generic Arithmetic Operations . . . . .	324
2.5.2	Combining Data of Different Types . . . . .	331
2.5.3	Example: Symbolic Algebra . . . . .	347
<b>3</b>	<b>Modularity, Objects, and State</b> . . . . .	<b>373</b>
3.1	Assignment and Local State . . . . .	375
3.1.1	Local State Variables . . . . .	377
3.1.2	The Benefits of Introducing Assignment . . . . .	387
3.1.3	The Costs of Introducing Assignment . . . . .	395
3.2	The Environment Model of Evaluation . . . . .	407
3.2.1	The Rules for Evaluation . . . . .	410
3.2.2	Applying Simple Procedures . . . . .	415
3.2.3	Frames as the Repository of Local State . . . . .	420
3.2.4	Internal Definitions . . . . .	427
3.3	Modeling with Mutable Data . . . . .	432
3.3.1	Mutable List Structure . . . . .	433
3.3.2	Representing Queues . . . . .	447
3.3.3	Representing Tables . . . . .	456
3.3.4	A Simulator for Digital Circuits . . . . .	466
3.3.5	Propagation of Constraints . . . . .	486
3.4	Concurrency: Time Is of the Essence . . . . .	504
3.4.1	The Nature of Time in Concurrent Systems . . . . .	506
3.4.2	Mechanisms for Controlling Concurrency . . . . .	515
3.5	Streams . . . . .	539
3.5.1	Streams Are Delayed Lists . . . . .	542
3.5.2	Infinite Streams . . . . .	556
3.5.3	Exploiting the Stream Paradigm . . . . .	570
3.5.4	Streams and Delayed Evaluation . . . . .	591
3.5.5	Modularity of Functional Programs and Modularity of Objects . . . . .	601

<b>4 Metalinguistic Abstraction</b>	<b>612</b>
4.1 The Metacircular Evaluator . . . . .	619
4.1.1 The Core of the Evaluator . . . . .	622
4.1.2 Representing Expressions . . . . .	630
4.1.3 Evaluator Data Structures . . . . .	643
4.1.4 Running the Evaluator as a Program . . . . .	649
4.1.5 Data as Programs . . . . .	655
4.1.6 Internal Definitions . . . . .	661
4.1.7 Separating Syntactic Analysis from Execution . . . . .	671
4.2 Variations on a Scheme --- Lazy Evaluation . . . . .	679
4.2.1 Normal Order and Applicative Order . . . . .	680
4.2.2 An Interpreter with Lazy Evaluation . . . . .	684
4.2.3 Streams as Lazy Lists . . . . .	697
4.3 Variations on a Scheme --- Nondeterministic Computing .	702
4.3.1 Amb and Search . . . . .	705
4.3.2 Examples of Nondeterministic Programs . . . . .	713
4.3.3 Implementing the Amb Evaluator . . . . .	727
4.4 Logic Programming . . . . .	746
4.4.1 Deductive Information Retrieval . . . . .	753
4.4.2 How the Query System Works . . . . .	773
4.4.3 Is Logic Programming Mathematical Logic? . . . .	790
4.4.4 Implementing the Query System . . . . .	801
4.4.4.1 The Driver Loop and Instantiation . . . .	801
4.4.4.2 The Evaluator . . . . .	804
4.4.4.3 Finding Assertions by Pattern Matching .	809
4.4.4.4 Rules and Unification . . . . .	814
4.4.4.5 Maintaining the Data Base . . . . .	821
4.4.4.6 Stream Operations . . . . .	825
4.4.4.7 Query Syntax Procedures . . . . .	827
4.4.4.8 Frames and Bindings . . . . .	831

<b>5 Computing with Register Machines</b>	<b>839</b>
5.1 Designing Register Machines . . . . .	842
5.1.1 A Language for Describing Register Machines . . . . .	847
5.1.2 Abstraction in Machine Design . . . . .	854
5.1.3 Subroutines . . . . .	858
5.1.4 Using a Stack to Implement Recursion . . . . .	864
5.1.5 Instruction Summary . . . . .	874
5.2 A Register-Machine Simulator . . . . .	875
5.2.1 The Machine Model . . . . .	878
5.2.2 The Assembler . . . . .	885
5.2.3 Generating Execution Procedures for Instructions .	890
5.2.4 Monitoring Machine Performance . . . . .	902
5.3 Storage Allocation and Garbage Collection . . . . .	907
5.3.1 Memory as Vectors . . . . .	909
5.3.2 Maintaining the Illusion of Infinite Memory . . . . .	919
5.4 The Explicit-Control Evaluator . . . . .	932
5.4.1 The Core of the Explicit-Control Evaluator . . . . .	935
5.4.2 Sequence Evaluation and Tail Recursion . . . . .	945
5.4.3 Conditionals, Assignments, and Definitions . . . . .	951
5.4.4 Running the Evaluator . . . . .	954
5.5 Compilation . . . . .	965
5.5.1 Structure of the Compiler . . . . .	971
5.5.2 Compiling Expressions . . . . .	980
5.5.3 Compiling Combinations . . . . .	991
5.5.4 Combining Instruction Sequences . . . . .	1002
5.5.5 An Example of Compiled Code . . . . .	1007
5.5.6 Lexical Addressing . . . . .	1023
5.5.7 Interfacing Compiled Code to the Evaluator . . . . .	1031
<b>References</b>	<b>1046</b>
<b>List of Exercises</b>	<b>1054</b>
<b>List of Figures</b>	<b>1056</b>

<b>Index</b>	<b>1057</b>
<b>Colophon</b>	<b>1063</b>

# Unofficial Texinfo Format

This is the second edition SICP book, from Unofficial Texinfo Format.

これは SICP の第二版非公式 Texinfo 版です。

You are probably reading it in an Info hypertext browser, such as the Info mode of Emacs. You might alternatively be reading it TeX-formatted on your screen or printer, though that would be silly. And, if printed, expensive.

あなたは恐らくこれを Emacs の Info モードの様なハイパーテキストブラウザで読んでいることでしょう。他にも TeX で組版した物を画面や印刷して読んでいるかもしれませんがそれはバカバカしい上に高くつきます。

The freely-distributed official HTML-and-GIF format was first converted personally to Unofficial Texinfo Format (UTF) version 1 by Lytha Ayth during a long Emacs lovefest weekend in April, 2001.

公式に無料で公開された HTML-and-GIF 版を Lytha Ayth が最初に私的に、2001 年 4 月の長い Emacs Lovefest Weekend の間に非公式 Texinfo 版 (UTF) バージョン 1 へと変換しました。

The UTF is easier to search than the HTML format. It is also much more accessible to people running on modest computers, such as donated '386-based PCs. A 386 can, in theory, run Linux, Emacs, and a Scheme interpreter simultaneously, but most 386s probably can't also run both Netscape and the necessary X Window System without prematurely introducing budding young underfunded hackers to the concept of *thrashing*. UTF can also fit uncompressed on a 1.44MB floppy diskette, which may come in handy for installing UTF on PCs that do not have Internet or LAN access.

UTF は HTML 版よりも検索がより簡単です。また寄付された古い 386 の様な質素な計算機上で行う人々にとってよりアクセスが容易です。386 は理論的には Linux、Emacs、Scheme インタプリタを同時に実行できます。しかし多く

の 386 は恐らく Netscape と必要な X Window System を事前に芽の出かけた資金不足の若いハッカーに *thrashing*(スラッシング) の概念を教えることなしに動かすことはできないでしょう。UTF はまた圧縮無しでも 1.44MB のフロッピーディスクettに収まります。これはインターネットや LAN への接続環境の無い PC にインストールする場合に役立つでしょう。

The Texinfo conversion has been a straight transliteration, to the extent possible. Like the TeX-to-HTML conversion, this was not without some introduction of breakage. In the case of Unofficial Texinfo Format, figures have suffered an amateurish resurrection of the lost art of ASCII. Also, it's quite possible that some errors of ambiguity were introduced during the conversion of some of the copious superscripts ('^') and subscripts ('\_'). Divining *which* has been left as an exercise to the reader. But at least we don't put our brave astronauts at risk by encoding the *greater-than-or-equal* symbol as <u>&gt; ; </u>.

Texinfo への変換は可能な範囲での直接的な翻字でした。TeX-to-HTML 変換の様にある程度の破れが含まれること無しにはできませんでした。非公式 TexInfo 形式においては図が「失なわれた技術」であるアスキーアートによる下手糞な“復活”を被りました。また多量の上付き文字と下付き文字のいくつかの変換の間に不明瞭さによる変換の失敗が含まれてしまった可能性が大いにあります。読者への課題として残されたと予測します。しかし、最低でも“以上”的記号を <u>&gt; ; </u> と符号化することで我等の勇敢な宇宙飛行士を危険に晒すようなことはしませんでした。

If you modify *sicp.texi* to correct errors or improve the ASCII art, then update the @set utfversion utfversion line to reflect your delta. For example, if you started with Lytha's version 1, and your name is Bob, then you could name your successive versions 1.bob1, 1.bob2, ..., 1.bobn. Also update utfversiondate. If you want to distribute your version on the Web, then embedding the string “sicp.texi” somewhere in the file or Web page will make it easier for people to find with Web search engines.

もしあなたが *sicp.texi* を変更しエラーを訂正したり、アスキーアートを向上させたなら @set utfversion utfversion の行を更新し、あなたの修正を反映して下さい。例えば、もしあなたが Lytha のバージョン 1 で開始し、あなたの名前が Bob なら、改訂版は 1.bob1, 1.bob2, ..., 1.bobn です。また utfversiondate も更新して下さい。もしあなたが自分の改訂版を Web 上で配布したいのなら文字列 “sicp.texi” をファイルや Web ページのどこかに埋め込んでおけば人々にとって Web 検索エンジンから探すことが簡単になるでしょう。

It is believed that the Unofficial Texinfo Format is in keeping with the spirit of the graciously freely-distributed HTML version. But you never know when someone's armada of lawyers might need something to do, and get their shorts all in a knot over some benign little thing, so think twice before you use your full name or distribute Info, DVI, PostScript, or PDF formats that might embed your account or machine name.

非公式 Texinfo 形式は寛大にも自由の下に配布された HTML 版の魂を引き継いでいると信じられています。しかし、いつ誰かの法律家の大艦隊が良心に基づく小さな事に対して非常に腹を立て何かを行わなければならなくなるかもしれません。ですのであなたのフルネームを使ったり、あなたのアカウントやマシン名を含む Info, DVI, PostScript, PDF 形式を配布する前に良く良く考えて下さい。

*Peath, Lytha Ayth*

**Addendum:** See also the SICP video lectures by Abelson and Sussman:  
at [MIT CSAIL](#) or [MIT OCW](#).

付録:Abelson と Sussman による SICP のビデオレクチャーもご覧下さい。  
[MIT CSAIL](#) [MIT OCW](#).

**Second Addendum:** Above is the original introduction to the UTF from 2001. Ten years later, UTF has been transformed: mathematical symbols and formulas are properly typeset, and figures drawn in vector graphics. The original text formulas and ASCII art figures are still there in the Texinfo source, but will display only when compiled to Info output. At the dawn of e-book readers and tablets, reading a PDF on screen is officially not silly anymore. Enjoy!

付録 2: 上記は 2001 年の元の UTF の紹介です。10 年後、UTF は一変しました。数学上の記号と式は適切に組版され、図はベクターグラフィックにより描かれています。元のテキスト形式とアスキーアートの図は今でも Texinfo のソースに残っていますが、Info 形式でコンパイルした場合のみ表示されます。電子書籍リーダーとタブレットの夜明けに画面上で PDF を読むことは正式に、最早バカバカしいことでは無くなりました。楽しんで下さい！

*A.R., May, 2011*

# Dedication

**T**HIS BOOK IS DEDICATED, in respect and admiration, to the spirit that lives in the computer.

この本を、尊敬と贊美を込めて、コンピュータの中に住む妖精に捧げます。

“I think that it’s extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free perfect use of these machines. I don’t think we are. I think we’re responsible for stretching them, setting them off in new directions, and keeping fun in the house. I hope the field of computer science never loses its sense of fun. Above all, I hope we don’t become missionaries. Don’t feel as if you’re Bible salesmen. The world has too many of those already. What you know about computing other people will learn. Don’t feel as if the key to successful computing is only in your hands. What’s in your hands, I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more.”

“コンピュータサイエンスに関わる私達にとってコンピュータを使用することを楽しむことはとても大事だと私は考えます。コンピュータサイエンスが始まった時、それはとても多くの楽しみに溢れていました。ご存知のとおり、お金を払うお客様達は時折酷く騙されました。そして暫くして私達は彼らの不満を真面目に受け取り始めてしまいました。私達は考え始めてしまったのです。成

功裏に、障害の無い完全なコンピュータの使用法について私達に責任があるのではないかと。私はそうは思いません。私は、私達がコンピュータサイエンスを伸展し、新しい方向に向かわせ、そして仲間達と共に楽しむことに責任があると考えます。私はコンピュータサイエンスの現場が楽しむことの感覚を失わないことを望みます。さらに、我々が伝道師になることは望みません。自分が聖書のセールスマンだとは思わないで下さい。世界には既にそのような人が溢れています。あなたが他の人々が学ぶコンピュータ利用法について何を知っているでしょう。コンピュータ利用に成功する鍵があなたの手の中にのみあるとは決つて思わないで下さい。私が思うに、そして期待することは、あなたの手の中にあるものは知性です。それはあなたが初めて計算機に出会った時よりもより多くのことを知ることができる能力であり、それはより多くのことを生むことができるのです。”

—Alan J. Perlis (April 1, 1922 – February 7, 1990)

# Foreword

Educators, generals, dieticians, psychologists, and parents program. Armies, students, and some societies are programmed. An assault on large problems employs a succession of programs, most of which spring into existence en route. These programs are rife with issues that appear to be particular to the problem at hand. To appreciate programming as an intellectual activity in its own right you must turn to computer programming; you must read and write computer programs—many of them. It doesn't matter much what the programs are about or what applications they serve. What does matter is how well they perform and how smoothly they fit with other programs in the creation of still greater programs. The programmer must seek both perfection of part and adequacy of collection. In this book the use of “program” is focused on the creation, execution, and study of programs written in a dialect of Lisp for execution on a digital computer. Using Lisp we restrict or limit not what we may program, but only the notation for our program descriptions.

教育者、将軍、栄養士、精神分析医、そして両親はプログラムします。軍隊、学生、そしていくつかの社会はプログラムされます。大きな問題に対する解決は一連のプログラムを利用します。それらのほとんどは途中でひょっこり表れます。これらのプログラムは手近な問題に特化されて現れる成果に溢れています。プログラミングを独立した知的な活動として理解するためにはあなたはコンピュータプログラミングに向かわねばなりません。コンピュータプログラムを読み、書かねばなりません。それも数多くです。そのプログラムが何についてであるか、またはどのような適用を担うのかは多くは関係ありません。重要なことはそれらがどのように実行され、どれだけ滑らかに他のプログラムに対してより大きなプログラムの作成のために適合するのかです。プログラマは部分の完全性と集合の妥当性の両方を追求せねばなりません。この本では“プログラム”的使用はデジタル計算機上にて実行されるための Lisp の方言で書かれ

たプログラムの創造、実行、それに学習に焦点を当てています。Lisp の使用はプログラム記述の表記法のみを制約、制限し、私達が何をプログラムするかについても影響を与えません。

Our traffic with the subject matter of this book involves us with three foci of phenomena: the human mind, collections of computer programs, and the computer. Every computer program is a model, hatched in the mind, of a real or mental process. These processes, arising from human experience and thought, are huge in number, intricate in detail, and at any time only partially understood. They are modeled to our permanent satisfaction rarely by our computer programs. Thus even though our programs are carefully handcrafted discrete collections of symbols, mosaics of interlocking functions, they continually evolve: we change them as our perception of the model deepens, enlarges, generalizes until the model ultimately attains a metastable place within still another model with which we struggle. The source of the exhilaration associated with computer programming is the continual unfolding within the mind and on the computer of mechanisms expressed as programs and the explosion of perception they generate. If art interprets our dreams, the computer executes them in the guise of programs!

この本の主題は3つの事象に焦点を当てます。人の心、コンピュータプログラムの集合、そしてコンピュータです。全てのコンピュータプログラムは人の心の中で生まれる現実の、または精神的な過程のモデルです。これらの過程は人の経験と思考から浮かび上がり、数はとても多く、詳細は入り組んで、いつでも部分的にしか理解されません。それらはコンピュータプログラムにより稀にしか永遠の充足としてモデル化されることはありません。従って、例え私達のプログラムが注意深く手作りされた別個の記号の集合だとしても、連動する機能の寄せ集めだとしても、それらは絶えず発展します。私達のモデルの知覚がより深まるにつれ、増えるにつれ、一般化されるにつれ、モデルが究極的に準安定な位置に達するまで変更を行い、その中には依然として私達が格闘するモデルが存在します。コンピュータプログラミングに関連する歓喜の源はプログラムとして表現された仕組みの心の中とコンピュータ上で絶え間無く続く発展であり、それにより生まれる知力の爆発です。もし技巧が私達の夢を解釈するならば、コンピュータはプログラムとして現わされるそれらを実行するのです！

For all its power, the computer is a harsh taskmaster. Its programs must be correct, and what we wish to say must be said accurately in every detail. As in every other symbolic activity, we become convinced of program truth through

argument. Lisp itself can be assigned a semantics (another model, by the way), and if a program's function can be specified, say, in the predicate calculus, the proof methods of logic can be used to make an acceptable correctness argument. Unfortunately, as programs get large and complicated, as they almost always do, the adequacy, consistency, and correctness of the specifications themselves become open to doubt, so that complete formal arguments of correctness seldom accompany large programs. Since large programs grow from small ones, it is crucial that we develop an arsenal of standard program structures of whose correctness we have become sure—we call them idioms—and learn to combine them into larger structures using organizational techniques of proven value. These techniques are treated at length in this book, and understanding them is essential to participation in the Promethean enterprise called programming. More than anything else, the uncovering and mastery of powerful organizational techniques accelerates our ability to create large, significant programs. Conversely, since writing large programs is very taxing, we are stimulated to invent new methods of reducing the mass of function and detail to be fitted into large programs.

その力全てに対して、コンピュータは厳しい親方です。そのプログラムは正しくなければなりません。私達が伝えたいと望む事柄は委細全て正確に伝えられねばなりません。全ての他の象徴的な活動と同じく、私達は議論を通してプログラムの真理を確信するようになります。Lisp それ自身に意味論を割り当てるこも可能です。(ところでこれはまた別のモデルです)。そしてもしプログラムの機能を指定できるのなら、例えば述語論理においてなら、論理の証明方法が容認可能な正確性の議論に使用できます。残念なことにプログラムが巨大で複雑になるにつれ、そしてほとんど常にそうなるのですが、仕様の妥当性、一貫性、正確さそれら自身が疑わしくなります。そのため完全に形式化された正確さの議論は巨大なプログラムには伴いません。巨大プログラムは小さな物から成長するため正確さに確信を持てる標準的なプログラム構造の武器庫を開発することは重要です。私達はこれを idiom(イディオム) と呼びます。そしてそれらを組み合わせて価値が検証された構成技術を用いてより大きな構造にすることを学びます。これらの技術はこの本の中で長々と扱われます。そしてそれらを理解することはプログラミングと呼ばれるプロメテウスの進取性(Promethean enterprise)に参加するのに絶対に必要なことです。他の何事でもなく、強力な構成技術を暴き熟達することは巨大で重要なプログラムを作成する能力を加速します。反対に、巨大なプログラムを書くことはとても苦労が多いため、私達は多大な機能や詳細を巨大プログラムに合うように減らす新しい

手法を開発することを促されています。

Unlike programs, computers must obey the laws of physics. If they wish to perform rapidly—a few nanoseconds per state change—they must transmit electrons only small distances (at most  $1\frac{1}{2}$  feet). The heat generated by the huge number of devices so concentrated in space has to be removed. An exquisite engineering art has been developed balancing between multiplicity of function and density of devices. In any event, hardware always operates at a level more primitive than that at which we care to program. The processes that transform our Lisp programs to “machine” programs are themselves abstract models which we program. Their study and creation give a great deal of insight into the organizational programs associated with programming arbitrary models. Of course the computer itself can be so modeled. Think of it: the behavior of the smallest physical switching element is modeled by quantum mechanics described by differential equations whose detailed behavior is captured by numerical approximations represented in computer programs executing on computers composed of ...!

プログラムとは異なり、コンピュータは物理法則に従わなければなりません。もしそれらを迅速に動かしたいのならば—状態変更当たり 2、3 ナノ秒で—コンピュータは電子を極小の距離で転送せねばなりません（高々  $1\frac{1}{2}$  フィート）。巨大な数の端子により生じる熱は空間に集中しますがこれは取り除かねばなりません。精緻な工学の技芸が機能の多重度と端子の密度の間のバランスを取るために開発されました。任意のイベントにおいて、ハードウェアは常に私達がプログラムを行うのに気にするよりもよりプリミティブなレベルで動作します。私達の Lisp プログラムを“機械”的”プログラムに変換する処理はそれ自体が私達がプログラムする抽象モデルです。それらの学習と作成はとても多くの見識をプログラミングの自由裁量なモデルに関連する組織的なプログラムに対して与えます。もちろんコンピュータそれ自身もそのようにモデル化可能です。そのことを考えてみましょう。最小の物理スイッチング要素の振舞は量子力学でモデル化され、微分方程式により記述され、その詳細な振舞は近似値の数値演算により獲得され、それはコンピュータプログラムにより表現され、それはコンピュータ上で実行され、それは組み立てられ...!

It is not merely a matter of tactical convenience to separately identify the three foci. Even though, as they say, it's all in the head, this logical separation induces an acceleration of symbolic traffic between these foci whose richness, vitality, and potential is exceeded in human experience only by the evolution of life itself. At best, relationships between the foci are metastable. The com-

puters are never large enough or fast enough. Each breakthrough in hardware technology leads to more massive programming enterprises, new organizational principles, and an enrichment of abstract models. Every reader should ask himself periodically “Toward what end, toward what end?”—but do not ask it too often lest you pass up the fun of programming for the constipation of bitter-sweet philosophy.

3つの焦点を別々に判別することは戦術上の利便性の問題でしかありません。例え良く言われるように全てが頭の中にあるとしても、この論理的分割はこれらの焦点の間の記号的通信量の加速を引き起します。焦点の豊かさ、活力、潜在力は人間の経験の中で人生自体の発展により増加します。最良時には焦点の間の関係は準安定になります。コンピュータは絶対に十分に大きく、速くはありません。ハードウェア技術の全ての飛躍的進歩がより大規模なプログラミング計画、新しい組織化原理、抽象モデルの向上へと導きます。読者の全員が自身に対し繰り返し“どの終点に向かって？どの終端に向かって？”と問わねばなりません。しかしあまり問い合わせてもいけません。ほろ苦い哲学の便秘のためにプログラミングの楽しさを逸してしまいます。

Among the programs we write, some (but never enough) perform a precise mathematical function such as sorting or finding the maximum of a sequence of numbers, determining primality, or finding the square root. We call such programs algorithms, and a great deal is known of their optimal behavior, particularly with respect to the two important parameters of execution time and data storage requirements. A programmer should acquire good algorithms and idioms. Even though some programs resist precise specifications, it is the responsibility of the programmer to estimate, and always to attempt to improve, their performance.

私達が書くプログラムの間で、いくつか（しかし絶対に十分ではない）は厳格な数学上の関数、例えばソートや数列の最大値を見つける、素数性判定、平方根を求める等が実行されます。私達はそのようなプログラムをアルゴリズムと呼びます。多数の物がそれらの最適な振舞を、特に2つの重要なパラメタである実行時間とデータストレージの必要量に関して知られています。プログラマは良いアルゴリズムとイディオムを獲得しなければなりません。例えいくつかのプログラムが厳格な仕様に反しても、それらのパフォーマンスに関して見積り、常に改善に努めることはプログラマの責務です。

Lisp is a survivor, having been in use for about a quarter of a century. Among the active programming languages only Fortran has had a longer life. Both languages have supported the programming needs of important areas

of application, Fortran for scientific and engineering computation and Lisp for artificial intelligence. These two areas continue to be important, and their programmers are so devoted to these two languages that Lisp and Fortran may well continue in active use for at least another quarter-century.

Lisp は“生存者”であり約四半世紀の間利用されてきました。活発なプログラミング言語の中で Fortran のみが Lisp より長い人生を経ています。Lisp と Fortran はどちらもアプリケーションの重要な領域のプログラミング上の必要性に対処してきました。すなわち Fortran は科学計算や工学計算に対して、Lisp は人工知能に対してです。これらの 2 つの領域は重要で有り続けており、そこに携わっているプログラマ達はこれら 2 つの言語に専念しているため、Lisp と Fortran は少なくとももう四半期は活発に使われ続けることでしょう。

Lisp changes. The Scheme dialect used in this text has evolved from the original Lisp and differs from the latter in several important ways, including static scoping for variable binding and permitting functions to yield functions as values. In its semantic structure Scheme is as closely akin to Algol 60 as to early Lisps. Algol 60, never to be an active language again, lives on in the genes of Scheme and Pascal. It would be difficult to find two languages that are the communicating coin of two more different cultures than those gathered around these two languages. Pascal is for building pyramids—imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms—imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place. The organizing principles used are the same in both cases, except for one extraordinarily important difference: The discretionary exportable functionality entrusted to the individual Lisp programmer is more than an order of magnitude greater than that to be found within Pascal enterprises. Lisp programs inflate libraries with functions whose utility transcends the application that produced them. The list, Lisp's native data structure, is largely responsible for such growth of utility. The simple structure and natural applicability of lists are reflected in functions that are amazingly nonidiosyncratic. In Pascal the plethora of declarable data structures induces a specialization within functions that inhibits and penalizes casual cooperation. It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures. As a result the pyramid must stand unchanged for a millennium; the organism must evolve or perish.

Lisp は変化します。このテキストで使用される Scheme 方言はオリジナル

の Lisp から発展していくつかの重要な手法に関して異なっています。違いには変数束縛に対する静的スコピングや関数の値として関数の生成を許可している点等が含まれます。その意味構造において Scheme は初期の Lisp と同様に Algol 60 に近い物です。Algol 60 は再び現役となることはないでしょうが、Scheme と Pascal の遺伝子に受け継がれています。これらの 2 つの言語の周りに集った言語よりも、もう 2 つの異なる文化の流通貨幣としての 2 つの言語を見つけることのほうが難しいでしょう。Pascal はピラミッドを建築するための物です—印象的で、息を飲むような、軍隊が重いブロックを所定の位置に押すことで建築された静的な構造物です。Lisp は有機体を構築するための物です—印象的で、息を飲むような、小分隊が不安定で無数のより単純な有機体を所定の位置に嵌め込むことで構築された動的な構造物です。使用された体系化の原則は両者の場合で同じです。ただし並外れて重要な違いが 1 つあります。個々の Lisp プログラムに委ねられた任意のエクスポート可能な機能の数は Pascal の進取性の中に見つかるそれらよりも桁違いに多いのです。Lisp プログラムは機能のライブラリを膨らませます。その機能の実用性はそれらを生成したアプリケーションを越えます。Lisp 生来のデータ構造であるリストがそのような実用性の成長の大きな原因です。簡単な構造と自然なリストの適用可能性が驚くべき程に非特異的に機能に反映されています。Pascal では宣言可能なデータ構造の過剰さがカジュアルな連携を抑止し、ペナルティを科す機能の中に特殊化することを促しています。1 つのデータ構造の上で操作する 100 の機能を持つほうが 10 のデータ構造の上で操作する 10 の機能を持つよりも優れています。結果としてピラミッドは 1000 年の間変わらぬままでいるに違いありませんが、有機体は発展できなければ滅んでしまうのです。

To illustrate this difference, compare the treatment of material and exercises within this book with that in any first-course text using Pascal. Do not labor under the illusion that this is a text digestible at MIT only, peculiar to the breed found there. It is precisely what a serious book on programming Lisp must be, no matter who the student is or where it is used.

この違いを説明するためにはこの本の中にある教材と課題の扱いを任意の初級課程の Pascal を用いるテキストのそれと比べてみて下さい。MIT だけが消費できる、そこで見つかる血統書付きの良馬のためのものという幻想の下で苦悩しないで下さい。学生が誰であるかとかどこで利用されるかが問題ではありません。まさに、Lisp プログラミングに対して真剣な本はどんな物であるべきかが問題です。

Note that this is a text about programming, unlike most Lisp books, which are used as a preparation for work in artificial intelligence. After all, the critical

programming concerns of software engineering and artificial intelligence tend to coalesce as the systems under investigation become larger. This explains why there is such growing interest in Lisp outside of artificial intelligence.

これはプログラミングに関するテキストであることに注意して下さい。人工知能の仕事のための予習に使われる他の多くの Lisp の本とは違います。結局、ソフトウェア工学と人工知能の重大なプログラミングの課題は研究がより大きくなるにつれシステムとして融合する傾向にあります。このことがなぜそのような Lisp への興味が人工知能の外側で大きくなっているのかを説明します。

As one would expect from its goals, artificial intelligence research generates many significant programming problems. In other programming cultures this spate of problems spawns new languages. Indeed, in any very large programming task a useful organizing principle is to control and isolate traffic within the task modules via the invention of language. These languages tend to become less primitive as one approaches the boundaries of the system where we humans interact most often. As a result, such systems contain complex language-processing functions replicated many times. Lisp has such a simple syntax and semantics that parsing can be treated as an elementary task. Thus parsing technology plays almost no role in Lisp programs, and the construction of language processors is rarely an impediment to the rate of growth and change of large Lisp systems. Finally, it is this very simplicity of syntax and semantics that is responsible for the burden and freedom borne by all Lisp programmers. No Lisp program of any size beyond a few lines can be written without being saturated with discretionary functions. Invent and fit; have fits and reinvent! We toast the Lisp programmer who pens his thoughts within nests of parentheses.

誰かがそのゴールから予測したように、人工知能研究は多くの明確なプログラミング上の問題を生成しました。他のプログラミング文化ではこの相次ぐ問題は新しい言語を生みます。実際にどんなとても大きなプログラミングタスクにおいても効果的な体系化原理はタスクモジュール内の情報量を言語の発明を通してコントロールし、分離することです。これらの言語は私達、人間が最も良く操作を行うシステムの境界へと辿り着くに従いプリミティブではなくなっていく傾向にあります。結果として、そのようなシステムは何度も複製された複雑な言語処理機能を含みます。Lisp はとてもシンプルな文法と意味論を持ち、ペースが初步的なタスクとして扱えます。従ってペースの技術は Lisp プログラムにおいてはほとんどルール無用の役割を演じます。そして言語処理機の

構築は巨大な Lisp システムの変化と成長の程度に対しほとんど障害になりません。最後に、全ての Lisp プログラマにより負われている義務と自由に対して責任を持つものこそがこのとても単純な文法と意味論です。数行のサイズを越える Lisp プログラムなら自由裁量による関数で満たすことなく書くことはできません。開発し、合わせる。合わせて、また開発する！括弧の入れ子の中に自身の考えを記述する Lisp プログラマに乾杯。

Alan J. Perlis  
New Haven, Connecticut

# Preface to the Second Edition

Is it possible that software is not like anything else, that it is meant to be discarded: that the whole point is to always see it as a soap bubble?

ソフトウェアが他の何物にも似ていないと言うことはできるでしょうか。それが捨てられるべき物だと。つまり、常にシャボン玉だと見なすことだと。

—Alan J. Perlis

The material in this book has been the basis of MIT's entry-level computer science subject since 1980. We had been teaching this material for four years when the first edition was published, and twelve more years have elapsed until the appearance of this second edition. We are pleased that our work has been widely adopted and incorporated into other texts. We have seen our students take the ideas and programs in this book and build them in as the core of new computer systems and languages. In literal realization of an ancient Talmudic pun, our students have become our builders. We are lucky to have such capable students and such accomplished builders.

この本の中の教材は 1980 年から MIT の入門者レベルの計算機科学の科目の中心となる物です。私達はこの教材を 4 年間、最初の版が出版された時点でお教えてきました。そしてこの第二版が出現するまでにさらに 12 年が経過しました。私達の成果が広く受け入れられ、他のテキストに取り込まれていています。私達の生徒がこの本の考え方とプログラムを学び新しい計算機システムと言語の核としてそれらを組み込んでいるのを見てきました。古代のタルムードの多義語の文字認識では、私達の生徒が開発者になってくれました。そのような能力有る学生と熟練した開発者を得たことはとても幸

運なことでした。

In preparing this edition, we have incorporated hundreds of clarifications suggested by our own teaching experience and the comments of colleagues at MIT and elsewhere. We have redesigned most of the major programming systems in the book, including the generic-arithmetic system, the interpreters, the register-machine simulator, and the compiler; and we have rewritten all the program examples to ensure that any Scheme implementation conforming to the IEEE Scheme standard ([IEEE 1990](#)) will be able to run the code.

この版を準備するにあたって、私達自身の教育上の経験と MIT や他の同僚達からのコメントにより提案された幾百もの説明を統合しました。この本の中の主なプログラミングシステムの多くを包括的数値演算システム、インタプリタ、レジスタマシンシミュレータ、コンパイラを含めて再設計しました。そして全てのプログラム例を、任意の IEEE Scheme 標準 ([IEEE 1990](#)) に従う Scheme 実装がそれのコードを実行できることを確実にするために、書き直しました。

This edition emphasizes several new themes. The most important of these is the central role played by different approaches to dealing with time in computational models: objects with state, concurrent programming, functional programming, lazy evaluation, and nondeterministic programming. We have included new sections on concurrency and nondeterminism, and we have tried to integrate this theme throughout the book.

この版はいくつかの新しいテーマを重視しています。これらの内、最も重要なものは計算モデル内での時間を取り扱うための異なる取り組みにより演じられる中心的な役割です。状態を伴うオブジェクト、並行プログラミング、関数型プログラミング、遅延評価、そして非決定性プログラミングです。私達は並行性と非決定性に関わる新しい節を含め、そしてこのテーマをこの本を通してまとめることを試みました。

The first edition of the book closely followed the syllabus of our MIT one-semester subject. With all the new material in the second edition, it will not be possible to cover everything in a single semester, so the instructor will have to pick and choose. In our own teaching, we sometimes skip the section on logic programming ([Section 4.4](#)), we have students use the register-machine simulator but we do not cover its implementation ([Section 5.2](#)), and we give only a cursory overview of the compiler ([Section 5.5](#)). Even so, this is still an intense course. Some instructors may wish to cover only the first three or four chapters, leaving the other material for subsequent courses.

この本の第一版は MIT の一学期の科目の講義概要を密接に追っていました。

第二版の全ての新しい教材により、一学期で全てをカバーすることは不可能となりました。そのためインストラクタは選択をしなければなりません。私達自身の教育現場では、時々論理プログラミング (Section 4.4) を飛ばします。学生にはレジスタマシンのシミュレータを使用させてるのでその実装 (Section 5.2) はカバーしません。そしてコンパイラ (Section 5.5) は概観のみを大雑把に教えていきます。それでもこれは依然として強烈な授業です。何人かのインストラクタは最初の 3 章から 4 章のみをカバーし、他の教材を続きの授業に残したいと願うでしょう。

The World-Wide-Web site <http://mitpress.mit.edu/sicp> provides support for users of this book. This includes programs from the book, sample programming assignments, supplementary materials, and downloadable implementations of the Scheme dialect of Lisp.

World-Wide-Web サイト <http://mitpress.mit.edu/sicp> はこの本のユーザへのサポートを提供します。これにはこの本のプログラム、プログラミング課題のサンプル、補助教材、ダウンロード可能な Lisp の Scheme 方言の実装が含まれます。

# Preface to the First Edition

A computer is like a violin. You can imagine a novice trying first a phonograph and then a violin. The latter, he says, sounds terrible. That is the argument we have heard from our humanists and most of our computer scientists. Computer programs are good, they say, for particular purposes, but they aren't flexible. Neither is a violin, or a typewriter, until you learn how to use it.

コンピュータはヴァイオリンのような物です。初心者が最初に蓄音機、そして次にヴァイオリンを試すことを想像して下さい。彼は後者の音は酷いと言います。これが人間主義者と多くの計算機科学者から聞こえてくる議論です。計算機のプログラムは特定の目的には良い物だ、しかし柔軟性が無いと彼らは言います。ヴァイオリンやタイプライタだって同じです。あなたがその使い方を学ぶまでは。

—Marvin Minsky, “Why Programming Is a Good Medium for Expressing Poorly-Understood and Sloppily-Formulated Ideas”

“The Structure and Interpretation of Computer Programs” is the entry-level subject in computer science at the Massachusetts Institute of Technology. It is required of all students at MIT who major in electrical engineering or in computer science, as one-fourth of the “common core curriculum,” which also includes two subjects on circuits and linear systems and a subject on the design of digital systems. We have been involved in the development of this subject since 1978, and we have taught this material in its present form since the fall of 1980 to between 600 and 700 students each year. Most of these students have had little or no prior formal training in computation, although many have

played with computers a bit and a few have had extensive programming or hardware-design experience.

“The Structure and Interpretation of Computer Programs”(SICP, 計算機プログラムの構造と解釈)はマサチューセッツ工科大学(MIT)での入門者レベルの計算機科学の科目です。MITにて電気工学、または計算機工学を専攻する全ての学生が“共通コアカリキュラム”的4つの内の1つとして履修しなければなりません。共通コアカリキュラムは回路と線形システムについて2つの科目とデジタルシステムの設計についての科目を含みます。私達はこの科目の開発を1978年から行なってきました。そしてこの教材を現行様式として1980年の秋から、600名から700名の学生に毎年、教えてきました。これらの学生の多くは少し、または全くの事前の公式な計算機利用についてのトレーニングを受けてはいませんでした。ただし、多くは事前に計算機で少々遊んだ経験があり、ほんの少数は広範囲のプログラミングの経験やハードウェア設計の経験がありました。

Our design of this introductory computer-science subject reflects two major concerns. First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. Second, we believe that the essential material to be addressed by a subject at this level is not the syntax of particular programming-language constructs, nor clever algorithms for computing particular functions efficiently, nor even the mathematical analysis of algorithms and the foundations of computing, but rather the techniques used to control the intellectual complexity of large software systems.

私達のこの計算機科学の入門科目の設計は2つの主な関心事を反映しています。1つは、コンピュータ言語はコンピュータに命令を実行させるための単なる方法等ではなく、新しい種類の方法論に関する考えを表現するための公式なメディアであるという考え方を証明することです。従ってプログラムは人々が読むために書かれねばならず、そしてただ偶然に機械にとって実行する物でなければなりません。2つ目は、このレベルの科目により扱われる本質的な教材とは、特定のプログラミング言語が構築する構文ではなく、また特定の関数を効率的に演算するための賢いアルゴリズムでもなく、増してアルゴリズムと演算基盤の数理解析でないという信念です。そうではなく、大きなソフトウェアシステムの知的な複雑性をコントロールするために用いる技術でなければなりません。

Our goal is that students who complete this subject should have a good feel for the elements of style and the aesthetics of programming. They should have command of the major techniques for controlling complexity in a large system. They should be capable of reading a 50-page-long program, if it is written in an exemplary style. They should know what not to read, and what they need not understand at any moment. They should feel secure about modifying a program, retaining the spirit and style of the original author.

私達の目標は、この教科を完了した学生がプログラミングの美学とスタイルの原理に対して必ず良い感触を得ることです。学生達が大きなシステムの複雑性をコントロールするための主な技術の能力を得られなければなりません。学生達が 50 ページの長さのプログラムを、それが模範的なスタイルで書かれているのならば、読めるようにならなければなりません。学生達がプログラムの変更を行う時に、元の作者の魂とスタイルを維持しながら安心できなければなりません。

These skills are by no means unique to computer programming. The techniques we teach and draw upon are common to all of engineering design. We control complexity by building abstractions that hide details when appropriate. We control complexity by establishing conventional interfaces that enable us to construct systems by combining standard, well-understood pieces in a “mix and match” way. We control complexity by establishing new languages for describing a design, each of which emphasizes particular aspects of the design and deemphasizes others.

これらのスキルは決してコンピュータプログラミングに対して独自なことではありません。私達が教え、利用する技術は全ての工学設計に対して共通な物です。私達は適切な場合に、詳細を隠す抽象概念を構築することにより複雑性をコントロールします。標準的な、良く理解された部品を“mix and match”(様々な物をうまく組み合わせる方法)の方法により組み合わせることにより、システムを構築することを可能にする慣習的なインターフェイスを確立することで、複雑性をコントロールします。私達は設計を記述するための新しい言語を確立することで複雑性をコントロールします。そして各言語は設計の特定の側面を重要視し、他の側面の重要性を緩和します。

Underlying our approach to this subject is our conviction that “computer science” is not a science and that its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called *procedural epistemology*—the study of the structure of

knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of “what is.” Computation provides a framework for dealing precisely with notions of “how to.”

私達のこの教科に対する取り組み方の根底を成す物は、“計算機科学”は科学ではなく、その意義は計算機とは関係が無いという信念です。計算機革命とは私達の考え方と私達の考えの表現方法における革命です。この変化の本質を恐らく最もうまく言い表わすのは*procedural epistemology*(手続的認識論)—古典的な数学上の主題により取られるより宣言的な視点に対立する、命令型の視点からの知識構造の研究—の出現でしょう。数学は“何であるか”的概念を正確に扱うためのフレームワークを提供します。計算機の使用は“行い方”的概念を正確に扱うためのフレームワークを提供します。

In teaching our material we use a dialect of the programming language Lisp. We never formally teach the language, because we don't have to. We just use it, and students pick it up in a few days. This is one great advantage of Lisp-like languages: They have very few ways of forming compound expressions, and almost no syntactic structure. All of the formal properties can be covered in an hour, like the rules of chess. After a short time we forget about syntactic details of the language (because there are none) and get on with the real issues—figuring out what we want to compute, how we will decompose problems into manageable parts, and how we will work on the parts. Another advantage of Lisp is that it supports (but does not enforce) more of the large-scale strategies for modular decomposition of programs than any other language we know. We can make procedural and data abstractions, we can use higher-order functions to capture common patterns of usage, we can model local state using assignment and data mutation, we can link parts of a program with streams and delayed evaluation, and we can easily implement embedded languages. All of this is embedded in an interactive environment with excellent support for incremental program design, construction, testing, and debugging. We thank all the generations of Lisp wizards, starting with John McCarthy, who have fashioned a fine tool of unprecedeted power and elegance.

私達の教材を教えるにあたって、プログラミング言語 Lisp の一方言を使用します。私達は正式にこの言語を教えることはしません。する必要がないからです。ただそれを使い、そして学生は 2、3 日で習熟してしまいます。これは Lisp の様な言語の 1 つの利点です。これらの言語は複合式を形成する方法があまり多くありません。そしてほとんど構文構造が存在しません。形式的な特性

の全ては一時間もあればカバーできます。まるでチェスのルールの様なものです。少しの時間の後にはこの言語の構文上の詳細を忘れてしまいます。(ほとんど存在しないからです)。そして本当の問題—私達が演算したい物を把握すること、どのように問題を扱いやすい部分へと分解するか、そしてどのようにその部品上で働くかについて取り掛かります。Lisp のもう 1 つの利点は私達が知っている他のどの言語よりもプログラムを分解したモジュラに対するより多くの大規模な戦略をサポートする(しかし強制はしない)ことです。手続化とデータ抽象化を行い、公開関数を用いて処理の共通なパターンを獲得し、代入とデータの変更を用いて局所状態のモデル化を行い、プログラムの部品をストリームと遅延評価に結び付け、簡単に組込言語を実装することができます。これら全てがインタラクティブ(相互作用)な環境にインクリメンタル(漸増的な)プログラム設計、構築、テスト、デバッグのための優れたサポートと共に組込まれています。私達は前例の無い力と洗練さを供えた素晴らしいツールを創り出した John McCarthy を始めとする全ての世代の Lisp wizard(ウィザード、魔法使い、最上級のプログラマの賞賛を込めた呼び名)に感謝します。

Scheme, the dialect of Lisp that we use, is an attempt to bring together the power and elegance of Lisp and Algol. From Lisp we take the metalinguistic power that derives from the simple syntax, the uniform representation of programs as data objects, and the garbage-collected heap-allocated data. From Algol we take lexical scoping and block structure, which are gifts from the pioneers of programming-language design who were on the Algol committee. We wish to cite John Reynolds and Peter Landin for their insights into the relationship of Church's  $\lambda$ -calculus to the structure of programming languages. We also recognize our debt to the mathematicians who scouted out this territory decades before computers appeared on the scene. These pioneers include Alonzo Church, Barkley Rosser, Stephen Kleene, and Haskell Curry.

私達が用いる Lisp の方言、Scheme は Lisp と Algol の力と洗練を一緒にもたらそうとしました。Lisp からは単純な構文から導き出されるメタ言語の力、データオブジェクトとしてのプログラムの単一の表現、ガベージコレクションを持つヒープ上に取得されるデータを得ました。Algol からは Algol 委員会に在籍したプログラム設計の開拓者からの贈り物であるレキシカルスコープとブロック構造を得ました。私達は John Reynolds と Peter Landin の Church(チャーチ)の *lambda-calculus*(ラムダ計算)のプログラミング言語の構造に対する関係についての彼等の洞察に対して言及したいと願います。またコンピュータがこの世界に現れる何十年も前にこの領域を偵察された数学者達に対する恩義も忘れておりません。これらの開拓者には Alonzo Church, Barkley Rosser,

Stephen Kleene, Haskell Curry 等が含まれております。

# Acknowledgements

We would like to thank the many people who have helped us develop this book and this curriculum.

この本とこのカリキュラムの開発を手助けして下さった多くの人々に感謝致します。

Our subject is a clear intellectual descendant of “6.231,” a wonderful subject on programming linguistics and the  $\lambda$ -calculus taught at MIT in the late 1960s by Jack Wozencraft and Arthur Evans, Jr.

私達の教科は明らかに 1960 年代の終わりに MIT にて Jack Wozencraft と Arthur Evans, Jr. により教えられたプログラミング言語学と  $\lambda$  演算上の素晴らしい科目、“6.231”の知的末裔です。

We owe a great debt to Robert Fano, who reorganized MIT’s introductory curriculum in electrical engineering and computer science to emphasize the principles of engineering design. He led us in starting out on this enterprise and wrote the first set of subject notes from which this book evolved.

私達は Robert Fano に大きな借りがあります。彼は MIT の電気工学と計算機科学の導入部のカリキュラムを再編成し、工学設計の原理を重視しました。彼はこの進取性への着手に導き、またこの本への発展の元となる最初の教科ノートのまとめを記述しました。

Much of the style and aesthetics of programming that we try to teach were developed in conjunction with Guy Lewis Steele Jr., who collaborated with Gerald Jay Sussman in the initial development of the Scheme language. In addition, David Turner, Peter Henderson, Dan Friedman, David Wise, and Will Clinger have taught us many of the techniques of the functional programming community that appear in this book.

私達が教えようとするプログラミングのスタイルと美学の多くは Guy

Lewis Steele Jr. の協力の下に開発されました。彼は初期の Scheme の開発において Gerald Jay Sussman と協力を行いました。加えて David Turner, Peter Henderson, Dan Friedman, David Wise, Will Clinger が私達にこの本の中に現れる関数型プログラミングのテクニックの多くを教えてくれました。

Joel Moses taught us about structuring large systems. His experience with the Macsyma system for symbolic computation provided the insight that one should avoid complexities of control and concentrate on organizing the data to reflect the real structure of the world being modeled.

Joel Moses は私達に巨大システムの構造化について教えてくれました。彼の記号演算のための Macsyma システムにおける経験が、人は制御の複雑性を回避し、データの体系化に集中してモデル化していく世界の真の構造を反映するべきだという見識を与えてくれました。

Marvin Minsky and Seymour Papert formed many of our attitudes about programming and its place in our intellectual lives. To them we owe the understanding that computation provides a means of expression for exploring ideas that would otherwise be too complex to deal with precisely. They emphasize that a student's ability to write and modify programs provides a powerful medium in which exploring becomes a natural activity.

Marvin Minsky と Seymour Papert は私達のプログラミングに関する態度の多くと、私達の知的な生活内にその場所を形作りました。彼等に対して、考えを探求するための式の意味を演算が与えることについての理解に借りがあります。そうでなければ、正確に取り扱うためには複雑過ぎることになってしまいます。彼らは学生のプログラムを書き、変更する能力が、その中で探求が自然な活動になる強力なメディアを提供すると強調します。

We also strongly agree with Alan Perlis that programming is lots of fun and we had better be careful to support the joy of programming. Part of this joy derives from observing great masters at work. We are fortunate to have been apprentice programmers at the feet of Bill Gosper and Richard Greenblatt.

私達はまたプログラミングは大いに楽しく、このプログラミングの楽しみをサポートするために十分に注意しなければならない点について Alan Perlis に強く同意します。この楽しみの一部は作業中の偉大な職人達を観察することから得られます。私達は幸運なことに、Bill Gosper と Richard Greenblatt の下で見習いプログラマでいることができました。

It is difficult to identify all the people who have contributed to the development of our curriculum. We thank all the lecturers, recitation instructors, and tutors who have worked with us over the past fifteen years and put in many

extra hours on our subject, especially Bill Siebert, Albert Meyer, Joe Stoy, Randy Davis, Louis Braida, Eric Grimson, Rod Brooks, Lynn Stein and Peter Szolovits. We would like to specially acknowledge the outstanding teaching contributions of Franklyn Turbak, now at Wellesley; his work in undergraduate instruction set a standard that we can all aspire to. We are grateful to Jerry Saltzer and Jim Miller for helping us grapple with the mysteries of concurrency, and to Peter Szolovits and David McAllester for their contributions to the exposition of nondeterministic evaluation in Chapter 4.

私達のカリキュラムの開発に貢献して下さった全ての人々を特定することは難しいことです。私達は過去 15 年私達と共に働き、多くの時間を私達の教科に費してくれた全ての講師、口答の指導者、チューターに、特に、Bill Siebert, Albert Meyer, Joe Stoy, Randy Davis, Louis Braida, Eric Grimson, Rod Brooks, Lynn Stein and Peter Szolovits に感謝します。私達は特に卓越した教育上の貢献として現在はウェルズリーの Franklyn Turbak に感謝します。彼の学部生向け指導要項は私達皆が目指す基準を打ち立てました。Jerry Saltzer と Jim Miller には私達が並行性のミステリーを取り組むのを手助けして下さったことに感謝します。そして Peter Szolovits と David McAllester にはChapter 4における非決定性評価の説明に対する貢献に感謝します。

Many people have put in significant effort presenting this material at other universities. Some of the people we have worked closely with are Jacob Katzenelson at the Technion, Hardy Mayer at the University of California at Irvine, Joe Stoy at Oxford, Elisha Sacks at Purdue, and Jan Komorowski at the Norwegian University of Science and Technology. We are exceptionally proud of our colleagues who have received major teaching awards for their adaptations of this subject at other universities, including Kenneth Yip at Yale, Brian Harvey at the University of California at Berkeley, and Dan Huttenlocher at Cornell.

多くの人々は他大学でこの資料を紹介するのに大きな努力を費してくださいました。私達が親密に働いたそれらの人々の幾人かはイスラエル工科大学の Jacob Katzenelson、カリフォルニア大学アーバイン校の Hardy Mayer、オックスフォード大学の Joe Stoy、パデュー大学の Elisha Sacks、ノルウェー技術科学大学の Jan Komorowski です。私達は他大学においてこの科目を受け入れることで主要な教育の賞を受けた同僚達を非常に誇りに思います。この中にはイェール大学の Kenneth Yip、カリフォルニア大学バークリー校の Brian Harvey、コーネル大学の Dan Huttenlocher を含みます。

Al Moyé arranged for us to teach this material to engineers at Hewlett-Packard, and for the production of videotapes of these lectures. We would like

to thank the talented instructors—in particular Jim Miller, Bill Siebert, and Mike Eisenberg—who have designed continuing education courses incorporating these tapes and taught them at universities and industry all over the world.

Al Moyé は私たちのためにこの教材を HP の技術者達に教える手筈とのレクチャーのビデオテープの製品化を準備してくれました。私たちはまた才能あるインストラクター達にも感謝致します。具体的には Jim Miller, Bill Siebert, Mike Eisenberg です。彼等はこれらのテープを組み込んで生涯教育のコースを設計し、世界中の大学と業界にて教育を行いました。

Many educators in other countries have put in significant work translating the first edition. Michel Briand, Pierre Chamard, and André Pic produced a French edition; Susanne Daniels-Herold produced a German edition; and Fumio Motoyoshi produced a Japanese edition. We do not know who produced the Chinese edition, but we consider it an honor to have been selected as the subject of an “unauthorized” translation.

他国の多くの教育者が多大な時間を第一版の翻訳に費して下さいました。 Michel Briand, Pierre Chamard, and André Pic はフランス語版をプロデュースして下さいました。 Susanne Daniels-Herold はドイツ語版をプロデュースして下さいました。元吉文男は日本語版をプロデュースして下さいました。私たちはどなたが中国語版をプロデュースして下さったのか知りません。しかし“未許可”の翻訳の題材として選ばれたことを光栄に思います。

It is hard to enumerate all the people who have made technical contributions to the development of the Scheme systems we use for instructional purposes. In addition to Guy Steele, principal wizards have included Chris Hanson, Joe Bowbeer, Jim Miller, Guillermo Rozas, and Stephen Adams. Others who have put in significant time are Richard Stallman, Alan Bawden, Kent Pitman, Jon Taft, Neil Mayle, John Lamping, Gwyn Osnos, Tracy Larabee, George Carrette, Soma Chaudhuri, Bill Chiarchiaro, Steven Kirsch, Leigh Klotz, Wayne Noss, Todd Cass, Patrick O'Donnell, Kevin Theobald, Daniel Weise, Kenneth Sinclair, Anthony Courtemanche, Henry M. Wu, Andrew Berlin, and Ruth Shyu.

私たちが教育の目的のために使用する Scheme システムの開発に技術的な貢献をされた全ての人々を列挙することは難しいことです。 Guy Steele に加えて、主要なウィザードの中には Chris Hanson, Joe Bowbeer, Jim Miller, Guillermo Rozas, and Stephen Adams が含まれます。多大な時間を費して下さった他の人々は Richard Stallman, Alan Bawden, Kent Pitman, Jon Taft, Neil Mayle, John Lamping, Gwyn Osnos, Tracy Larabee, George Carrette, Soma

Chaudhuri, Bill Chiarchiaro, Steven Kirsch, Leigh Klotz, Wayne Noss, Todd Cass, Patrick O'Donnell, Kevin Theobald, Daniel Weise, Kenneth Sinclair, Anthony Courtemanche, Henry M. Wu, Andrew Berlin, それに Ruth Shyu です。

Beyond the MIT implementation, we would like to thank the many people who worked on the IEEE Scheme standard, including William Clinger and Jonathan Rees, who edited the R<sup>4</sup>RS, and Chris Haynes, David Bartley, Chris Hanson, and Jim Miller, who prepared the IEEE standard.

MIT の実装を越えて、私たちは IEEE の Scheme 標準仕様について働いた多くの人々に感謝したいと思います。R<sup>4</sup>RS を編集した William Clinger と Jonathan Rees、IEEE 標準を準備した Chris Haynes, David Bartley, Chris Hanson, Jim Miller を含みます。

Dan Friedman has been a long-time leader of the Scheme community. The community's broader work goes beyond issues of language design to encompass significant educational innovations, such as the high-school curriculum based on EdScheme by Schemer's Inc., and the wonderful books by Mike Eisenberg and by Brian Harvey and Matthew Wright.

Dan Friedman は長い間 Scheme コミュニティのリーダーでした。コミュニティの広範な仕事は言語設計の問題を越えて、Schemer's Inc. による EdScheme を基にした高校生向けカリキュラムや Mike Eisenberg や Brian Harvey と Matthew Wright による素晴らしい本のような、特筆すべき教育上のイノベーションを含むまでに至りました。

We appreciate the work of those who contributed to making this a real book, especially Terry Ehling, Larry Cohen, and Paul Bethge at the MIT Press. Ella Mazel found the wonderful cover image. For the second edition we are particularly grateful to Bernard and Ella Mazel for help with the book design, and to David Jones, T<sub>E</sub>X wizard extraordinaire. We also are indebted to those readers who made penetrating comments on the new draft: Jacob Katzenelson, Hardy Mayer, Jim Miller, and especially Brian Harvey, who did unto this book as Julie did unto his book *Simply Scheme*.

私たちはこの本を現実にすることに貢献して下さった人々の働きに感謝致します。特に MIT 出版の Terry Ehling, Larry Cohen, Paul Bethge です。Ella Mazel は素晴らしいカバーの絵を見つけてくれました。第二版に対しては特にこの本のデザインを助けてくれた Bernard と Ella の Mazel 夫妻、非凡な T<sub>E</sub>X ウィザードである David Jones に感謝致します。私たちはまた新しいドラフトに対し洞察力のあるコメントをして下さった読者の方々、Jacob Katzenelson, Hardy Mayer, Jim Miller, そして特に Brian Harvey に対して、Julie が彼の本

*Simply Scheme* に行ったように、この本に行ってくれたことを感謝致します。

Finally, we would like to acknowledge the support of the organizations that have encouraged this work over the years, including support from Hewlett-Packard, made possible by Ira Goldstein and Joel Birnbaum, and support from DARPA, made possible by Bob Kahn.

最後に、何年にも渡ったこの仕事を励まして下さった組織のサポートに感謝したいと思います。Hewlett-Packard からのサポートを可能にして下さった Ira Goldstein と Joel Birnbaum、それに DARPA からのサポートを可能にして下さった Bob Kahn を含みます。

# 1

## 手続を用いた抽象化の構築

The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three: 1. Combining several simple ideas into one compound one, and thus all complex ideas are made. 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations. 3. The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.

心がその中で、その力を単純な考えの上に発揮する“心の働き”は、主としてこれら3つです。1. いくつかの簡単な考えを1つの複合物に組み合わせます。全ての複雑な考えはこのようにして作られます。2. 2つの考えをそれが簡単であるか複雑であるかに係らず一緒にもたらし、お互いに合わせることでそれらを統合することは無しに、全ての関係性の考えを得ることで、一度にそれらを見渡します。3. 考えをそれらの実在に付随する全ての他の考えから分離します。これは抽象化と呼ばれ、このようにして全てのその一般的な考えは作られます。

—John Locke, *An Essay Concerning Human Understanding* (1690)

We are about to study the idea of a *computational process*. Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called *data*. The evolution of a process is directed by a pattern of rules called a *program*. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells.

私達は*computational process*(演算プロセス)について学びます。演算プロセスとは抽象的な存在でコンピュータの中に複数が住んでいます。それらが進化するとプロセスは*data*(データ)と呼ばれるまた別の抽象的な物を扱います。プロセスの進化は*program*(プログラム)と呼ばれるルールのパターン(型、類型)により命じられます。人はプログラムを作成してプロセスに命ずるのです。つまり私達はコンピュータの精霊に私達の呪文で魔法をかけるのです。

A computational process is indeed much like a sorcerer's idea of a spirit. It cannot be seen or touched. It is not composed of matter at all. However, it is very real. It can perform intellectual work. It can answer questions. It can affect the world by disbursing money at a bank or by controlling a robot arm in a factory. The programs we use to conjure processes are like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric *programming languages* that prescribe the tasks we want our processes to perform.

演算プロセスは本当に魔法使いの精霊の考えに似ています。それは見たり触れたりはできません。物理的な物では構成されていません。しかしどもリアルな存在です。知的な仕事を行います。質問にも答えます。銀行でお金を払ったり、工場でロボットの腕をコントロールすることで世界に影響を与えることも可能です。私達が利用するプロセスに魔法をかけるプログラムとは魔法使いの呪文のような物です。難解な秘伝の *programming languages*(プログラミング言語)の中で記号的表現にて慎重に組み立てられプロセスに実行してほしいタスク(仕事、任務)を指示します。

A computational process, in a correctly working computer, executes programs precisely and accurately. Thus, like the sorcerer's apprentice, novice programmers must learn to understand and to anticipate the consequences of their conjuring. Even small errors (usually called *bugs* or *glitches*) in programs can have complex and unanticipated consequences.

演算プロセスは、正しく動くコンピュータでは、精密に正しくプログラムを実行します。従って魔法使いの見習いのように、初心者のプログラマは魔法の結果について理解し、予測することを学ばねばなりません。例えプログラム

の小さなエラー（通常は*bugs*(バグ)、または*glitches*(グリッチ、誤作動)と呼ばれます）でも複雑で予測不可能な結果をもたらす場合もあるのです。

Fortunately, learning to program is considerably less dangerous than learning sorcery, because the spirits we deal with are conveniently contained in a secure way. Real-world programming, however, requires care, expertise, and wisdom. A small bug in a computer-aided design program, for example, can lead to the catastrophic collapse of an airplane or a dam or the self-destruction of an industrial robot.

幸運なことに、プログラムを学ぶことは魔法を学ぶことより少しも危険ではありません。私達が相手にする精靈は都合良く安全な方法で封じ込まれています。しかし、実際の世界でのプログラミングには注意力、専門知識、堅実さを必要とします。例えばCAD(計算機による設計支援)プログラムの小さなバグが飛行機やダムの最悪な崩壊に繋ったり、工業ロボットの自己破壊を起こしたりします。

Master software engineers have the ability to organize programs so that they can be reasonably sure that the resulting processes will perform the tasks intended. They can visualize the behavior of their systems in advance. They know how to structure programs so that unanticipated problems do not lead to catastrophic consequences, and when problems do arise, they can *debug* their programs. Well-designed computational systems, like well-designed automobiles or nuclear reactors, are designed in a modular manner, so that the parts can be constructed, replaced, and debugged separately.

ソフトウェアエンジニアのマスター達は最終的にプロセスが望まれたタスクを実行することに自信持てるだけの技能を、プログラムの構築に対して持っています。彼らは前もってシステムの行いを図で説明することができます。予測不可能な問題が最悪な結果をもたらさぬようプログラムをどのように構造化を行うのか知っています。そして問題が発生した時にはプログラムの *debug*(デバッグ、バグ取り)を行えます。良い設計のコンピュータシステムは、良い設計の自動車や原子炉のようにモジュール方式で設計されており、パーツは個別に組み立て、置き換え、デバッグが可能です。

## Lisp プログラミング

We need an appropriate language for describing processes, and we will use for this purpose the programming language Lisp. Just as our everyday thoughts are usually expressed in our natural language (such as English, French,

or Japanese), and descriptions of quantitative phenomena are expressed with mathematical notations, our procedural thoughts will be expressed in Lisp. Lisp was invented in the late 1950s as a formalism for reasoning about the use of certain kinds of logical expressions, called *recursion equations*, as a model for computation. The language was conceived by John McCarthy and is based on his paper “Recursive Functions of Symbolic Expressions and Their Computation by Machine” (McCarthy 1960).

私達はプロセスを記述するのに適切な言語を必要とします。この目的に対しプログラミング言語 Lisp を利用します。私達の日々の考えが通常、自然言語(例えば英語やフランス語、日本語)で表されるように、定量的な現象が数学の記号で表されるように、手続的な思考は Lisp で表現されます。Lisp は 1950 年代後半に *recursion equations*(再帰方程式) と呼ばれるある種の論理表現に関する推論のための形式化として開発されました。この言語は John McCarthy により着想され、彼の論文“記号式の再帰方程式とそれらの機械による演算”(McCarthy 1960) を基にしています。

Despite its inception as a mathematical formalism, Lisp is a practical programming language. A Lisp *interpreter* is a machine that carries out processes described in the Lisp language. The first Lisp interpreter was implemented by McCarthy with the help of colleagues and students in the Artificial Intelligence Group of the MIT Research Laboratory of Electronics and in the MIT Computation Center.<sup>1</sup> Lisp, whose name is an acronym for LISt Processing, was designed to provide symbol-manipulating capabilities for attacking programming problems such as the symbolic differentiation and integration of algebraic expressions. It included for this purpose new data objects known as atoms and lists, which most strikingly set it apart from all other languages of the period.

数学上の形式主義としての始まりにも関わらず、Lisp は実用的なプログラミング言語です。Lisp *interpreter*(インタプリタ、逐次翻訳処理器) は Lisp 言語にて記述されたプロセスを実行する機械です。最初の Lisp インタプリタは McCarthy と MIT 研究所の人工知能部門の同僚、学生による手助けにて実装されました。<sup>2</sup> Lisp はその名前を LISt Processing(リスト処理) の頭文字から取つ

---

<sup>1</sup>The *Lisp 1 Programmer’s Manual* appeared in 1960, and the *Lisp 1.5 Programmer’s Manual* (McCarthy et al. 1965) was published in 1962. The early history of Lisp is described in McCarthy 1978.

<sup>2</sup>*Lisp 1 Programmer’s Manual* は 1960 年に初出し、*Lisp 1.5 Programmer’s Manual* (McCarthy et al. 1965) は 1962 年に出版されました。Lisp の初期の歴史は McCarthy 1978 にて説明されています。

ており、記号微分や代数式の積分の様なプログラミング上の問題に着手するための記号操作能力を提供するために設計されました。この目的のためにアトムとリストとして知られる新しいデータオブジェクトを含みます。これはその時代の他の全ての言語から著しく際立たせる物でした。

Lisp was not the product of a concerted design effort. Instead, it evolved informally in an experimental manner in response to users' needs and to pragmatic implementation considerations. Lisp's informal evolution has continued through the years, and the community of Lisp users has traditionally resisted attempts to promulgate any "official" definition of the language. This evolution, together with the flexibility and elegance of the initial conception, has enabled Lisp, which is the second oldest language in widespread use today (only Fortran is older), to continually adapt to encompass the most modern ideas about program design. Thus, Lisp is by now a family of dialects, which, while sharing most of the original features, may differ from one another in significant ways. The dialect of Lisp used in this book is called Scheme.<sup>3</sup>

Lisp は計画的な設計の取り組みから生まれた製品ではありませんでした。そうではなく、非公式に試験的なやり方で、ユーザの要求と実利的な実装上の考慮への対応として発展しました。Lisp の非公式な進化は何年も続き、Lisp ユーザのコミュニティは伝統的に言語の“公式な”どんな定義の公表に対しても抵抗しました。この進化は初期構想の柔軟性と洗練さと共に、今日世界中で広く使用される言語で 2 番目に古い (Fortran のみがより古い) 言語として、Lisp に継続的に最新のプログラム設計についての考えを受け入れることを可能にしてきました。従って Lisp は今では複数の方言の系統が存在し、それらはオリジ

<sup>3</sup>The two dialects in which most major Lisp programs of the 1970s were written are MacLisp (Moon 1978; Pitman 1983), developed at the MIT Project MAC, and Interlisp (Teitelman 1974), developed at Bolt Beranek and Newman Inc. and the Xerox Palo Alto Research Center. Portable Standard Lisp (Hearn 1969; Griss 1981) was a Lisp dialect designed to be easily portable between different machines. MacLisp spawned a number of subdialects, such as Franz Lisp, which was developed at the University of California at Berkeley, and Zetalisp (Moon and Weinreb 1981), which was based on a special-purpose processor designed at the MIT Artificial Intelligence Laboratory to run Lisp very efficiently. The Lisp dialect used in this book, called Scheme (Steele and Sussman 1975), was invented in 1975 by Guy Lewis Steele Jr. and Gerald Jay Sussman of the MIT Artificial Intelligence Laboratory and later reimplemented for instructional use at MIT. Scheme became an IEEE standard in 1990 (IEEE 1990). The Common Lisp dialect (Steele 1982, Steele 1990) was developed by the Lisp community to combine features from the earlier Lisp dialects to make an industrial standard for Lisp. Common Lisp became an ANSI standard in 1994 (ANSI 1994).

ナルの機能の多くを共有しながらも、お互いに大きな違いを持ちます。この本で使用される Lisp の方言は Scheme と呼ばれます。<sup>4</sup>

Because of its experimental character and its emphasis on symbol manipulation, Lisp was at first very inefficient for numerical computations, at least in comparison with Fortran. Over the years, however, Lisp compilers have been developed that translate programs into machine code that can perform numerical computations reasonably efficiently. And for special applications, Lisp has been used with great effectiveness.<sup>5</sup> Although Lisp has not yet overcome its old reputation as hopelessly inefficient, Lisp is now used in many applications where efficiency is not the central concern. For example, Lisp has become a language of choice for operating-system shell languages and for extension languages for editors and computer-aided design systems.

実験的であるという特徴と記号操作の重要性のため、Lisp は初期においては数値演算に対し少くとも Fortran との比較にてとても非効率でした。しかし年を追って、プログラムを機械語に変換し、数値演算を適度に効率良く実行可能な Lisp コンパイラが開発されました。特別なアプリケーションに対しては Lisp は最高の効果を発揮しています。<sup>6</sup> Lisp は今でもどうしようもなく非効率

<sup>4</sup>1970 年代に最もメジャーな Lisp プログラムの記述に用いられた 2 つの方言は MIT のプロジェクト MAC で開発された MacLisp (Moon 1978; Pitman 1983) と Bolt Beranek and Newman Inc. と Xerox Palo Alto 研究センターにて開発された Interlisp (Teitelman 1974) でした。Portable Standard Lisp (Hearn 1969; Griss 1981) は簡単に、異なるマシンの間で移植可能にするよう設計された Lisp 方言です。MacLisp はカリフォルニア大学バークレー校により開発された Franz Lisp や MIT 人工知能研究所が Lisp をとても効率良く実行するために設計した特定目的プロセッサ (処理機) をベースにした Zetalisp (Moon and Weinreb 1981) といつたいくつかの下位方言を生みました。この本で使用する Lisp 方言は Scheme (Steele and Sussman 1975) と呼ばれ、1975 年に MIT 人工知能研究所の Guy Lewis Steele Jr. と Gerald Jay Sussman により開発され、後に MIT にて教育目的のために再実装されました。Common Lisp (Steele 1982, Steele 1990) は Lisp コミュニティにより初期の Lisp 方言の機能を集約し、Lisp の業界標準を作成するために開発されました。Common Lisp は 1994 年に ANSI 標準 (ANSI 1994) になりました。

<sup>5</sup>One such special application was a breakthrough computation of scientific importance—an integration of the motion of the Solar System that extended previous results by nearly two orders of magnitude, and demonstrated that the dynamics of the Solar System is chaotic. This computation was made possible by new integration algorithms, a special-purpose compiler, and a special-purpose computer all implemented with the aid of software tools written in Lisp (Abelson et al. 1992; Sussman and Wisdom 1992).

<sup>6</sup>そのような特別なアプリケーションの 1 つは自然科学上の重大な計算、太陽系の動き

であるという古い評判を乗り越えられてはいませんが、Lisp は今では多くのアプリケーションにて、効率が問題の中心ではない場合において利用されています。例えば Lisp は OS のシェル言語やエディタの拡張言語、CAD システム等において選択言語となっています。

If Lisp is not a mainstream language, why are we using it as the framework for our discussion of programming? Because the language possesses unique features that make it an excellent medium for studying important programming constructs and data structures and for relating them to the linguistic features that support them. The most significant of these features is the fact that Lisp descriptions of processes, called *procedures*, can themselves be represented and manipulated as Lisp data. The importance of this is that there are powerful program-design techniques that rely on the ability to blur the traditional distinction between “passive” data and “active” processes. As we shall discover, Lisp’s flexibility in handling procedures as data makes it one of the most convenient languages in existence for exploring these techniques. The ability to represent procedures as data also makes Lisp an excellent language for writing programs that must manipulate other programs as data, such as the interpreters and compilers that support computer languages. Above and beyond these considerations, programming in Lisp is great fun.

もし Lisp がメインストリームの言語でなければなぜ私達はプログラミングの議論のためのフレームワークとしてそれを用いるのでしょうか? なぜならこの言語は重要なプログラミング構成概念とデータ構造を学ぶため、またそれらをサポートする言語上の機能にそれらを関連付けするために、言語自身を洗練された媒体と成す個有の機能を持っているためです。これらの機能で最も著しい物は、Lisp による *procedures*(プロシージャ、手続) と呼ばれるプロセスの記述が、それ自身が Lisp のデータとして表現され、また操作されることが可能であるという事実です。これ的重要性は、伝統的な“受動的な”データと“能動的な”プロセスとの間の区別をぼかす能力に依存する、強力なプログラム設計のテクニックが存在するということです。私達がそれを発見するにつれ、手続をデータとして扱う Lisp の柔軟性は Lisp をこれらのテクニックを探求するのに、既存で最も便利な言語の 1 つとします。手続をデータとして表現する能力はまた、Lisp を他のプログラムをデータとして操作しなければならないプログラム

---

の統合におけるブレイクスルーでした。これは以前の結果より二桁も良く、太陽系の活動が混沌であることを実演しました。この計算は全て Lisp で書かれたソフトウェアツールの手助けにより実装された新しい統合アルゴリズム、特定目的のコンパイラ、特定目的の計算機により可能となりました。(Abelson et al. 1992; Sussman and Wisdom 1992)

を書く目的に対し洗練された言語にします。例えばコンピュータ言語に対応するインタプリタやコンパイラのようなプログラムです。これらの考慮点に加えて、Lispによるプログラミングはとても楽しいのです。

## 1.1 The Elements of Programming

A powerful programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organize our ideas about processes. Thus, when we describe a language, we should pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas. Every powerful language has three mechanisms for accomplishing this:

強力なプログラミング言語はコンピュータにタスクの実行を指示するだけではありません。そのような言語は私達がプロセスについての自らの考えを体系化するフレームワークとしての役目を担います。従って言語を記述する時、簡単なアイデアを組み合わせてより複雑なアイデアを形成するという手段をその言語が提供することには特に注意を払わねばなりません。強力な言語全てがこれを達成するために3つのメカニズムを持っています。

- プリミティブな式, 言語に関わる最も単純な要素を表現する
- 合成化の手段, by which compound elements are built from simpler ones, and これにより、より単純なものより複合要素が構築される
- 抽象化の手段, by which compound elements can be named and manipulated as units. これにより複合要素は名前を付けて個体として扱える

In programming, we deal with two kinds of elements: procedures and data. (Later we will discover that they are really not so distinct.) Informally, data is “stuff” that we want to manipulate, and procedures are descriptions of the rules for manipulating the data. Thus, any powerful programming language should be able to describe primitive data and primitive procedures and should have methods for combining and abstracting procedures and data.

プログラミングにおいては2つの種類の要素を扱います。手続(procedure)とデータです。(後でそれらはあまりはっきりとは区別できないことを明かします。)簡単に説明するとデータは操作対象の“物”で手続はデータの操作のためのルールの記述です。従って強力なプログラミング言語はどれもプリミティブ(原始的な、最低レベルの、組込の)なデータとプリミティブな手続を記述可

能でなければならず、また手続とデータを合成化、抽象化する手法を持たなければなりません。

In this chapter we will deal only with simple numerical data so that we can focus on the rules for building procedures.<sup>7</sup> In later chapters we will see that these same rules allow us to build procedures to manipulate compound data as well.

この章では単純な数値データのみを扱うことにより、手続構築のためのルールに集中します。<sup>8</sup>後の章では同じこれらのルールにより複合データもまた構築できることを学びます。

---

<sup>7</sup>The characterization of numbers as “simple data” is a barefaced bluff. In fact, the treatment of numbers is one of the trickiest and most confusing aspects of any programming language. Some typical issues involved are these: Some computer systems distinguish *integers*, such as 2, from *real numbers*, such as 2.71. Is the real number 2.00 different from the integer 2? Are the arithmetic operations used for integers the same as the operations used for real numbers? Does 6 divided by 2 produce 3, or 3.0? How large a number can we represent? How many decimal places of accuracy can we represent? Is the range of integers the same as the range of real numbers? Above and beyond these questions, of course, lies a collection of issues concerning roundoff and truncation errors—the entire science of numerical analysis. Since our focus in this book is on large-scale program design rather than on numerical techniques, we are going to ignore these problems. The numerical examples in this chapter will exhibit the usual roundoff behavior that one observes when using arithmetic operations that preserve a limited number of decimal places of accuracy in noninteger operations.

<sup>8</sup>数値を“単純なデータ”と特徴付けるのは公然なウソです。実際に数値の扱いは任意のプログラミング言語において最も油断ならない、混乱を招く要素です。いくつかの典型的な問題は次のものです。いくつかのコンピュータシステムは2のような*integers*(整数)と2.71のような*real numbers*(実数)の区別をします。実数2.00は整数2とは異なるでしょうか?整数に用いられる算術演算は実数に対する物と同じでしょうか?6を2で割ったら3?それとも3.0?どれだけ大きな数値を表示できますか?精度は小数何桁まで正しく表わされますか?整数の範囲は実数の範囲と同じですか?もちろんこれらの質問の他にも丸めと切り捨てに関する誤差の問題の蓄積といった数値解析の科学全体が存在します。この本のフォーカスは大規模なプログラム設計であり数値演算向けのテクニックではないのでこれらの問題は無視することにします。この章の数値演算の例では非整数演算において精度上正確な桁数に制限を持つ算術演算を用いる場合に一般的な丸めの方法を示します。

### 1.1.1 Expressions

One easy way to get started at programming is to examine some typical interactions with an interpreter for the Scheme dialect of Lisp. Imagine that you are sitting at a computer terminal. You type an *expression*, and the interpreter responds by displaying the result of its *evaluating* that expression.

プログラミングを始める 1 つの簡単な方法はいくつかの典型的な対話を Lisp の方言である Scheme のインタプリタを用いて試してみることです。コンピュータの端末の前に座っていると想像してみてください。あなたが *expression(式)* を入力するとインタプリタはその式の *evaluation(評価)* の結果を表示することで応答します。

One kind of primitive expression you might type is a number. (More precisely, the expression that you type consists of the numerals that represent the number in base 10.) If you present Lisp with a number

あなたが入力するプリミティブな式の一種として数値があります。(より正確にはあなたが入力する式は 10 進数の数値を表す数字から成り立ちます。) もし数値を Lisp に与えた場合、

486

the interpreter will respond by printing <sup>9</sup>

インタプリタは以下を表示することで応答します。<sup>10</sup>

486

Expressions representing numbers may be combined with an expression representing a primitive procedure (such as + or \*) to form a compound expression that represents the application of the procedure to those numbers. For example:

数値を表す式はプリミティブな手続を表す式(例えば + や \*) と接続することで複合式を形成し、それら数値に対し手続を適用することを表現します。例えば:

(+ 137 349)

486

---

<sup>9</sup>Throughout this book, when we wish to emphasize the distinction between the input typed by the user and the response printed by the interpreter, we will show the latter in slanted characters.

<sup>10</sup>この本を通して、ユーザの入力とインタプリタが表示した応答を区別したい場合、傾いた文字で表します。

(- 1000 334)

666

(\* 5 99)

495

(/ 10 5)

2

(+ 2.7 10)

12.7

Expressions such as these, formed by delimiting a list of expressions within parentheses in order to denote procedure application, are called *combinations*. The leftmost element in the list is called the *operator*, and the other elements are called *operands*. The value of a combination is obtained by applying the procedure specified by the operator to the *arguments* that are the values of the operands.

これらのような式は括弧の中の式のリストを区切ることにより形成され手続の適用を示し、*combinations*(組み合わせ)と呼ばれます。リストの最も左の要素は*operator*(オペレータ、演算子)と呼ばれ、他の要素は*operand*(オペランド、被演算数)と呼ばれます。組み合わせの値はオペレータにより与えられた手続をオペランドの値である*arguments*(引数)に適用することで得られます。

The convention of placing the operator to the left of the operands is known as *prefix notation*, and it may be somewhat confusing at first because it departs significantly from the customary mathematical convention. Prefix notation has several advantages, however. One of them is that it can accommodate procedures that may take an arbitrary number of arguments, as in the following examples:

オペレータをオペランドの左に置く決まりは*prefix notation*(前置表記法)として知られています。最初の内は数学の決まりから明らかに逸脱するので混乱するかもしれません。しかし、前置表記法にはいくつかの利点が存在します。その1つは以下の例のように、任意の数の引数を取る手続に適応できることです。

(+ 21 35 12 7)

75

```
(* 25 4 12)  
1200
```

No ambiguity can arise, because the operator is always the leftmost element and the entire combination is delimited by the parentheses.

曖昧さが全くありません。オペレータが常に最も左の要素であり、合成全体は括弧で区切られているためです。

A second advantage of prefix notation is that it extends in a straightforward way to allow combinations to be *nested*, that is, to have combinations whose elements are themselves combinations:

前置表記法の2つ目の利点は直接的な方法にて組み合わせを *nested*(ネスト、入れ子) にすることが可能です。つまり、組み合わせの要素それ自体が組み合わせである場合です。

```
(+ (* 3 5) (- 10 6))  
19
```

There is no limit (in principle) to the depth of such nesting and to the overall complexity of the expressions that the Lisp interpreter can evaluate. It is we humans who get confused by still relatively simple expressions such as

原理的にはそのようなネストの深さと Lisp インタプリタが評価可能な式全体の複雑さには制限がありません。しかし私達人間は以下のような比較的単純な式でも混乱してしまいます。

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

which the interpreter would readily evaluate to be 57. We can help ourselves by writing such an expression in the form

インタプリタは直ちに 57 だと評価するでしょう。このような式を次のような形式で記述することで私達自身を助けることが可能です。

```
(+ (* 3  
      (+ (* 2 4)  
           (+ 3 5)))  
    (+ (- 10 7)  
       6))
```

following a formatting convention known as *pretty-printing*, in which each long combination is written so that the operands are aligned vertically. The resulting

indentations display clearly the structure of the expression.<sup>11</sup>

*pretty-printing*(プリティプリント、整形)として知られるフォーマットの決まりに個々の長いオペランドを従わせることで、オペランドが垂直方向で位置合わせされます。結果的に式の構造が明確にインデント(字下げ)されることになります。<sup>12</sup>

Even with complex expressions, the interpreter always operates in the same basic cycle: It reads an expression from the terminal, evaluates the expression, and prints the result. This mode of operation is often expressed by saying that the interpreter runs in a *read-eval-print loop*. Observe in particular that it is not necessary to explicitly instruct the interpreter to print the value of the expression.<sup>13</sup>

例え複雑な式でもインタプリタは常に同じ基本的なサイクルにて処理を行います。式を端末から読み、その式を評価し、結果を表示します。この操作モードはしばしばインタプリタが*read-eval-print loop*(REPL: レップル)で実行されていると呼ばれます。特に明示的にインタプリタに式の値を表示しろと命令する必要がないことに注意して下さい。<sup>14</sup>

### 1.1.2 Naming and the Environment

A critical aspect of a programming language is the means it provides for using names to refer to computational objects. We say that the name identifies a *variable* whose *value* is the object.

---

<sup>11</sup>Lisp systems typically provide features to aid the user in formatting expressions. Two especially useful features are one that automatically indents to the proper pretty-print position whenever a new line is started and one that highlights the matching left parenthesis whenever a right parenthesis is typed.

<sup>12</sup>典型的な Lisp システムは式を整形しユーザを手助けするための機能を提供します。特に便利な 2 つの機能において、1 つは新しい行がどこで始まろうとも自動的に正しい整形位置にインデントします。もう 1 つは右括弧が入力された時に対応する左括弧がハイライトされます。

<sup>13</sup>Lisp obeys the convention that every expression has a value. This convention, together with the old reputation of Lisp as an inefficient language, is the source of the quip by Alan Perlis (paraphrasing Oscar Wilde) that “Lisp programmers know the value of everything but the cost of nothing.”

<sup>14</sup>Lisp は各式が値を持つという決まりに従います。Lisp が非効率な言語であるという古い噂と共に、この決まりが Alan Perlis による Oscar Wilde をもじった皮肉のネタ元になっています。曰く“Lisp プログラマは全ての値を知っているがそのコストはどれについても知らない”

プログラミング言語の重要な特徴は演算対象を参照するための名前を利用するためにはそれが提供する手段です。名前は *value(値)* としてオブジェクトを持つ *variable(変数)* を識別します。

In the Scheme dialect of Lisp, we name things with `define`. Typing Lisp の方言 Scheme では対象に `define(定義)` を用いて名前を付けます。以下のように入力すると

```
(define size 2)
```

causes the interpreter to associate the value 2 with the name `size`.<sup>15</sup> Once the name `size` has been associated with the number 2, we can refer to the value 2 by name:

インタプリタは名前 `size` と値 2 を関連付けます。<sup>16</sup> 一度名前 `size` が数値 2 に関連付けられれば値 2 を名前で参照することが可能です。

```
size  
2  
  
(* 5 size)  
10
```

Here are further examples of the use of `define`:

より多くの `define` の使用例を見ましょう。

```
(define pi 3.14159)  
(define radius 10)  
(* pi (* radius radius))  
314.159  
(define circumference (* 2 pi radius))  
circumference  
62.8318
```

`Define` is our language's simplest means of abstraction, for it allows us to use simple names to refer to the results of compound operations, such as the `circumference` computed above. In general, computational objects may have

---

<sup>15</sup>In this book, we do not show the interpreter's response to evaluating definitions, since this is highly implementation-dependent.

<sup>16</sup>この本ではインタプリタの定義に対する評価の応答を表示しません。とても実装依存であるためです。

very complex structures, and it would be extremely inconvenient to have to remember and repeat their details each time we want to use them. Indeed, complex programs are constructed by building, step by step, computational objects of increasing complexity. The interpreter makes this step-by-step program construction particularly convenient because name-object associations can be created incrementally in successive interactions. This feature encourages the incremental development and testing of programs and is largely responsible for the fact that a Lisp program usually consists of a large number of relatively simple procedures.

`define` は言語の最も単純な抽象化の手段です。簡単な名前を使用して合成命令の結果を参照することを可能にします。例えば上の例で計算した `circumference`(円周) です。一般的に演算対象はとても複雑な構造を持ち、それを覚えて使用時に詳細を繰り返し記述することはとても面倒です。実際に、複雑なプログラムは少しづつ複雑さを増していく演算対象を 1 つづつ構築して組み立てられます。インタプリタはこのプログラム組立の各ステップに特に便利です。なぜなら名前とオブジェクトの関連性が連続した対話を通して少しづつ作成可能なためです。この機能は漸進的開発とプログラムのテストを促進し、Lisp プログラムが通常数多くの比較的単純な手続により構成される理由です。

It should be clear that the possibility of associating values with symbols and later retrieving them means that the interpreter must maintain some sort of memory that keeps track of the name-object pairs. This memory is called the *environment* (more precisely the *global environment*, since we will see later that a computation may involve a number of different environments).<sup>17</sup>

値に記号を関連付け、後にそれらを取り出すことがあるのは、インタプリタが名前とオブジェクトのペアを追跡するためのある種のメモリを持たなければいけないことを意味することは明白でしょう。このメモリは *environment*(環境)(より正確には *global environment*(大域環境)、演算には複数の異なる環境が利用されることを後に学ぶため) と呼ばれます。<sup>18</sup>

---

<sup>17</sup> Chapter 3 will show that this notion of environment is crucial, both for understanding how the interpreter works and for implementing interpreters.

<sup>18</sup> Chapter 3 にてこの環境という概念がインタプリタがどのように働くか、またどのようにインタプリタを実装するかにおいて重要であることを示します。

### 1.1.3 Evaluating Combinations

One of our goals in this chapter is to isolate issues about thinking procedurally. As a case in point, let us consider that, in evaluating combinations, the interpreter is itself following a procedure.

この章の目標の 1 つは手続的な思考上の問題を分離することです。代表例として、組み合わせの評価においてインタプリタはそれ自身が手続に従うことを考えてみましょう。

To evaluate a combination, do the following: 組み合わせを評価するため、以下を行います

1. Evaluate the subexpressions of the combination. 組み合わせの部分式を評価する
2. Apply the procedure that is the value of the leftmost subexpression (the operator) to the arguments that are the values of the other subexpressions (the operands). 最も左の部分式(オペレータ)の値である手続を他の部分式の値である引数(オペランド)に対し適用する

Even this simple rule illustrates some important points about processes in general. First, observe that the first step dictates that in order to accomplish the evaluation process for a combination we must first perform the evaluation process on each element of the combination. Thus, the evaluation rule is *recursive* in nature; that is, it includes, as one of its steps, the need to invoke the rule itself.<sup>19</sup>

この単純なルールでさえ、一般的な過程におけるいくつかの重要な点を示します。最初に第一のステップが組み合わせの評価過程を達成すためには先に組み合わせの各要素の評価過程の実行を行う必要があります。従って評価ルールは事実上 *recursive*(再帰的) です。つまり評価ルールの 1 ステップとしてそれ自身を実行する必要性があります。<sup>20</sup>

---

<sup>19</sup>It may seem strange that the evaluation rule says, as part of the first step, that we should evaluate the leftmost element of a combination, since at this point that can only be an operator such as + or \* representing a built-in primitive procedure such as addition or multiplication. We will see later that it is useful to be able to work with combinations whose operators are themselves compound expressions.

<sup>20</sup>評価ルールが第一のステップの部分として組み合わせの最も左の要素を評価しなければいけないというのは奇妙に写るかもしれません。この時点ではそれは + や \* が表す

Notice how succinctly the idea of recursion can be used to express what, in the case of a deeply nested combination, would otherwise be viewed as a rather complicated process. For example, evaluating

再帰の考えがいかに簡潔に、深くネストした複合式を表現できるかに注目してください。再帰でなければとても複雑な経過に見えてしまうでしょう。例えれば以下の式を評価してみます。

```
(* (+ 2 (* 4 6))  
  (+ 3 5 7))
```

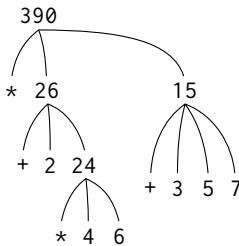
requires that the evaluation rule be applied to four different combinations. We can obtain a picture of this process by representing the combination in the form of a tree, as shown in Figure 1.1. Each combination is represented by a node with branches corresponding to the operator and the operands of the combination stemming from it. The terminal nodes (that is, nodes with no branches stemming from them) represent either operators or numbers. Viewing evaluation in terms of the tree, we can imagine that the values of the operands percolate upward, starting from the terminal nodes and then combining at higher and higher levels. In general, we shall see that recursion is a very powerful technique for dealing with hierarchical, treelike objects. In fact, the “percolate values upward” form of the evaluation rule is an example of a general kind of process known as *tree accumulation*.

この式は評価ルールが4つの異なる組み合わせに適用される必要があります。この過程を式の組立を木形式にて表現することで図解することができます。Figure 1.1をご覧下さい。各組み合わせは枝の付いたノードで表され、枝にはオペレータと別の組み合わせへの茎となるオペランドが付いています。終端ノード(他のノードへのと続く枝の無い物)はオペレータか数値を表しています。評価を木の用語で表すと、オペランドの値は上へと流れしていくことが想像できます。終端ノードから始まり上のレベル、さらに上のレベルにて合成されます。一般的に、再帰は階層的な木のような対象を扱うのにも強力な技術です。実際に評価ルールの“値を情報に流す”形式は*tree accumulation*(集積木)として知られます。

Next, observe that the repeated application of the first step brings us to the point where we need to evaluate, not combinations, but primitive expressions

---

足し算やかけ算のような組込のプリミティブな手続でしかないとめです。後に組み合せのオペレータそのものが組み合せである場合を扱えることが便利であることを学びます。



**Figure 1.1:** Tree representation, showing the value of each subcombination.

such as numerals, built-in operators, or other names. We take care of the primitive cases by stipulating that

次に第一ステップの適用の繰り返しが、組み合わせでなく、プリミティブな式、例えば数値や組込オペレータ、その他の名前を評価する必要となる点へと導くことに注目して下さい。以下を規定することにより、プリミティブな場合を取り扱います。

- the values of numerals are the numbers that they name, 数字の値はそれが意味する値です
- the values of built-in operators are the machine instruction sequences that carry out the corresponding operations, and  
組込オペレータの値は機械語の列であり対応する操作を実行します。
- the values of other names are the objects associated with those names in the environment. その他の名前の値は現在の環境にてその名前に対応するオブジェクトです。

We may regard the second rule as a special case of the third one by stipulating that symbols such as `+` and `*` are also included in the global environment, and are associated with the sequences of machine instructions that are their “values.” The key point to notice is the role of the environment in determining the meaning of the symbols in expressions. In an interactive language such as Lisp, it is meaningless to speak of the value of an expression such as `(+ x 1)` without specifying any information about the environment that would provide a meaning for the symbol `x` (or even for the symbol `+`). As we shall see in

**Chapter 3**, the general notion of the environment as providing a context in which evaluation takes place will play an important role in our understanding of program execution.

2つ目のルールは + と \* のような記号もまたグローバル(大域)環境に含まれており、それらの“値”として一連の機械語命令に関係付けられると規定することにより、3番目のルールの特別な場合であると見做すことができます。注意すべき鍵となる点は式の中の記号の意味の決定に環境が果たす役割です。Lispのようなインタラクティブな言語では (+ x 1) のような式の値について記号 x (またはの + のような記号についてさえ) 意味を与える環境の説明無しに話すことは無意味です。**Chapter 3**にて学びますが、評価が行われる文脈を提供する環境の一般的概念は我々がプログラムの実行を理解する上で重要な役割を果たします。

Notice that the evaluation rule given above does not handle definitions. For instance, evaluating (`define x 3`) does not apply `define` to two arguments, one of which is the value of the symbol `x` and the other of which is 3, since the purpose of the `define` is precisely to associate `x` with a value. (That is, `(define x 3)` is not a combination.)

上で与えられた評価ルールが定義を扱わないことにも注意して下さい。例えば `(define x 3)` の評価は `define` を 2つの引数、シンボル `x` の値と 3 に適用しません。`define` の目的はまさに `x` に対する値の関連付けだからです。(つまり `(define x 3)` は合成式ではありません。)

Such exceptions to the general evaluation rule are called *special forms*. `Define` is the only example of a special form that we have seen so far, but we will meet others shortly. Each special form has its own evaluation rule. The various kinds of expressions (each with its associated evaluation rule) constitute the syntax of the programming language. In comparison with most other programming languages, Lisp has a very simple syntax; that is, the evaluation rule for expressions can be described by a simple general rule together with specialized rules for a small number of special forms.<sup>21</sup>

---

<sup>21</sup>Special syntactic forms that are simply convenient alternative surface structures for things that can be written in more uniform ways are sometimes called *syntactic sugar*, to use a phrase coined by Peter Landin. In comparison with users of other languages, Lisp programmers, as a rule, are less concerned with matters of syntax. (By contrast, examine any Pascal manual and notice how much of it is devoted to descriptions of syntax.) This disdain for syntax is due partly to the flexibility of Lisp, which makes it easy to change surface syntax, and partly to the observation that many “convenient” syntactic constructs, which make the language less uniform,

そのような一般的な評価ルールに対する例外は特殊形式と呼ばれます。`define` は特殊形式の一例に過ぎません。すぐに他の例に出会うことになります。特殊形式は全てそれ自身の評価ルールを持ちます。色々な種類の式(それぞれが関連する評価ルールを持つ)はプログラミング言語の構文を構成します。他の多くのプログラミング言語と比較して Lisp はとても簡単な構文を持ちます。式の評価ルールは簡単な一般ルールと少しの特殊形式にて説明可能です。

22

### 1.1.4 Compound Procedures

We have identified in Lisp some of the elements that must appear in any powerful programming language:

他の強力なプログラミング言語に必ず存在する要素をいくつか Lisp でも確認しました。

- Numbers and arithmetic operations are primitive data and procedures.  
数値と算術命令はプリミティブなデータと手続です。
- Nesting of combinations provides a means of combining operations.  
組み合わせのネストは演算の結合手法を提供します。
- Definitions that associate names with values provide a limited means of abstraction.  
名前と値を関連付けする定義は抽象化の限定された手法を与えます。

Now we will learn about *procedure definitions*, a much more powerful abstraction technique by which a compound operation can be given a name and then referred to as a unit.

---

end up causing more trouble than they are worth when programs become large and complex. In the words of Alan Perlis, “Syntactic sugar causes cancer of the semicolon.”

<sup>22</sup>事物をより統一的な方法で表記可能な、簡単で便利な代替的表面構造である特別な構文形式を、Peter Landin の作成した語句ですが、*syntactic sugar* (シンタックスシュガー、構文糖) と呼ぶ場合があります。他言語のユーザと比較して Lisp プログラムは一般に構文上の問題に気をつかいません。(Pascal のマニュアルを調査するとどれだけ多くのページが構文の記述に割り当てられているのかに気付くのは逆です。) この構文の軽視は Lisp の柔軟性の理由の一部になります。Lisp の柔軟性は表面上の構文の変更を簡単になります。また多くの“便利な”構文の構築を見かける理由にも繋ります。それらの構文は言語をあまり統一的でないものにし、プログラムが巨大で複雑になるにつれ元の価値よりも多くの問題を起こすことになります。Alan Peril 曰く、“構文糖はセミコロンの癌を引き起こす”

ここでは*procedure definitions*(手続の定義) を学びます。より強力な抽象化のテクニックであり組み立てられた操作に名前を与え、1つの単位としてアクセス可能にします。

We begin by examining how to express the idea of “squaring.” We might say, “To square something, multiply it by itself.” This is expressed in our language as “二乗の値”をどのように表現するかから始めましょう。“二乗の値を求めるためにはその値をその値自身にかける”と言えるでしょう。

```
(define (square x) (* x x))
```

We can understand this in the following way: これを以下のように理解することができます。

(define (square x) (\* x x))  
 定義 二乗する xをかける xを xで.

We have here a *compound procedure*, which has been given the name `square`. The procedure represents the operation of multiplying something by itself. The thing to be multiplied is given a local name, `x`, which plays the same role that a pronoun plays in natural language. Evaluating the definition creates this compound procedure and associates it with the name `square`.<sup>23</sup>

ここで `square` と名付けられた *compound procedure*(複合手続) が出てきました。この手続はある数値をそれ自身にてかけ算することを表しています。かけられる数には `x` という名前が付けられており代名詞が自然言語にて果たすのと同じ役割を果たします。この定義の評価はこの複合手続を作成し、私達はそれに `square` という名前を与えていました。<sup>24</sup>

The general form of a procedure definition is  
一般的な手続の定義形式は以下の通りです。

<sup>23</sup>Observe that there are two different operations being combined here: we are creating the procedure, and we are giving it the name `square`. It is possible, indeed important, to be able to separate these two notions—to create procedures without naming them, and to give names to procedures that have already been created. We will see how to do this in Section 1.3.2.

24 ここでは次の2つを区別できるようになることが本当に重要だと言えます。1つは手続を名前を付けて作成すること、もう1つは既に作成された手続に名前を付けることです。どのように行うかについてはSection 1.3.2にて学びます。

```
(define (<name> <formal parameters>)
  <body>)
```

The *<name>* is a symbol to be associated with the procedure definition in the environment.<sup>25</sup> The *<formal parameters>* are the names used within the body of the procedure to refer to the corresponding arguments of the procedure. The *<body>* is an expression that will yield the value of the procedure application when the formal parameters are replaced by the actual arguments to which the procedure is applied.<sup>26</sup> The *<name>* and the *<formal parameters>* are grouped within parentheses, just as they would be in an actual call to the procedure being defined.

*<name>* はその環境における手続定義に関連付けられる記号です。<sup>27</sup> *<formal parameters>* は手続の中で利用される名前で手続の関連する引数を参照します。*<body>* は形式上のパラメータが、適用される手続の実際の引数に置換される時、手続適用の値を返す式です。<sup>28</sup> *<name>* と *<formal parameters>* は括弧を用いてグループ化され、実際の手続呼出しのように定義されます。

Having defined **square**, we can now use it: **square** を定義したので使ってみましょう。

```
(square 21)
441
(square (+ 2 5))
49
(square (square 3))
81
```

We can also use **square** as a building block in defining other procedures. For example,  $x^2 + y^2$  can be expressed as

**square** を他の手続の構築に用いることも可能です。

---

<sup>25</sup>Throughout this book, we will describe the general syntax of expressions by using italic symbols delimited by angle brackets—e.g., *<name>*—to denote the “slots” in the expression to be filled in when such an expression is actually used.

<sup>26</sup>More generally, the body of the procedure can be a sequence of expressions. In this case, the interpreter evaluates each expression in the sequence in turn and returns the value of the final expression as the value of the procedure application.

<sup>27</sup>この本では一般的な式の構文をかぎ括弧にて閉じたイタリックの記号—例えば、*<name>*—を用いて実際に式が利用される時に埋められるべき式中の“枠”を示します。

<sup>28</sup>より実際には、手続のボディは連続する式です。この場合インタプリタは連続する各式を順に評価し最後の式の値を手続適用全体の値として返します。

```
(+ (square x) (square y))
```

We can easily define a procedure `sum-of-squares` that, given any two numbers as arguments, produces the sum of their squares:

2つの数値を引数として取りそれらの二乗の和を求める `sum-of-squares` を定義することも簡単です。

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(sum-of-squares 3 4)
```

25

Now we can use `sum-of-squares` as a building block in constructing further procedures:

`sum-of-squares` をさらに別の手続構築に利用することもできます。

```
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
(f 5)
```

136

Compound procedures are used in exactly the same way as primitive procedures. Indeed, one could not tell by looking at the definition of `sum-of-squares` given above whether `square` was built into the interpreter, like `+` and `*`, or defined as a compound procedure.

複合手続はプリミティブな手続と全く同じように利用可能です。実際に上の `sum-of-squares` の定義を見ても `square` が `+` や `*` のようにインタプリタに組込まれているのか、複合手続として定義されているのか見分けがつかないでしょう。

### 1.1.5 手続適用の置換モデル

To evaluate a combination whose operator names a compound procedure, the interpreter follows much the same process as for combinations whose operators name primitive procedures, which we described in [Section 1.1.3](#). That is, the interpreter evaluates the elements of the combination and applies the procedure (which is the value of the operator of the combination) to the arguments (which are the values of the operands of the combination).

オペレータの名前が合成式を示す組合せを評価する時、インタプリタは Section 1.1.3で説明した組合せのオペレータがプリミティブである場合とほぼ同じ手順を追います。インタプリタは合成の各要素を評価し、(組合せのオペレータの値である) 手続を(組合せのオペランドである) 引数に対して適用します。

We can assume that the mechanism for applying primitive procedures to arguments is built into the interpreter. For compound procedures, the application process is as follows:

プリミティブな手続を引数に対して適用するメカニズムはインタプリタに組込まれていることが想像できます。複合手続に対しては適用プロセスは以下のようになります。

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

複合手続を引数に適用するために、手続のボディを、各形式パラメタを対応する引数にて置換してから評価します。

To illustrate this process, let's evaluate the combination 過程を説明するため以下とのコンビネーションを評価してみましょう。

(**f** 5)

where **f** is the procedure defined in Section 1.1.4. We begin by retrieving the body of **f**:

**f** はSection 1.1.4にて定義された手続です。**f** のボディを取得することから始めます。

(**sum-of-squares** (+ **a** 1) (\* **a** 2))

Then we replace the formal parameter **a** by the argument 5: 次に形式パラメタの **a** を引数 5 で置き換えます。

(**sum-of-squares** (+ 5 1) (\* 5 2))

Thus the problem reduces to the evaluation of a combination with two operands and an operator **sum-of-squares**. Evaluating this combination involves three subproblems. We must evaluate the operator to get the procedure to be applied, and we must evaluate the operands to get the arguments. Now (+ 5 1) produces 6 and (\* 5 2) produces 10, so we must apply the **sum-of-squares** procedure

to 6 and 10. These values are substituted for the formal parameters `x` and `y` in the body of `sum-of-squares`, reducing the expression to

従って問題は 2 つのオペラントとオペレータ `sum-of-squares` に換算されます。この組み合わせの評価は 3 つの部分問題に分かれます。まずオペレータを評価して適用する手続を得て、オペラントを評価して引数を得る必要があります。さて `(+ 5 1)` は 6 になり、`(* 5 2)` は 10 になりますので `sum-of-squares` 手続を 6 と 10 に適用しなければなりません。これらの値は `sum-of-squares` のボディのパラメタ `x` and `y` を置き換え、式は以下のように置換されます。

```
(+ (square 6) (square 10))
```

If we use the definition of `square`, this reduces to

`square` の定義を用いるとこれはさらに以下のように置換されます。

```
(+ (* 6 6) (* 10 10))
```

which reduces by multiplication to

乗算を置換することで以下になります。

```
(+ 36 100)
```

and finally to

最終的には次のとおりです。

136

The process we have just described is called the *substitution model* for procedure application. It can be taken as a model that determines the “meaning” of procedure application, insofar as the procedures in this chapter are concerned. However, there are two points that should be stressed:

ここまで説明したプロセスは手続適用の *substitution model* (置換モデル、置換モデル) と呼ばれます。この章にて扱われた手続の過程においては、手続適用の“意味”を決定するモデルとして捉えることができます。しかし、強調すべき 2 つの事があります。

- The purpose of the substitution is to help us think about procedure application, not to provide a description of how the interpreter really works. Typical interpreters do not evaluate procedure applications by manipulating the text of a procedure to substitute values for the formal parameters. In practice, the “substitution” is accomplished by using a

local environment for the formal parameters. We will discuss this more fully in Chapter 3 and Chapter 4 when we examine the implementation of an interpreter in detail.

置換の目的は私達が手続適用について考えることを手助けすることであり、インタプリタが実際にどのように働くかの説明を与えることではありません。典型的なインタプリタは形式パラメータのための値を置き換えるために手続のテキストを操作することで、手続適用を評価することはしません。実際には“置換”は形式パラメタにローカルの環境を用いることで行われます。このことについてはより完全にChapter 3 と Chapter 4 にてインタプリタの実装の詳細について調査する時に議論します。

- Over the course of this book, we will present a sequence of increasingly elaborate models of how interpreters work, culminating with a complete implementation of an interpreter and compiler in Chapter 5. The substitution model is only the first of these models—a way to get started thinking formally about the evaluation process. In general, when modeling phenomena in science and engineering, we begin with simplified, incomplete models. As we examine things in greater detail, these simple models become inadequate and must be replaced by more refined models. The substitution model is no exception. In particular, when we address in Chapter 3 the use of procedures with “mutable data,” we will see that the substitution model breaks down and must be replaced by a more complicated model of procedure application.<sup>29</sup>

この本のコース全体ではインタプリタがどのように働くかについて、一連の徐々に精巧なモデルを紹介して行きます。最終的にはインタプリタとコンパイラーの完全な実装を Chapter 5 で見せます。置換モデルはこれらのモデルの最初—評価手続について正式な考え得るための始まりに過ぎません。一般的に科学とエンジニアリングについての事象をモデリングする場合、単純化した不完全なモデルから始めます。より詳細な調査を行つれ、これらの単純なモデルは不適切になり、より正確なモデル

---

<sup>29</sup>Despite the simplicity of the substitution idea, it turns out to be surprisingly complicated to give a rigorous mathematical definition of the substitution process. The problem arises from the possibility of confusion between the names used for the formal parameters of a procedure and the (possibly identical) names used in the expressions to which the procedure may be applied. Indeed, there is a long history of erroneous definitions of *substitution* in the literature of logic and programming semantics. See Stoy 1977 for a careful discussion of substitution.

にて置き換えられます。置換モデルもまた例外ではありません。実際に Chapter 3で示しますが手続を“mutable(変わりやすい)データ”と共に扱う場合に置換モデルは破綻し、より複雑な手続適用のモデルにより置き換えなければならなくなります。<sup>30</sup>

## Applicative order versus normal order

According to the description of evaluation given in Section 1.1.3, the interpreter first evaluates the operator and operands and then applies the resulting procedure to the resulting arguments. This is not the only way to perform evaluation. An alternative evaluation model would not evaluate the operands until their values were needed. Instead it would first substitute operand expressions for parameters until it obtained an expression involving only primitive operators, and would then perform the evaluation. If we used this method, the evaluation of (f 5) would proceed according to the sequence of expansions

Section 1.1.3で与えられた評価の記述に従えば、インタプリタは最初にオペレータとオペランドを評価し、次に結果の手続を結果の引数に適用します。評価の仕方はこれだけではありません。代替としての評価モデルはオペランドをそれらの値が必要になるまで評価しません。その代わりに最初はオペランドの式にそれがプリミティブなオペレータのみを持つまでパラメタで置換します。それから評価を実行します。この手法を用いた場合、(f 5) の評価は展開の流れに従って進行します。

```
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2))))
```

followed by the reductions 簡約が続きます。

```
(+ (* 6 6) (* 10 10))
(+ 36 100)
```

136

<sup>30</sup>置換のアイデアの簡明性にもかかわらず、置換処理の厳密な数学上の定義を与えることは驚くほど複雑になることが知られています。問題は手続の形式パラメタの名前と手続が適用される式で利用されている(同じである可能性のある)名前の間の混乱の可能性から生じます。実際に論理とプログラミング意味論の文献における置換の間違った定義には長い歴史があります。Stoy 1977の置換に関する注意深い議論を参照下さい。

This gives the same answer as our previous evaluation model, but the process is different. In particular, the evaluations of  $(+ 5 1)$  and  $(* 5 2)$  are each performed twice here, corresponding to the reduction of the expression  $(* x x)$  with  $x$  replaced respectively by  $(+ 5 1)$  and  $(* 5 2)$ .

今回も前回の評価モデルと同じ答になりました。しかし経過が異なります。具体的には  $(+ 5 1)$  と  $(* 5 2)$  がここではそれぞれ二回づつ実行されます。式  $(* x x)$  における  $x$  がそれぞれ  $(+ 5 1)$  と  $(* 5 2)$  に置き換えることで換算されているのに相当しています。

This alternative “fully expand and then reduce” evaluation method is known as *normal-order evaluation*, in contrast to the “evaluate the arguments and then apply” method that the interpreter actually uses, which is called *applicative-order evaluation*. It can be shown that, for procedure applications that can be modeled using substitution (including all the procedures in the first two chapters of this book) and that yield legitimate values, normal-order and applicative-order evaluation produce the same value. (See [Exercise 1.5](#) for an instance of an “illegitimate” value where normal-order and applicative-order evaluation do not give the same result.)

この代替である“完全に展開してから簡約する”評価方法は正規順序評価として知られています。一方、“引数を評価してから適用”する方法はインタプリタが実際に利用するもので適用順序評価と呼ばれます。、(この本の最初の 2 つの章の手続全てを含めて) 置換を使用してモデリング可能、かつ正当な値を生む手続適用において正規順序と適用順序の評価は同じ値を生むことが見てとれるでしょう。(正規順序と適用順序の評価が同じ値を返さない“不当な”値の例は[Exercise 1.5](#)をご覧下さい)

Lisp uses applicative-order evaluation, partly because of the additional efficiency obtained from avoiding multiple evaluations of expressions such as those illustrated with  $(+ 5 1)$  and  $(* 5 2)$  above and, more significantly, because normal-order evaluation becomes much more complicated to deal with when we leave the realm of procedures that can be modeled by substitution. On the other hand, normal-order evaluation can be an extremely valuable tool, and we will investigate some of its implications in [Chapter 3](#) and [Chapter 4](#).<sup>31</sup>

Lisp は適用順序評価を用いています。理由の一部は上の  $(+ 5 1)$  と  $(* 5$

---

<sup>31</sup>In [Chapter 3](#) we will introduce *stream processing*, which is a way of handling apparently “infinite” data structures by incorporating a limited form of normal-order evaluation. In [Section 4.2](#) we will modify the Scheme interpreter to produce a normal-order variant of Scheme.

2) で示されたような式の複数回評価を避けることで付加的な効率を得るためです。そしてより重要な理由は正規順序評価は置換によりモデル化可能な手続の範囲を離れる時の取扱がとても複雑なためです。一方で、正規順序評価はとても価値のあるツールです。その意味のいくらかをChapter 3とChapter 4にて調査します。<sup>32</sup>

### 1.1.6 Conditional Expressions and Predicates

The expressive power of the class of procedures that we can define at this point is very limited, because we have no way to make tests and to perform different operations depending on the result of a test. For instance, we cannot define a procedure that computes the absolute value of a number by testing whether the number is positive, negative, or zero and taking different actions in the different cases according to the rule

この時点で私達が定義可能な種類の手続の表現力はとても限られています。テストを作成し、テストの結果により異なる命令を実行する方法が無いためです。例えば数値の絶対値を演算する手続を定義できません。数値が正、負、零であるかテストを行いルールに従い異なる場合に対し異なる行動をしなければなりません。

$$|x| = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x = 0, \\ -x & \text{if } x < 0. \end{cases}$$

This construct is called a *case analysis*, and there is a special form in Lisp for notating such a case analysis. It is called **cond** (which stands for “conditional”), and it is used as follows:

この考えは*case analysis*(ケース分析)と呼び Lisp にはそのようなケース分析のための特殊形式が存在します。**cond**(“conditional”(条件文)を表わす)と呼ばれ、以下のように利用されます。

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

---

<sup>32</sup>Chapter 3では*stream processing*を紹介します。これは一見して“無限”的データ構造を正規順評価の制約形式に立脚して取り扱う手法です。Section 4.2ではScheme インタプリタを変更し Scheme の正規順異種を作成します。

The general form of a conditional expression is 条件式の一般的な形式は以下のとおりです。

```
(cond ((p1) (e1))
      ((p2) (e2))
      ...
      ((pn) (en)))
```

consisting of the symbol `cond` followed by parenthesized pairs of expressions

記号 `cond` から構成され、続く括弧で括った複数の式のペア

```
((p) (e))
```

called *clauses*. The first expression in each pair is a *predicate*—that is, an expression whose value is interpreted as either true or false.<sup>33</sup>

*clauses*(クローズ、節)と呼ばれます。各ペアの最初の式は *predicate*(述語)<sup>34</sup>—値が真か偽になる式です。

Conditional expressions are evaluated as follows. The predicate  $\langle p_1 \rangle$  is evaluated first. If its value is false, then  $\langle p_2 \rangle$  is evaluated. If  $\langle p_2 \rangle$ 's value is also false, then  $\langle p_3 \rangle$  is evaluated. This process continues until a predicate is found whose value is true, in which case the interpreter returns the value of the corresponding *consequent expression*  $\langle e \rangle$  of the clause as the value of the conditional expression. If none of the  $\langle p \rangle$ 's is found to be true, the value of the `cond` is undefined.

条件式は次のように評価されます。まず述語  $\langle p_1 \rangle$  が最初に評価されます。もしその値が偽であれば次に  $\langle p_2 \rangle$  が評価されます。もし  $\langle p_2 \rangle$  の値もまた偽であるならば、その次は  $\langle p_3 \rangle$  が評価されます。この過程は値が真となる述語が見つかるまで続きます。その場合インタプリタは対応するクローズの *consequent*

---

<sup>33</sup>“Interpreted as either true or false” means this: In Scheme, there are two distinguished values that are denoted by the constants `#t` and `#f`. When the interpreter checks a predicate's value, it interprets `#f` as false. Any other value is treated as true. (Thus, providing `#t` is logically unnecessary, but it is convenient.) In this book we will use names `true` and `false`, which are associated with the values `#t` and `#f` respectively.

<sup>34</sup>“真か偽のどちらに解釈される”とは次を意味します。Scheme では 2 つの区別される値が存在し、それらは `#t` と `#f` の定数で示されます。インタプリタが述語の値をチェックする時、`#f` を偽と訳します。それ以外の任意の値は全て真だと扱われます。(従って `#t` を与えることは論理的には必要ありません。しかしそのほうが便利です。) この本では `true` と `false` という名前を用います。それらは `#t` と `#f` という値にそれぞれ関連付けられます。

*expression*(結果式)  $\langle e \rangle$  の値が条件式の値として返されます。もし真となる  $\langle p \rangle$  が見つからない場合には *cond* の値は未定義です。

The word *predicate* is used for procedures that return true or false, as well as for expressions that evaluate to true or false. The absolute-value procedure *abs* makes use of the primitive predicates  $>$ ,  $<$ , and  $=$ .<sup>35</sup> These take two numbers as arguments and test whether the first number is, respectively, greater than, less than, or equal to the second number, returning true or false accordingly.

述語という単語は真か偽を返す手続に利用されます。真か偽と評価される式にも用いられます。絶対値の手続 *abs* はプリミティブな述語  $>$ ,  $<$ ,  $=$  を利用します。<sup>36</sup>

これらは 2 つの数値を引数として取り最初の数が 2 つ目の数に対し式の順に、より大きい、より小さい、等しいかどうかテストを行い適宜に真か偽を返します。

Another way to write the absolute-value procedure is  
絶対値の手続を書くもう 1 つの方法が次です。

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

which could be expressed in English as “If  $x$  is less than zero return  $-x$ ; otherwise return  $x$ .” *Else* is a special symbol that can be used in place of the  $\langle p \rangle$  in the final clause of a *cond*. This causes the *cond* to return as its value the value of the corresponding  $\langle e \rangle$  whenever all previous clauses have been bypassed. In fact, any expression that always evaluates to a true value could be used as the  $\langle p \rangle$  here.

これは日本語で“もし  $x$  が零より小さい場合  $-x$  を返す。そうでなければ  $x$  を返す”と表現できます。*else* は特別なシンボルで *cond* の最後の節(クローズ)の  $\langle p \rangle$  の場所にて利用可能です。こうすることで *cond* がその値として対応する  $\langle e \rangle$  の値をそれ以前のクローズ全てが回避された場合に返すことが可能です。本当の所はここで  $\langle p \rangle$  に常に値が真となる任意の式を使用することも可能です。

Here is yet another way to write the absolute-value procedure:

---

<sup>35</sup>*Abs* also uses the “minus” operator  $-$ , which, when used with a single operand, as in  $(- x)$ , indicates negation.

<sup>36</sup>*abs* はまた“マイナス”演算子  $-$  を使用します。 $(- x)$  のように单一のオペランドに利用された時、符号の反転を示します。

次はさらにもう 1 つ別の絶対値手続の書き方です。

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

This uses the special form `if`, a restricted type of conditional that can be used when there are precisely two cases in the case analysis. The general form of an `if` expression is

これは特殊形式の `if` という制約のある条件の型を使用しており、ケース分析にて正確に 2 つのケースが存在する場合に用います。`if` 式の一般的な形式は以下のとおりです。

```
(if <code><predicate> <code><consequent> <code><alternative></code></code>)
```

To evaluate an `if` expression, the interpreter starts by evaluating the `<predicate>` part of the expression. If the `<predicate>` evaluates to a true value, the interpreter then evaluates the `<consequent>` and returns its value. Otherwise it evaluates the `<alternative>` and returns its value.<sup>37</sup>

`if` 式を評価するためにインタプリタは式の `<predicate>` の部分を評価することから始めます。もし `<predicate>` の評価が真値になる場合、インタプリタは次に `<consequent>` を評価しその値を返します。そうでなければ `<alternative>` を評価しその値を返します。<sup>38</sup>

In addition to primitive predicates such as `<`, `=`, and `>`, there are logical composition operations, which enable us to construct compound predicates. The three most frequently used are these:

`<` や `=`, `>` のようなプリミティブな述語に追加して論理複合命令が存在し、複合述語を構築することを可能にします。最も良く利用される 3 つは以下の物です。

---

<sup>37</sup> A minor difference between `if` and `cond` is that the `<e>` part of each `cond` clause may be a sequence of expressions. If the corresponding `<p>` is found to be true, the expressions `<e>` are evaluated in sequence and the value of the final expression in the sequence is returned as the value of the `cond`. In an `if` expression, however, the `<consequent>` and `<alternative>` must be single expressions.

<sup>38</sup> `if` と `cond` の小さな違いは `cond` の各クローズの `<e>` は連続する式になつても良いことです。もし対応する `<p>` が真になる場合、`<e>` 内の式は順に評価され連なりの最後の式の値が `cond` の値として返されます。しかし `if` 式の中では `<consequent>` と `<alternative>` は単一の式でなければなりません

- (**and**  $\langle e_1 \rangle \dots \langle e_n \rangle$ )

The interpreter evaluates the expressions  $\langle e \rangle$  one at a time, in left-to-right order. If any  $\langle e \rangle$  evaluates to false, the value of the **and** expression is false, and the rest of the  $\langle e \rangle$ 's are not evaluated. If all  $\langle e \rangle$ 's evaluate to true values, the value of the **and** expression is the value of the last one.

インタプリタは式  $\langle e \rangle$  を左から右へ 1 つずつ評価します。もし  $\langle e \rangle$  のどれかが偽と評価された場合 **and** 式の値は偽となり、残りの  $\langle e \rangle$  は評価されません。もし全ての  $\langle e \rangle$  の評価が真となれば **and** 式の値は最後の値になります。

- (**or**  $\langle e_1 \rangle \dots \langle e_n \rangle$ )

The interpreter evaluates the expressions  $\langle e \rangle$  one at a time, in left-to-right order. If any  $\langle e \rangle$  evaluates to a true value, that value is returned as the value of the **or** expression, and the rest of the  $\langle e \rangle$ 's are not evaluated. If all  $\langle e \rangle$ 's evaluate to false, the value of the **or** expression is false.

インタプリタは式  $\langle e \rangle$  を 1 つずつ左から右へ評価します。もし  $\langle e \rangle$  のどれかが真と評価されればその値が **or** 式の値として返され、 $\langle e \rangle$  の残りは評価されません。もし全ての  $\langle e \rangle$  が偽と評価された場合、**or** の値は偽となります。

- (**not**  $\langle e \rangle$ )

The value of a **not** expression is true when the expression  $\langle e \rangle$  evaluates to false, and false otherwise.

**not** 式の値は式  $\langle e \rangle$  が偽と評価される時は真であり、そうでなければ偽となります。

Notice that **and** and **or** are special forms, not procedures, because the subexpressions are not necessarily all evaluated. **Not** is an ordinary procedure.

**and** と **or** が特殊形式であり手続ではないことに注意して下さい。部分式が全て評価される必要が無いためです。**not** は通常の手続です。

As an example of how these are used, the condition that a number  $x$  be in the range  $5 < x < 10$  may be expressed as

これらがどのように利用されるかの例として、数値  $x$  が  $5 < x < 10$  の値域に存在するかという条件は次のように表現されます。

(**and** ( $>$  x 5) ( $<$  x 10))

As another example, we can define a predicate to test whether one number is greater than or equal to another as

別の例として、ある数値が別の数値に対し等しいかより大きいかを示す述語は以下の通りです。

```
(define (>= x y) (or (> x y) (= x y)))
```

or alternatively as

または代替法として

```
(define (>= x y) (not (< x y)))
```

**Exercise 1.1:** Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented.

以下の一連の式について、各式に対するインタプリタの応答としての表示結果は何か？式の列は下記に表示された順で評価されるものと考えよ。

```
10
(+ 5 3 4)
(- 9 1)
(/ 6 2)
(+ (* 2 4) (- 4 6))
(define a 3)
(define b (+ a 1))
(+ a b (* a b))
(= a b)
(if (and (> b a) (< b (* a b)))
    b
    a)

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))

(+ 2 (if (> b a) b a))
```

```
(* (cond ((> a b) a)
          ((< a b) b)
          (else -1))
(+ a 1))
```

**Exercise 1.2:** Translate the following expression into prefix form:  
以下の式を接頭辞形式にて翻訳せよ

$$\frac{5 + 4 + (2 - (3 - (6 + \frac{4}{5}))))}{3(6 - 2)(2 - 7)}.$$

**Exercise 1.3:** Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

3つの数値を引数として取り内2つの大きな数値の二乗の和を返す手続を定義せよ。

**Exercise 1.4:** Observe that our model of evaluation allows for combinations whose operators are compound expressions. Use this observation to describe the behavior of the following procedure:

我々の評価モデルがオペレータが複合式である組み合わせを可能にすることを観察せよ。この観察結果を用いて次の手続の挙動を説明せよ:

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

**Exercise 1.5:** Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two procedures:

Ben Bitdiddleは自分が直面するインタプリタが適用順評価と正規順評価のどちらを用いるか決定するテストを開発した。まず以下の2つの手続を定義する。

```
(define (p) (p))
(define (test x y)
  (if (= x 0) 0 y))
```

Then he evaluates the expression

次に以下の式を評価する。

```
(test 0 (p))
```

What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form `if` is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

Ben は適用順評価を用いるインタプリタではどのような挙動を観察するだろうか? Ben は正規順評価を用いるインタプリタではどのような挙動を観察するだろうか? あなたの回答を説明せよ。(特殊形式の `if` はインタプリタが適用順評価でも正規順評価でも同じ挙動を行うと仮定せよ: 述語式が最初に評価され、結果が `consequent` と `alternative` のどちらを評価するか決定する)

### 1.1.7 Example: Square Roots by Newton's Method

Procedures, as introduced above, are much like ordinary mathematical functions. They specify a value that is determined by one or more parameters. But there is an important difference between mathematical functions and computer procedures. Procedures must be effective.

ここまでで説明された通り、手続は普通の数学の関数にとても似ています。手続は 1 つ以上のパラメタにより決定される値を特定します。しかし数学の関数と計算機の手続の間には重要な違いが存在します。手続は効果的である必要があります。

As a case in point, consider the problem of computing square roots. We can define the square-root function as

その一例として、平方根の演算問題について考えましょう。square-root 関数を以下のように定義できます。

$$\sqrt{x} = \text{the } y \text{ such that } y \geq 0 \text{ and } y^2 = x.$$

This describes a perfectly legitimate mathematical function. We could use it to recognize whether one number is the square root of another, or to derive facts about square roots in general. On the other hand, the definition does not describe a procedure. Indeed, it tells us almost nothing about how to actually find the square root of a given number. It will not help matters to rephrase this definition in pseudo-Lisp:

これは完全に正しい数学の関数です。これを用いてある数値が他の数値の平方根であるか分かりますし、平方根の一般的な事実を導出可能です。しかし一方でこの定義は手続の記述ではありません。与えられた数値から実際どのようにして平方根を求めるのか、これはほとんど何も教えてくれません。この定義を疑似 Lisp にて言い換えようとも問題の何の手助けにもなりません。

```
(define (sqrt x)
  (the y (and (>= y 0)
               (= (square y) x))))
```

This only begs the question.

これはただ問題を提起するだけです。

The contrast between function and procedure is a reflection of the general distinction between describing properties of things and describing how to do things, or, as it is sometimes referred to, the distinction between declarative knowledge and imperative knowledge. In mathematics we are usually concerned with declarative (what is) descriptions, whereas in computer science we are usually concerned with imperative (how to) descriptions.<sup>39</sup>

関数と手続の間の対称性は事物の属性の説明と行きの説明との間の一般的区別に関する反映です。または時には宣言的知識と手続的知識の間の区別だと

---

<sup>39</sup> Declarative and imperative descriptions are intimately related, as indeed are mathematics and computer science. For instance, to say that the answer produced by a program is “correct” is to make a declarative statement about the program. There is a large amount of research aimed at establishing techniques for proving that programs are correct, and much of the technical difficulty of this subject has to do with negotiating the transition between imperative statements (from which programs are constructed) and declarative statements (which can be used to deduce things). In a related vein, an important current area in programming-language design is the exploration of so-called very high-level languages, in which one actually programs in terms of declarative statements. The idea is to make interpreters sophisticated enough so that, given “what is” knowledge specified by the programmer, they can generate “how to” knowledge automatically. This cannot be done in general, but there are important areas where progress has been made. We shall revisit this idea in Chapter 4.

参照できるでしょう。数学では通常宣言的 (what is) 記述を用い、コンピュータサイエンスでは通常手続的 (how to) 記述を用います。<sup>40</sup>

How does one compute square roots? The most common way is to use Newton's method of successive approximations, which says that whenever we have a guess  $y$  for the value of the square root of a number  $x$ , we can perform a simple manipulation to get a better guess (one closer to the actual square root) by averaging  $y$  with  $x/y$ .<sup>41</sup> For example, we can compute the square root of 2 as follows. Suppose our initial guess is 1:

人はどのようにして平方根を求めるができるのでしょうか? 最も一般的な方法はニュートンの漸次接近法を用いる方法です。ニュートン法はある数値  $x$  の平方根の推定値として  $y$  を持つ場合に、より良い推定値(実際の平方根により近い値)を求めるために  $y$  と  $x/y$  の平均を取るという簡単な操作を実行します。<sup>42</sup> 例として、2 の平方根は以下のようにして求められます。推定値の初期値を 1 とします:

推定値	商	平均
1	$(2/1) = 2$	$((2 + 1)/2) = 1.5$
1.5	$(2/1.5) = 1.3333$	$((1.3333 + 1.5)/2) = 1.4167$
1.4167	$(2/1.4167) = 1.4118$	$((1.4167 + 1.4118)/2) = 1.4142$
1.4142	...	...

<sup>40</sup>宣言的、手続的記述は数学とコンピュータサイエンスのように実際に深く関わっています。例えばプログラムの生成した答が“正しい”ということはプログラムについて宣言的な文を作成することです。プログラムが正しいことを証明するための立証技術を目的とした非常に多くの研究が存在します。この問題の技術的難度の多くは(プログラムが構築される)手続的文と(事象を推論するのに用いられる)宣言的文との間の移行に関連します。関連領域において、プログラミング言語の設計における現在の重要な領域は超高速準言語と呼ばれる物の調査です。それは実際にプログラムを宣言的文の用語にて作成します。その意図はインタプリタを十分に洗練することでプログラマより与えられた“what is”的知識より“how to”的知識を自動的に生成可能とします。これは一般的には可能ではありませんが、成果が達成された重要な領域が存在します。この考え方についてはChapter 4にて再度触れることに致します。

<sup>41</sup>This square-root algorithm is actually a special case of Newton's method, which is a general technique for finding roots of equations. The square-root algorithm itself was developed by Heron of Alexandria in the first century A.D. We will see how to express the general Newton's method as a Lisp procedure in Section 1.3.4.

<sup>42</sup>この平方根アルゴリズムは実際にはニュートン法の特別なケースです。ニュートン法は方程式の根を求める一般的な技法です。平方根アルゴリズム自体はアレキサンドリアの Heron により A.D.1 世紀に開発されました。一般的なニュートン法を Lisp の手続によりどのように表わすかについてはSection 1.3.4にて学びます。

Continuing this process, we obtain better and better approximations to the square root.

この過程を繰り返すことにより平方根のより良い近似値を得られます。

Now let's formalize the process in terms of procedures. We start with a value for the radicand (the number whose square root we are trying to compute) and a value for the guess. If the guess is good enough for our purposes, we are done; if not, we must repeat the process with an improved guess. We write this basic strategy as a procedure:

では手続の表現にてこの過程を形式化してみましょう。radicand(被開法数: 根号の中身。平方根を求める値) と guess(推定値) を用います。もし推定値の品質が十分であれば終了します。そうでなければ処理をより良い推定値にて繰り返さなければなりません。この基本的戦略を手続として以下のように記述しました。

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
```

A guess is improved by averaging it with the quotient of the radicand and the old guess:

推定値は旧推定値と商の平均を取ることで改善されます。

```
(define (improve guess x)
  (average guess (/ x guess)))
```

average の定義は以下です。

```
(define (average x y)
  (/ (+ x y) 2))
```

We also have to say what we mean by “good enough.” The following will do for illustration, but it is not really a very good test. (See [Exercise 1.7.](#)) The idea is to improve the answer until it is close enough so that its square differs from the radicand by less than a predetermined tolerance (here 0.001).<sup>43</sup>

---

<sup>43</sup>We will usually give predicates names ending with question marks, to help us remember that they are predicates. This is just a stylistic convention. As far as the interpreter is concerned, the question mark is just an ordinary character.

“十分に良い”の定義を決めねばなりません。以下に説明しますが、これは本当はあまり良いテストではありません。(Exercise 1.7をご覧下さい) 考え方は回答を十分に近い値にするために、その二乗と被開法数の差が事前に決定した許容誤差(ここでは 0.001)より小さくなるまで改善します。<sup>44</sup>

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

Finally, we need a way to get started. For instance, we can always guess that the square root of any number is 1:<sup>45</sup>

最後にどのように始めるかが必要です。例えば任意の数値の平方根の推定値を常に 1 とすることも可能です。<sup>46</sup>

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
```

If we type these definitions to the interpreter, we can use `sqrt` just as we can use any procedure:

---

<sup>44</sup>通常は述語にはクエスチョンマークで終わる名前を与えます。そうすることでそれが述語だと理解できるようにです。これは単にスタイル上の慣例です。インタプリタの受け取り方に関する限り、クエスチョンマークは通常の文字でしかありません。

<sup>45</sup>Observe that we express our initial guess as 1.0 rather than 1. This would not make any difference in many Lisp implementations. MIT Scheme, however, distinguishes between exact integers and decimal values, and dividing two integers produces a rational number rather than a decimal. For example, dividing 10 by 6 yields  $5/3$ , while dividing 10.0 by 6.0 yields 1.6666666666666667. (We will learn how to implement arithmetic on rational numbers in Section 2.1.1.) If we start with an initial guess of 1 in our square-root program, and  $x$  is an exact integer, all subsequent values produced in the square-root computation will be rational numbers rather than decimals. Mixed operations on rational numbers and decimals always yield decimals, so starting with an initial guess of 1.0 forces all subsequent values to be decimals.

<sup>46</sup>推定値の初期値を 1 ではなく 1.0 と表現していることに注意して下さい。これは多くの Lisp の実装では何の違いもありません。しかし MIT Scheme は整数と小数の値を厳格に区別します。2 つの整数を割ると小数ではなく分数を返します。例として 10 を 6 で割ると  $5/3$  を返します。しかし 10.0 を 6.0 で割れば 1.6666666666666667 を返すのです。(分数の演算の実装法についてはSection 2.1.1で学びます。) もし推定値の初期値を square-root プログラムにおいて 1 にして開始した場合、 $x$  も実際に整数である場合には全ての続く square-root の演算により生成される値は小数ではなく分数になります。分数と小数を混ぜた演算は小数を返します。従って推定値の初期値を 1.0 にすることで全ての続く値を小数にすることが可能です。

この定義をインタプリタに入力すれば `sqrt` を他の手続のように利用可能です。

```
(sqrt 9)
3.00009155413138

(sqrt (+ 100 37))
11.704699917758145

(sqrt (+ (sqrt 2) (sqrt 3)))
1.7739279023207892

(square (sqrt 1000))
1000.000369924366
```

The `sqrt` program also illustrates that the simple procedural language we have introduced so far is sufficient for writing any purely numerical program that one could write in, say, C or Pascal. This might seem surprising, since we have not included in our language any iterative (looping) constructs that direct the computer to do something over and over again. `Sqrt-iter`, on the other hand, demonstrates how iteration can be accomplished using no special construct other than the ordinary ability to call a procedure.<sup>47</sup>

`sqrt` プログラムはまた私達がここまで紹介した単純な手続き型言語が C や Pascal で記述可能などんな純粋数値演算プログラムを書くのにも十分であることを示しています。これには驚かれるかもしれません。私達はまだコンピュータに何かを繰り返し繰り返し行わせるどのような繰り返し(ループ)要素もこの言語には入れていないためです。一方で `Sqrt-iter` はどのように繰り返しが特別な記法を全く使わず通常の手続呼出能力のみで成し遂げられるかを実演して見せています。<sup>48</sup>

**Exercise 1.6:** Alyssa P. Hacker doesn't see why `if` needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of `cond`?" she asks. Alyssa's friend Eva Lu Ator

---

<sup>47</sup> Readers who are worried about the efficiency issues involved in using procedure calls to implement iteration should note the remarks on "tail recursion" in Section 1.2.1.

<sup>48</sup> 反復実装における手続呼出上の効率の問題を気にされている読者の方は Section 1.2.1 の "末尾再帰" 上の備考に注目して下さい

claims this can indeed be done, and she defines a new version of if:

Alyssa P. Hacker はなぜ if が特殊形式として提供される必要があるのか理解できなかった。“なぜ cond を用いた通常手続として定義できないのだろう?”と彼女は訝った。Alyssa の友達である Eva Lu Ator はこれは実際にできると主張し、if の新バージョンを定義した。

```
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
```

Eva demonstrates the program for Alyssa: Eva は Alyssa に対してプログラムのデモを行った。

```
(new-if (= 2 3) 0 5)
5
(new-if (= 1 1) 0 5)
0
```

Delighted, Alyssa uses new-if to rewrite the square-root program: 喜びながら Alyssa は new-if を用いて square-root プログラムを書き直した。

```
(define (sqrt-iter guess x)
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x) x)))
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

Alyssa が平方根の計算にこれの使用を試した時に何が起こるだろうか?

**Exercise 1.7:** The good-enough? test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements,

with examples showing how the test fails for small and large numbers. An alternative strategy for implementing `good-enough?` is to watch how `guess` changes from one iteration to the next and to stop when the change is a very small fraction of the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

平方根の演算で使用された `good-enough?` テストはとても小さい数値の平方根を見つける場合にはあまり効果的ではないだろう。また実際のコンピュータでは数値演算命令はほとんど常に精度に制限のある状態で実行される。これが我々のテストをとても大きな数値に対して不適切にする。ここまで記述についてテストがどのように小さな値と大きな値にて失敗するか例を用いて説明せよ。`good-enough?` 実装の代替戦略は `guess` がある試行から次に向かどのように変化するか監視し、変化が推定値の割合においてとても小さい時に止めることがある。このような終了テストを用いる `square-root` を設計せよ。これは小さな、及び、大きな数値に対してより良く働くだろうか？

**Exercise 1.8:** Newton's method for cube roots is based on the fact that if  $y$  is an approximation to the cube root of  $x$ , then a better approximation is given by the value

立方根に対するニュートン法は  $y$  が  $x$  の立方根である場合において以下の値により良く近似される。

$$\frac{x/y^2 + 2y}{3}$$

Use this formula to implement a cube-root procedure analogous to the square-root procedure. (In [Section 1.3.4](#) we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root procedures.)

この式を用いて `square-root` に類似した `cube-root` を実装せよ。[\(Section 1.3.4\)](#) にてこれらの `square-root` と `cube-root` の抽象化としての汎用なニュートン法の実装方法を学習します。 )

### 1.1.8 Procedures as Black-Box Abstractions

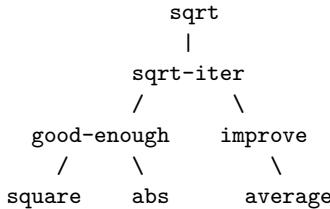
Sqrt is our first example of a process defined by a set of mutually defined procedures. Notice that the definition of `sqrt-iter` is *recursive*; that is, the procedure is defined in terms of itself. The idea of being able to define a procedure in terms of itself may be disturbing; it may seem unclear how such a “circular” definition could make sense at all, much less specify a well-defined process to be carried out by a computer. This will be addressed more carefully in [Section 1.2](#). But first let’s consider some other important points illustrated by the `sqrt` example.

`sqrt` は私達にとり相互に定義された手続の集合により定義されたプロセスの例でした。`sqrt-iter` の定義が *recursive*(再帰的) であることに注意して下さい。再帰とは手続がそれ自身の語により定義されていることです。手続をそれ自体の名前を用いて定義する考え方には不安になるかもしれません。そのような“循環的”な定義がどのようにしてつじつまを合わせるのか全く不明に見えるかもしれません。コンピュータにより実行するために良く定義された手続には指定が足りなく見えるかもしれません。これについては [Section 1.2](#) にてより注意深く触れることにします。最初はしかし `sqrt` の例にて説明されたいつか別の重要な点について考えましょう。

Observe that the problem of computing square roots breaks up naturally into a number of subproblems: how to tell whether a guess is good enough, how to improve a guess, and so on. Each of these tasks is accomplished by a separate procedure. The entire `sqrt` program can be viewed as a cluster of procedures (shown in [Figure 1.2](#)) that mirrors the decomposition of the problem into subproblems.

平方根を演算する問題が自然にいくつかの部分的な問題へ分割されることに注意して下さい。推定値が十分に良いかどのように判断するか、推定値をどのように改善するか、等です。これらのタスクの 1 つ 1 つは分離された手続により達成されます。`sqrt` プログラム全体は ([Figure 1.2](#) にて表される) 手続の群れに見てとることが可能です。この図が問題を部分問題へと分解することを映し出しています。

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts—the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures. For example, when we define the `good-enough?` procedure in terms of `square`, we are able to regard



**Figure 1.2:** Procedural decomposition of the `sqrt` program.

the `square` procedure as a “black box.” We are not at that moment concerned with *how* the procedure computes its result, only with the fact that it computes the square. The details of how the square is computed can be suppressed, to be considered at a later time. Indeed, as far as the `good-enough?` procedure is concerned, `square` is not quite a procedure but rather an abstraction of a procedure, a so-called *procedural abstraction*. At this level of abstraction, any procedure that computes the square is equally good.

この分解戦略の重要性はプログラムを部分—最初の 10 行、次の 10 行、その他へと分割するような単純なものではありません。そうではなく、各手続が他の手続の定義にてモジュールとして利用可能な特定のタスクを担うことが不可欠です。例えば `good-enough?` 手続を `square` の語を用いて定義する時、`square` 手続を“ブラックボックス”として考えることが可能です。その時、その手続がどのように結果を計算するのか気にしていません。それが二乗を計算するという事実のみです。二乗がどのように計算されるかという詳細は隠し、後の時点で考慮することができます。実際に `good-enough?` 手続について考える限り、`square` は手続では無く手続の抽象に過ぎないです。手続の抽象化と呼ばれるものです。この抽象化のレベルでは二乗を計算するどんな手続も等しく相応しいのです。

Thus, considering only the values they return, the following two procedures for squaring a number should be indistinguishable. Each takes a numerical argument and produces the square of that number as the value.<sup>49</sup> 従って返り

---

<sup>49</sup>It is not even clear which of these procedures is a more efficient implementation. This depends upon the hardware available. There are machines for which the “obvious” implementation is the less efficient one. Consider a machine that has extensive tables of logarithms and antilogarithms stored in a very efficient manner.

値のみを考えるため、以下の 2 つの二乗する手続は区別不可能となります。それぞれが数値の引数を取りその数値の二乗を値として生成します。<sup>50</sup>

```
(define (square x) (* x x))
(define (square x) (exp (double (log x))))
(define (double x) (+ x x))
```

So a procedure definition should be able to suppress detail. The users of the procedure may not have written the procedure themselves, but may have obtained it from another programmer as a black box. A user should not need to know how the procedure is implemented in order to use it.

従って手続定義は詳細を隠すことができなければなりません。手続のユーザはその手続を彼等自身で書いたとは限りません。しかし他のプログラマからブラックボックスとして取得したかもしれません。ユーザはその手続がどのように実装されているのかそれを使用するためには知る必要が無いのです。

## Local names

One detail of a procedure's implementation that should not matter to the user of the procedure is the implementer's choice of names for the procedure's formal parameters. Thus, the following procedures should not be distinguishable:

手続のユーザにとっては問題とならない手続実装の詳細の 1 つには手続の形式パラメタに対する実装者が選択した名前があります。従って以下の異なる手続は区別不可能でなければなりません。

```
(define (square x) (* x x))
(define (square y) (* y y))
```

This principle—that the meaning of a procedure should be independent of the parameter names used by its author—seems on the surface to be self-evident, but its consequences are profound. The simplest consequence is that the parameter names of a procedure must be local to the body of the procedure. For example, we used `square` in the definition of `good-enough?` in our square-root procedure:

---

<sup>50</sup>これら手続のどちらがより効率的な実装であるかは全く明確ではありません。これは実行環境依存です。“明白な”実装が効率的ではない機械が存在します。広範な対数と逆対数のテーブルをとても効率の良い方法で持つ機械について考えてみて下さい

この指針—手続の意味はその作者が使用したパラメタの名前から独立すべきである—は表面上では自明なことに思えますがそこから導きださる結論は重要です。最も単純な結論は手続のパラメタ名はその手続のボディに対してローカルであるべきというものです。例えば、まず square-root 手続の中の good-enough? の定義においては square を使用しました。

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x))
      0.001))
```

The intention of the author of `good-enough?` is to determine if the square of the first argument is within a given tolerance of the second argument. We see that the author of `good-enough?` used the name `guess` to refer to the first argument and `x` to refer to the second argument. The argument of `square` is `guess`. If the author of `square` used `x` (as above) to refer to that argument, we see that the `x` in `good-enough?` must be a different `x` than the one in `square`. Running the procedure `square` must not affect the value of `x` that is used by `good-enough?`, because that value of `x` may be needed by `good-enough?` after `square` is done computing.

`good-enough?` の作者の意図する所は第一引数の二乗が第二引数にて与えられた許容差の範囲であるかを決定することです。`good-enough?` の作者が `guess` を第一引数の参照に用い `x` を第二引数に用いたことが見てとれます。`square` の引数は `guess` です。もし `square` の作者が `x` を（上で見たように）使用した場合 `good-enough?` の `x` は `square` の `x` とは異ならなければならぬことがわかります。手続 `square` の実行は `good-enough?` の `x` の値に影響を与えてはいけません。なぜならその `x` の値は `square` が演算を終えた後にも `good-enough?` にて必要だからです。

If the parameters were not local to the bodies of their respective procedures, then the parameter `x` in `square` could be confused with the parameter `x` in `good-enough?`, and the behavior of `good-enough?` would depend upon which version of `square` we used. Thus, `square` would not be the black box we desired.

もしパラメタがそれらが関連する手続のボディに対してローカルでない場合、`square` のパラメタ `x` は `good-enough?` のパラメタ `x` と混同される可能性があります。そして `good-enough?` の挙動はどのバージョンの `square` を利用するかに依存するでしょう。従って `square` は私達が望んだブラックボックスではなくなるでしょう。

A formal parameter of a procedure has a very special role in the procedure

definition, in that it doesn't matter what name the formal parameter has. Such a name is called a *bound variable*, and we say that the procedure definition *binds* its formal parameters. The meaning of a procedure definition is unchanged if a bound variable is consistently renamed throughout the definition.<sup>51</sup> If a variable is not bound, we say that it is *free*. The set of expressions for which a binding defines a name is called the *scope* of that name. In a procedure definition, the bound variables declared as the formal parameters of the procedure have the body of the procedure as their scope.

手続きの形式パラメタは手続き定義においてとても特別な役割を持ちます。形式パラメタにはどんな名前を用いてもかまわないのです。そのような名前は*bound variable*(束縛変数)と呼ばれます。そして手続き定義はその形式パラメタを*binds(束縛する)*と呼びます。もし束縛変数が静的に定義中においてリネームされても手続き定義の意味は変わりません。<sup>52</sup> もし変数が束縛されていなければそれは*free(自由)*だと呼びます。束縛が名前を定義する式の集合はその名前の*scope(スコープ)*と呼ばれます。手続き定義においてはその手続きの形式パラメタとして宣言された束縛変数はその手続きのボディをそのスコープとします。

In the definition of `good-enough?` above, `guess` and `x` are bound variables but `<`, `-`, `abs`, and `square` are free. The meaning of `good-enough?` should be independent of the names we choose for `guess` and `x` so long as they are distinct and different from `<`, `-`, `abs`, and `square`. (If we renamed `guess` to `abs` we would have introduced a bug by *capturing* the variable `abs`. It would have changed from free to bound.) The meaning of `good-enough?` is not independent of the names of its free variables, however. It surely depends upon the fact (external to this definition) that the symbol `abs` names a procedure for computing the absolute value of a number. `Good-enough?` will compute a different function if we substitute `cos` for `abs` in its definition.

上記の `good-enough?` の定義において、`guess` と `x` は束縛変数ですが、`<`, `-`, `abs`, `square` は自由変数です。私達が選んだ `guess` と `x` の名前が、`<`, `-`, `abs`, `square` と異なり区別可能である限り、`good-enough?` の意味はそれらの名前から独立せねばなりません。(もし `guess` を `abs` にリネームした場合、変数 `abs` を *capturing(占領)*することで自由変数を束縛変数に変化させてバグを持ち込むことになるでしょう。) しかしながら `good-enough?` の意味はその自由

---

<sup>51</sup>The concept of consistent renaming is actually subtle and difficult to define formally. Famous logicians have made embarrassing errors here.

<sup>52</sup> 静的なリネームのコンセプトは実際には微妙で正式に定義するのは難しいことです。有名な論理学者達も恥しい間違いをここで犯してきました

変数の名前からは独立していません。記号 `abs` は (この定義の外部の) 数値の絶対値を求める手続に名付けられているという事実に当然、依存します。もし `cos` を `abs` にその定義において置き換えれば `good-enough?` は異なる関数を計算することでしょう。

## 内部定義とブロック構造

We have one kind of name isolation available to us so far: The formal parameters of a procedure are local to the body of the procedure. The square-root program illustrates another way in which we would like to control the use of names. The existing program consists of separate procedures:

私達は今の所、一種類の名前の分離について学びました。手続の形式パラメタは手続のボディに対してローカルです。square-root プログラムは我々が望むだろう名前使用をコントロールする別な方法を示します。

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
(define (improve guess x)
  (average guess (/ x guess)))
```

The problem with this program is that the only procedure that is important to users of `sqrt` is `sqrt`. The other procedures (`sqrt-iter`, `good-enough?`, and `improve`) only clutter up their minds. They may not define any other procedure called `good-enough?` as part of another program to work together with the square-root program, because `sqrt` needs it. The problem is especially severe in the construction of large systems by many separate programmers. For example, in the construction of a large library of numerical procedures, many numerical functions are computed as successive approximations and thus might have procedures named `good-enough?` and `improve` as auxiliary procedures. We would like to localize the subprocedures, hiding them inside `sqrt` so that `sqrt` could coexist with other successive approximations, each having its own

private `good-enough?` procedure. To make this possible, we allow a procedure to have internal definitions that are local to that procedure. For example, in the square-root problem we can write

このプログラムの問題は `sqrt` のユーザにとって重要な手続は `sqrt` のみであることです。他の手続 (`sqrt-iter`, `good-enough?`, `improve`) は彼らにとって余計なものです。ユーザは他に `good-enough?` という名の手続を、square-root プログラムと一緒に使用する他のプログラムの一部として定義することができません。なぜなら `sqrt` がそれを必要とするからです。この問題は多くの異なるプログラマにより巨大システムを構築する場合に特に深刻な問題となります。例えば数値演算の巨大ライブラリの構築において、多くの数値演算関数は一連の近似値演算として計算されるため補助的な手続として `good-enough?` と `improve` と名付けられた手続を持つかもしれません。私達は部分手続を局所化し `sqrt` の中に隠したいと思うでしょう。そうすれば `sqrt` が他の一連の近似値演算と共存し、それぞれが自身のプライベートな `good-enough?` 手続を持つことができます。これを可能にするために手続はその手続に対して局所的な内部定義を持つことが可能です。例えば `sqaure-root` プログラムは以下のように書き換えることが可能です。

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x) (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

Such nesting of definitions, called *block structure*, is basically the right solution to the simplest name-packaging problem. But there is a better idea lurking here. In addition to internalizing the definitions of the auxiliary procedures, we can simplify them. Since `x` is bound in the definition of `sqrt`, the procedures `good-enough?`, `improve`, and `sqrt-iter`, which are defined internally to `sqrt`, are in the scope of `x`. Thus, it is not necessary to pass `x` explicitly to each of these procedures. Instead, we allow `x` to be a free variable in the internal definitions, as shown below. Then `x` gets its value from the argument with which the enclosing procedure `sqrt` is called. This discipline is called *lexical*

*scoping*.<sup>53</sup>

このような定義の入れ子は *block structure*(ブロック構造) と呼ばれ、最も単純な名前パッケージ問題解決に基本的に正しい解決方法です。しかしながら良いアイデアがここに隠れています。補助的な手続を内在化させることに加えてそれらを簡潔化することができます。*x* は *sqrt* に束縛されているため、*sqrt* の内部に定義された手続 *good-enough?*, *improve*, *sqrt-iter* は *x* のスコープ内にあります。従って *x* を明示的にこれらの手続それぞれに渡す必要はありません。その代わりに *x* を以下で示すように内部の定義にて自由変数にすることができます。そして *x* は包括する手続 *sqrt* が呼ばれた時にその値を得ます。<sup>54</sup> このような規律を *lexical scoping*(レキシカルスコープ) と呼びます。

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

We will use block structure extensively to help us break up large programs into tractable pieces.<sup>55</sup> The idea of block structure originated with the programming language Algol 60. It appears in most advanced programming languages and is an important tool for helping to organize the construction of large programs.

ブロック構造は巨大なプログラムを取り扱う簡単な部品に分割するために広

<sup>53</sup>Lexical scoping dictates that free variables in a procedure are taken to refer to bindings made by enclosing procedure definitions; that is, they are looked up in the environment in which the procedure was defined. We will see how this works in detail in chapter 3 when we study environments and the detailed behavior of the interpreter.

<sup>54</sup>レキシカルスコープは手続内の自由変数が包括する手続定義により作られた束縛を参照するため用いられるよう指示します。それはつまり、手続が定義された環境の中でそれらが探されることを意味します。これがどのように働くのか、その詳細については第3章にて環境とインタプリタの詳細な挙動について学ぶ時に理解します。

<sup>55</sup>Embedded definitions must come first in a procedure body. The management is not responsible for the consequences of running programs that intertwine definition and use.

範囲にて利用されます。<sup>56</sup> ブロック構造の考え方はプログラミング言語 Algol 60 に起源を持ちます。多くの先進的なプログラミング言語に存在し、巨大プログラム構築の体系化を手助けする重要なツールです。

## 1.2 Procedures and the Processes They Generate

We have now considered the elements of programming: We have used primitive arithmetic operations, we have combined these operations, and we have abstracted these composite operations by defining them as compound procedures. But that is not enough to enable us to say that we know how to program. Our situation is analogous to that of someone who has learned the rules for how the pieces move in chess but knows nothing of typical openings, tactics, or strategy. Like the novice chess player, we don't yet know the common patterns of usage in the domain. We lack the knowledge of which moves are worth making (which procedures are worth defining). We lack the experience to predict the consequences of making a move (executing a procedure).

私達はプログラミングの要素について考えてきました。プリミティブな算術演算子を用いこれらの演算子を組み合わせ、その合成演算子を複合手続として定義することで抽象化を行ってきました。しかしそれらは私達がプログラムをどのように書くか知っていると言えるためには十分ではありません。私達の状況はチェスにおいて各駒がどのように動くのかルールを覚えたが典型的な序盤や戦術、戦略について何も知らない人に似ています。チェスの初心者棋士のように、私達はまだこの領域での慣習として的一般的パターンを知りません。私達はどの手が打つ価値があるのか (どの手続が定義する価値があるのか) の知識を欠いています。打った手の(手続実行の)結果を予想する経験を欠いています。

The ability to visualize the consequences of the actions under consideration is crucial to becoming an expert programmer, just as it is in any synthetic, creative activity. In becoming an expert photographer, for example, one must learn how to look at a scene and know how dark each region will appear on a print for each possible choice of exposure and development conditions. Only

---

<sup>56</sup>組込定義は手続本体の最初に来なければなりません。相互依存の定義と使用を行うプログラムの実行結果についてはこのような管理も責任を持つことができません

then can one reason backward, planning framing, lighting, exposure, and development to obtain the desired effects. So it is with programming, where we are planning the course of action to be taken by a process and where we control the process by means of a program. To become experts, we must learn to visualize the processes generated by various types of procedures. Only after we have developed such a skill can we learn to reliably construct programs that exhibit the desired behavior.

熟慮下の行動の結果を思い描く能力はエキスパートプログラマになるために重大です。それはどんな統合的、かつ創造的な活動についても同じです。熟練の写真家になるには例えば、景色の見方を学び、各可能な露出と現像条件の組み合わせにおいて各領域がどれだけ暗く写真に表れるかを知らなければなりません。そうして初めてフレームの計画、光量、露出、現像を逆向きに推測して望んだ効果を得ることが可能となります。プログラミングにおいてもまた同じです。プロセスが取り得る行動がどのような進行を経るのか計画し、プログラムを用いてプロセスをコントロールします。

エキスパートになるためには、数多くの種類の手続により生成されるプロセスを心に描けられるようにならなければなりません。そのようなスキルを開発した後に初めて望んだ挙動を示すプログラムを信頼できる形で構築する方法を学ぶことが可能になります。

A procedure is a pattern for the *local evolution* of a computational process. It specifies how each stage of the process is built upon the previous stage. We would like to be able to make statements about the overall, or *global*, behavior of a process whose local evolution has been specified by a procedure. This is very difficult to do in general, but we can at least try to describe some typical patterns of process evolution.

手続は計算過程の*local evolution*(局所展開)のためのパターンです。プロセスの各ステージが以前のステージの上にどのように構築されるかを指定します。ここで手続により局所展開が指示されたプロセスの全体的な、または*global*(大域的)な挙動について説明を行えればと思います。しかしこれは一般的にはとても難しいので、最低でもいくつかのプロセス展開の典型的パターンについて説明することを試してみましょう。

In this section we will examine some common “shapes” for processes generated by simple procedures. We will also investigate the rates at which these processes consume the important computational resources of time and space. The procedures we will consider are very simple. Their role is like that played by test patterns in photography: as oversimplified prototypical patterns, rather

than practical examples in their own right.

この節では簡単な手続により生成されたプロセスのためのいくつかの共通な“形”について検討してみます。またこれらのプロセスが時間と記憶域の重要な計算資源をどの程度消費するかについても調査してみます。ここで考慮する手続はとても簡単なものです。それらの役割は写真撮影におけるテストパターンにより演じられる様なものです。非常に単純化した原型的なパターンであり、それら自身の目的に沿った現実的な例ではありません。

### 1.2.1 Linear Recursion and Iteration

We begin by considering the factorial function, defined by  
階乗を求める関数を考えることから始めましょう。定義を以下に示します。

$$n! = n \cdot [(n - 1) \cdot (n - 2) \dots 3 \cdot 2 \cdot 1] = n \cdot (n - 1)!$$

There are many ways to compute factorials. One way is to make use of the observation that  $n!$  is equal to  $n$  times  $(n - 1)!$  for any positive integer  $n$ :

階乗を計算する方法は数多くあります。1つの方法は任意の正の整数  $n$ において、 $n!$  は  $n$  と  $(n - 1)!$  の積に等しいという観察結果を利用します。

$$n! = n \cdot [(n - 1)(n - 2) \dots 3 \cdot 2 \cdot 1] = n \cdot (n - 1)!$$

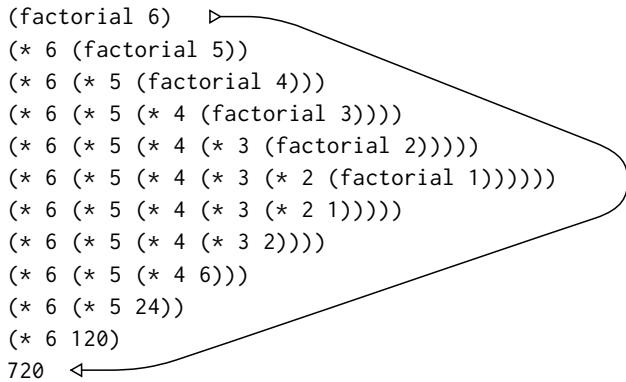
Thus, we can compute  $n!$  by computing  $(n - 1)!$  and multiplying the result by  $n$ . If we add the stipulation that  $1!$  is equal to 1, this observation translates directly into a procedure:

従って  $(n - 1)!$  を演算し、 $n$  を掛けることで  $n!$  を求めることができます。もし  $1!$  が 1 に等しいという規約を追加すればこの観察結果は直接手続に翻訳できます。

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

We can use the substitution model of Section 1.1.5 to watch this procedure in action computing  $6!$ , as shown in Figure 1.3.

Section 1.1.5の置換モデルを用いてこの手続が  $6!$  の計算を実行する様子を Figure 1.3 に示すように観察できます。



**Figure 1.3:** A linear recursive process for computing 6!.

Now let's take a different perspective on computing factorials. We could describe a rule for computing  $n!$  by specifying that we first multiply 1 by 2, then multiply the result by 3, then by 4, and so on until we reach  $n$ . More formally, we maintain a running product, together with a counter that counts from 1 up to  $n$ . We can describe the computation by saying that the counter and the product simultaneously change from one step to the next according to the rule

では階乗の演算について異なる視点を得てみましょう。 $n!$  を計算するルールを最初に 1 を 2 で掛け、その結果を 3 に掛け、次に 4 に掛け  $n$  に辿り着くまで繰り返すと説明することも可能でした。より形式的には、積の実行と、1 から  $n$  までカウントするカウンタとと一緒に保持します。カウンタと積は同時にあるステップから次へとルールに従い変更されると言うことでこの演算を説明できます。

```

product ← counter * product
counter ← counter + 1

```

and stipulating that  $n!$  is the value of the product when the counter exceeds  $n$ .

そして  $n!$  とはカウンタが  $n$  を越えた時点での積の値であると規定します。

Once again, we can recast our description as a procedure for computing factorials:<sup>57</sup>

---

<sup>57</sup>In a real program we would probably use the block structure introduced in the

繰り返します。今までの階乗を求める手続の説明を次のように書き換えられます。<sup>58</sup>

```
(define (factorial n)
  (fact-iter 1 1 n))
(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

As before, we can use the substitution model to visualize the process of computing  $6!$ , as shown in Figure 1.4.

前回と同じく、置換モデルを用いて  $6!$  の演算を Figure 1.4 として示します。

Compare the two processes. From one point of view, they seem hardly different at all. Both compute the same mathematical function on the same

---

last section to hide the definition of fact-iter:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1)))))

(iter 1 1))
```

We avoided doing this here so as to minimize the number of things to think about at once.

<sup>58</sup> 実際のプログラムでは恐らく前の節で紹介したブロック構造を fact-iter の定義を隠すために用いるでしょう。

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1)))))

(iter 1 1))
```

ここでそれを避けたのは一度に考えなければならなことを最小にするためです。

```

(factorial 6) ▷
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720 ◀

```

**Figure 1.4:** A linear iterative process for computing  $6!$ .

domain, and each requires a number of steps proportional to  $n$  to compute  $n!$ . Indeed, both processes even carry out the same sequence of multiplications, obtaining the same sequence of partial products. On the other hand, when we consider the “shapes” of the two processes, we find that they evolve quite differently.

2つのプロセスを比べてみて下さい。1つの見方としては、これらはほとんど同じに見えます。両者は同じ数学の関数を同じ領域で計算し、それぞれが  $n!$  を求めるのに  $n$  に比例したステップ数を必要とします。実際に両者のプロセスが全く同じ一連の乗算を実行し、全く同じ一連の部分的な積を得ます。一方で2つのプロセスの“形”を考えた時、全く異なった展開をしていることに気がつきます。

Consider the first process. The substitution model reveals a shape of expansion followed by contraction, indicated by the arrow in Figure 1.3. The expansion occurs as the process builds up a chain of *deferred operations* (in this case, a chain of multiplications). The contraction occurs as the operations are actually performed. This type of process, characterized by a chain of deferred operations, is called a *recursive process*. Carrying out this process requires that the interpreter keep track of the operations to be performed later on. In the computation of  $n!$ , the length of the chain of deferred multiplications, and hence the amount of information needed to keep track of it, grows linearly with  $n$  (is proportional to  $n$ ), just like the number of steps. Such a process is called a *linear recursive process*.

最初のプロセスについて考えます。Figure 1.3の矢印で示されるように、置

換モデルが展開の後、収縮する状態を明らかにしています。展開は*deferred operations*(遅延演算)の連鎖(このケースでは乗算の連なり)を構築するプロセスとして起こります。収縮は演算が実際に実行されることにより起こります。遅延演算の連鎖として示されるこのタイプのプロセスは*recursive process*(再帰プロセス)と呼ばれます。このプロセスの実行にはインタプリタが後の実行ために操作の過程を記録する必要があります。 $n!$  の演算では遅延乗算の連鎖の長さ、そしてそれに従う追跡の必要な情報の量が、 $n$  に従い線形に( $n$  に比例して)ステップ数と同様に増加します。このようなプロセスは*linear recursive process*(線形再帰プロセス)と呼ばれます。

By contrast, the second process does not grow and shrink. At each step, all we need to keep track of, for any  $n$ , are the current values of the variables `product`, `counter`, and `max-count`. We call this an *iterative process*. In general, an iterative process is one whose state can be summarized by a fixed number of *state variables*, together with a fixed rule that describes how the state variables should be updated as the process moves from state to state and an (optional) end test that specifies conditions under which the process should terminate. In computing  $n!$ , the number of steps required grows linearly with  $n$ . Such a process is called a *linear iterative process*.

対照的に、2つ目のプロセスは展開も収縮もしません。各ステップにおいて追跡が必要な物はどの  $n$  に対しても変数 `product`, `counter`, `max-count` の現在値です。これを*iterative process*(反復プロセス)と呼びます。一般的に、反復プロセスは限られた数の*state variables*(状態変数)により状態が、集約されることができ可能な物です。状態変数がプロセスが状態毎にどのように更新されるかという固定ルールとプロセスが停止する条件を指定する(任意の)終了試験と一緒に用います。 $n!$  の演算では  $n$  に従い必要なステップ数が線形に増加します。このようなプロセスは*linear iterative process*(線形反復プロセス)と呼ばれます。

The contrast between the two processes can be seen in another way. In the iterative case, the program variables provide a complete description of the state of the process at any point. If we stopped the computation between steps, all we would need to do to resume the computation is to supply the interpreter with the values of the three program variables. Not so with the recursive process. In this case there is some additional “hidden” information, maintained by the interpreter and not contained in the program variables, which indicates “where the process is” in negotiating the chain of deferred operations. The longer the chain, the more information must be maintained.<sup>59</sup>

---

<sup>59</sup>When we discuss the implementation of procedures on register machines in Chap-

2つのプロセスの対称性は他の見方もできます。反復の場合、プログラムの変数は任意のポイントにおいてプロセスの状態について完全な描写を提供します。もしステップの間で計算を停止した場合に、計算の再開を行うのに必要な全てはインタプリタに対し3つのプログラム変数の値を提供することです。再帰プロセスではそうはいきません。この場合、いくつかの追加の“隠された”情報が存在し、インタプリタにより保持されており、プログラムの変数には保存されていません。その情報には遅延命令の連鎖を辿る中での“プロセスの現在地”が示されています。鎖が長い程、より多くの情報が保持される必要があります。<sup>60</sup>

In contrasting iteration and recursion, we must be careful not to confuse the notion of a recursive *process* with the notion of a recursive *procedure*. When we describe a procedure as recursive, we are referring to the syntactic fact that the procedure definition refers (either directly or indirectly) to the procedure itself. But when we describe a process as following a pattern that is, say, linearly recursive, we are speaking about how the process evolves, not about the syntax of how a procedure is written. It may seem disturbing that we refer to a recursive procedure such as `fact-iter` as generating an iterative process. However, the process really is iterative: Its state is captured completely by its three state variables, and an interpreter need keep track of only three variables in order to execute the process.

反復と再帰の対称性において、再帰プロセスの概念と再帰手続の概念を混同しないように注意せねばなりません。私達が手続を再帰だと説明する時、手続の定義が(直接、または間接的に)その手続自身を参照するという構文上の事実を参照します。しかし、プロセスがあるパターン、例えば、線形再帰に従うと説明する時、私達はプロセスがどのように展開するかについて話しており、手続がどのように書かれているかという構文については話していません。`fact-iter`のような再帰手続を反復プロセスの生成として言及することは当惑させるかもしれません。しかし、そのプロセスは実際に反復的です。その状態は3つの状態変数により完全に補足され、インタプリタはプロセスを実行するため

---

ter 5, we will see that any iterative process can be realized “in hardware” as a machine that has a fixed set of registers and no auxiliary memory. In contrast, realizing a recursive process requires a machine that uses an auxiliary data structure known as a *stack*.

<sup>60</sup>Chapter 5にてレジスタマシン上での手続の実装について議論する時に、任意の反復プロセスが“ハードウェア内にて”固定長のレジスタ集合を持ち、補助的なメモリは持たない機械であると認識できることを学びます。対照的に、再帰プロセスを理解するには`stack`(スタック)として知られる補助的なデータ構造が必要です。

に、ただ 3 つの変数を追跡することのみが必要です。

One reason that the distinction between process and procedure may be confusing is that most implementations of common languages (including Ada, Pascal, and C) are designed in such a way that the interpretation of any recursive procedure consumes an amount of memory that grows with the number of procedure calls, even when the process described is, in principle, iterative. As a consequence, these languages can describe iterative processes only by resorting to special-purpose “looping constructs” such as `do`, `repeat`, `until`, `for`, and `while`. The implementation of Scheme we shall consider in Chapter 5 does not share this defect. It will execute an iterative process in constant space, even if the iterative process is described by a recursive procedure. An implementation with this property is called *tail-recursive*. With a tail-recursive implementation, iteration can be expressed using the ordinary procedure call mechanism, so that special iteration constructs are useful only as syntactic sugar.<sup>61</sup>

プロセスとプロシージャ(手続)の区別が混乱を招き易いのは、(Ada や Pascal, C 言語を含む)多くの一般的言語の実装が、例えプロセスが本質的には反復で記述されていても、任意の再帰手続の逐次実行が手続呼出の回数に伴い多くのメモリ容量を消費するように設計されているためです。結果としてこれらの言語は反復プロセスのみを特別な目的の“ループ構成概念”である `do`, `repeat`, `until`, `for`, `while` のような物を用いて記述します。私達が Chapter 5 にて考える Scheme の実装はこの短所を共有しません。例え反復プロセスが再帰手続により記述されても定量的な記憶域にて実行します。この属性を持つ実装は *tail-recursive*(末尾再帰) と呼ばれます。末尾再帰の実装を用いれば反復は一般的な手続呼出メカニズムを用いて表現可能であり、特別な反復構成概念は構文糖としてのみ実益のあるものとなります。<sup>62</sup>

---

<sup>61</sup>Tail recursion has long been known as a compiler optimization trick. A coherent semantic basis for tail recursion was provided by Carl Hewitt (1977), who explained it in terms of the “message-passing” model of computation that we shall discuss in Chapter 3. Inspired by this, Gerald Jay Sussman and Guy Lewis Steele Jr. (see Steele and Sussman 1975) constructed a tail-recursive interpreter for Scheme. Steele later showed how tail recursion is a consequence of the natural way to compile procedure calls (Steele 1977). The IEEE standard for Scheme requires that Scheme implementations be tail-recursive.

<sup>62</sup>末尾再帰は長い間コンパイラの最適化のための裏技として知られてきました。末尾再帰の論理的な意味論上の基礎は Carl Hewitt (1977) により与えられました。彼はそれを演算の“メッセージパッシング”モデルにて説明しました。Chapter 3 にて議論します。これに影響を受けて、Gerald Jay Sussman と Guy Lewis Steele Jr. (Steele and Sussman

**Exercise 1.9:** Each of the following two procedures defines a method for adding two positive integers in terms of the procedures `inc`, which increments its argument by 1, and `dec`, which decrements its argument by 1.

次の 2 つの各手続は 2 つの正の整数を加算する手段を定義している。手続 `inc` は引数を 1 増やし、`dec` は引数を 1 減らす。

```
(define (+ a b)
  (if (= a 0) b (inc (+ (dec a) b))))
(define (+ a b)
  (if (= a 0) b (+ (dec a) (inc b))))
```

Using the substitution model, illustrate the process generated by each procedure in evaluating  $(+ 4 5)$ . Are these processes iterative or recursive?

置換モデルを用いて各手続が  $(+ 4 5)$  の評価において生成するプロセスを図示せよ。これらのプロセスは反復であるか、再帰であるか？

**Exercise 1.10:** The following procedure computes a mathematical function called Ackermann's function.

以下の手続はアッカーマン関数と呼ばれる数学の関数を計算する。

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1) (A x (- y 1))))))
```

What are the values of the following expressions?

以下の式の値はいくつであるか？

```
(A 1 10)
(A 2 4)
(A 3 3)
```

---

1975 参照) は Scheme のための末尾再帰インタプリタを構築しました。Steele は後に末尾再帰が手続呼出をコンパイルするのにどれだけ自然な方法の結果であるかを示しました (Steele 1977)。Scheme の IEEE 標準仕様は Scheme の実装が末尾再帰であることを必須要件としています。

Consider the following procedures, where  $A$  is the procedure defined above:

$A$  が上で定義された手続である時、以下の手続について考察せよ。

```
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
(define (k n) (* 5 n n))
```

Give concise mathematical definitions for the functions computed by the procedures  $f$ ,  $g$ , and  $h$  for positive integer values of  $n$ . For example,  $(k n)$  computes  $5n^2$ .

$n$  が正の整数である場合に手続  $f$ ,  $g$ ,  $h$  により計算される関数の数学上の定義について簡明に答えよ。例として  $(k n)$  は  $5n^2$  を計算する。

### 1.2.2 Tree Recursion

Another common pattern of computation is called *tree recursion*. As an example, consider computing the sequence of Fibonacci numbers, in which each number is the sum of the preceding two:

もう 1 つの演算の一般的パターンは *tree recursion*(木再帰) と呼ばれます。例として、フィボナッチ数の計算について考えてみましょう。各数値は先行する 2 つの数の和となります。

0, 1, 1, 2, 3, 5, 8, 13, 21, ....

In general, the Fibonacci numbers can be defined by the rule

一般に、フィボナッチ数は次のルールにて定義可能です。

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \text{Fib}(n - 1) + \text{Fib}(n - 2) & \text{otherwise.} \end{cases}$$

We can immediately translate this definition into a recursive procedure for computing Fibonacci numbers:

私達は直ぐにこの定義をフィボナッチ数を計算する再帰手続の定義に翻訳が可能です。

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2)))))))
```

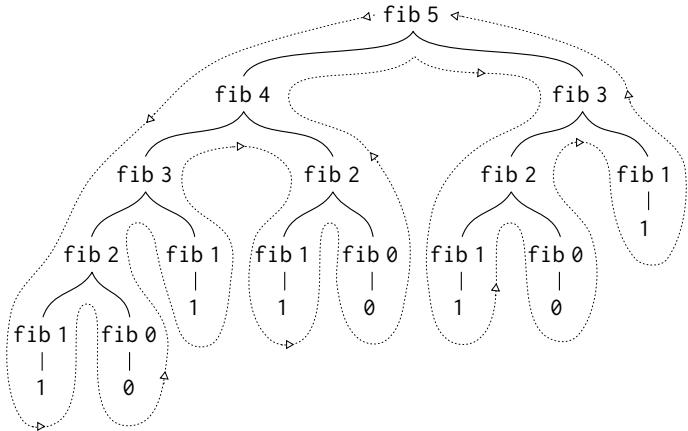
Consider the pattern of this computation. To compute `(fib 5)`, we compute `(fib 4)` and `(fib 3)`. To compute `(fib 4)`, we compute `(fib 3)` and `(fib 2)`. In general, the evolved process looks like a tree, as shown in Figure 1.5. Notice that the branches split into two at each level (except at the bottom); this reflects the fact that the `fib` procedure calls itself twice each time it is invoked.

この計算のパターンについて考えてみましょう。`(fib 5)` を計算するには `(fib 4)` と `(fib 3)` を計算します。`(fib 4)` を計算するには `(fib 3)` と `(fib 2)` を計算します。一般的に展開されたプロセスはFigure 1.5で示すように木のように見えます。枝が各レベル(最下層を除く)にて 2 つに分かれることに注意して下さい。これが `fib` 手続が実行される度に毎回、自身を二回呼び出す事實を反映しています。

This procedure is instructive as a prototypical tree recursion, but it is a terrible way to compute Fibonacci numbers because it does so much redundant computation. Notice in Figure 1.5 that the entire computation of `(fib 3)`—almost half the work—is duplicated. In fact, it is not hard to show that the number of times the procedure will compute `(fib 1)` or `(fib 0)` (the number of leaves in the above tree, in general) is precisely  $\text{Fib}(n+1)$ . To get an idea of how bad this is, one can show that the value of  $\text{Fib}(n)$  grows exponentially with  $n$ . More precisely (see Exercise 1.13),  $\text{Fib}(n)$  is the closest integer to  $\varphi^n/\sqrt{5}$ , where

この手続は典型的な木再帰としては有益です。しかしフィボナッチ数を計算するには酷い方法です。あまりにも冗長な計算を行うためです。Figure 1.5において `(fib 3)` の計算全体が—ほぼ仕事の半分が—重複していることに注意して下さい。実際には手続が `(fib 1)` や `(fib 0)` の演算回数(上記の木全体においての葉の数)が正確に  $\text{Fib}(n+1)$  であることを示すのは難しくありません。この方法の酷さを理解するために、 $\text{Fib}(n)$  の値が  $n$  に対し指数関数的に増加することを示すことができます。より正確には  $\text{Fib}(n)$  は以下の条件の場合に  $\varphi^n/\sqrt{5}$  に最も近い整数になります。(Exercise 1.13参照)

$$\varphi = \frac{1 + \sqrt{5}}{2} \approx 1.6180$$



**Figure 1.5:** The tree-recursive process generated in computing (fib 5).

is the *golden ratio*, which satisfies the equation

$\varphi$  は golden ratio(黄金比) であり次の等式を満たします。

$$\varphi^2 = \varphi + 1.$$

Thus, the process uses a number of steps that grows exponentially with the input. On the other hand, the space required grows only linearly with the input, because we need keep track only of which nodes are above us in the tree at any point in the computation. In general, the number of steps required by a tree-recursive process will be proportional to the number of nodes in the tree, while the space required will be proportional to the maximum depth of the tree.

従ってプロセスは入力に伴ない指数関数的に増加するステップ数を要します。一方で要求される記憶域は入力に対し線形にしか増加しません。なぜなら計算過程の任意のポイントにおいて、木の中のどのノードが上にあるのかのみ追跡する必要があるためです。一般的に、木再帰プロセスにおいて必要とされるステップ数は木の中のノードの数に比例します。必要とされる記憶域は木の最大の深さに対して比例します。

We can also formulate an iterative process for computing the Fibonacci numbers. The idea is to use a pair of integers  $a$  and  $b$ , initialized to  $\text{Fib}(1) = 1$  and  $\text{Fib}(0) = 0$ , and to repeatedly apply the simultaneous transformations

フィボナッチ数の計算を反復プロセスに定式化することも可能です。この考えは  $a$  と  $b$  の整数のペアを用い、 $\text{Fib}(1) = 1$  と  $\text{Fib}(0) = 0$  の初期化を行い、以下の変換を同時に行うというものです。

$$\begin{aligned} a &\leftarrow a + b, \\ b &\leftarrow a. \end{aligned}$$

It is not hard to show that, after applying this transformation  $n$  times,  $a$  and  $b$  will be equal, respectively, to  $\text{Fib}(n+1)$  and  $\text{Fib}(n)$ . Thus, we can compute Fibonacci numbers iteratively using the procedure

この変換を  $n$  回行った後に  $a$  と  $b$  がそれぞれ  $\text{Fib}(n+1)$  と  $\text{Fib}(n)$  に等しいことを示すのは難しくありません。従ってフィボナッチ数を反復的に以下の手続を用いて計算可能です。

```
(define (fib n)
  (fib-iter 1 0 n))
(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```

This second method for computing  $\text{Fib}(n)$  is a linear iteration. The difference in number of steps required by the two methods—one linear in  $n$ , one growing as fast as  $\text{Fib}(n)$  itself—is enormous, even for small inputs.

この  $\text{Fib}(n)$  を計算する 2 つ目の方法は線形反復です。2 つの方法により要求されるステップ数の差は、一方は  $n$  に対し線形であり、もう一方は  $\text{Fib}(n)$  自身の値の速さで増加するため、例え入力値が小さくてもその差は非常に大きくなります。

One should not conclude from this that tree-recursive processes are useless. When we consider processes that operate on hierarchically structured data rather than numbers, we will find that tree recursion is a natural and powerful tool.<sup>63</sup> But even in numerical operations, tree-recursive processes can be useful in helping us to understand and design programs. For instance, although

---

<sup>63</sup>An example of this was hinted at in Section 1.1.3: The interpreter itself evaluates expressions using a tree-recursive process.

the first `fib` procedure is much less efficient than the second one, it is more straightforward, being little more than a translation into Lisp of the definition of the Fibonacci sequence. To formulate the iterative algorithm required noticing that the computation could be recast as an iteration with three state variables.

これより木再帰プロセスが役に立たないと結論づけるべきではありません。数値ではなく階層構造のデータを操作するプロセスを考えた場合、木再帰は自然で強力なツールです。<sup>64</sup>しかし、例え数値演算においても木再帰はプログラムの設計と理解を手助けするのに役立ちます。例えば最初の `fib` 手続は 2 つ目に比べてとても非効率ですが、より直感的でフィボナッチ数列の定義と Lisp 翻訳の違いは大差ありません。反復アルゴリズムの定式化を行うためには、計算が 3 つの状態変数に再定義できることに気付く必要があります。

### Example: Counting change

It takes only a bit of cleverness to come up with the iterative Fibonacci algorithm. In contrast, consider the following problem: How many different ways can we make change of \$1.00, given half-dollars, quarters, dimes, nickels, and pennies? More generally, can we write a procedure to compute the number of ways to change any given amount of money?

反復的フィボナッチアルゴリズムに至るには多少の知恵が必要です。一方で、次の問題について考えてみて下さい：\$1.00 を両替するにはいくつの方法があるでしょうか？50 セント、25 セント、10 セント、5 セント、1 セント硬貨があります。より一般的に、任意の量の金額に対して両替方法がいくつ存在するか計算する手続を書くことができますか？

This problem has a simple solution as a recursive procedure. Suppose we think of the types of coins available as arranged in some order. Then the following relation holds:

この問題には再帰手続としての簡単な答が存在します。利用可能なコインのタイプがある順序で並べると考えてみましょう。すると以下の関係が成り立ちます。

The number of ways to change amount  $a$  using  $n$  kinds of coins equals  $n$  種類の硬貨を用いた場合、金額  $a$  の両替方法の数は

---

<sup>64</sup> これの例はSection 1.1.3: インタプリタ自身が木再帰プロセスを用いて式を評価することから暗示されます

- the number of ways to change amount  $a$  using all but the first kind of coin,  
最初の種類の硬貨を除いた残り全てを用いた金額  $a$  の両替方法の数、  
plus プラス
- the number of ways to change amount  $a - d$  using all  $n$  kinds of coins,  
where  $d$  is the denomination of the first kind of coin.  
 $d$  が最初の種類の硬貨の額面である場合に、 $n$  種類の硬貨全てを用いた  
金額  $a - d$  の両替方法の数

To see why this is true, observe that the ways to make change can be divided into two groups: those that do not use any of the first kind of coin, and those that do. Therefore, the total number of ways to make change for some amount is equal to the number of ways to make change for the amount without using any of the first kind of coin, plus the number of ways to make change assuming that we do use the first kind of coin. But the latter number is equal to the number of ways to make change for the amount that remains after using a coin of the first kind.

なぜこれが正しいのか考えるためには両替方法が 2 つのグループに分けられることに注目します。最初の種類の硬貨を用いないものと、用いるものです。従ってある金額に対する両替方法の数の総数は最初の種類の硬貨を全く使わないその金額に対する両替方法の数と最初のコインを用いる両替方法の数の和です。しかし後者の数は最初の種類の硬貨を用いた後の残りの金額に対する両替方法の数に等しくなります。

Thus, we can recursively reduce the problem of changing a given amount to the problem of changing smaller amounts using fewer kinds of coins. Consider this reduction rule carefully, and convince yourself that we can use it to describe an algorithm if we specify the following degenerate cases:<sup>65</sup>

従って与えられた金額の両替問題から少ない種類の硬貨を用いたより少ない金額の両替問題へと再帰的に縮小することができます。この集約ルールについて注意深く考えてください。そして自分自身でそのルールを用いて以下の縮退ケースを指定した場合アルゴリズムを記述できるように準備して下さい。<sup>66</sup>

---

<sup>65</sup>For example, work through in detail how the reduction rule applies to the problem of making change for 10 cents using pennies and nickels.

<sup>66</sup>例えば 5 セント硬貨と 1 セント硬貨を用いて 10 セントの両替を行う問題に縮退ルールをどのように適用するかについて、詳細に通じてやって見て下さい

- If  $a$  is exactly 0, we should count that as 1 way to make change. もし  $a$  が 0 である場合、両替方法は 1 と数える
- If  $a$  is less than 0, we should count that as 0 ways to make change. もし  $a$  が 0 未満の場合、両替方法は 0 と数える
- If  $n$  is 0, we should count that as 0 ways to make change. もし  $n$  が 0 の場合、両替方法は 0 と数える

We can easily translate this description into a recursive procedure: この記述は簡単に再帰手続に翻訳できます。

```
(define (count-change amount) (cc amount 5))
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                      (- kinds-of-coins 1))
                  (cc (- amount
                           (first-denomination
                             kinds-of-coins))
                      kinds-of-coins)))))
(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

(The `first-denomination` procedure takes as input the number of kinds of coins available and returns the denomination of the first kind. Here we are thinking of the coins as arranged in order from largest to smallest, but any order would do as well.) We can now answer our original question about changing a dollar:

(`first-denomination` 手続は利用可能な硬貨の種類の数を取り、最初の種類の硬貨の額面を返します。ここでは硬貨は最大額面から最小への順に並んでいいると仮定しますが、どのような順でもうまく行きます) これで元々の質問である \$1 の両替について回答ができます。

```
(count-change 100)
```

`Count-change` generates a tree-recursive process with redundancies similar to those in our first implementation of `fib`. (It will take quite a while for that 292 to be computed.) On the other hand, it is not obvious how to design a better algorithm for computing the result, and we leave this problem as a challenge. The observation that a tree-recursive process may be highly inefficient but often easy to specify and understand has led people to propose that one could get the best of both worlds by designing a “smart compiler” that could transform tree-recursive procedures into more efficient procedures that compute the same result.<sup>67</sup>

`count-change` は `fib` の最初の実装と同様に冗長な木再帰プロセスを生成します。(292 が演算されるのに暫く時間がかかるでしょう) 一方で結果を求めるのにより良いアルゴリズムをどのように設計するかは自明ではありません。この問題は読者への宿題とします。木再帰プロセスはとても非効率ですが多くの場合、指示と理解が簡単であることが人々に対し、ユーザが両者の世界の良い面を得られる、木再帰手続をより効率的で等価な手続へと変換を行う“賢いコンパイラ”の設計を提案する方向へと向かわせています。<sup>68</sup>

**Exercise 1.11:** A function  $f$  is defined by the rule that  $f(n) = n$  if  $n < 3$  and  $f(n) = f(n - 1) + 2f(n - 2) + 3f(n - 3)$  if  $n \geq 3$ . Write a procedure that computes  $f$  by means of a recursive process. Write a procedure that computes  $f$  by means of an iterative process.

---

<sup>67</sup>One approach to coping with redundant computations is to arrange matters so that we automatically construct a table of values as they are computed. Each time we are asked to apply the procedure to some argument, we first look to see if the value is already stored in the table, in which case we avoid performing the redundant computation. This strategy, known as *tabulation* or *memoization*, can be implemented in a straightforward way. Tabulation can sometimes be used to transform processes that require an exponential number of steps (such as `count-change`) into processes whose space and time requirements grow linearly with the input. See [Exercise 3.27](#).

<sup>68</sup>冗長な演算に対処する 1 つの取り組み方法は演算結果に従い、自動的に値のテーブルを構築することです。手続がある引数に適用するよう要求される度に、最初にその値が既にテーブルに存在するかを確認します。その場合、冗長な演算を防ぐことが可能です。この戦略は *tabulation*(表形式化) や *memoization*(メモ化) として知られ直感的な方法で実装が可能です。表形式化は時折、(`count-change` のような) 指数関数的なステップ数を要するプロセスを、入力に対し時間と記憶域の要求が線形に増加するプロセスへと変換するのに利用されます。(Exercise 3.27 参照)

関数  $f$  は  $n < 3$  の場合  $f(n) = n$  と  $n \geq 3$  の場合、 $f(n) = f(n - 1) + 2f(n - 2) + 3f(n - 3)$  のルールの下に定義される。 $f$  を演算する手続を再帰プロセスを用いて書け。また  $f$  を演算する手続を反復プロセスを用いて書け。

**Exercise 1.12:** The following pattern of numbers is called *Pascal's triangle*.

以下の数値のパターンは *Pascal's triangle*(パスカルの三角形) と呼ばれる。

		1		
	1	1		
	1	2	1	
	1	3	3	1
1	4	6	4	1
		.	.	.

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it.<sup>69</sup> Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

三角の端の数値は全て 1 であり、三角内部の各数値はそれの上 2 つの数値の和である。<sup>70</sup> パスカルの三角形の要素を再帰プロセス

---

<sup>69</sup>The elements of Pascal's triangle are called the *binomial coefficients*, because the  $n^{\text{th}}$  row consists of the coefficients of the terms in the expansion of  $(x + y)^n$ . This pattern for computing the coefficients appeared in Blaise Pascal's 1653 seminal work on probability theory, *Traité du triangle arithmétique*. According to Knuth (1973), the same pattern appears in the *Szu-yuen Yü-chien* ("The Precious Mirror of the Four Elements"), published by the Chinese mathematician Chu Shih-chieh in 1303, in the works of the twelfth-century Persian poet and mathematician Omar Khayyam, and in the works of the twelfth-century Hindu mathematician Bháskara Áchárya.

<sup>70</sup>パスカルの三角形の各要素は *binomial coefficients*(二項係数) と呼ばれます。 $n$  番目の行が  $(x + y)^n$  の展開式における各項の係数であるためです。係数を計算するこのパターンは Blaise Pascal の 1653 年の確率理論の独創的な成果である *Traité du triangle arithmétique* に現れました。Knuth (1973) によると、1303 年に同様のパターンが中国の数学者、朱世傑により出版された *Szu-yuen Yü-chien* ("The Precious Mirror of the Four Elements") (四元玉鑑) の中にも記載されています。また 12 世紀のペルシャの詩人であり数学者であった Omar Khayyam、同じく 12 世紀のインド人数学者 Bháskara Áchárya についても同様です。

を用いて求める手続を書け。

**Exercise 1.13:** Prove that  $\text{Fib}(n)$  is the closest integer to  $\varphi^n / \sqrt{5}$ , where  $\varphi = (1 + \sqrt{5})/2$ . Hint: Let  $\psi = (1 - \sqrt{5})/2$ . Use induction and the definition of the Fibonacci numbers (see [Section 1.2.2](#)) to prove that  $\text{Fib}(n) = (\varphi^n - \psi^n) / \sqrt{5}$ .

$\text{Fib}(n)$  が  $\varphi^n / \sqrt{5}$  に最も近い整数であることを証明せよ。 $\varphi = (1 + \sqrt{5})/2$  とする。ヒント： $\psi = (1 - \sqrt{5})/2$  と置く。帰納法とフィボナッチ数の定義 ([Section 1.2.2](#)参照) を用いて  $\text{Fib}(n) = (\varphi^n - \psi^n) / \sqrt{5}$  であることを証明せよ。

### 1.2.3 Orders of Growth

The previous examples illustrate that processes can differ considerably in the rates at which they consume computational resources. One convenient way to describe this difference is to use the notion of *order of growth* to obtain a gross measure of the resources required by a process as the inputs become larger.

前節の例ではプロセスが消費する計算資源の割合が大幅に異なることを示しました。この違いを説明する1つの便利な方法には*order of growth*の記法を用いて入力が大きくなるにつれプロセスが要求するリソース(資源)の総体的量の説明を得ることができます。

Let  $n$  be a parameter that measures the size of the problem, and let  $R(n)$  be the amount of resources the process requires for a problem of size  $n$ . In our previous examples we took  $n$  to be the number for which a given function is to be computed, but there are other possibilities. For instance, if our goal is to compute an approximation to the square root of a number, we might take  $n$  to be the number of digits accuracy required. For matrix multiplication we might take  $n$  to be the number of rows in the matrices. In general there are a number of properties of the problem with respect to which it will be desirable to analyze a given process. Similarly,  $R(n)$  might measure the number of internal storage registers used, the number of elementary machine operations performed, and so on. In computers that do only a fixed number of operations at a time, the time required will be proportional to the number of elementary machine operations performed.

$n$  が問題サイズを測るパラメータ、 $R(n)$  をサイズ  $n$  の問題に対しプロセスが要求するリソースの量だとします。前節の例では  $n$  を与えられた関数が何回計算されるかの数としました。しかし他の可能性もあります。例えば、もし私達のゴールが数値の平方根の近似値を求めることがあれば、 $n$  を必要な精度の桁数と取ることもありえるでしょう。行列の乗算では  $n$  を行列の行数と取るかもしれません。一般的に与えられた問題を分析するのに望ましい問題の属性はいくつもあります。同様に、 $R(n)$  が使用される内部保管レジスタの数を量ったり、実行された基本的機械語命令の数であったり等します。一度に固定数の命令を実行する計算機においては必要とされる時間は実行される基本的機械語命令の数に比例します。

We say that  $R(n)$  has order of growth  $\Theta(f(n))$ , written  $R(n) = \Theta(f(n))$  (pronounced “theta of  $f(n)$ ”), if there are positive constants  $k_1$  and  $k_2$  independent of  $n$  such that  $k_1 f(n) \leq R(n) \leq k_2 f(n)$  for any sufficiently large value of  $n$ . (In other words, for large  $n$ , the value  $R(n)$  is sandwiched between  $k_1 f(n)$  and  $k_2 f(n)$ .)

もし任意の十分に大きな  $n$  の値に対して正の定数  $k_1$  と  $k_2$  が  $n$  に独立して存在し  $k_1 f(n) \leq R(n) \leq k_2 f(n)$  を満たす時、 $R(n)$  は増加の次数  $\Theta(f(n))$  を持ち  $R(n) = \Theta(f(n))$  (“シータ  $f(n)$ ”と発音する) と記述されます。

For instance, with the linear recursive process for computing factorial described in Section 1.2.1 the number of steps grows proportionally to the input  $n$ . Thus, the steps required for this process grows as  $\Theta(n)$ . We also saw that the space required grows as  $\Theta(n)$ . For the iterative factorial, the number of steps is still  $\Theta(n)$  but the space is  $\Theta(1)$ —that is, constant.<sup>71</sup> The tree-recursive Fibonacci computation requires  $\Theta(\varphi^n)$  steps and space  $\Theta(n)$ , where  $\varphi$  is the golden ratio described in Section 1.2.2.

例として、Section 1.2.1で説明した階乗を求める線形再帰プロセスではステップ数は入力  $n$  に比例します。従ってこのプロセスに必要なステップ数は  $\Theta(n)$  に従い増加します。必要とされる記憶域もまた  $\Theta(n)$  に従い増加します。反復式階乗ではステップ数はまだ  $\Theta(n)$  ですが、記憶域は  $\Theta(1)$ —定数です。<sup>72</sup> 木再帰フィボナッチ演算は  $\Theta(\varphi^n)$  ステップと記憶域  $\Theta(n)$  を必要とします。こ

---

<sup>71</sup>These statements mask a great deal of oversimplification. For instance, if we count process steps as “machine operations” we are making the assumption that the number of machine operations needed to perform, say, a multiplication is independent of the size of the numbers to be multiplied, which is false if the numbers are sufficiently large. Similar remarks hold for the estimates of space. Like the design and description of a process, the analysis of a process can be carried out at various levels of abstraction.

<sup>72</sup>これらの記述は多くの過度な単純化を隠しています。例えばもし“機械語命令”をブ

の時  $\varphi$  はSection 1.2.2で示したとおりの黄金比です。

Orders of growth provide only a crude description of the behavior of a process. For example, a process requiring  $n^2$  steps and a process requiring  $1000n^2$  steps and a process requiring  $3n^2 + 10n + 17$  steps all have  $\Theta(n^2)$  order of growth. On the other hand, order of growth provides a useful indication of how we may expect the behavior of the process to change as we change the size of the problem. For a  $\Theta(n)$  (linear) process, doubling the size will roughly double the amount of resources used. For an exponential process, each increment in problem size will multiply the resource utilization by a constant factor. In the remainder of Section 1.2 we will examine two algorithms whose order of growth is logarithmic, so that doubling the problem size increases the resource requirement by a constant amount.

増加の次数はプロセスの行いについて概観的な説明のみを与えます。例えば  $n^2$  ステップ、 $1000n^2$  ステップ、 $3n^2 + 10n + 17$  ステップを必要とするプロセスは全て增加の次数は  $\Theta(n^2)$  になります。一方で增加の次数は問題のサイズを変更した場合にどの程度プロセスの挙動が変化するかを推測するのに実用的な指標です。 $\Theta(n)$  の線形プロセスに対しサイズを 2 倍にした場合、概ね 2 倍のリソースを使用します。指数関数的プロセスに対しては問題サイズを 1 増やす度、定数因子をリソース使用率にかけることになります。Section 1.2 の最後にて增加の次数が対数である 2 つのアルゴリズムをでは、問題サイズを倍にした時に必要とするリソースが定数量増えることを調査します。

**Exercise 1.14:** Draw the tree illustrating the process generated by the `count-change` procedure of Section 1.2.2 in making change for 11 cents. What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

Section 1.2.2 の `count-change` 手続により 11 セントの両替を求めた場合に生成されるプロセスの木を図示せよ。量が増えるに従いこのプロセスにより使用される記憶域とステップ数の增加の次数はいくつか？

**Exercise 1.15:** The sine of an angle (specified in radians) can be

---

ロセスのステップ数として数えた場合に、一例として、乗算に対し必要な機械語命令の数は乗算される 2 つの数に対し独立していると想定するとします。それは数値が十分に大きな場合は間違います。同様の見解が記憶域に対する見積に対しても取られます。プロセスの設計と記述のように、プロセスの分析は抽象化の色々なレベルに対して行えます。

computed by making use of the approximation  $\sin x \approx x$  if  $x$  is sufficiently small, and the trigonometric identity

(ラジアンで指定される) 角度の正弦値は  $x$  が十分に小さい時  $\sin x \approx x$  の近似式を用いることで計算できる。そして三角法の恒等式、

$$\sin x = 3 \sin \frac{x}{3} - 4 \sin^3 \frac{x}{3}$$

to reduce the size of the argument of sin. (For purposes of this exercise an angle is considered “sufficiently small” if its magnitude is not greater than 0.1 radians.) These ideas are incorporated in the following procedures:

を用いて sin の引数の大きさを縮小することができる。(この課題の目的では“十分に小さい”とはその大きさが 0.1 ラジアンよりも大きくないことをとする) これらの考えが以下の手続に組込まれている。

```
(define (cube x) (* x x x))
(define (p x) (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
  (if (not (> (abs angle) 0.1))
      angle
      (p (sine (/ angle 3.0)))))
```

a How many times is the procedure p applied when (sine 12.15) is evaluated?

手続 p は (sine 12.15) を評価した時、何回適用されるか？

b What is the order of growth in space and number of steps (as a function of a) used by the process generated by the sine procedure when (sine a) is evaluated?

(sine a) が評価された時、sine 手続により生成されたプロセスにより使用された (a の関数としての) 記憶域とステップ数の増加の次数を求めよ。

## 1.2.4 Exponentiation

Consider the problem of computing the exponential of a given number. We would like a procedure that takes as arguments a base  $b$  and a positive integer exponent  $n$  and computes  $b^n$ . One way to do this is via the recursive definition

与えられた数値の指数関数を求める問題について考えましょう。基底  $b$  と正の整数である指数  $n$  を引数に取り  $b^n$  を求める手続にします。再帰定義によりこれを行う 1 つの方法は次の通りです。

$$\begin{aligned} b^n &= b \cdot b^{n-1}, \\ b^0 &= 1, \end{aligned}$$

which translates readily into the procedure

早速、手続に翻訳します。

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

This is a linear recursive process, which requires  $\Theta(n)$  steps and  $\Theta(n)$  space. Just as with factorial, we can readily formulate an equivalent linear iteration:

これは線形再帰プロセスであり、 $\Theta(n)$  ステップと記憶域  $\Theta(n)$  を必要とします。階乗と同様にすぐに等価な線形反復へと定式化可能です。

```
(define (expt b n)
  (expt-iter b n 1))
(define (expt-iter b counter product)
  (if (= counter 0)
      product
      (expt-iter b
                 (- counter 1)
                 (* b product))))
```

This version requires  $\Theta(n)$  steps and  $\Theta(1)$  space.

このバージョンは  $\Theta(n)$  ステップと記憶域  $\Theta(1)$  を必要とします。

We can compute exponentials in fewer steps by using successive squaring. For instance, rather than computing  $b^8$  as

指数関数は二乗を連続して用いることでより少ないステップで計算できます。例えば、 $b^8$  を以下のように計算するのではなく、

$$b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b)))))),$$

we can compute it using three multiplications: 3回の乗算で求めることが可能です。

$$\begin{aligned}b^2 &= b \cdot b, \\b^4 &= b^2 \cdot b^2, \\b^8 &= b^4 \cdot b^4.\end{aligned}$$

This method works fine for exponents that are powers of 2. We can also take advantage of successive squaring in computing exponentials in general if we use the rule この方法は 2 の幂乗である指数関数についてはうまく働きます。また連続する二乗の利点を一般的な指数関数の演算に対し以下のルールに従うこと で利用可能です。

$$\begin{aligned}b^n &= (b^{n/2})^2 && \text{if } n \text{ is even,} \\b^n &= b \cdot b^{n-1} && \text{if } n \text{ is odd.}\end{aligned}$$

We can express this method as a procedure: この方法を手続として表現します。

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

where the predicate to test whether an integer is even is defined in terms of the primitive procedure `remainder` by

整数が偶数であるかテストする述語はプリミティブな手続、`remainder` を用い以下のように定義される。

```
(define (even? n)
  (= (remainder n 2) 0))
```

The process evolved by `fast-expt` grows logarithmically with  $n$  in both space and number of steps. To see this, observe that computing  $b^{2n}$  using `fast-expt` requires only one more multiplication than computing  $b^n$ . The size of the exponent we can compute therefore doubles (approximately) with every new multiplication we are allowed. Thus, the number of multiplications required for

an exponent of  $n$  grows about as fast as the logarithm of  $n$  to the base 2. The process has  $\Theta(\log n)$  growth.<sup>73</sup>

`fast-expt` により展開されるプロセスは  $n$  の対数に従い記憶域とステップ数の両者が増加します。これを理解するために  $b^{2n}$  を `fast-expt` を用いて演算するのに  $b^n$  の演算よりただ 1 度のみ多くの乗算が必要であることに注目して下さい。従って計算可能な指数のサイズは、可能な新規の乗算の度に（大体）倍になります。このため  $n$  の指数により必要とされる乗算の数は 2 を底とする  $n$  の対数と同等の早さにて増加します。このプロセスは  $\Theta(\log n)$  で増加します。

<sup>74</sup>

The difference between  $\Theta(\log n)$  growth and  $\Theta(n)$  growth becomes striking as  $n$  becomes large. For example, `fast-expt` for  $n = 1000$  requires only 14 multiplications.<sup>75</sup> It is also possible to use the idea of successive squaring to devise an iterative algorithm that computes exponentials with a logarithmic number of steps (see [Exercise 1.16](#)), although, as is often the case with iterative algorithms, this is not written down so straightforwardly as the recursive algorithm.<sup>76</sup>

$\Theta(\log n)$  の増加と  $\Theta(n)$  の増加の違いは  $n$  が大きくなる程顕著になります。例えば  $n = 1000$  の時 `fast-expt` は 14 回しか乗算を必要としません。<sup>77</sup> 連続する二乗の考え方を用いて対数ステップ数の指數関数を求める反復アルゴリズムを考案することも可能です。（[Exercise 1.16](#)参照）しかし反復アルゴリズムでは良くあることですが、これは再帰アルゴリズムのように直接的に書下すことが

<sup>73</sup>More precisely, the number of multiplications required is equal to 1 less than the log base 2 of  $n$  plus the number of ones in the binary representation of  $n$ . This total is always less than twice the log base 2 of  $n$ . The arbitrary constants  $k_1$  and  $k_2$  in the definition of order notation imply that, for a logarithmic process, the base to which logarithms are taken does not matter, so all such processes are described as  $\Theta(\log n)$ .

<sup>74</sup>より正確に言えば、必要とされる乗算の数は 1 から  $n$  の基底 2 の対数未満と  $n$  の二進数表現における 1 の数の和になります。この合計が常に  $n$  の基底 2 の対数の 2 倍よりも小さくなります。オーダ（次数）記法の定義に従う任意の定数  $k_1$  と  $k_2$  により、対数プロセスに対し、その対数の基底は問題ではないため、そのようなプロセス全ては  $\Theta(\log n)$  と説明されることが示されます

<sup>75</sup>You may wonder why anyone would care about raising numbers to the 1000th power. See [Section 1.2.6](#).

<sup>76</sup>This iterative algorithm is ancient. It appears in the *Chandah-sutra* by Áchárya Pingala, written before 200 B.C. See [Knuth 1981](#), section 4.6.3, for a full discussion and analysis of this and other methods of exponentiation.

<sup>77</sup>誰が数値を 1000 乗まで上げることを気にするのだろうかと思うかもしれません。 [Section 1.2.6](#)を参照して下さい

できません。<sup>78</sup> Knuth 1981 の 4.6.3 節にてこれと指數関数の他の方法について完全な議論と分析を行っています。

**Exercise 1.16:** Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does `fast-expt`. (Hint: Using the observation that  $(b^{n/2})^2 = (b^2)^{n/2}$ , keep, along with the exponent  $n$  and the base  $b$ , an additional state variable  $a$ , and define the state transformation in such a way that the product  $ab^n$  is unchanged from state to state. At the beginning of the process  $a$  is taken to be 1, and the answer is given by the value of  $a$  at the end of the process. In general, the technique of defining an *invariant quantity* that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

連続二乗と対数ステップ数を用いる `fast-expt` のような反復指數関数プロセスを展開する手続を設計せよ。(ヒント:( $b^{n/2})^2 = (b^2)^{n/2}$  を用い、指數  $n$ 、基數  $b$  と共に追加の状態変数  $a$  を保持し状態変換を積  $ab^n$  が状態間において一定であるという方法にて定義せよ。プロセスの最初において  $a$  は 1 を取り、回答はプロセスの終了時に  $a$  の値として得られる。一般的に、状態間において一定である *invariant quantity*(不变量) を定義する技法は反復アルゴリズムの設計を考える上で強力な方法である。)

**Exercise 1.17:** The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (in which it is assumed that our language can only add, not multiply) is analogous to the `expt` procedure:

この節における指數演算アルゴリズムは指數関数を乗算の繰り返しを用いて実行することを基本としている。同様な手段で、整数の乗算を加算の繰り返しを用いて実行することも可能だ。以下の乗算手続(私達の言語が足し算だけ可能で乗算はできないと仮定する)は `expt` 手続の類似である。

---

<sup>78</sup> この反復アルゴリズムは古代から存在します。紀元前 200 年以前に Áchárya Pingala により書かれた *Chandah-sutra* には現れています

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1))))))
```

This algorithm takes a number of steps that is linear in  $b$ . Now suppose we include, together with addition, operations `double`, which doubles an integer, and `halve`, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to `fast-expt` that uses a logarithmic number of steps.

このアルゴリズムは  $b$  に対し線形のステップ数を取る。今、和に加えて整数を倍にする `double` と偶数を 2 で割る `halve` があるとする。これらを用いて `fast-expt` に類似して対数ステップ数を持つ乗算手続を設計せよ。

**Exercise 1.18:** Using the results of [Exercise 1.16](#) and [Exercise 1.17](#), devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.<sup>79</sup>

[Exercise 1.16](#)と[Exercise 1.17](#)の結果を用いて反復プロセスを生成する 2 つの整数を乗算する手続を考案せよ。足し算、`double`、`halve` を用い対数ステップ数のアルゴリズムを使用すること。<sup>80</sup>

**Exercise 1.19:** There is a clever algorithm for computing the Fibonacci numbers in a logarithmic number of steps. Recall the transformation of the state variables  $a$  and  $b$  in the `fib-iter` process of [Section 1.2.2](#):  $a \leftarrow a+b$  and  $b \leftarrow a$ . Call this transformation  $T$ , and observe that applying  $T$  over and over again  $n$  times, starting with 1 and 0, produces the pair  $\text{Fib}(n+1)$  and  $\text{Fib}(n)$ . In other words, the Fibonacci numbers are produced by applying  $T^n$ , the

---

<sup>79</sup>This algorithm, which is sometimes known as the “Russian peasant method” of multiplication, is ancient. Examples of its use are found in the Rhind Papyrus, one of the two oldest mathematical documents in existence, written about 1700 B.C. (and copied from an even older document) by an Egyptian scribe named A’h-mose.

<sup>80</sup>このアルゴリズムは時折“ロシア農民のかけ算”として知られており、古くから存在します。その使用例は最も古い数学の書籍の 1 つ、リンドパビルスにも見られます。これは紀元前 1700 年頃にエジプトの筆記者、A’h-mose により書かれた（そしてより古い書物から写本された）本です。

$n^{\text{th}}$  power of the transformation  $T$ , starting with the pair  $(1, 0)$ . Now consider  $T$  to be the special case of  $p = 0$  and  $q = 1$  in a family of transformations  $T_{pq}$ , where  $T_{pq}$  transforms the pair  $(a, b)$  according to  $a \leftarrow bq + aq + ap$  and  $b \leftarrow bp + aq$ . Show that if we apply such a transformation  $T_{pq}$  twice, the effect is the same as using a single transformation  $T'_{p'q'}$  of the same form, and compute  $p'$  and  $q'$  in terms of  $p$  and  $q$ . This gives us an explicit way to square these transformations, and thus we can compute  $T^n$  using successive squaring, as in the **fast-expt** procedure. Put this all together to complete the following procedure, which runs in a logarithmic number of steps.<sup>81</sup>

フィボナッチ数を対数ステップ数にて求める巧みなアルゴリズムが存在する。Section 1.2.2 の **fib-iter** にて  $a$  と  $b$  状態変数の変換  $a \leftarrow a + b$  と  $b \leftarrow a$  を思い出そう。この変換を  $T$  と呼び、1 と 0 から始めて  $n$  回繰り返して  $T$  を適用した時に  $\text{Fib}(n+1)$  と  $\text{Fib}(n)$  のペアを算出することに注意せよ。言い換えれば、フィボナッチ数は変換  $T$  の  $n$  乗である  $T^n$  をペア  $(1, 0)$  から始めて適用することである。ここで  $T$  は  $p = 0, q = 1$  である時の変換  $T_{pq}$  の特別な形であると考えてみよう。この時  $T_{pq}$  は  $(a, b)$  を  $a \leftarrow bq + aq + ap$  and  $b \leftarrow bp + aq$  とする。もしそのような変換  $T_{pq}$  を二回適用した場合にその効果は同形変換  $T'_{p'q'}$  を一回適用した場合と同じであることを示せ。また  $p$  と  $q$  に対する  $p'$  and  $q'$  を求めよ。これは **fast-expt** 手続におけるように、 $T^n$  を連続する平方にて求める。これらを全て一緒に考慮して次の手続を完成させよ。これは対数ステップ数にて実行される。<sup>82</sup>

```
(define (fib n)
  (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   p
                   (+ (* 2 p) q)
                   (/ count 2)))
        (else (fib-iter (+ (* (- b) q) (* (- a) p))
                        (+ (* (- b) p) (* (- a) q))
                        p
                        q
                        (- count 1))))))
```

---

<sup>81</sup>This exercise was suggested to us by Joe Stoy, based on an example in Kaldewaij 1990.

<sup>82</sup>この課題はKaldewaij 1990の例をベースに Joe Stoy により提案されました

```

    ⟨??⟩      ; compute p'
    ⟨??⟩      ; compute q'
    (/ count 2)))
(else (fib-iter (+ (* b q) (* a q) (* a p))
                  (+ (* b p) (* a q)))
                  p
                  q
                  (- count 1))))))

```

## 1.2.5 Greatest Common Divisors

Greatest Common divisor (GCD: 最大公約数) of two integers  $a$  and  $b$  is defined to be the largest integer that divides both  $a$  and  $b$  with no remainder. For example, the GCD of 16 and 28 is 4. In [Chapter 2](#), when we investigate how to implement rational-number arithmetic, we will need to be able to compute GCDS in order to reduce rational numbers to lowest terms. (To reduce a rational number to lowest terms, we must divide both the numerator and the denominator by their GCD. For example, 16/28 reduces to 4/7.) One way to find the GCD of two integers is to factor them and search for common factors, but there is a famous algorithm that is much more efficient.

2つの整数  $a$  と  $b$  の Greatest Common divisor(GCD: 最大公約数) とは  $a$  と  $b$  の両者を余り無しで割り切れる最大の整数だと定義されます。例えば 16 と 28 の GCD は 4 です。[Chapter 2](#)で分数の計算の実装方法について調査する時に分数を約分するために GCD を求められるようになる必要が出てきます。(分数を約分するためには分母と分子をそれらの GCD で割らねばなりません。例えば 16/28 は 4/7 になります) 2つの整数の GCD を求める 1つの方法はそれらを因数分解し、共通因数を求める方法です。しかしそれより効率的な有名なアルゴリズムが存在します。

The idea of the algorithm is based on the observation that, if  $r$  is the remainder when  $a$  is divided by  $b$ , then the common divisors of  $a$  and  $b$  are precisely the same as the common divisors of  $b$  and  $r$ . Thus, we can use the equation

そのアルゴリズムの考えはもし  $r$  が  $a$  を  $b$  で割った時の余りである場合に  $a$  と  $b$  の共通因数は正確に  $b$  と  $r$  の共通な因数であるという結果を基にしています。従って次の等式を利用可能です。

$$\text{GCD}(a, b) = \text{GCD}(b, r)$$

to successively reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers. For example,

引き続いて GCD を求める問題からより小さな値の整数のペアの GCD を求める問題へと縮小していくことができます。例えば、

$$\begin{aligned}\text{GCD}(206, 40) &= \text{GCD}(40, 6) \\ &= \text{GCD}(6, 4) \\ &= \text{GCD}(4, 2) \\ &= \text{GCD}(2, 0) \\ &= 2\end{aligned}$$

reduces  $\text{GCD}(206, 40)$  to  $\text{GCD}(2, 0)$ , which is 2. It is possible to show that starting with any two positive integers and performing repeated reductions will always eventually produce a pair where the second number is 0. Then the GCD is the other number in the pair. This method for computing the GCD is known as *Euclid's Algorithm*.<sup>83</sup>

上の例では  $\text{GCD}(206, 40)$  を  $\text{GCD}(2, 0)$  へと縮小しています。その答は 2 です。任意の 2 つの正の整数から始めて収縮を繰り返し実行することで常に最終的には 2 つ目の数値が 0 であるペアにすることができます。その時、GCD の値はもう 1 つの値です。この GCD を求める方法は *Euclid's Algorithm*(ユークリッドの互除法)として知られています。<sup>84</sup>

It is easy to express Euclid's Algorithm as a procedure:

ユークリッドの互除法を手続として表すのは簡単です。

```
(define (gcd a b)
```

<sup>83</sup>Euclid's Algorithm is so called because it appears in Euclid's *Elements* (Book 7, ca. 300 B.C.). According to Knuth (1973), it can be considered the oldest known nontrivial algorithm. The ancient Egyptian method of multiplication (Exercise 1.18) is surely older, but, as Knuth explains, Euclid's algorithm is the oldest known to have been presented as a general algorithm, rather than as a set of illustrative examples.

<sup>84</sup>ユークリッドの互除法はユークリッドの *Elements*(およそ紀元前 300 年の原論第 7 卷)に載っていたためにそう呼ばれます。Knuth (1973)によると最も古く良く知られた重要なアルゴリズムであると考えられるそうです。(Exercise 1.18) の古代のエジプト人の乗算方法は確かにこれよりも古いのですが、Knuth の説明ではユークリッドのアルゴリズムは最も古く知られた一般的なアルゴリズムとして紹介されたものであり、説明的な例の集合では無いとのことです

```
(if (= b 0)
  a
  (gcd b (remainder a b))))
```

This generates an iterative process, whose number of steps grows as the logarithm of the numbers involved.

これが反復プロセスを生成し、そのステップ数は与えられた数値の対数で増加します。

The fact that the number of steps required by Euclid's Algorithm has logarithmic growth bears an interesting relation to the Fibonacci numbers:

ユーリッドの互除法により必要とされるステップ数が対数増加する事実がフィボナッチ数に対する興味深い関係を持ちます。

**Lamé's Theorem:** If Euclid's Algorithm requires  $k$  steps to compute the GCD of some pair, then the smaller number in the pair must be greater than or equal to the  $k^{\text{th}}$  Fibonacci number.<sup>85</sup>

**Lamé の定理:** もしユーリッドの互除法があるペアの GCD を求めるのに  $k$  ステップを必要とする場合、必ずペアの小さな値が  $k$  番目のフィボナッチ数より大きいか等しい。<sup>86</sup>

---

<sup>85</sup>This theorem was proved in 1845 by Gabriel Lamé, a French mathematician and engineer known chiefly for his contributions to mathematical physics. To prove the theorem, we consider pairs  $(a_k, b_k)$ , where  $a_k \geq b_k$ , for which Euclid's Algorithm terminates in  $k$  steps. The proof is based on the claim that, if  $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$  are three successive pairs in the reduction process, then we must have  $b_{k+1} \geq b_k + b_{k-1}$ . To verify the claim, consider that a reduction step is defined by applying the transformation  $a_{k-1} = b_k, b_{k-1} = \text{remainder of } a_k \text{ divided by } b_k$ . The second equation means that  $a_k = qb_k + b_{k-1}$  for some positive integer  $q$ . And since  $q$  must be at least 1 we have  $a_k = qb_k + b_{k-1} \geq b_k + b_{k-1}$ . But in the previous reduction step we have  $b_{k+1} = a_k$ . Therefore,  $b_{k+1} = a_k \geq b_k + b_{k-1}$ . This verifies the claim. Now we can prove the theorem by induction on  $k$ , the number of steps that the algorithm requires to terminate. The result is true for  $k = 1$ , since this merely requires that  $b$  be at least as large as  $\text{Fib}(1) = 1$ . Now, assume that the result is true for all integers less than or equal to  $k$  and establish the result for  $k + 1$ . Let  $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$  be successive pairs in the reduction process. By our induction hypotheses, we have  $b_{k-1} \geq \text{Fib}(k - 1)$  and  $b_k \geq \text{Fib}(k)$ . Thus, applying the claim we just proved together with the definition of the Fibonacci numbers gives  $b_{k+1} \geq b_k + b_{k-1} \geq \text{Fib}(k) + \text{Fib}(k - 1) = \text{Fib}(k + 1)$ , which completes the proof of Lamé's Theorem.

<sup>86</sup>この定理は 1845 年にフランスの数学者でありかつエンジニアでもある Gabriel

We can use this theorem to get an order-of-growth estimate for Euclid's Algorithm. Let  $n$  be the smaller of the two inputs to the procedure. If the process takes  $k$  steps, then we must have  $n \geq \text{Fib}(k) \approx \varphi^k / \sqrt{5}$ . Therefore the number of steps  $k$  grows as the logarithm (to the base  $\varphi$ ) of  $n$ . Hence, the order of growth is  $\Theta(\log n)$ .

この定理を用いてユークリッドの互除法の増加のオーダーを推測することができます。 $n$  が手続の入力値の小さな値だとします。もしプロセスが  $k$  ステップ必要とする場合、 $n \geq \text{Fib}(k) \approx \varphi^k / \sqrt{5}$  が必ず成り立ちます。従ってステップ数  $k$  は  $n$  の ( $\varphi$  を底とする) 対数で増加します。つまり増加のオーダーは  $\Theta(\log n)$  となります。

**Exercise 1.20:** The process that a procedure generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative `gcd` procedure given above. Suppose we were to interpret this procedure using normal-order evaluation, as discussed in Section 1.1.5. (The normal-order-evaluation rule for `if` is described in Exercise 1.5.) Using the substitution method (for normal order), illustrate the process generated in evaluating `(gcd 206 40)` and indicate the `remainder` operations that are actually performed. How many `remainder` operations are actually performed in

---

Laméにより証明されました。彼は数理物理学への貢献の第一人者としても有名です。この定理を証明するには  $a_k \geq b_k$  であるペア  $(a_k, b_k)$  がユークリッドの互除法にて  $k$  ステップで停止するか考えます。証明は  $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$  が縮小プロセスにおける連続する 3 つのペアである場合、必ず  $b_{k+1} \geq b_k + b_{k-1}$  であることを基にします。この仮定を確認するために縮小ステップの変換定義が  $a_{k-1} = b_k, b_{k-1} = "a_k$  を  $b_k$  で割った余り" であることについて考えます。2 つ目の等式はある整数  $q$  に対し  $a_k = qb_k + b_{k-1}$  が成り立つことを意味します。 $q$  は少なくとも 1 ですから  $a_k = qb_k + b_{k-1} \geq b_k + b_{k-1}$  が成り立ちます。しかし以前の収縮ステップより  $b_{k+1} = a_k$  です。従って  $b_{k+1} = a_k \geq b_k + b_{k-1}$  が成り立ちます。これで先程の仮定は立証できました。アルゴリズムが停止するのに必要なステップ数を  $k$  とした場合に、これで定理は  $k$  を用いた数学的帰納法にて証明可能となりました。 $k = 1$  の時、これは単に  $b$  が少なくとも  $\text{Fib}(1) = 1$  と同じ大きさであることを必要としますので真です。次に  $k$  に等しいかより小さい整数全てにおいて定理が真であると仮定します。そして  $k+1$  でも成立することを証明します。 $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$  が縮小プロセスにおける連続するペアである場合に、数学的帰納法の仮定より、 $b_{k-1} \geq \text{Fib}(k-1)$  と  $b_k \geq \text{Fib}(k)$  が成り立ちます。ここで先程フィボナッチ数の定義と共に証明した式を適用すると  $b_{k+1} \geq b_k + b_{k-1} \geq \text{Fib}(k) + \text{Fib}(k-1) = \text{Fib}(k+1)$  が導出されます。これで定理の証明は終了です。

the normal-order evaluation of `(gcd 206 40)`? In the applicative-order evaluation?

ある手続が生成するプロセスはもちろんインタプリタにより使用されるルールに依存する。例として上で説明した反復 `gcd` 手続について考える。この手続を [Section 1.1.5](#) で議論した正規順評価で解釈、実行したと想定する。`(if)` に対する正規順評価ルールは [Exercise 1.5](#) を参照)。置換法を(正規順に)用いて `(gcd 206 40)` の評価により生成されるプロセスを説明せよ。次に実際に実行された `remainder` 命令を示せ。`(gcd 206 40)` の正規順評価において実際に実行された `remainder` 命令は何回だろうか? 適用順評価では?

### 1.2.6 Example: Testing for Primality

This section describes two methods for checking the primality of an integer  $n$ , one with order of growth  $\Theta(\sqrt{n})$ , and a “probabilistic” algorithm with order of growth  $\Theta(\log n)$ . The exercises at the end of this section suggest programming projects based on these algorithms.

この節では整数  $n$  が素数であるかをテストする 2 つの方法について述べます。1 つは増加のオーダーが  $\Theta(\sqrt{n})$  であり、他は“確率的”なアルゴリズムで増加のオーダーが  $\Theta(\log n)$  です。この節の最後の課題ではこれらのアルゴリズムに基づいたプログラミングのプロジェクトを提案します。

#### Searching for divisors

Since ancient times, mathematicians have been fascinated by problems concerning prime numbers, and many people have worked on the problem of determining ways to test if numbers are prime. One way to test if a number is prime is to find the number’s divisors. The following program finds the smallest integral divisor (greater than 1) of a given number  $n$ . It does this in a straightforward way, by testing  $n$  for divisibility by successive integers starting with 2.

古代の時代から数学者は素数についての問題に魅惑されてきました。多くの人々が数値が素数であるかの決定法の問題に取り組んできました。数値が素数であるかのテストの 1 つの方法は数値の約数を求めることです。次のプログラムは 1 より大きな最も小さい整因子(約数)を与えられた  $n$  に対して求めま

す。このプログラムはそれを直接的な方法、つまり 2 で始まる一連の整数により割り切れるかどうかをテストすることにより行います。

```
(define (smallest-divisor n) (find-divisor n 2))
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))
(define (divides? a b) (= (remainder b a) 0))
```

We can test whether a number is prime as follows:  $n$  is prime if and only if  $n$  is its own smallest divisor.

数値が素数であるか以下のようにテストします:  $n$  は  $n$  自身が最小の約数である場合、かつその場合に限り素数である。

```
(define (prime? n)
  (= n (smallest-divisor n)))
```

The end test for `find-divisor` is based on the fact that if  $n$  is not prime it must have a divisor less than or equal to  $\sqrt{n}$ .<sup>87</sup> This means that the algorithm need only test divisors between 1 and  $\sqrt{n}$ . Consequently, the number of steps required to identify  $n$  as prime will have order of growth  $\Theta(\sqrt{n})$ .

`find-divisor` の終了条件はもし  $n$  が素数でないならば  $\sqrt{n}$  より小さいかまたは等しい約数を持つという事実に基づいています。<sup>88</sup> これはこのアルゴリズムが 1 から  $\sqrt{n}$  までの約数についてのみテストすれば良いことを示します。結果として、 $n$  が素数であるかを判定するのに必要なステップ数の増加のオーダーは  $\Theta(\sqrt{n})$  となります。

## The Fermat test

The  $\Theta(\log n)$  primality test is based on a result from number theory known as Fermat's Little Theorem.<sup>89</sup>

---

<sup>87</sup>If  $d$  is a divisor of  $n$ , then so is  $n/d$ . But  $d$  and  $n/d$  cannot both be greater than  $\sqrt{n}$ .

<sup>88</sup>もし  $d$  が  $n$  の約数である時、 $n/d$  もまた約数です。しかし  $d$  と  $n/d$  の両者が共に  $\sqrt{n}$  より大きいことは有りません

<sup>89</sup>Pierre de Fermat (1601-1665) is considered to be the founder of modern number theory. He obtained many important number-theoretic results, but he usually announced just the results, without providing his proofs. Fermat's Little Theorem was

$\Theta(\log n)$  の素数判定はフェルマーの小定理として知られる数論の結果に基づきます。<sup>90</sup>

**Fermat's Little Theorem:** If  $n$  is a prime number and  $a$  is any positive integer less than  $n$ , then  $a$  raised to the  $n^{\text{th}}$  power is congruent to  $a$  modulo  $n$ .

フェルマーの小定理:  $n$  が素数かつ  $a$  が  $n$  より小さい任意の正の整数である時、 $a$  の  $n$  乗は法  $n$  に関して  $a$  と合同である。

(Two numbers are said to be *congruent modulo n* if they both have the same remainder when divided by  $n$ . The remainder of a number  $a$  when divided by  $n$  is also referred to as the *remainder of a modulo n*, or simply as  $a \bmod n$ .)

(2つの数値はその両方が  $n$  で割った時に同じ余りを持つ場合、*congruent modulo n* (法  $n$  に関して合同) と呼ばれます。また  $a$  を  $n$  で割った時の余りは  $a \bmod n$  の*remainder*(剩余)、または単純に  $a \bmod n$  と呼ばれます。)

If  $n$  is not prime, then, in general, most of the numbers  $a < n$  will not satisfy the above relation. This leads to the following algorithm for testing primality: Given a number  $n$ , pick a random number  $a < n$  and compute the remainder of  $a^n \bmod n$ . If the result is not equal to  $a$ , then  $n$  is certainly not prime. If it is  $a$ , then chances are good that  $n$  is prime. Now pick another random number

---

stated in a letter he wrote in 1640. The first published proof was given by Euler in 1736 (and an earlier, identical proof was discovered in the unpublished manuscripts of Leibniz). The most famous of Fermat's results—known as Fermat's Last Theorem—was jotted down in 1637 in his copy of the book *Arithmetic* (by the third-century Greek mathematician Diophantus) with the remark “I have discovered a truly remarkable proof, but this margin is too small to contain it.” Finding a proof of Fermat's Last Theorem became one of the most famous challenges in number theory. A complete solution was finally given in 1995 by Andrew Wiles of Princeton University.

<sup>90</sup>Pierre de Fermat (1601-1665) は現在の整数論の創始者と考えられています。彼は多くの重要な数論上の事実について発見しました。しかし彼は通常その結果のみを公表し、証明を与えませんでした。フェルマーの小定理は彼が 1640 年に書いた手紙に記録されています。最初に出版された証明は 1736 年にオイラーにより与えられました。(それより早く、同様の証明がライブニッツの出版されなかった原稿に見つかっています)。最も有名なフェルマーの数式は—フェルマーの最終定理として知られ—1637 年に彼の所有した書籍(3 世紀のギリシャ人数学者 Diophantus による)*Arithmetic* に“私は真に驚くべき証明を発見したが、書き残すにはこの余白は狭すぎる”という所感と共にメモされた物です。フェルマーの最終定理の証明を見つけることは数論において最も有名な挑戦の 1 つとなりました。完全な解はついに 1995 年にプリンストン大学の Andrew Wiles により与えされました。

$a$  and test it with the same method. If it also satisfies the equation, then we can be even more confident that  $n$  is prime. By trying more and more values of  $a$ , we can increase our confidence in the result. This algorithm is known as the Fermat test.

もし  $n$  が素数でなければ一般に  $a < n$  の多くの値は上記の関係を満しません。これが次の素数判定のアルゴリズムへと導きます: ある値  $n$  が与えられた時、 $a < n$  となる乱数を取り  $a^n \bmod n$  の剰余を求めます。もし結果が  $a$  に等しくない時、 $n$  は確実に素数ではありません。もし  $a$  に等しいならば  $n$  が素数である確率は良いと言えます。ここで別の乱数  $a$  を取り同じ方法でテストを行います。それもまた等式を満すのであれば  $n$  が素数である確率はより確からしくなります。より多くの  $a$  について試験を行えば、結果の確からしさを増すことが可能です。このアルゴリズムはフェルマーテストとして知られています。

To implement the Fermat test, we need a procedure that computes the exponential of a number modulo another number:

フェルマーテストを実装するには(ある数値の指数関数  $\bmod$  別の数値)を求める手続が必要です。

```
(define (expmod base exp m)
  (cond ((= exp 0)
         1)
        ((even? exp)
         (remainder
          (square
           (expmod base (/ exp 2) m))
          m))
        (else
         (remainder
          (* base
             (expmod base (- exp 1) m)))
          m))))
```

This is very similar to the `fast-expt` procedure of Section 1.2.4. It uses successive squaring, so that the number of steps grows logarithmically with the exponent.<sup>91</sup>

---

<sup>91</sup>The reduction steps in the cases where the exponent  $e$  is greater than 1 are based on the fact that, for any integers  $x$ ,  $y$ , and  $m$ , we can find the remainder of  $x$  times  $y$  modulo  $m$  by computing separately the remainders of  $x$  modulo  $m$  and  $y$  modulo  $m$ ,

これはSection 1.2.4の `fast-expt` 手続にとても似ています。連続する二乗を用いるため、ステップ数の増加は”指数”引数の対数になります。<sup>92</sup>

The Fermat test is performed by choosing at random a number  $a$  between 1 and  $n - 1$  inclusive and checking whether the remainder modulo  $n$  of the  $n^{\text{th}}$  power of  $a$  is equal to  $a$ . The random number  $a$  is chosen using the procedure `random`, which we assume is included as a primitive in Scheme. `Random` returns a nonnegative integer less than its integer input. Hence, to obtain a random number between 1 and  $n - 1$ , we call `random` with an input of  $n - 1$  and add 1 to the result:

フェルマーテストは 1 から  $n - 1$  までの乱数  $a$  を選択し、 $a$  の  $n$  乗の modulo  $n$  が  $a$  に等しいかをチェックすることで行います。乱数  $a$  は手続 `random` を用いて選択しますが、それは Scheme のプリミティブな手続に存在する前提です。`random` は入力の整数よりも少ない非負数な整数を返します。そのため 1 から  $n - 1$  の乱数を得るには `random` に  $n - 1$  を入力とし、結果に 1 を足します。

```
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))
```

The following procedure runs the test a given number of times, as specified by a parameter. Its value is true if the test succeeds every time, and false otherwise.

以下の手続はパラメータにより与えられた数値の回数分、テストを実行します。テストが毎回成功すれば `true` を、そうでなければ `false` を返します。

```
(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1))))
```

---

multiplying these, and then taking the remainder of the result modulo  $m$ . For instance, in the case where  $e$  is even, we compute the remainder of  $b^{e/2}$  modulo  $m$ , square this, and take the remainder modulo  $m$ . This technique is useful because it means we can perform our computation without ever having to deal with numbers much larger than  $m$ . (Compare Exercise 1.25.)

<sup>92</sup>指数  $e$  が 1 より大きい場合の縮小ステップは、任意の整数  $x, y, m$  に対し  $x$  modulo  $m$  と  $y$  modulo  $m$  を別々に求め、これらを掛け、その結果の法  $m$  に関する剰余を求ることで ( $x$  と  $y$  の積 modulo  $m$ ) を求めることができるという事実に基づきます。例えば  $e$  が偶数の場合に  $b^{e/2}$  modulo  $m$  を求め、その二乗を取り、法  $m$  に関する剰余を得ます。このテクニックはとても役に立ちます。 $m$  よりもはるかに大きな数値を一切扱う必要無しに演算を行うことが可能だからです。(Exercise 1.25 と比較せよ)

```
(else false)))
```

## Probabilistic methods

The Fermat test differs in character from most familiar algorithms, in which one computes an answer that is guaranteed to be correct. Here, the answer obtained is only probably correct. More precisely, if  $n$  ever fails the Fermat test, we can be certain that  $n$  is not prime. But the fact that  $n$  passes the test, while an extremely strong indication, is still not a guarantee that  $n$  is prime. What we would like to say is that for any number  $n$ , if we perform the test enough times and find that  $n$  always passes the test, then the probability of error in our primality test can be made as small as we like.

フェルマーテストは正確さが保証された多くの親しみのあるアルゴリズムとは性格が異なっています。ここでは得られた結果は確率的にのみ正しいと言えます。より正確には、 $n$  が常にフェルマーテストに失敗するのであれば  $n$  が素数でないことは確実です。しかし  $n$  がテストをパスしたという結果は、とても強い目安ではありますが、 $n$  が素数であることを保証しません。ここで言いたかったのは任意の数値  $n$  に対し、十分な回数のテストを行い  $n$  が常にテストをパスする場合、この素数判定が間違いである可能性は思い通りに小さくすることが可能だということでした。

Unfortunately, this assertion is not quite correct. There do exist numbers that fool the Fermat test: numbers  $n$  that are not prime and yet have the property that  $a^n$  is congruent to  $a$  modulo  $n$  for all integers  $a < n$ . Such numbers are extremely rare, so the Fermat test is quite reliable in practice.<sup>93</sup>

残念ながらこの主張は完全には正しくありません。実はフェルマーテストを騙してしまう数値が存在します。 $a < n$  となる全ての整数において、 $n$  が素数ではなく、しかし、 $a^n$  が  $n$  を法とする  $a$  に合同であるような数値  $n$ 。そのよ

---

<sup>93</sup> Numbers that fool the Fermat test are called *Carmichael numbers*, and little is known about them other than that they are extremely rare. There are 255 Carmichael numbers below 100,000,000. The smallest few are 561, 1105, 1729, 2465, 2821, and 6601. In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a “correct” algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering.

うな数値はとても稀です。そのためフェルマーテストは実際にとても信用が高いと言えます。<sup>94</sup>

There are variations of the Fermat test that cannot be fooled. In these tests, as with the Fermat method, one tests the primality of an integer  $n$  by choosing a random integer  $a < n$  and checking some condition that depends upon  $n$  and  $a$ . (See Exercise 1.28 for an example of such a test.) On the other hand, in contrast to the Fermat test, one can prove that, for any  $n$ , the condition does not hold for most of the integers  $a < n$  unless  $n$  is prime. Thus, if  $n$  passes the test for some random choice of  $a$ , the chances are better than even that  $n$  is prime. If  $n$  passes the test for two random choices of  $a$ , the chances are better than 3 out of 4 that  $n$  is prime. By running the test with more and more randomly chosen values of  $a$  we can make the probability of error as small as we like.

フェルマーテストのバリエーションには騙されない物も複数あります。これらのテストではフェルマーテストと同様に、整数  $n$  の素数判定を乱数  $a < n$  を選択し、 $n$  と  $a$  に依る何らかの条件をチェックします。(そのようなテストの例は Exercise 1.28 を参照して下さい)。一方でフェルマーテストとは対照的に、任意の  $n$  に対し  $n$  が素数でなければ  $a < n$  の多くに対し条件が成立しないことを証明できます。従って  $n$  がいくつかの乱数  $a$  に対してテストが通るのであれば、 $n$  が素数である可能性は五分五分より高くなります。もし  $n$  が 2 つの乱数である  $a$  に対してテストを通過すれば、 $n$  が素数である確率は 4 分の 3 よりも高くなります。テストを何度も乱数  $a$  を選択しながら実行することでエラーの確率を思い通りに小さくすることが可能です。

The existence of tests for which one can prove that the chance of error becomes arbitrarily small has sparked interest in algorithms of this type, which have come to be known as *probabilistic algorithms*. There is a great deal of research activity in this area, and probabilistic algorithms have been fruitfully applied to many fields.<sup>95</sup>

<sup>94</sup> フェルマーテストを騙してしまう数は *Carmichael numbers*(カーマイケル数) と呼ばれとても稀であるということ以外はあまり良くわかっていません。100,000,000 未満には 255 のカーマイケル数が存在します。最小の物からいくつか上げると 561, 1105, 1729, 2465, 2821, 6601 です。任意に選ばれたとても巨大な数値の素数性をテストする場合にフェルマーテストを騙す数値に当る確率はコンピュータが“正確な”アルゴリズムを実行する際に宇宙放射線がエラーを引き起す確率よりも低いです。2 つ目の理由でなく、最初の理由からアルゴリズムを不適切だと考えることは数学とエンジニアリングの間の違いを示しています。

<sup>95</sup> One of the most striking applications of probabilistic prime testing has been to

エラーの確率が自由裁量で小さくできることが証明可能なテストの存在はこのタイプのアルゴリズムへの興味を起こしました。それらは*probabilistic algorithms*（確率的アルゴリズム）と呼ばれます。この領域にはとても多くの研究活動が存在し、確率的アルゴリズムは多くの現場に効果的に適用されてきました。<sup>96</sup>

**Exercise 1.21:** Use the `smallest-divisor` procedure to find the smallest divisor of each of the following numbers: 199, 1999, 19999.

`smallest-divisor` 手続を用いて次の数値の最小の約数を求めよ：  
199、1999、19999

**Exercise 1.22:** Most Lisp implementations include a primitive called `runtime` that returns an integer that specifies the amount of time the system has been running (measured, for example, in microseconds). The following `timed-prime-test` procedure, when called with an integer  $n$ , prints  $n$  and checks to see if  $n$  is prime. If  $n$  is prime, the procedure prints three asterisks followed by the amount of time used in performing the test.

多くの Lisp 実装は `runtime` と呼ばれるプリミティブな手続を持つておりそれはシステムが実行している間の（例えばマイクロ秒で測定された）時間を整数にて返す。次の `timed-prime-test` 手続は

---

the field of cryptography. Although it is now computationally infeasible to factor an arbitrary 200-digit number, the primality of such a number can be checked in a few seconds with the Fermat test. This fact forms the basis of a technique for constructing “unbreakable codes” suggested by Rivest et al. (1977). The resulting RSA algorithm has become a widely used technique for enhancing the security of electronic communications. Because of this and related developments, the study of prime numbers, once considered the epitome of a topic in “pure” mathematics to be studied only for its own sake, now turns out to have important practical applications to cryptography, electronic funds transfer, and information retrieval.

<sup>96</sup>確率的素数判定法の最も特筆すべき適用例は暗号の領域です。現時点では任意の 200 衡の数値を因数分解することは計算能力上不可能ですが、そのような数値の素数判定はフェルマーテストにより数秒で行うことが可能です。この事実がRivest et al. (1977)により提案された“解読不能な符号”を構築するためのテクニックの基を形成しました。その結果としてRSA アルゴリズムは電子通信上のセキュリティを拡張するテクニックとして広く利用されるようになりました。このことと関連する開発により、素数の研究は一時は“純粹”数学のトピックの典型例でありその世界自身のためにのみ研究される物だと考えられてきましたが、現在では暗号、電子資産の転送と情報検索に対する重要な実的な適用例を持つことが判明しました。

整数  $n$  と共に呼んだ時、 $n$  を表示し、 $n$  が素数であるかチェックする。 $n$  が素数であれば手続は 3 つのアスタリスクとテスト実行に掛けた時間を表示する。

```
(define (timed-prime-test n)
  (newline) (display n) (start-prime-test n (runtime)))
(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime (- (runtime) start-time)))
(define (report-prime elapsed-time)
  (display " *** ") (display elapsed-time))
```

Using this procedure, write a procedure `search-for-primes` that checks the primality of consecutive odd integers in a specified range. Use your procedure to find the three smallest primes larger than 1000; larger than 10,000; larger than 100,000; larger than 1,000,000. Note the time needed to test each prime. Since the testing algorithm has order of growth of  $\Theta(\sqrt{n})$ , you should expect that testing for primes around 10,000 should take about  $\sqrt{10}$  times as long as testing for primes around 1000. Do your timing data bear this out? How well do the data for 100,000 and 1,000,000 support the  $\Theta(\sqrt{n})$  prediction? Is your result compatible with the notion that programs on your machine run in time proportional to the number of steps required for the computation?

この手続を用いて指定した範囲の連続した奇数について素数判定を行う手続、`search-for-primes` を書け。その手続を用いて 1000、10,000、100,000 より大きな素数を 3 つ見つけよ。各素数のテストに必要な時間を記録せよ。テストアルゴリズムは 10,000 辺りの素数を判定する時、約  $\sqrt{10}$  倍、1000 辺りの素数をテストするより時間がかかるはずである。あなたの結果はこれに従っているか?  
100,000 や 1,000,000 のデータに対して  $\Theta(\sqrt{n})$  の予想は当たっているか? あなたの結果は演算に必要なステップ数に比例して実行時間が増えるという考えに矛盾していないか?

**Exercise 1.23:** The `smallest-divisor` procedure shown at the start of this section does lots of needless testing: After it checks to see if the number is divisible by 2 there is no point in checking to see if it is divisible by any larger even numbers. This suggests

that the values used for `test-divisor` should not be 2, 3, 4, 5, 6, ..., but rather 2, 3, 5, 7, 9, .... To implement this change, define a procedure `next` that returns 3 if its input is equal to 2 and otherwise returns its input plus 2. Modify the `smallest-divisor` procedure to use `(next test-divisor)` instead of `(+ test-divisor 1)`. With `timed-prime-test` incorporating this modified version of `smallest-divisor`, run the test for each of the 12 primes found in [Exercise 1.22](#). Since this modification halves the number of test steps, you should expect it to run about twice as fast. Is this expectation confirmed? If not, what is the observed ratio of the speeds of the two algorithms, and how do you explain the fact that it is different from 2?

この節の最初で示された `smallest-divisor` 手続は必要の無いテストを数多く行っている：数値が 2 で割ることができるかチェックした後に、それがより大きな偶数にて割り切れるかチェックを行う必要は無い。これにより `test-divisor` の値は 2, 3, 4, 5, 6, ... ではなく、2, 3, 5, 7, 9, ... であるべきだと提案できる。この変更を実装するために、入力が 2 であれば 3 を返し、それ以外では入力に 2 を足した値を返す手続 `next` を定義せよ。`smallest-divisor` 手続を変更し、`(+ test-divisor 1)` の代わりに `(next test-divisor)` を使用せよ。`timed-prime-test` をこの変更したバージョンの `smallest-divisor` を用いて [Exercise 1.22](#)で見つけた 12 の素数に対しテストを行え。この変更はテストステップを半分にするため 2 倍速く実行されることを予測される。この予測が確認できるだろうか？もしそうでなければ 2 つのアルゴリズムのスピードの比率はどのような値が確認できるか？2 と異なる結果をどのように説明するか？

**Exercise 1.24:** Modify the `timed-prime-test` procedure of [Exercise 1.22](#) to use `fast-prime?` (the Fermat method), and test each of the 12 primes you found in that exercise. Since the Fermat test has  $\Theta(\log n)$  growth, how would you expect the time to test primes near 1,000,000 to compare with the time needed to test primes near 1000? Do your data bear this out? Can you explain any discrepancy you find?

[Exercise 1.22](#)の `timed-prime-test` 手続を変更し `fast-prime?`(フェルマー法) を用い、課題で見つけた 12 の素数をそれぞれテストせよ。フェルマーテストは  $\Theta(\log n)$  で増加するが、1000 に近い素数をテストするのに必要な時間と

比べ 1,000,000 付近の素数をテストするのに必要な時間をどれ程と見繕うだろうか? 実際との相違をどのように説明できるか?

**Exercise 1.25:** Alyssa P. Hacker complains that we went to a lot of extra work in writing `expmod`. After all, she says, since we already know how to compute exponentials, we could have simply written Alyssa P. Hacker は `expmod` を書くにあたって多くの余分な仕事を行ったと文句を言った。結局のところ我々は既に指数演算のやり方を知っているのだから単純に以下のように書くことができたはずだと彼女は言った。

```
(define (expmod base exp m)
  (remainder (fast-expt base exp) m))
```

Is she correct? Would this procedure serve as well for our fast prime tester? Explain.

彼女は正しいだろうか? この手続は最初の素数判定と同様にうまく行えるだろうか?

**Exercise 1.26:** Louis Reasoner is having great difficulty doing Exercise 1.24. His `fast-prime?` test seems to run more slowly than his `prime?` test. Louis calls his friend Eva Lu Ator over to help. When they examine Louis's code, they find that he has rewritten the `expmod` procedure to use an explicit multiplication, rather than calling `square`:

Louis Reasoner は Exercise 1.24 を行うのに随分と苦労した。彼の `fast-prime?` テストは彼の `prime?` テストよりも随分遅いようだ。Louis は友達の Eva Lu Ator を呼んで助けを求めた。彼らが Louis のコードを試してみると、Louis が `expmod` 手続を `square` 手続を呼ぶのではなく、明示的に乗算を用いていることを見つけた。

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (* (expmod base (/ exp 2) m)
                      (expmod base (/ exp 2) m))
                    m))
        (else
```

```
(remainder (* base
              (expmod base (- exp 1) m))
              m))))
```

“I don’t see what difference that could make,” says Louis. “I do.” says Eva. “By writing the procedure like that, you have transformed the  $\Theta(\log n)$  process into a  $\Theta(n)$  process.” Explain.

“これがどんな違いを生んでいるのかわからないよ”と Louis は言った。“私にはわかる”と Eva が言う。“手続をそのように記述することで、 $\Theta(\log n)$  のプロセスを  $\Theta(n)$  のプロセスに変えてしまったの。”説明せよ。

**Exercise 1.27:** Demonstrate that the Carmichael numbers listed in Footnote 1.47 really do fool the Fermat test. That is, write a procedure that takes an integer  $n$  and tests whether  $a^n$  is congruent to  $a$  modulo  $n$  for every  $a < n$ , and try your procedure on the given Carmichael numbers.

Footnote 1.47にて並べられたカーマイケル数が実際にフェルマー テストを騙すか実演せよ。整数  $n$  を取り  $a^n$  が  $a$  と法  $n$  に関して 合同であるか全ての  $a < n$  に対しテストを行う手続を書き、与えられたカーマイケル数に対してその手続を試せ。

**Exercise 1.28:** One variant of the Fermat test that cannot be fooled is called the *Miller-Rabin test* (Miller 1976; Rabin 1980). This starts from an alternate form of Fermat’s Little Theorem, which states that if  $n$  is a prime number and  $a$  is any positive integer less than  $n$ , then  $a$  raised to the  $(n - 1)$ -st power is congruent to 1 modulo  $n$ . To test the primality of a number  $n$  by the Miller-Rabin test, we pick a random number  $a < n$  and raise  $a$  to the  $(n - 1)$ -st power modulo  $n$  using the `expmod` procedure. However, whenever we perform the squaring step in `expmod`, we check to see if we have discovered a “nontrivial square root of 1 modulo  $n$ ,” that is, a number not equal to 1 or  $n - 1$  whose square is equal to 1 modulo  $n$ . It is possible to prove that if such a nontrivial square root of 1 exists, then  $n$  is not prime. It is also possible to prove that if  $n$  is an odd number that is not prime, then, for at least half the numbers  $a < n$ , computing  $a^{n-1}$  in this way will reveal a nontrivial square root of 1 modulo  $n$ . (This is why the Miller-Rabin

test cannot be fooled.) Modify the `expmod` procedure to signal if it discovers a nontrivial square root of 1, and use this to implement the Miller-Rabin test with a procedure analogous to `fermat-test`. Check your procedure by testing various known primes and non-primes. Hint: One convenient way to make `expmod` signal is to have it return 0.

騙されないフェルマーテストの 1 つの変形として *Miller-Rabin test* (Miller 1976; Rabin 1980) がある。これはフェルマーの小定理の代替形から始めるが、それは  $n$  が素数でかつ  $a$  が  $n$  以下の任意の正の整数である時、 $a$  の  $(n - 1)$  乗は法  $n$  に関して 1 と合同であると定める。Miller-Rabin テストで数値  $n$  の素数判定を行うには乱数  $a < n$  を選択し、 $a$  の  $(n - 1)$  乗の  $n$  を法とする剰余を `expmod` 手続を用いて求める。しかし、`expmod` 手続中で二乗するステップにおいて毎回、“自明でない法  $n$  に関する 1 の平方根”を見つけたかチェックを行う。これは 1 または  $n - 1$  に等しくない数値でかつ、法  $n$  に関して二乗した値の剰余が 1 に等しい数値である。そのような自明でない 1 の平方根が存在すれば  $n$  が素数ではないことが証明可能である。またもし  $n$  が素数でない奇数である時、少なくとも  $a < n$  の半分においてこの方法で  $a^{n-1}$  を演算すると自明でない法  $n$  に関する 1 の平方根が現れることが証明可能である。(これがなぜ Miller-Rabin テストが騙されないかである)。`expmod` 手続を変更し自明でない 1 の平方根を見つけた時合図を送るようにし、それを用いて `fermat-test` に似た Miller-Rabin テストを実装せよ。既知の素数、非素数を用いてあなたの手続をチェックせよ。ヒント：`expmod` に合図を送らせる簡単な方法は 0 を返させることである。

## 1.3 Formulating Abstractions with Higher-Order Procedures

We have seen that procedures are, in effect, abstractions that describe compound operations on numbers independent of the particular numbers. For example, when we

私達はここまで手続が事実上、特定の値から独立した数値への複合命令を記述する抽象化であることを見てきました。例えば、

```
(define (cube x) (* x x x))
```

we are not talking about the cube of a particular number, but rather about a method for obtaining the cube of any number. Of course we could get along without ever defining this procedure, by always writing expressions such as

これは特定の値の立方について述べているのではなく、任意の数値の立方を得るための手法について述べている訳です。もちろんこの手続を定義することなく常に以下のような式を書くことでやっていくことも可能です。

```
(* 3 3 3)  
(* x x x)  
(* y y y)
```

and never mentioning `cube` explicitly. This would place us at a serious disadvantage, forcing us to work always at the level of the particular operations that happen to be primitives in the language (multiplication, in this case) rather than in terms of higher-level operations. Our programs would be able to compute cubes, but our language would lack the ability to express the concept of cubing. One of the things we should demand from a powerful programming language is the ability to build abstractions by assigning names to common patterns and then to work in terms of the abstractions directly. Procedures provide this ability. This is why all but the most primitive programming languages include mechanisms for defining procedures.

そして明示的に `cube` について触れないことも可能でしょう。しかしこれはとても大きな不便を与えます。高レベルな命令の用語ではなく常に言語内にプリミティブとして偶然存在するレベルの特定の命令レベルにて働くざるを得なくなります(このケースでは乗算です)。私達のプログラムは立方を計算可能ですが、私達の言語は立方のコンセプトを表現する能力が欠けているかもしれません。私達が強力なプログラミング言語から望むべき物の1つは共通のパターンに対し名前を付けることで抽象を構築し、その後抽象化的用語にて直接働く能力です。手続はこの能力を与えます。これがなぜ原始的な物を除いた全てのプログラミング言語にて手続を定義するメカニズムが含まれているかの理由です。

Yet even in numerical processing we will be severely limited in our ability to create abstractions if we are restricted to procedures whose parameters must

be numbers. Often the same programming pattern will be used with a number of different procedures. To express such patterns as concepts, we will need to construct procedures that can accept procedures as arguments or return procedures as values. Procedures that manipulate procedures are called *higher-order procedures*. This section shows how higher-order procedures can serve as powerful abstraction mechanisms, vastly increasing the expressive power of our language.

それにも関わらず数値演算ですら、もし手続のパラメータが数値のみであると制約されていれば抽象化を行うには我々の能力は非常に大きく制限されていると言えるでしょう。しばしば同じプログラムのパターンがいくつもの異なる手続にて使用されます。そのようなパターンを表現するには引数として手続を受け入れができるか、手続を値として返すような手続を構築する必要が出てきます。手続を操作する手続は*higher-order procedures*(高階手続)と呼ばれます。この節では高階手続がどのように強力な抽象化メカニズムを果たし、言語の表現力を幅広く増大するかを示します。

### 1.3.1 引数としての手続

Consider the following three procedures. The first computes the sum of the integers from `a` through `b`:

次の3つの手続について考えてみて下さい。1つ目は `a` から `b` の整数の合計を計算します。

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

The second computes the sum of the cubes of the integers in the given range:

2つ目は与えられた範囲の整数の立方の合計を計算します。

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a)
          (sum-cubes (+ a 1) b))))
```

The third computes the sum of a sequence of terms in the series

3つ目は以下の級数の一連の項の合計を計算します。

$$\frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \dots,$$

which converges to  $\pi/8$  (very slowly):<sup>97</sup>

これは  $\pi/8$  に (とてもゆっくりと) 収束します。<sup>98</sup>

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2)))
          (pi-sum (+ a 4) b))))
```

These three procedures clearly share a common underlying pattern. They are for the most part identical, differing only in the name of the procedure, the function of `a` used to compute the term to be added, and the function that provides the next value of `a`. We could generate each of the procedures by filling in slots in the same template:

これらの3つの手続は明確に共通な基礎をなすパターンを共有しています。それらはほとんどの部分が同一で、手続の名前、和を求める項を `a` を用いて演算する関数、`a` の次の値を与える関数のみが異なります。各手続を同じテンプレートを用いて枠を埋めることで生成することができそうです。

```
(define (<name>) a b)
  (if (> a b)
      0
      (+ (<term> a)
          (<name> (<next> a) b))))
```

The presence of such a common pattern is strong evidence that there is a useful abstraction waiting to be brought to the surface. Indeed, mathematicians long ago identified the abstraction of *summation of a series* and invented “sigma notation,” for example

---

<sup>97</sup>This series, usually written in the equivalent form  $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$ , is due to Leibniz. We'll see how to use this as the basis for some fancy numerical tricks in Section 3.5.3.

<sup>98</sup>この級数は一般に等価である形式  $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$  にて記述される Leibniz による物です。私達はこれがある高級な数学上のトリックに使われるのを Section 3.5.3 で見ることになります。

このような共通パターンの存在は便利な抽象化が表に浮かび上がるのを待っていることを示す強力な証拠です。実際に数学者は大昔に*summation of a series*(級数の和)の抽象化を特定し“シグマ記法”を開発しました。つまり、

$$\sum_{n=a}^b f(n) = f(a) + \dots + f(b),$$

to express this concept. The power of sigma notation is that it allows mathematicians to deal with the concept of summation itself rather than only with particular sums—for example, to formulate general results about sums that are independent of the particular series being summed.

このように表現します。シグマ記法の力は数学者に特定の合計のみについてではなく、総和のコンセプト自身について取り扱うことを可能にしました。例えば特定の級数の和を求めるところから独立して一般的な総和についての結果を形式化することを可能としたのです。

Similarly, as program designers, we would like our language to be powerful enough so that we can write a procedure that expresses the concept of summation itself rather than only procedures that compute particular sums. We can do so readily in our procedural language by taking the common template shown above and transforming the “slots” into formal parameters:

同様に、プログラムの設計者である私達は言語に、特定の総和を求める手続のみでなく、総和自身のコンセプトを表現する手続を書くことができるのに十分に強力になって欲しいと願うでしょう。そうすることが直ぐに私達の手続言語にて上記にて示された共通テンプレートを用いて、“枠”を形式パラメータに変換することで可能です。

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
          (sum term (next a) next b))))
```

Notice that `sum` takes as its arguments the lower and upper bounds `a` and `b` together with the procedures `term` and `next`. We can use `sum` just as we would any procedure. For example, we can use it (along with a procedure `inc` that increments its argument by 1) to define `sum-cubes`:

`sum` が引数として下限と上限の `a` と `b` を手続 `term` と `next` と一緒に取ることに注意して下さい。`sum` はこれから行うように好きな手続を使用することができます

できます。例えはそれを(引数に1を足す手続`inc`と共に)`sum-cubes`の定義に利用可能です。

```
(define (inc n) (+ n 1))
(define (sum-cubes a b)
  (sum cube a inc b))
```

Using this, we can compute the sum of the cubes of the integers from 1 to 10:  
これを用いて整数1から10の立方の和を求めることができます。

```
(sum-cubes 1 10)
3025
```

With the aid of an identity procedure to compute the term, we can define `sum-integers` in terms of `sum`:

termを求めるidentityプロシージャの助けを借りて、`sum`を用いて`sum-integers`の定義ができます。

```
(define (identity x) x)
(define (sum-integers a b)
  (sum identity a inc b))
```

Then we can add up the integers from 1 to 10:

これで1から10までの整数の和を求められます。

```
(sum-integers 1 10)
55
```

We can also define `pi-sum` in the same way:<sup>99</sup>

`pi-sum`も同様に定義可能です。<sup>100</sup>

```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2)))))
```

---

<sup>99</sup>Notice that we have used block structure (Section 1.1.8) to embed the definitions of `pi-next` and `pi-term` within `pi-sum`, since these procedures are unlikely to be useful for any other purpose. We will see how to get rid of them altogether in Section 1.3.2.

<sup>100</sup>ブロック構造(Section 1.1.8)を`pi-sum`の中に`pi-next`と`pi-term`の定義を埋め込むために使用していることに注意して下さい。これらの手続は任意の他の手続に対し有用ではなかろうためです。それらと一緒にどのように取り除くかについてはSection 1.3.2で説明します。

```
(define (pi-next x)
  (+ x 4))
(define pi-term a pi-next b))
```

Using these procedures, we can compute an approximation to  $\pi$ :  
これらの手続を用いて  $\pi$  の近似値を求められます。

```
(* 8 (pi-sum 1 1000))
3.139592655589783
```

Once we have `sum`, we can use it as a building block in formulating further concepts. For instance, the definite integral of a function  $f$  between the limits  $a$  and  $b$  can be approximated numerically using the formula

`sum` を手に入れたことで、それを構築用ブロックとしてより多くのコンセプトの形式化にて利用可能です。例えば関数  $f$  の  $a$  と  $b$  の限度値の間の定積分は以下の式を用いて小さな値  $dx$  に対して数値的に近似可能です。

$$\int_a^b f = \left[ f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \dots \right] dx$$

for small values of  $dx$ . We can express this directly as a procedure:  
これを直接、手続として表現します。

```
(define (integral f a b dx)
  (define (add-dx x)
    (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
     dx))

(integral cube 0 1 0.01)
.24998750000000042
```

```
(integral cube 0 1 0.001)
.24999875000001
```

(The exact value of the integral of `cube` between 0 and 1 is 1/4.)  
(`cube` の 0 から 1 の実際の定積分の値は 1/4 です。)

**Exercise 1.29:** Simpson's Rule is a more accurate method of numerical integration than the method illustrated above. Using Simpson's Rule, the integral of a function  $f$  between  $a$  and  $b$  is approximated as

シンプソンの公式は上記にて示された方法よりもより正確な数値積分の方法である。シンプソンの公式を用いて  $a$  と  $b$  の間の  $f$  の定積分は次のように近似される。

$$\frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \dots + 2y_{n-2} + 4y_{n-1} + y_n),$$

where  $h = (b - a)/n$ , for some even integer  $n$ , and  $y_k = f(a + kh)$ . (Increasing  $n$  increases the accuracy of the approximation.) Define a procedure that takes as arguments  $f$ ,  $a$ ,  $b$ , and  $n$  and returns the value of the integral, computed using Simpson's Rule. Use your procedure to integrate `cube` between 0 and 1 (with  $n = 100$  and  $n = 1000$ ), and compare the results to those of the `integral` procedure shown above.

ここで  $h = (b - a)/n$ 、 $n$  は偶数、 $y_k = f(a + kh)$  である。 $(n$  を増やすことで近似の精度を高めることができる)。 $f$ ,  $a$ ,  $b$ ,  $n$  を引数に取りシンプソンの公式を用いて求めた定積分の値を返す手続を定義せよ。

**Exercise 1.30:** The `sum` procedure above generates a linear recursion. The procedure can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

上の `sum` 手続は線形再帰を生成する。手続は和の計算が線形で行われるよう書き直すことが可能だ。次の定義にて消された表記を埋め、どのように行うのか示せ。

```
(define (sum term a next b)
  (define (iter a result)
    (if (??)
        (??)
        (iter (??) (??))))
  (iter (??) (??)))
```

**Exercise 1.31:**

- a The `sum` procedure is only the simplest of a vast number of similar abstractions that can be captured as higher-order

procedures.<sup>101</sup> Write an analogous procedure called `product` that returns the product of the values of a function at points over a given range. Show how to define `factorial` in terms of `product`. Also use `product` to compute approximations to  $\pi$  using the formula<sup>102</sup>

`sum` 手続は高階手続として捉えられる非常に多くの数の同様な抽象化の最も簡単な物にすぎない。<sup>103</sup> `product` と呼ぶ与えられた範囲の点の関数値の積を返す類似の手続を書け。どのようにして `product` を用いて `factorial` を定義するのか示せ。また `product` を用いて次の式を使用して  $\pi$  の近似値を計算せよ。<sup>104</sup>

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \dots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \dots}.$$

- b If your `product` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

---

<sup>101</sup>The intent of Exercise 1.31 through Exercise 1.33 is to demonstrate the expressive power that is attained by using an appropriate abstraction to consolidate many seemingly disparate operations. However, though accumulation and filtering are elegant ideas, our hands are somewhat tied in using them at this point since we do not yet have data structures to provide suitable means of combination for these abstractions. We will return to these ideas in Section 2.2.3 when we show how to use *sequences* as interfaces for combining filters and accumulators to build even more powerful abstractions. We will see there how these methods really come into their own as a powerful and elegant approach to designing programs.

<sup>102</sup>This formula was discovered by the seventeenth-century English mathematician John Wallis.

<sup>103</sup>Exercise 1.31からExercise 1.33の目的は、多くの一見では異なる操作を統一するために、適切な抽象化を利用することで達成される表現力を実演することです。しかし、集積やフィルタリングは洗練された考えではありますが、この時点ではそれらを使用するのに我々の両手が縛られているようなものです。私達はまだこれらの抽象化のための適切な組み合わせの手段を与えるためのデータ構造を持っていないためです。私達はSection 2.2.3にてこれらの考えに立ち戻り、集積とフィルタを組み合わせるためのインターフェイスとして*sequences* (列) をどのように使うのかを示しさらに一層強力な抽象化を構築します。そこではこれらの手法が実際にどのようにしてプログラムを設計するのに強力で洗練されたアプローチとして役に立つかを学びます。

<sup>104</sup>この式は 17 世紀に英国人數学者 John Wallis により発見されました。

もしあなたの `product` 手続が再帰プロセスを生成するのであれば線形プロセスを生成するものを書け。もし線形プロセスを生成するのであれば再帰プロセスを生成するものを書け。

**Exercise 1.32:**

- a. Show that `sum` and `product` (Exercise 1.31) are both special cases of a still more general notion called `accumulate` that combines a collection of terms, using some general accumulation function:

`sum` と `product` (Exercise 1.31) は両方とも、汎用の集積関数を用いて項の集合を結合する `accumulate`(集積) と呼ばれるより一般的な目的の特別なケースに過ぎない。

```
(accumulate combiner null-value term a next b)
```

`Accumulate` takes as arguments the same term and range specifications as `sum` and `product`, together with a `combiner` procedure (of two arguments) that specifies how the current term is to be combined with the accumulation of the preceding terms and a `null-value` that specifies what base value to use when the terms run out. Write `accumulate` and show how `sum` and `product` can both be defined as simple calls to `accumulate`.

`accumulate` は引数として `sum` と `product` と同じく項と範囲の指定を(2つの引数の) `combiner` 手続と `null-value` を共に得る。`combiner` はどのように現在の項が以前の項の集積と結合されるかを指定し、`null-value` は項が尽きた時に使用する基となる値を指定する。`accumulate` を書き `sum` と `product` の両者がどのように簡単な `accumulate` の呼び出しで定義できるかを示せ。

- b. If your `accumulate` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

あなたの `accumulate` が再帰プロセスを生成するのなら線形プロセスを生成する物を書け。もし線形プロセスを生成するのならば再帰プロセスを生成する物を書け。

**Exercise 1.33:** You can obtain an even more general version of `accumulate` (Exercise 1.32) by introducing the notion of a *filter* on the terms to be combined. That is, combine only those terms derived from values in the range that satisfy a specified condition. The resulting `filtered-accumulate` abstraction takes the same arguments as `accumulate`, together with an additional predicate of one argument that specifies the filter. Write `filtered-accumulate` as a procedure. Show how to express the following using `filtered-accumulate`:

より汎用的なバージョンの `accumulate` (Exercise 1.32) を結合される項の *filter*(フィルタ) の概念を紹介することで得ることが可能だ。指定された条件を満たす範囲の値から導かれる項のみを連結する。結果としての `filtered-accumulate` 抽象は `accumulate` と同じ引数を追加の 1 引数の述語と共に取り、述語はフィルタを指定する。手続としての `filtered-accumulate` を書け。以下を `filtered-accumulate` を用いてどのように表現するかを示せ。

- a the sum of the squares of the prime numbers in the interval  $a$  to  $b$  (assuming that you have a `prime?` predicate already written)  
 $a$  と  $b$  の区間の素数の二乗の和 (あなたは既に `prime?` 述語を書いていると前提する)
- b the product of all the positive integers less than  $n$  that are relatively prime to  $n$  (i.e., all positive integers  $i < n$  such that  $\text{GCD}(i, n) = 1$ ).  
全ての  $n$  未満の正の整数でかつ  $n$  に対して互いに素 (つまり  $\text{GCD}(i, n) = 1$  となる全ての整数  $i < n$ ) の積

### 1.3.2 Constructing Procedures Using Lambda

In using `sum` as in Section 1.3.1, it seems terribly awkward to have to define trivial procedures such as `pi-term` and `pi-next` just so we can use them as arguments to our higher-order procedure. Rather than define `pi-next` and `pi-term`, it would be more convenient to have a way to directly specify “the procedure that returns its input incremented by 4” and “the procedure that

returns the reciprocal of its input times its input plus 2.” We can do this by introducing the special form `lambda`, which creates procedures. Using `lambda` we can describe what we want as

Section 1.3.1に示すよう `sum` を使用する時、`pi-term` や `pi-next` のような自明な手続を高階手続にて引数として使うためだけに定義せねばならないのはひどく不恰好に見えます。`pi-next` や `pi-term` を定義する代わりに、“入力値に 4 を足す手続”や“入力値と入力値に 2 を足した数の積の逆数を返す手続”を直接指定する方法を持つほうがより便利になるでしょう。これは手続を作成する特殊形式 `lambda` を紹介することで可能です。`lambda` を用いることで先程行いたかったことを以下のように記述できます。

```
(lambda (x) (+ x 4))
```

そして

```
(lambda (x) (/ 1.0 (* x (+ x 2))))
```

Then our `pi-sum` procedure can be expressed without defining any auxiliary procedures as

次に `pi-sum` 手続は補助的な手続を定義すること無しに表現が可能となります。

```
(define (pi-sum a b)
  (sum (lambda (x) (/ 1.0 (* x (+ x 2))))
    a
    (lambda (x) (+ x 4))
    b))
```

Again using `lambda`, we can write the `integral` procedure without having to define the auxiliary procedure `add-dx`:

同様に `lambda` を使用して、`integral` 手続を補助的な手続 `add-dx` を定義することなく書くことが可能です。

```
(define (integral f a b dx)
  (* (sum f
    (+ a (/ dx 2.0))
    (lambda (x) (+ x dx))
    b)
  dx))
```

In general, `lambda` is used to create procedures in the same way as `define`, except that no name is specified for the procedure:

一般的に、`lambda` は `define` と同様に手続を作成しますが、手続に対して名前が指定されないことが異なります。

```
(lambda (<formal-parameters>) <body>)
```

The resulting procedure is just as much a procedure as one that is created using `define`. The only difference is that it has not been associated with any name in the environment. In fact,

結果としての手続は `define` を用いて作成した手続と同じです。ただ 1 つの違いはそれが環境においてどのような名前にも結び付けられていないことです。

```
(define (plus4 x) (+ x 4))
```

is equivalent to

上記は以下と等価です。

```
(define plus4 (lambda (x) (+ x 4)))
```

We can read a `lambda` expression as follows:

`lambda` 式は以下のように読むことができます。

(lambda   手続は	(x)            (+        x        4)                              引数 x を持ち 足す x と 4
---------------------	---

Like any expression that has a procedure as its value, a `lambda` expression can be used as the operator in a combination such as

任意の値として手続を持つ式と同様に、`lambda` 式は複合式においてオペレータとして使用することができます。例えば、

```
((lambda (x y z) (+ x y (square z)))  
 1 2 3)  
12
```

or, more generally, in any context where we would normally use a procedure name.<sup>105</sup>

---

<sup>105</sup>It would be clearer and less intimidating to people learning Lisp if a name more obvious than `lambda`, such as `make-procedure`, were used. But the convention is firmly

またはより一般的に、私達が通常手続の名前を使用する任意の文脈において使用可能です。<sup>106</sup>

## Using `let` to create local variables

Another use of `lambda` is in creating local variables. We often need local variables in our procedures other than those that have been bound as formal parameters. For example, suppose we wish to compute the function

別の `lambda` 使用法にはローカル変数の作成があります。形式的パラメータに束縛されていないローカル変数を手続で必要とする場合は良くあります。例えば以下の関数を演算したいとします。

$$f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y),$$

which we could also express as これは以下のようにも表現できます。

$$\begin{aligned} a &= 1 + xy, \\ b &= 1 - y, \\ f(x, y) &= xa^2 + yb + ab. \end{aligned}$$

In writing a procedure to compute  $f$ , we would like to include as local variables not only  $x$  and  $y$  but also the names of intermediate quantities like  $a$  and  $b$ . One way to accomplish this is to use an auxiliary procedure to bind the local variables:

$f$  を求める手続を書く場合、 $x$  と  $y$  のみでなく中間値の名前として  $a$  や  $b$  をローカル変数として含みたくなるでしょう。これを実現する 1 つの方法として補助的な手続をローカル変数を束縛するため使用することができます。

---

entrenched. The notation is adopted from the  $\lambda$ -calculus, a mathematical formalism introduced by the mathematical logician Alonzo Church (1941). Church developed the  $\lambda$ -calculus to provide a rigorous foundation for studying the notions of function and function application. The  $\lambda$ -calculus has become a basic tool for mathematical investigations of the semantics of programming languages.

<sup>106</sup>Lisp を学ぶ人にとっては `lambda` という名前よりは `make-procedure` の様な名前を使用したほうがより判りやすいか、または恐しく思われたりはしないでしょう。しかしこの慣習はしっかりと根付いた物です。この表記は  $\lambda$ -calculus(ラムダ計算) という数理論理学者 Alonzo Church (1941)により発表された数学上の形式主義の名から受け入れられています。Church は  $\lambda$  演算を関数と関数適用の概念を学ぶための厳格な基礎として与えるために開発しました。 $\lambda$  演算はプログラミング言語の意味の数学上の研究のための基礎的なツールとなりました。

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (f-helper (+ 1 (* x y))
            (- 1 y)))
```

Of course, we could use a **lambda** expression to specify an anonymous procedure for binding our local variables. The body of **f** then becomes a single call to that procedure:

もちろん、**lambda** 式を用いて無名手続をローカル変数の束縛のため指定することも可能です。**f** のボディはすると手続への单一の呼び出しになります。

```
(define (f x y)
  ((lambda (a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
   (+ 1 (* x y))
   (- 1 y)))
```

This construct is so useful that there is a special form called **let** to make its use more convenient. Using **let**, the **f** procedure could be written as

この構成はとても便利で **let** と呼ばれる特殊形式がその使用をより便利にするために用意されています。**let** を用いることで手続 **f** は以下のようになります。

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x (square a))
       (* y b)
       (* a b))))
```

The general form of a **let** expression is  
**let** 式の一般的な形式は次のとおりです。

```
(let ((<var1> <exp1>)
```

```
((⟨var2⟩ ⟨exp2⟩)
 ...
(⟨varn⟩ ⟨expn⟩))
⟨body⟩)
```

which can be thought of as saying

これは以下のように考えることが可能です。

```
let ⟨var1⟩ have the value ⟨exp1⟩ and
    ⟨var2⟩ have the value ⟨exp2⟩ and
    ...
    ⟨varn⟩ have the value ⟨expn⟩
in ⟨body⟩
```

(let は使役の意ですので、⟨body⟩ の中では ⟨var<sub>1</sub>⟩ は ⟨exp<sub>1</sub>⟩ の値を持たせる、以下繰り返しと読みます。)

The first part of the let expression is a list of name-expression pairs. When the let is evaluated, each name is associated with the value of the corresponding expression. The body of the let is evaluated with these names bound as local variables. The way this happens is that the let expression is interpreted as an alternate syntax for

let 式の最初の部分は名前と式のペアのリストです。let が評価される時、各名前は関連する式の値と関連付けされます。let のボディはこれらのローカルな値に束縛された名前と共に評価されます。let 式は以下の代替文法として評価されるためこれが起こります。

```
((lambda ⟨var1⟩ ... ⟨varn⟩
    ⟨body⟩)
  ⟨exp1⟩
  ...
  ⟨expn⟩)
```

No new mechanism is required in the interpreter in order to provide local variables. A let expression is simply syntactic sugar for the underlying lambda application.

インタプリタ内にはローカル変数を提供するために新しいメカニズムが必要とされません。let 式は中で行われる lambda 適用に対する構文糖でしかありません。

We can see from this equivalence that the scope of a variable specified by a `let` expression is the body of the `let`. This implies that:

この等価式から `let` 式にて指定された変数のスコープが `let` のボディであることがわかります。これが以下のことを暗示します。

- Let allows one to bind variables as locally as possible to where they are to be used. For example, if the value of `x` is 5, the value of the expression `let` は変数を可能な限り使用される場所に局地的に束縛します。例えばもし `x` の値が 5 である時、次の式の値は

```
(+ (let ((x 3))
      (+ x (* x 10)))
     x)
```

is 38. Here, the `x` in the body of the `let` is 3, so the value of the `let` expression is 33. On the other hand, the `x` that is the second argument to the outermost `+` is still 5.

38 です。ここで `let` のボディの中の `x` は 3 ですので `let` 式の値は 33 です。一方で最も外側の `+` の第二引数である `x` は依然 5 です。

- The variables' values are computed outside the `let`. This matters when the expressions that provide the values for the local variables depend upon variables having the same names as the local variables themselves. For example, if the value of `x` is 2, the expression

変数の値は `let` の外側にて計算されます。これはローカル変数の値を提供する式がローカル変数自身と同じ名前を持っている変数に依存する場合に問題となります。例えば、もし `x` の値が 2 の時、次の式では

```
(let ((x 3)
      (y (+ x 2)))
  (* x y))
```

will have the value 12 because, inside the body of the `let`, `x` will be 3 and `y` will be 4 (which is the outer `x` plus 2).

式の値は 12 になります。`let` のボディ内部では `x` は 3、`y` は 4(外側の `x` 足す 2) になるためです。

Sometimes we can use internal definitions to get the same effect as with `let`. For example, we could have defined the procedure `f` above as

時には `let` と同様の効果を得るために内部定義を利用することもあります。例として、上記の手続 `f` を次のように定義することも可能でした。

```
(define (f x y)
  (define a (+ 1 (* x y)))
  (define b (- 1 y))
  (+ (* x (square a))
    (* y b)
    (* a b)))
```

We prefer, however, to use `let` in situations like this and to use internal `define` only for internal procedures.<sup>107</sup>

しかしこのような状況では `let` を使用し、内部定義は内部手続のみのために利用することを好みます。<sup>108</sup>

**Exercise 1.34:** Suppose we define the procedure  
以下の手続を定義したとする。

```
(define (f g) (g 2))
```

Then we have

すると以下の結果を得る。

```
(f square)
4
(f (lambda (z) (* z (+ z 1))))
```

What happens if we (perversely) ask the interpreter to evaluate the combination `(f f)`? Explain.

もし(天邪鬼にも) インタプリタに `(f f)` の組み合わせを評価させたらどのような結果が起こるか? 説明せよ。

---

<sup>107</sup> Understanding internal definitions well enough to be sure a program means what we intend it to mean requires a more elaborate model of the evaluation process than we have presented in this chapter. The subtleties do not arise with internal definitions of procedures, however. We will return to this issue in [Section 4.1.6](#), after we learn more about evaluation.

<sup>108</sup> 内部定義を十分に良く理解し、プログラムが私達がそれに意図した意味を意味することを確実にするには私達がこの章で紹介したよりもより複雑な評価過程のモデルを必要とします。しかし手続の内部定義と共ににはその機微は浮かび上がりません。この問題については[Section 4.1.6](#)にて評価についてより学んだ後に立ち戻ります。

### 1.3.3 Procedures as General Methods

We introduced compound procedures in [Section 1.1.4](#) as a mechanism for abstracting patterns of numerical operations so as to make them independent of the particular numbers involved. With higher-order procedures, such as the `integral` procedure of [Section 1.3.1](#), we began to see a more powerful kind of abstraction: procedures used to express general methods of computation, independent of the particular functions involved. In this section we discuss two more elaborate examples—general methods for finding zeros and fixed points of functions—and show how these methods can be expressed directly as procedures.

[Section 1.1.4](#)にて複合手続を数値演算の抽象化パターンのメカニズムとして紹介し、関係する特定の数値から独立させました。[Section 1.3.1](#)の `integral` 手続のような高階手続ではより強力な種類の抽象化について学び始めました。関係する特定の関数から独立した汎用的演算手法を表現するのに利用される手続でした。この節では 2 つより複雑な例—零と関数の不動点を見付けるための汎用手法—について議論します。そしてこれらの手法がどのように手続として直接的に表現されるのかを示します。

#### Finding roots of equations by the half-interval method

The *half-interval method* is a simple but powerful technique for finding roots of an equation  $f(x) = 0$ , where  $f$  is a continuous function. The idea is that, if we are given points  $a$  and  $b$  such that  $f(a) < 0 < f(b)$ , then  $f$  must have at least one zero between  $a$  and  $b$ . To locate a zero, let  $x$  be the average of  $a$  and  $b$ , and compute  $f(x)$ . If  $f(x) > 0$ , then  $f$  must have a zero between  $a$  and  $x$ . If  $f(x) < 0$ , then  $f$  must have a zero between  $x$  and  $b$ . Continuing in this way, we can identify smaller and smaller intervals on which  $f$  must have a zero. When we reach a point where the interval is small enough, the process stops. Since the interval of uncertainty is reduced by half at each step of the process, the number of steps required grows as  $\Theta(\log(L/T))$ , where  $L$  is the length of the original interval and  $T$  is the error tolerance (that is, the size of the interval we will consider “small enough”). Here is a procedure that implements this strategy:

*half-interval method*(半区間手法) は方程式  $f(x) = 0$  の根を求めるのに単純ながら強力なテクニックです。ここで  $f$  は連続関数とします。この考えは  $f(a) < 0 < f(b)$  となる点  $a$  と  $b$  を与えた時、 $f$  は最低でも 1 つの 0 を  $a$  と  $b$  の

間に持つことになります。ゼロを特定するために  $a$  と  $b$  の平均  $x$  を求め  $f(x)$  を計算します。もし  $f(x) > 0$  なら  $f$  は 0 を  $a$  と  $x$  の間に持ちます。もし  $f(x) < 0$  なら  $f$  は 0 を  $x$  と  $b$  の間に持ちます。このように繰り返すことで  $f$  が 0 を持つより小さな区間を特定できます。区間が十分に小さな時点に辿りついたら処理は停止します。不確かな区間が処理の各ステップにて半分になるため、必要とされるステップ数は  $\Theta(\log(L/T))$  に従い増加します。このとき  $L$  は元の区間の長さで  $T$  は許容誤差(私達が“十分に小さい”と考える区間のサイズ)になります。この戦略を実装した手続が以下になります。

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                  (search f neg-point midpoint))
                ((negative? test-value)
                  (search f midpoint pos-point))
                (else midpoint)))))))
```

We assume that we are initially given the function  $f$  together with points at which its values are negative and positive. We first compute the midpoint of the two given points. Next we check to see if the given interval is small enough, and if so we simply return the midpoint as our answer. Otherwise, we compute as a test value the value of  $f$  at the midpoint. If the test value is positive, then we continue the process with a new interval running from the original negative point to the midpoint. If the test value is negative, we continue with the interval from the midpoint to the positive point. Finally, there is the possibility that the test value is 0, in which case the midpoint is itself the root we are searching for.

最初に関数  $f$  を値が負と正になる 2 つの点と共に与えられると想定します。最初に 2 つの与えられた点の中間点を求めます。次に与えられた区間が十分に小さいかチェックし、もしそうであれば単純に中間点を答とします。そうでなければ中間点における  $f$  の値を test-value(試験値) として計算します。もし試験値が正ならば、元の負の地点から中間点までの新しい区間にて処理を続けます。もし試験値が負ならば中間値から正の地点までの区間にて処理を続けます。最終的に試験値が 0 になる可能性がありますが、その場合、中間地点そのもの

が我々が探している根となります。

To test whether the endpoints are “close enough” we can use a procedure similar to the one used in Section 1.1.7 for computing square roots:<sup>109</sup>

終了地点が“十分に近い”か試験するためにはSection 1.1.7にて平方根を求めるために利用した物と同様の手続が利用可能です。<sup>110</sup>

```
(define (close-enough? x y) (< (abs (- x y)) 0.001))
```

`Search` is awkward to use directly, because we can accidentally give it points at which  $f$ ’s values do not have the required sign, in which case we get a wrong answer. Instead we will use `search` via the following procedure, which checks to see which of the endpoints has a negative function value and which has a positive value, and calls the `search` procedure accordingly. If the function has the same sign on the two given points, the half-interval method cannot be used, in which case the procedure signals an error.<sup>111</sup>

`search` は直接利用するのは扱いにくいです。 $f$  の値が必要な符号を持たない点を意図せず与えてしまうことが可能なためです。そのような場合では間違った答を得てしまいます。代わりに `search` を次の手続を経由して使用することにしましょう。これは終端のどちらが負の関数値を持ち、どちらが正の関数値を持つか検査します。そして `search` 手續を適切に呼び出します。もし関数が 2 つの与えられた点にて同じ符号を持つ場合、半区間手法は使用できません。この場合この手続はエラーを伝えます。<sup>112</sup>

```
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b))))
```

---

<sup>109</sup>We have used 0.001 as a representative “small” number to indicate a tolerance for the acceptable error in a calculation. The appropriate tolerance for a real calculation depends upon the problem to be solved and the limitations of the computer and the algorithm. This is often a very subtle consideration, requiring help from a numerical analyst or some other kind of magician.

<sup>110</sup>私達は“小さな”値の表現として 0.001 を用い、計算にて受け入れられる誤差の許容範囲を示しました。実際の演算における適切な許容範囲は解決すべき問題、計算機とアルゴリズムの制約に依存します。これはしばしばとても微妙な考慮事項であり、数値解析者や他の魔法使いのような人達の助けを必要とします。

<sup>111</sup>This can be accomplished using `error`, which takes as arguments a number of items that are printed as error messages.

<sup>112</sup>これは `error` を用いて達成できます。`error` は引数としていくつかの項目を受け取りそれらをエラーメッセージとして出力します。

```
(cond ((and (negative? a-value) (positive? b-value))
       (search f a b))
      ((and (negative? b-value) (positive? a-value))
       (search f b a))
      (else (error "Values are not of
opposite sign" a b)))))
```

The following example uses the half-interval method to approximate  $\pi$  as the root between 2 and 4 of  $\sin x = 0$ :

次の例は半区間手法を使用して  $\pi$  の近似を  $\sin x = 0$  の 2 と 4 の間の根として求めています。

```
(half-interval-method sin 2.0 4.0)
3.14111328125
```

Here is another example, using the half-interval method to search for a root of the equation  $x^3 - 2x - 3 = 0$  between 1 and 2: また次の別の例では半区間手法を用いて方程式  $x^3 - 2x - 3 = 0$  において 1 と 2 の間で根を探しています。

```
(half-interval-method (lambda (x) (- (* x x x) (* 2 x) 3))
1.0
2.0)
1.89306640625
```

## Finding fixed points of functions

A number  $x$  is called a *fixed point* of a function  $f$  if  $x$  satisfies the equation  $f(x) = x$ . For some functions  $f$  we can locate a fixed point by beginning with an initial guess and applying  $f$  repeatedly,

数値  $x$  は  $x$  が等式  $f(x) = x$  を満たす時、関数  $f$  の *fixed point* と呼ばれます。いくつかの関数  $f$  に対し不動点を初期推測値から始めて  $f$  を値があまり変わらなくなるまで繰り返し適用することで求めることができます。

$$f(x), \quad f(f(x)), \quad f(f(f(x))), \quad \dots,$$

until the value does not change very much. Using this idea, we can devise a procedure **fixed-point** that takes as inputs a function and an initial guess and produces an approximation to a fixed point of the function. We apply the

function repeatedly until we find two successive values whose difference is less than some prescribed tolerance:

この考え方を用いて関数と初期推定値を入力とし、関数の不動点への近似を生成する手続、`fixed-point` を開発できます。指示した許容範囲未満の差に二点が収まるまで関数を繰り返し適用します。

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
        tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

For example, we can use this method to approximate the fixed point of the cosine function, starting with 1 as an initial approximation:<sup>113</sup>

例えばこの手法をコサイン関数の不動点を近似するのに利用できます。初期推測値は1とします。<sup>114</sup>

```
(fixed-point cos 1.0)
.7390822985224023
```

Similarly, we can find a solution to the equation

同様にして、次の方程式の答を見つけられます。 $y = \sin y + \cos y$ :

```
(fixed-point (lambda (y) (+ (sin y) (cos y))) 1.0)
1.2587315962971173
```

The fixed-point process is reminiscent of the process we used for finding square roots in Section 1.1.7. Both are based on the idea of repeatedly improving a guess until the result satisfies some criterion. In fact, we can readily formulate

---

<sup>113</sup>Try this during a boring lecture: Set your calculator to radians mode and then repeatedly press the `cos` button until you obtain the fixed point.

<sup>114</sup>以下を暇な授業の間に実行してみて下さい：電卓をラジアンモードに設定し不動点に到達するまで `cos` を連打してみましょう。

the square-root computation as a fixed-point search. Computing the square root of some number  $x$  requires finding a  $y$  such that  $y^2 = x$ . Putting this equation into the equivalent form  $y = x/y$ , we recognize that we are looking for a fixed point of the function<sup>115</sup>  $y \mapsto x/y$ , and we can therefore try to compute square roots as

不動点処理はSection 1.1.7にて平方根を求めるのに使用した処理を思い出させます。両者は共に結果がある判定基準を満たすまで推測値を繰り返し改善する考えを基にしています。実際に直ぐに平方根の計算を不動点検索として形式化が可能です。ある数値  $x$  の平方根を求めるには  $y^2 = x$  を満たす  $y$  を探す必要があります。この等式を等価な形の  $y = x/y$ <sup>116</sup> にするとその関数  $y \mapsto x/y$  の不動点を探してることに気付きます。従って平方根を以下のように求めることを試すことができます。

```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y)) 1.0))
```

Unfortunately, this fixed-point search does not converge. Consider an initial guess  $y_1$ . The next guess is  $y_2 = x/y_1$  and the next guess is  $y_3 = x/y_2 = x/(x/y_1) = y_1$ . This results in an infinite loop in which the two guesses  $y_1$  and  $y_2$  repeat over and over, oscillating about the answer.

残念ながらこの不動点検索は収束しません。初期推測値を  $y_1$  とします。次の推測値は  $y_2 = x/y_1$  でさらに次は  $y_3 = x/y_2 = x/(x/y_1) = y_1$  です。この結果は 2 つの推測値  $y_1$  と  $y_2$  がずっと繰り返し、答が振動する無限ループです。

One way to control such oscillations is to prevent the guesses from changing so much. Since the answer is always between our guess  $y$  and  $x/y$ , we can make a new guess that is not as far from  $y$  as  $x/y$  by averaging  $y$  with  $x/y$ , so that the next guess after  $y$  is  $\frac{1}{2}(y + x/y)$  instead of  $x/y$ . The process of making such a sequence of guesses is simply the process of looking for a fixed point of  $y \mapsto \frac{1}{2}(y + x/y)$ :

そのような振動をコントロールする 1 つの方法は推測値が大きく変化することを防ぐことです。回答は常に推測値  $y$  と  $x/y$  の間ににあるはずですから  $y$  と  $x/y$  の両方から同じ位遠くはない地点にできるはずです。従って  $y$  と  $x/y$  の平均を取って  $y$  の次の推測値は  $\frac{1}{2}(y + x/y)$  となります。

---

<sup>115</sup>  $\mapsto$  (pronounced “maps to”) is the mathematician’s way of writing `lambda`.  $y \mapsto x/y$  means `(lambda (y) (/ x y))`, that is, the function whose value at  $y$  is  $x/y$ .

<sup>116</sup>  $\mapsto$  (“maps to”(写す)と読みます。)は數学者による `lambda` の記述法です。 $y \mapsto x/y$  は `(lambda (y) (/ x y))` を意味し、 $y$  における関数の値は  $x/y$  ということです。

```
(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y))) 1.0))
```

(Note that  $y = \frac{1}{2}(y + x/y)$  is a simple transformation of the equation  $y = x/y$ ; to derive it, add  $y$  to both sides of the equation and divide by 2.)

( $y = \frac{1}{2}(y + x/y)$  は等式  $y = x/y$  を単純に変形したものであることに注意して下さい。得るためにには等式の両辺に  $y$  を足し、2で割ります。)

With this modification, the square-root procedure works. In fact, if we unravel the definitions, we can see that the sequence of approximations to the square root generated here is precisely the same as the one generated by our original square-root procedure of [Section 1.1.7](#). This approach of averaging successive approximations to a solution, a technique that we call *average damping*, often aids the convergence of fixed-point searches.

この変更により平方根手続がうまく行きます。実際に、もし定義をひも解いた場合、ここで生成された平方根の近似の連続は元々の[Section 1.1.7](#)の平方根手続が生成するものと正確に同じです。この一連の近似値の平均から回答へのアプローチは、*average damping*(平均減衰)と呼ぶテクニックであり、良く不動点検索の収束に対し手助けとなります。

**Exercise 1.35:** Show that the golden ratio  $\varphi$  ([Section 1.2.2](#)) is a fixed point of the transformation  $x \mapsto 1 + 1/x$ , and use this fact to compute  $\varphi$  by means of the `fixed-point` procedure.

黄金比率  $\varphi$ ([Section 1.2.2](#)) は変形  $x \mapsto 1 + 1/x$  の不動点であること示せ。この比率を用いて  $\varphi$  を `fixed-point` 手続を用いて求めよ。

**Exercise 1.36:** Modify `fixed-point` so that it prints the sequence of approximations it generates, using the `newline` and `display` primitives shown in [Exercise 1.22](#). Then find a solution to  $x^x = 1000$  by finding a fixed point of  $x \mapsto \log(1000)/\log(x)$ . (Use Scheme's primitive `log` procedure, which computes natural logarithms.) Compare the number of steps this takes with and without average damping. (Note that you cannot start `fixed-point` with a guess of 1, as this would cause division by  $\log(1) = 0$ .)

`fixed-point` を変更し、[Exercise 1.22](#)にて示された `newline` と `display` プリミティブを用いて生成する一連の近似値を表示するようにせよ。次に  $x^x = 1000$  の答を  $x \mapsto \log(1000)/\log(x)$  の不動点を求める方法で求めよ。(Scheme のプリミティブである `log` 手続を利用せよ。これは自然対数を計算する)。平均減衰を用いる場

合と用いない場合にてステップ数を比較せよ。(fixed-point を推定値 1 では開始できないことに注意せよ。これは  $\log(1) = 0$  での割り算を引き起すためである。)

### Exercise 1.37:

- a An infinite *continued fraction* is an expression of the form  
無限*continued fraction*(連分数) とは以下の形式の式である。

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \dots}}}.$$

As an example, one can show that the infinite continued fraction expansion with the  $N_i$  and the  $D_i$  all equal to 1 produces  $1/\varphi$ , where  $\varphi$  is the golden ratio (described in Section 1.2.2). One way to approximate an infinite continued fraction is to truncate the expansion after a given number of terms. Such a truncation—a so-called *k-term finite continued fraction*—has the form

例として、無限連分数の展開として  $N_i$  と  $D_i$  の全てが 1 の場合、 $1/\varphi$  を生成し、この時  $\varphi$  は (Section 1.2.2で説明した) 黄金比率である。無限連分数の近似を求める 1 つの方法として与えられた項の数を越えた後、展開を切り捨てる方法がある。そのような切り捨て—所謂 *k-term finite continued fraction*(k 項有限連分数)—は以下の形式になる。

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{\ddots + \cfrac{N_k}{D_k}}}.$$

Suppose that `n` and `d` are procedures of one argument (the term index  $i$ ) that return the  $N_i$  and  $D_i$  of the terms of the continued fraction. Define a procedure `cont-frac` such that evaluating (`cont-frac n d k`) computes the value of the  $k$ -term finite continued fraction. Check your procedure by approximating  $1/\varphi$  using

`n` と `d` は連分数の  $N_i D_i$  の項を返す 1 引数 (項の索引  $i$ ) の手続であると考える。`(cont-frac n d k)` を評価すると  $k$  項有

限連分数の値を求める手続、`cont-frac` を定義せよ。あなたの手続を以下を用いて  $1/\varphi$  の近似を求めてることで、一連の  $k$  の値についてチェックせよ。

```
(cont-frac (lambda (i) 1.0)
            (lambda (i) 1.0)
            k)
```

for successive values of  $k$ . How large must you make  $k$  in order to get an approximation that is accurate to 4 decimal places? 小数点以下 4 桁の精度の近似を得るには  $k$  はどれだけの大きさでなければならないか？

- b If your `cont-frac` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

もしあなたの `cont-frac` 手続が再帰プロセスを生成するのならば線形プロセスを生成するものを書け。もし線形プロセスを生成するのであれば、再帰プロセスを生成するものを書け。

**Exercise 1.38:** In 1737, the Swiss mathematician Leonhard Euler published a memoir *De Fractionibus Continuis*, which included a continued fraction expansion for  $e - 2$ , where  $e$  is the base of the natural logarithms. In this fraction, the  $N_i$  are all 1, and the  $D_i$  are successively 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, .... Write a program that uses your `cont-frac` procedure from [Exercise 1.37](#) to approximate  $e$ , based on Euler's expansion.

1737 年にスイスの数学者、Leonhard Euler(レオンハルト・オイラー)は学術論文 *De Fractionibus Continuis* を出版した。それには  $e$  が自然対数の底である時の、 $e - 2$  に対する連分数展開が含まれている。この分数では  $N_i$  は全て 1 であり、 $D_i$  は数列 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, ... である。[Exercise 1.37](#) のあなたの `cont-frac` 手続を用いてオイラー展開を基にし  $e$  の近似を求めるプログラムを書け。

**Exercise 1.39:** A continued fraction representation of the tangent function was published in 1770 by the German mathematician J.H. Lambert:

タンジェント(正接)関数の連分数表現は1770年にドイツの数学者J.H. Lambert(ヨハン・ハインリッヒ・ランベルト)により発表された。

$$\tan x = \frac{x}{1 - \frac{x^2}{3 - \frac{x^2}{5 - \dots}}},$$

where  $x$  is in radians. Define a procedure (`(tan-cf x k)`) that computes an approximation to the tangent function based on Lambert's formula.  $k$  specifies the number of terms to compute, as in [Exercise 1.37](#).

ここで  $x$  はラジアンである。ランベルトの式を基にして正接関数の近似値を求める手続 (`(tan-cf x k)`) を定義せよ。 $k$  は[Exercise 1.37](#)と同様に求める項の数を指定する。

### 1.3.4 Procedures as Returned Values

The above examples demonstrate how the ability to pass procedures as arguments significantly enhances the expressive power of our programming language. We can achieve even more expressive power by creating procedures whose returned values are themselves procedures.

ここまでの一連の例は手続を引数として渡す能力が著しく私達のプログラミング言語の表現力を拡張することを実演しました。返り値自体が手続である手続を作成することでさらに表現力を獲得することができます。

We can illustrate this idea by looking again at the fixed-point example described at the end of [Section 1.3.3](#). We formulated a new version of the square-root procedure as a fixed-point search, starting with the observation that  $\sqrt{x}$  is a fixed-point of the function  $y \mapsto x/y$ . Then we used average damping to make the approximations converge. Average damping is a useful general technique in itself. Namely, given a function  $f$ , we consider the function whose value at  $x$  is equal to the average of  $x$  and  $f(x)$ .

この考えを[Section 1.3.3](#)の終わりにて説明された不動点の例を振り返ることで説明できます。`square-root`手続の新しいバージョンを、 $\sqrt{x}$ は  $y \mapsto x/y$  関数の不動点であるという観察結果から始めて、不動点検索の形で形式化しました。次に平均減衰を用いて近似値を収束させました。平均減衰はそれ自体が便利な汎用技法です。即ち関数  $f$  を与えられた時、 $x$  における関数の値が  $x$  と  $f(x)$  の平均だと考えます。

We can express the idea of average damping by means of the following procedure:

平均減衰の考えを次の手続を用いて説明できます。

```
(define (average-damp f) (lambda (x) (average x (f x))))
```

**Average-damp** is a procedure that takes as its argument a procedure **f** and returns as its value a procedure (produced by the **lambda**) that, when applied to a number **x**, produces the average of **x** and **(f x)**. For example, applying **average-damp** to the **square** procedure produces a procedure whose value at some number **x** is the average of **x** and  **$x^2$** . Applying this resulting procedure to 10 returns the average of 10 and 100, or 55:<sup>117</sup>

**average-damp** は引数として手続 **f** を取りその値として (**lambda** で生成された) 手続を返します。その手続は数値 **x** に適用された時、**x** と **(f x)** の平均を返します。例えば **average-damp** を **square** 手続に適用した時、生成された手続の値は、ある値 **x** において **x** と  **$x^2$**  の平均となります。この結果の手続に 10 を適用すれば 10 と 100 の平均として 55 を返します。<sup>118</sup>

```
((average-damp square) 10)  
55
```

Using **average-damp**, we can reformulate the square-root procedure as follows: **average-damp** を用いて **square-root** 手続を次のように再公式化できます。

```
(define (sqrt x)  
  (fixed-point (average-damp (lambda (y) (/ x y))) 1.0))
```

Notice how this formulation makes explicit the three ideas in the method: fixed-point search, average damping, and the function  $y \mapsto x/y$ . It is instructive to compare this formulation of the square-root method with the original version given in Section 1.1.7. Bear in mind that these procedures express the same

---

<sup>117</sup>Observe that this is a combination whose operator is itself a combination. Exercise 1.4 already demonstrated the ability to form such combinations, but that was only a toy example. Here we begin to see the real need for such combinations—when applying a procedure that is obtained as the value returned by a higher-order procedure.

<sup>118</sup>これは組み合わせであり、かつそのオペレータもまた合成式であることに注意して下さい。Exercise 1.4にて既にそのような合成式を形式化する能力については実演しました。しかしあれは単に簡単な例にすぎません。ここではそのような合成式に対する真の要求—高階手続により値として返されることで得られた手続をいつ適用するのかについて学び始めます。

process, and notice how much clearer the idea becomes when we express the process in terms of these abstractions. In general, there are many ways to formulate a process as a procedure. Experienced programmers know how to choose procedural formulations that are particularly perspicuous, and where useful elements of the process are exposed as separate entities that can be reused in other applications. As a simple example of reuse, notice that the cube root of  $x$  is a fixed point of the function  $y \mapsto x/y^2$ , so we can immediately generalize our square-root procedure to one that extracts cube roots:<sup>119</sup>

この形式化が手法内においてどれだけ 3 つの考え方、不動点検索、平均減衰、関数  $y \mapsto x/y$  について明快にしているかに注目して下さい。この square-root の手法の形式化と Section 1.1.7 で与えた元のバージョンの比較は示唆的です。これらの手続が同じ処理について表現していることを心に留めて下さい。そして同じ処理をこれらの抽象化を用いて表現した時にどれだけ明白になるのかに注目して下さい。一般的に処理を手続に形式化する手法はとても多くの数有ります。経験の豊富なプログラマはどのように手続形式化を選ぶのか、特に明快な方法を知っています。そしてどこで処理の便利な要素が他のアプリケーションにて再使用可能な独立した要素として浮かび上がるかについて知っているのです。再使用の簡単な例として  $x$  の立方根は関数  $y \mapsto x/y^2$  の不動点であることに注意して下さい。従って直ぐに square-root 手続を立方根を求める手続に汎化することが可能です。<sup>120</sup>

```
(define (cube-root x)
  (fixed-point (average-damp (lambda (y) (/ x (square y))))
               1.0))
```

## Newton's method

When we first introduced the square-root procedure, in Section 1.1.7, we mentioned that this was a special case of *Newton's method*. If  $x \mapsto g(x)$  is a differentiable function, then a solution of the equation  $g(x) = 0$  is a fixed point of the function  $x \mapsto f(x)$ , where

Section 1.1.7 にて初めて square-root 手続を紹介した時に、これは *Newton's method*(ニュートン法) の特別な場合であると伝えました。もし  $x \mapsto g(x)$  が微分可能な関数である時、方程式  $g(x) = 0$  の答は以下の条件で関数  $x \mapsto f(x)$  の

---

<sup>119</sup> See Exercise 1.45 for a further generalization.

<sup>120</sup> さらなる一般化については Exercise 1.45 を参照して下さい。

不動点となります。

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

and  $Dg(x)$  is the derivative of  $g$  evaluated at  $x$ . Newton's method is the use of the fixed-point method we saw above to approximate a solution of the equation by finding a fixed point of the function  $f$ .<sup>121</sup>

そして  $Dg(x)$  は  $x$  により微分した導関数です。ニュートン法は上で学んだ不動点を用いる手法で、関数  $f$  の不動点を探すことで方程式の解の近似を求めます。<sup>122</sup>

For many functions  $g$  and for sufficiently good initial guesses for  $x$ , Newton's method converges very rapidly to a solution of  $g(x) = 0$ .<sup>123</sup>

多くの関数  $g$  において、また十分に良い初期推測値  $x$  においてニュートン法は  $g(x) = 0$  の解に急速に収束します。<sup>124</sup>

In order to implement Newton's method as a procedure, we must first express the idea of derivative. Note that “derivative,” like average damping, is something that transforms a function into another function. For instance, the derivative of the function  $x \mapsto x^3$  is the function  $x \mapsto 3x^2$ . In general, if  $g$  is a function and  $dx$  is a small number, then the derivative  $Dg$  of  $g$  is the function whose value at any number  $x$  is given (in the limit of small  $dx$ ) by

ニュートン法を手続として実装するために、最初に微分の考え方を表現せねばなりません。“微分”は平均減衰と同様にある関数を別の関数へと変形することに注目して下さい。例えば関数  $x \mapsto x^3$  の微分は  $x \mapsto 3x^2$  です。一般的に  $g$  が関数であり  $dx$  が小さな値である時、 $g$  の導関数  $Dg$  はその値が任意の数  $x$  が

<sup>121</sup>Elementary calculus books usually describe Newton's method in terms of the sequence of approximations  $x_{n+1} = x_n - g(x_n)/Dg(x_n)$ . Having language for talking about processes and using the idea of fixed points simplifies the description of the method.

<sup>122</sup> 初歩的な微積分学の教科書は通常ニュートン法を近似的な数列  $x_{n+1} = x_n - g(x_n)/Dg(x_n)$  を用いて説明しています。処理に関する言語を持ち不動点の考え方を用いることで手法の説明を平易にできます。

<sup>123</sup>Newton's method does not always converge to an answer, but it can be shown that in favorable cases each iteration doubles the number-of-digits accuracy of the approximation to the solution. In such cases, Newton's method will converge much more rapidly than the half-interval method.

<sup>124</sup>ニュートン法は常に解へと収束はしません。しかし好ましい場合においては各繰り返しにおいて解の近似値の精度の桁数は二倍になることが示されます。そのような場合にはニュートン法は半区間手法よりも大変速く収束します。

与えられた時に (小さな値  $dx$  の極限において)

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}.$$

Thus, we can express the idea of derivative (taking  $dx$  to be, say, 0.00001) as the procedure

従って微分の考えを ( $dx$  を例えば 0.00001 として) 手続として次のように表現できます。

```
(define (deriv g)
  (lambda (x) (/ (- (g (+ x dx)) (g x)) dx)))
```

along with the definition

次の定義と一緒に用います。

```
(define dx 0.00001)
```

Like `average-damp`, `deriv` is a procedure that takes a procedure as argument and returns a procedure as value. For example, to approximate the derivative of  $x \mapsto x^3$  at 5 (whose exact value is 75) we can evaluate

`average-damp` と同様に、`deriv` は引数として手続を取り、値として手続を返す手続です。例えば導関数  $x \mapsto x^3$  の 5 における値(正確な値は 75 です)の近似を求めるために以下のように評価できます。

```
(define (cube x) (* x x x))
((deriv cube) 5)
75.00014999664018
```

With the aid of `deriv`, we can express Newton's method as a fixed-point process:

`deriv` の助けを借りて、ニュートン法を不動点処理として次のように表現できます。

```
(define (newton-transform g)
  (lambda (x) (- x (/ (g x) ((deriv g) x)))))
(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
```

The `newton-transform` procedure expresses the formula at the beginning of this section, and `newtons-method` is readily defined in terms of this. It takes as

arguments a procedure that computes the function for which we want to find a zero, together with an initial guess. For instance, to find the square root of  $x$ , we can use Newton's method to find a zero of the function  $y \mapsto y^2 - x$  starting with an initial guess of 1.<sup>125</sup>

`newton-transform` 手続はこの節の最初の式を表現しています。そして `newtons-method` が直ぐにそれを用いて定義されています。これは初期推測値と一緒に手続を引数として取りその手続はゼロを見つけたい関数を計算します。例えば、 $x$  の平方根を見つけたい時、ニュートン法を用いて関数  $y \mapsto y^2 - x$  のゼロを初期推測値 1 から始めて探すことが可能です。<sup>126</sup>

This provides yet another form of the square-root procedure:  
これが平方根手続の別の形を与えます。

```
(define (sqrt x)
  (newtons-method
    (lambda (y) (- (square y) x)) 1.0))
```

## Abstractions and first-class procedures

We've seen two ways to express the square-root computation as an instance of a more general method, once as a fixed-point search and once using Newton's method. Since Newton's method was itself expressed as a fixed-point process, we actually saw two ways to compute square roots as fixed points. Each method begins with a function and finds a fixed point of some transformation of the function. We can express this general idea itself as a procedure:

より一般的な手法の事例として、平方根の演算の表現方法を 2 つ見てきました。1 つは不動点検索で、もう 1 つはニュートン法です。ニュートン法はそれ自体が不動点処理として表現されているため、実際には平方根を不動点として計算する 2 つの方法を見た訳になります。各手法は関数と共に開始し、その関数のある変形の不動点を探します。この一般的な考え方自身を手続として表現できます。

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))
```

---

<sup>125</sup>For finding square roots, Newton's method converges rapidly to the correct solution from any starting point.

<sup>126</sup>平方根を探す場合、ニュートン法は任意の開始値から急速に正しい解に収束します。

This very general procedure takes as its arguments a procedure  $g$  that computes some function, a procedure that transforms  $g$ , and an initial guess. The returned result is a fixed point of the transformed function.

このとても汎用的な手続は引数としてある関数を計算する手続  $g$ 、 $g$  を変形する手続、初期推測値を取ります。結果としての返り値は変形された関数の不動点です。

Using this abstraction, we can recast the first square-root computation from this section (where we look for a fixed point of the average-damped version of  $y \mapsto x/y$ ) as an instance of this general method:

この抽象化を用いて、この節最初の(平均減衰バージョンの  $y \mapsto x/y$  の不動点を探した)平方根演算をこの汎用手法の例として変更できます。

```
(define (sqrt x)
  (fixed-point-of-transform
    (lambda (y) (/ x y)) average-damp 1.0))
```

Similarly, we can express the second square-root computation from this section (an instance of Newton's method that finds a fixed point of the Newton transform of  $y \mapsto y^2 - x$ ) as

同様にこの節の2つ目の平方根演算( $y \mapsto y^2 - x$  のニュートン変形の不動点を探すニュートン法の例)を以下の様に表現できます。

```
(define (sqrt x)
  (fixed-point-of-transform
    (lambda (y) (- (square y) x)) newton-transform 1.0))
```

We began [Section 1.3](#) with the observation that compound procedures are a crucial abstraction mechanism, because they permit us to express general methods of computing as explicit elements in our programming language. Now we've seen how higher-order procedures permit us to manipulate these general methods to create further abstractions.

[Section 1.3](#)では複合手続は重大な抽象化メカニズムであるという考え方から始めました。私達のプログラミング言語において演算の一般的な手法を明示的な要素として表現することを可能にするためです。ここでは高階手続がどのようにこれらの一般的手法を操作してさらなる抽象化を作成することを可能にするのかについて学びます。

As programmers, we should be alert to opportunities to identify the underlying abstractions in our programs and to build upon them and generalize

them to create more powerful abstractions. This is not to say that one should always write programs in the most abstract way possible; expert programmers know how to choose the level of abstraction appropriate to their task. But it is important to be able to think in terms of these abstractions, so that we can be ready to apply them in new contexts. The significance of higher-order procedures is that they enable us to represent these abstractions explicitly as elements in our programming language, so that they can be handled just like other computational elements.

プログラマとして、私達のプログラムに内在する抽象化を判別する機会を迅速に学ばねばなりません。そしてその上に構築し、それらを汎化してより強力な抽象化を作成する術を学ばねばなりません。これは常にプログラムを可能な限り抽象化して書かねばならないと言う訳ではありません。エキスパートプログラマは彼等のタスクにとって適切な抽象化レベルの選択方法を知っています。しかし、これらの抽象化を用いて考えられるようになることが重要です。そうすればそれらを新しいコンテキストでも適用することに準備することができます。高階手続の有用性は、それらがここまで抽象化を私達のプログラミング言語の要素として明示的に表現することを可能にしてくれることです。そうすることで抽象化は他の計算上の要素と同様に扱われることが可能となります。

In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have *first-class* status. Some of the “rights and privileges” of first-class elements are:<sup>127</sup>

通常、プログラミング言語は計算要素が取扱可能になるような方法に制約を課します。制約が最も少ない要素は*first-class*(第一級)の地位にあると言われます。第一級要素の“権利と特権”的いくつかを次に示します。<sup>128</sup>

- They may be named by variables.  
変数により名付けることが可能
- They may be passed as arguments to procedures.  
手続に対し引数として渡すことが可能
- They may be returned as the results of procedures.

---

<sup>127</sup>The notion of first-class status of programming-language elements is due to the British computer scientist Christopher Strachey (1916-1975).

<sup>128</sup>プログラミング言語の要素の第一級の地位の概念はイギリスの計算機科学者 Christopher Strachey (1916-1975) によるものです。

手続の結果として返すことが可能

- They may be included in data structures.<sup>129</sup>

データ構造に含まれることが可能<sup>130</sup>

Lisp, unlike other common programming languages, awards procedures full first-class status. This poses challenges for efficient implementation, but the resulting gain in expressive power is enormous.<sup>131</sup>

Lisp は他の言語と異なり、手続に完全な第一級の地位を与えます。このことが効率の良い実装に対して課題を課しますが、結果的に表現力に得る物は莫大な物となります。<sup>132</sup>

**Exercise 1.40:** Define a procedure `cubic` that can be used together with the `newtons-method` procedure in expressions of the form

`newtons-method` 手續と共に以下の形式の式にて使用が可能な手続 `cubic` を定義せよ。

```
(newtons-method (cubic a b c) 1)
```

to approximate zeros of the cubic  $x^3 + ax^2 + bx + c$ .

次に三次方程式  $x^3 + ax^2 + bx + c = 0$  の近似解を求めよ。

**Exercise 1.41:** Define a procedure `double` that takes a procedure of one argument as argument and returns a procedure that applies the original procedure twice. For example, if `inc` is a procedure that adds 1 to its argument, then `(double inc)` should be a procedure that adds 2. What value is returned by

引数が 1 つの手続を引数として取り、その手続を二回適用する手続を返す手続 `double` を定義せよ。例えば `inc` が引数に 1 を足す手続であれば、`(double inc)` は 2 を足す手続になる。次の式はどんな値を返すか?

<sup>129</sup>We'll see examples of this after we introduce data structures in Chapter 2.

<sup>130</sup>これの例はChapter 2にてデータ構造を紹介した後に学びます。

<sup>131</sup>The major implementation cost of first-class procedures is that allowing procedures to be returned as values requires reserving storage for a procedure's free variables even while the procedure is not executing. In the Scheme implementation we will study in Section 4.1, these variables are stored in the procedure's environment.

<sup>132</sup>第一級手続の主な実装コストは、手続が値として返すことを可能とするために、手続の自由変数に対して予備の領域を、例え手続が実行中でなくとも必要とします。私達が Section 4.1 にて学ぶ Scheme の実装では、これらの変数は手続の環境に保存されます。

```
((double (double double)) inc) 5)
```

**Exercise 1.42:** Let  $f$  and  $g$  be two one-argument functions. The composition  $f$  after  $g$  is defined to be the function  $x \mapsto f(g(x))$ . Define a procedure `compose` that implements composition. For example, if `inc` is a procedure that adds 1 to its argument,

$f$  と  $g$  が 2 つの 1 引数関数だとする。 $g$  に  $f$  を composition 合成するとは関数  $x \mapsto f(g(x))$  と定義される。合成を実装する手続 `compose` を定義せよ。例えば `inc` が引数に 1 を足す手続である場合、

```
((compose square inc) 6)
```

49

**Exercise 1.43:** If  $f$  is a numerical function and  $n$  is a positive integer, then we can form the  $n^{\text{th}}$  repeated application of  $f$ , which is defined to be the function whose value at  $x$  is  $f(f(\dots(f(x))\dots))$ . For example, if  $f$  is the function  $x \mapsto x + 1$ , then the  $n^{\text{th}}$  repeated application of  $f$  is the function  $x \mapsto x + n$ . If  $f$  is the operation of squaring a number, then the  $n^{\text{th}}$  repeated application of  $f$  is the function that raises its argument to the  $2^n$ -th power. Write a procedure that takes as inputs a procedure that computes  $f$  and a positive integer  $n$  and returns the procedure that computes the  $n^{\text{th}}$  repeated application of  $f$ . Your procedure should be able to be used as follows:

$f$  が数値演算関数であり  $n$  が正の整数である時、 $f$  を  $n$  回適用する、 $x$  における値が  $f(f(\dots(f(x))\dots))$  である関数を定義できる。例えば  $f$  が関数  $x \mapsto x + 1$  である時、 $f$  を  $n$  回適用した関数は  $x \mapsto x + n$  となる。もし  $f$  が数値を二乗する操作ならば、 $f$  を  $n$  回適用した関数は引数を  $2^n$  乗する。入力として  $f$  を計算する手続と正の整数  $n$  を取り、 $f$  の  $n$  回適用を計算する手続を返す手続を書け。その手続は以下のように使用可能でなければならない。

```
((repeated square 2) 5)
```

625

Hint: You may find it convenient to use `compose` from Exercise 1.42.

ヒント : Exercise 1.42 の `compose` を使うと便利でしょう。

**Exercise 1.44:** The idea of *smoothing* a function is an important concept in signal processing. If  $f$  is a function and  $dx$  is some small number, then the smoothed version of  $f$  is the function whose value at a point  $x$  is the average of  $f(x - dx)$ ,  $f(x)$ , and  $f(x + dx)$ . Write a procedure `smooth` that takes as input a procedure that computes  $f$  and returns a procedure that computes the smoothed  $f$ . It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtain the *n-fold smoothed function*. Show how to generate the  $n$ -fold smoothed function of any given function using `smooth` and `repeated` from Exercise 1.43.

関数の *smoothing*(補間) という考えは信号処理において重要な概念である。 $f$  が関数であり  $dx$  がある小さな値である時、 $f$  の補間とは  $x$  における値が  $f(x - dx)$ ,  $f(x)$ , and  $f(x + dx)$  の平均である関数である。入力として  $f$  を計算する手続を取り、補間された  $f$  を計算する手続を返す手続 `smooth` を書け。時には関数の補正を繰り返し(つまり補間された関数をさらに補間することを繰り返す)、*n-fold smoothed function*( $n$  次畳み込み補間関数)を得ることには価値がある。任意の与えられた関数の  $n$  次畳み込み補間関数を Exercise 1.43 の `smooth` と `repeated` を用いてどのように生成するかを示せ。

**Exercise 1.45:** We saw in Section 1.3.3 that attempting to compute square roots by naively finding a fixed point of  $y \mapsto x/y$  does not converge, and that this can be fixed by average damping. The same method works for finding cube roots as fixed points of the average-damped  $y \mapsto x/y^2$ . Unfortunately, the process does not work for fourth roots—a single average damp is not enough to make a fixed-point search for  $y \mapsto x/y^3$  converge. On the other hand, if we average damp twice (i.e., use the average damp of the average damp of  $y \mapsto x/y^3$ ) the fixed-point search does converge. Do some experiments to determine how many average damps are required to compute  $n^{\text{th}}$  roots as a fixed-point search based upon repeated average damping of  $y \mapsto x/y^{n-1}$ . Use this to implement a simple procedure for computing  $n^{\text{th}}$  roots using `fixed-point`, `average-damp`, and the `repeated` procedure of Exercise 1.43. Assume that

any arithmetic operations you need are available as primitives.

Section 1.3.3 にて平方根を求める試みにおいて単純に  $y \mapsto x/y$  の不動点を探すのでは収束しないのを見た。この問題は平均減衰にて解決できた。同じ手法が平均減衰を行った  $y \mapsto x/y^2$  の不動点として立方根を求める場合においてもうまく行く。残念ながらこの処理は 4 乗根ではうまくいかない—单一の平均減衰は  $y \mapsto x/y^3$  の不動点検索を収束させるのに十分ではない。一方でもし平均減衰を二回行えば(すなわち  $y \mapsto x/y^3$ )の平均減衰の平均減衰)不動点検索は収束する。 $n$  乗根を  $y \mapsto x/y^{n-1}$  の平均減衰の繰り返しを基として不動点探索して求める場合に何回の平均減衰が必要であるかを試行せよ。この結果を用いて Exercise 1.43 の fixed-point, average-damp, repeated 手続を用いて  $n$  乗根を求める单一の手続を実装せよ。必要な数値演算はプリミティブとして存在すると仮定する。

**Exercise 1.46:** Several of the numerical methods described in this chapter are instances of an extremely general computational strategy known as *iterative improvement*. Iterative improvement says that, to compute something, we start with an initial guess for the answer, test if the guess is good enough, and otherwise improve the guess and continue the process using the improved guess as the new guess. Write a procedure `iterative-improve` that takes two procedures as arguments: a method for telling whether a guess is good enough and a method for improving a guess. `Iterative-improve` should return as its value a procedure that takes a guess as argument and keeps improving the guess until it is good enough. Rewrite the `sqrt` procedure of Section 1.1.7 and the `fixed-point` procedure of Section 1.3.3 in terms of `iterative-improve`.

この章にて説明されたいいくつかの数値解析手法は非常に汎用的な計算戦略であり *iterative improvement*(反復改善法)として知られている。反復改善法は何かを求めるために解の初期推測値から始め、推測値が十分に良いかをテストし、そうでなければ推測値を改善し、改善された推測値を新しい推測値として用いて処理を継続する。2 つの手続を引数として取る手続 `iterative-improve` を書け。1 つは推測値が十分に良いか判断する手続であり、もう 1 つは推測値を改善する手続である。`iterative-improve` は推測値を引数として取り、推測値を十分に良くなるまで繰り返す手続

をその値として返さなければならない。[Section 1.1.7](#)の `sqrt` 手続と [Section 1.3.3](#)の `fixed-point` 手続を `iterative-improve` を用いて書き直せ。

# 2

## Building Abstractions with Data

We now come to the decisive step of mathematical abstraction: we forget about what the symbols stand for. . .[The mathematician] need not be idle; there are many operations which he may carry out with these symbols, without ever having to look at the things they stand for.

私達は今、数学上の抽象化の重要なステップに到達しました。記号がどんな意味を持つのか忘れるのです。. . .[数学者]に遊んでいる暇はありません。これらの記号を用いて実行する演算はいくらでもあります、これらが何を意味するのか全く考へる必要無しに。

—Hermann Weyl, *The Mathematical Way of Thinking*

We concentrated in Chapter 1 on computational processes and on the role of procedures in program design. We saw how to use primitive data (numbers) and primitive operations (arithmetic operations), how to combine procedures to form compound procedures through composition, conditionals, and the use of parameters, and how to abstract procedures by using `define`. We saw that a procedure can be regarded as a pattern for the local evolution of a process, and we classified, reasoned about, and performed simple algorithmic analyses of some common patterns for processes as embodied in procedures. We also saw that higher-order procedures enhance the power of our language by en-

abling us to manipulate, and thereby to reason in terms of, general methods of computation. This is much of the essence of programming.

Chapter 1では演算処理とプログラム設計における手続の役割について集中しました。私達はプリミティブなデータ(数値)とプリミティブな命令(算術演算)の使い方、組み合わせ、条件式、パラメタの使用を通して複合手続を形成するための手続の結合方法、`define`を用いた抽象化の方法について学びました。また手続が処理の局地展開のためのパターンとして見なされ得ることを学びました。そして手続内で具体化されたプロセスに対するいくつかの共通パターンの、簡単なアルゴリズム上の解析を分類し、推論し、実行しました。また高階手続が、一般的な演算の手段を操作し、その結果を用いて推測することを可能にすることにより、私達の言語を強化することも学びました。これはプログラミングの本質の大部分です。

In this chapter we are going to look at more complex data. All the procedures in chapter 1 operate on simple numerical data, and simple data are not sufficient for many of the problems we wish to address using computation. Programs are typically designed to model complex phenomena, and more often than not one must construct computational objects that have several parts in order to model real-world phenomena that have several aspects. Thus, whereas our focus in chapter 1 was on building abstractions by combining procedures to form compound procedures, we turn in this chapter to another key aspect of any programming language: the means it provides for building abstractions by combining data objects to form *compound data*.

この章では私達はより複雑なデータについて目を向けることにします。第一章での全ての手続は単純な数値データを操作しましたが、単純なデータは私達が演算を用いて解決したいと願う多くの問題には不十分です。プログラムは一般的に複雑な事象をモデル化するために設計され、大抵の場合、複数の側面を持つ実世界の事象をモデル化するため、いくつかのパートを持つ演算対象のオブジェクトを構築せねばなりません。従って第一章での焦点は手續を組み合わせることで複合手続を形成し抽象化を構築することでしたが、この章では任意のプログラミング言語においてもう1つの鍵となる側面に向かいます。データオブジェクトを組み合わせ *compound data*(複合データ)を形成することによる、プログラミング言語が抽象化の構築に対して与える意味です。

Why do we want compound data in a programming language? For the same reasons that we want compound procedures: to elevate the conceptual level at which we can design our programs, to increase the modularity of our designs, and to enhance the expressive power of our language. Just as the ability to

define procedures enables us to deal with processes at a higher conceptual level than that of the primitive operations of the language, the ability to construct compound data objects enables us to deal with data at a higher conceptual level than that of the primitive data objects of the language.

私達はなぜプログラミング言語にてデータを組合せたいのでしょうか? 手続を組み合わせたいのと同じ理由のためです。プログラムを設計可能な概念上のレベルに持ち上げ、設計の部品化を進め、言語の表現力を拡張したいがためです。手続を定義する能力が、言語のプリミティブな命令のレベルよりもより高い概念のレベルにおいて処理を扱うことを可能にしてくれるのと同様に、複合データオブジェクトを構築できる能力は、言語のプリミティブなデータオブジェクトが与えるよりもより高い概念レベルのデータを扱うことを可能にします。

Consider the task of designing a system to perform arithmetic with rational numbers. We could imagine an operation `add-rat` that takes two rational numbers and produces their sum. In terms of simple data, a rational number can be thought of as two integers: a numerator and a denominator. Thus, we could design a program in which each rational number would be represented by two integers (a numerator and a denominator) and where `add-rat` would be implemented by two procedures (one producing the numerator of the sum and one producing the denominator). But this would be awkward, because we would then need to explicitly keep track of which numerators corresponded to which denominators. In a system intended to perform many operations on many rational numbers, such bookkeeping details would clutter the programs substantially, to say nothing of what they would do to our minds. It would be much better if we could “glue together” a numerator and denominator to form a pair—a *compound data object*—that our programs could manipulate in a way that would be consistent with regarding a rational number as a single conceptual unit.

分数を用いて数値演算を実行するシステム設計の課題を考えます。2つの分数を取りそれらの和を実行する命令 `add-rat` を想像します。単純なデータを用いる場合、分数は2つの整数として考えられます。分子と分母です。すると各分数が2つの整数(分子と分母)で表現されるプログラムの設計が可能です。そして `add-rat` は2つの手続(1つは和の分子を求め、もう1つは分母を求める)にて実装されるでしょう。しかしこれは不恰好です。それではどの分子がどの分母に関係するのか明示的に追跡をせねばなりません。多くの分数に對して多くの命令を実行する目的のシステムにおいては、そのような詳細な記

録はプログラムを大幅に散乱させるのみでなく、それらが私達の心にどんな影響を与えるかについては言うまでもありません。もし分子と分母をプログラムが分数を单一の概念上の単位として見做し静的な方法で扱えることができるペア—*compound data object*(複合データオブジェクト)—に“貼り合せ”られればずっと良くなることでしょう。

The use of compound data also enables us to increase the modularity of our programs. If we can manipulate rational numbers directly as objects in their own right, then we can separate the part of our program that deals with rational numbers per se from the details of how rational numbers may be represented as pairs of integers. The general technique of isolating the parts of a program that deal with how data objects are represented from the parts of a program that deal with how data objects are used is a powerful design methodology called *data abstraction*. We will see how data abstraction makes programs much easier to design, maintain, and modify.

複合データの使用はまたプログラムのモジュラリティ(部品化)を推進します。もし分数を独自に、直接それ自身をオブジェクトとして扱うことができれば、分数それ自体を扱うプログラムの一部を、分数が整数のペアとして表現されるだろうという詳細から分離することができます。データオブジェクトがどのように表現されるかを扱うプログラムの部分を、データオブジェクトがどのように利用されるかを扱うプログラムの部分から分離する一般的なテクニックは強力な設計手法であり *data abstraction*(データ抽象化)と呼ばれます。どのようにデータ抽象化がプログラムの設計、保守、変更をより簡単にするかをこれから学びます。

The use of compound data leads to a real increase in the expressive power of our programming language. Consider the idea of forming a “linear combination”  $ax + by$ . We might like to write a procedure that would accept  $a$ ,  $b$ ,  $x$ , and  $y$  as arguments and return the value of  $ax + by$ . This presents no difficulty if the arguments are to be numbers, because we can readily define the procedure

複合データの使用はプログラミング言語の表現力を実際に増加させます。“一次結合” $ax + by$ の形式化について考えてみて下さい。 $a$ ,  $b$ ,  $x$ ,  $y$ を引数として取り、 $ax + by$ の値を返す手続を書こうと思うかもしれません。これは引数が数値であるならば少しも難しいとは思えません。私達は既に手続を定義できます。

```
(define (linear-combination a b x y)
  (+ (* a x) (* b y)))
```

But suppose we are not concerned only with numbers. Suppose we would like to express, in procedural terms, the idea that one can form linear combinations whenever addition and multiplication are defined—for rational numbers, complex numbers, polynomials, or whatever. We could express this as a procedure of the form

しかし数値のみが対象ではないと考えてみましょう。手続の項目として加算と乗算が定義されているならば分数、複素数、多項式、その他何でも一次結合を形式化できるというアイデアを表現したいとします。これを以下の形式の手続として表現できるでしょう。

```
(define (linear-combination a b x y)
  (add (mul a x) (mul b y)))
```

where `add` and `mul` are not the primitive procedures `+` and `*` but rather more complex things that will perform the appropriate operations for whatever kinds of data we pass in as the arguments `a`, `b`, `x`, and `y`. The key point is that the only thing `linear-combination` should need to know about `a`, `b`, `x`, and `y` is that the procedures `add` and `mul` will perform the appropriate manipulations. From the perspective of the procedure `linear-combination`, it is irrelevant what `a`, `b`, `x`, and `y` are and even more irrelevant how they might happen to be represented in terms of more primitive data. This same example shows why it is important that our programming language provide the ability to manipulate compound objects directly: Without this, there is no way for a procedure such as `linear-combination` to pass its arguments along to `add` and `mul` without having to know their detailed structure.<sup>1</sup>

`add` と `mul` はプリミティブな手続 `+` と `*` ではなく、より複雑なものです。適切な操作を引数 `a`, `b`, `x`, `y` として与えたなどのような種類のデータに対しても行います。キーポイントは `linear-combination` が `a`, `b`, `x`, `y` について知らねば

---

<sup>1</sup>The ability to directly manipulate procedures provides an analogous increase in the expressive power of a programming language. For example, in Section 1.3.1 we introduced the `sum` procedure, which takes a procedure `term` as an argument and computes the sum of the values of `term` over some specified interval. In order to define `sum`, it is crucial that we be able to speak of a procedure such as `term` as an entity in its own right, without regard for how `term` might be expressed with more primitive operations. Indeed, if we did not have the notion of “a procedure,” it is doubtful that we would ever even think of the possibility of defining an operation such as `sum`. Moreover, insofar as performing the summation is concerned, the details of how `term` may be constructed from more primitive operations are irrelevant.

ならないことは手続 `add` とは `mul` が適切な操作を行うだろうことのみです。手続 `linear-combination` の視点からは `a`, `b`, `x`, `y` が何であるかは無関係であり、それらがどのようにによりプリミティブなデータを用いて表現されるのかについては尚更無関係です。この同じ例がなぜプログラミング言語が複合オブジェクトを直接操作する能力を提供することが重要であるのかを示しています。もしこれが無ければ、`linear-combination` のような手続に対してその引数を `add` と `mul` に向けてそれらの詳細な構造を知らずに渡す方法がありません。<sup>2</sup>

We begin this chapter by implementing the rational-number arithmetic system mentioned above. This will form the background for our discussion of compound data and data abstraction. As with compound procedures, the main issue to be addressed is that of abstraction as a technique for coping with complexity, and we will see how data abstraction enables us to erect suitable *abstraction barriers* between different parts of a program.

私達はこの章を先に触れられた分数の数値演算システムを実装すること始めます。これが複合データとデータ抽象化の議論の背景を形作ります。複合手続と同様に、解決すべき主な問題は複雑さを対処するための技術としての抽象化でありどのようにデータ抽象化が適切な *abstraction barriers*(抽象化バリア)を異なるプログラムの部分の間に構築することを可能にするかについて学びます。

We will see that the key to forming compound data is that a programming language should provide some kind of “glue” so that data objects can be combined to form more complex data objects. There are many possible kinds of glue. Indeed, we will discover how to form compound data using no special “data” operations at all, only procedures. This will further blur the distinction between “procedure” and “data,” which was already becoming tenuous toward the end of chapter 1. We will also explore some conventional techniques for representing sequences and trees. One key idea in dealing with compound data is the notion of *closure*—that the glue we use for combining data objects should allow us to combine not only primitive data objects, but compound data objects

---

<sup>2</sup>手続を直接操作する能力はプログラミング言語の表現力に対して類似の増強を与えます。例えばSection 1.3.1において `sum` 手続を紹介しましたが、これは手続 `term` を引数として取り、ある指定した区間の `term` の値の和を求めました。`sum` を定義するためには `term` のような手続を、`term` がよりプリミティブな命令にてどのように表現されているのかに関わらず、それ自身の要素として表現できることが重要でした。実際に、もし“手続”という概念が無ければ `sum` のような命令の定義の可能性について考えつくことすら疑わしいことだったでしょう。その上、加算の実行を考慮する範囲では `term` がどのようにによりプリミティブな命令から構築され得るのかの詳細は無関係なのです。

as well. Another key idea is that compound data objects can serve as *conventional interfaces* for combining program modules in mix-and-match ways. We illustrate some of these ideas by presenting a simple graphics language that exploits closure.

複合データを形成するための鍵はプログラミング言語はある種の“糊”を提供しなければならないことであり、そうすることでデータオブジェクトはより複雑なデータオブジェクトを形成するために組み合わせることが可能になります。多くの有力な種類の糊が存在します。実際に、全く特別ではない手続のみの“データ”操作を用いて複合データをどのように形成するかについて発見するでしょう。これは第一章の終りに向かい既に希薄となっていた“手続”と“データ”的区別をよりボカすことになるでしょう。また列と木を表現するいくつかの保守的な技術についても探検します。複合データを扱う場合の鍵となる考え方の1つはclosure(クロージャ)の概念です—そのデータオブジェクトを組み合わせるのに用いる糊はプリミティブなデータオブジェクトのみではなく、複合データオブジェクトもまた組み合わせられなければなりません。もう1つの鍵となる考えは複合データオブジェクトは種々様々な方法でプログラムモジュールを組み立てるための *conventional interfaces*(慣習的インターフェイス)の役を演じることができます。これらのアイデアのいくつかについてはクロージャを用いる簡単なグラフィック言語を与えることで説明します。

We will then augment the representational power of our language by introducing *symbolic expressions*—data whose elementary parts can be arbitrary symbols rather than only numbers. We explore various alternatives for representing sets of objects. We will find that, just as a given numerical function can be computed by many different computational processes, there are many ways in which a given data structure can be represented in terms of simpler objects, and the choice of representation can have significant impact on the time and space requirements of processes that manipulate the data. We will investigate these ideas in the context of symbolic differentiation, the representation of sets, and the encoding of information.

次に *symbolic expressions*(記号表現)—その基本的な部分は任意の記号であり数字のみではないデータ—を紹介することで言語の具象的な力を増補します。オブジェクトの集合を表現するための様々な代替方法について探検します。与えられた数値関数が多くの異なる演算処理により計算され得るのと同様に、与えられたデータ構造が多くの方法にてより単純なオブジェクトを用いて表現され得ること、表現の選択がデータを扱う処理の時間と記憶域の要件に対し重大な影響を与えることについて発見するでしょう。記号微分、集合の表現、情

報符号化のコンテキストにてこれらの考えについて調査します。

Next we will take up the problem of working with data that may be represented differently by different parts of a program. This leads to the need to implement *generic operations*, which must handle many different types of data. Maintaining modularity in the presence of generic operations requires more powerful abstraction barriers than can be erected with simple data abstraction alone. In particular, we introduce *data-directed programming* as a technique that allows individual data representations to be designed in isolation and then combined *additively* (i.e., without modification). To illustrate the power of this approach to system design, we close the chapter by applying what we have learned to the implementation of a package for performing symbolic arithmetic on polynomials, in which the coefficients of the polynomials can be integers, rational numbers, complex numbers, and even other polynomials.

次にプログラムの異なる部分において異なって表現され得るデータを用いて処理する問題にとりかかります。これが*generic operations*(総称命令)の実装の必要性へと導きます。総称命令は多くの異なるデータの型を扱わなければなりません。総称命令の存在時における部品化の保守は単純なデータ抽象化のみにより構築可能な場合に比べて、より強力な抽象化バリアを、必要とします。具体的には*data-directed programming*(データ指示プログラミング)を個別のデータ表現に対し分離した設計と付加的(*additively*)に(つまり変更無しに)組み合わせることを可能にする技術として紹介します。システム設計に対するこのアプローチを説明するために、多項式上の記号演算の実行向けパッケージを実装するために私達が学んだことを適用することでこの章を終わります。その実装の中では多項式の係数は整数、分数、複素数、さらには多項式にもなり得ます。

## 2.1 Introduction to Data Abstraction

In Section 1.1.8, we noted that a procedure used as an element in creating a more complex procedure could be regarded not only as a collection of particular operations but also as a procedural abstraction. That is, the details of how the procedure was implemented could be suppressed, and the particular procedure itself could be replaced by any other procedure with the same overall behavior. In other words, we could make an abstraction that would separate the way the procedure would be used from the details of how the procedure would be implemented in terms of more primitive procedures. The analogous notion for

compound data is called *data abstraction*. Data abstraction is a methodology that enables us to isolate how a compound data object is used from the details of how it is constructed from more primitive data objects.

Section 1.1.8においてより複雑な手続を作成する要素として使われる手続は特定の命令の集合としてのみでなく、手続の抽象化としても見做されることを伝えました。その手続がどのように実装されたのかの詳細は抑制可能であり、特定の手続それ自身は相対的に同じ振舞を持つ任意の他の手続で置き換えられます。言い換えれば、手続がどのように使われるかをその手続がどのようによりプリミティブな手続を用いて実装されたかの詳細から分離する抽象化を作成できます。複合データのための類似の概念は*data abstraction*(データ抽象化)と呼ばれます。データ抽象化はどのように複合データオブジェクトが使用されるかをどれがどのようによりプリミティブなデータオブジェクトから構築されたのかの詳細から分離することを可能にする方法論(methodology, メソドロジ)です。

The basic idea of data abstraction is to structure the programs that are to use compound data objects so that they operate on “abstract data.” That is, our programs should use data in such a way as to make no assumptions about the data that are not strictly necessary for performing the task at hand. At the same time, a “concrete” data representation is defined independent of the programs that use the data. The interface between these two parts of our system will be a set of procedures, called *selectors* and *constructors*, that implement the abstract data in terms of the concrete representation. To illustrate this technique, we will consider how to design a set of procedures for manipulating rational numbers.

データ抽象化の基本的なアイデアは複合データオブジェクトを使用するためのプログラムを構造化することで“抽象データ”上で操作を行うことです。それはつまり、私達のプログラムが手元でタスクを実行するためには厳密には必要と言えないデータに関する想定を一切持たないような方法でデータを利用しなければいけないということです。同時に、“具体的”なデータ表現はデータを利用するプログラムとは独立に定義されます。システムにおけるこれらの2つのパートの間のインターフェイスは手続の集合であり、*selectors*(セレクタ)と*constructors*(コンストラクタ)と呼ばれ、抽象データを具体的な表現を用いて実装します。このテクニックを説明するために、分数を扱う手続の集合をどのように設計するかについてこれから考えます。

### 2.1.1 Example: Arithmetic Operations for Rational Numbers

Suppose we want to do arithmetic with rational numbers. We want to be able to add, subtract, multiply, and divide them and to test whether two rational numbers are equal.

分数を用いて数値演算を行いたいとします。足し算、引き算、かけ算、割り算をそれらに対して行い 2 つの分数が等しいかテストします。

Let us begin by assuming that we already have a way of constructing a rational number from a numerator and a denominator. We also assume that, given a rational number, we have a way of extracting (or selecting) its numerator and its denominator. Let us further assume that the constructor and selectors are available as procedures:

分子と分母から分数を構築する方法を既に持っていると仮定することから始めましょう。分数を与えられた時にその分子と分母を抽出する(または選択(セレクト)する)方法を持っているとも仮定します。さらにコンストラクタとセレクタが手続として存在すると仮定します。

- `(make-rat  $\langle n \rangle$   $\langle d \rangle$ )` returns the rational number whose numerator is the integer  $\langle n \rangle$  and whose denominator is the integer  $\langle d \rangle$ .  
`(make-rat  $\langle n \rangle$   $\langle d \rangle$ )` は分子が整数  $\langle n \rangle$  であり、かつ分母が整数  $\langle d \rangle$  である分数を返す。
- `(numer  $\langle x \rangle$ )` returns the numerator of the rational number  $\langle x \rangle$ .  
`(numer  $\langle x \rangle$ )` は分数  $\langle x \rangle$  の分子を返す。
- `(denom  $\langle x \rangle$ )` returns the denominator of the rational number  $\langle x \rangle$ .  
`(denom  $\langle x \rangle$ )` は分数  $\langle x \rangle$  の分母を返す。

We are using here a powerful strategy of synthesis: *wishful thinking*. We haven't yet said how a rational number is represented, or how the procedures `numer`, `denom`, and `make-rat` should be implemented. Even so, if we did have these three procedures, we could then add, subtract, multiply, divide, and test equality by using the following relations:

ここで統合のための強力な戦略、*wishful thinking*(希望的観測)を用います。私達はまだ分数がどのように表現されるのか、または手続 `numer`, `denom`, `make-rat` がどのように実装されるべきであるのかについて語っていません。そうであっても、もし私達がこれら 3 つの手続を持っているのならば、足し算、引き

算、かけ算、割り算、等値テストを以下の関係性を用いて行うことができるでしょう。

$$\begin{aligned}\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}, \\ \frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}, \\ \frac{n_1}{d_1} \cdot \frac{n_2}{d_2} &= \frac{n_1 n_2}{d_1 d_2}, \\ \frac{n_1/d_1}{n_2/d_2} &= \frac{n_1 d_2}{d_1 n_2}, \\ \frac{n_1}{d_1} &= \frac{n_2}{d_2} \quad \text{if and only if } n_1 d_2 = n_2 d_1.\end{aligned}$$

We can express these rules as procedures:

これらのルールを手続として表現できます。

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
                (* (numer y) (denom x)))
             (* (denom x) (denom y)))))

(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
                (* (numer y) (denom x)))
             (* (denom x) (denom y)))))

(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y)))))

(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y)))))

(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

Now we have the operations on rational numbers defined in terms of the selector and constructor procedures `numer`, `denom`, and `make-rat`. But we haven't yet defined these. What we need is some way to glue together a numerator and a denominator to form a rational number.

これでセレクタとコンストラクタの手続である `numer`, `denom`, `make-rat` を用いて分数の操作を定義できました。必要な物は分子と分母を貼り合せて分数を形成する何らかの方法です。

## Pairs

To enable us to implement the concrete level of our data abstraction, our language provides a compound structure called a *pair*, which can be constructed with the primitive procedure `cons`. This procedure takes two arguments and returns a compound data object that contains the two arguments as parts. Given a pair, we can extract the parts using the primitive procedures `car` and `cdr`.<sup>3</sup> Thus, we can use `cons`, `car`, and `cdr` as follows:

データ抽象化の具体的なレベルを実装できるようになるために、私達の言語は *pair*(ペア) と呼ばれる複合構造を提供します。それはプリミティブな手続 `cons` を用いて構築できます。この手続は 2 つの引数を取り、2 つの引数を部分として持つ複合データオブジェクトを返します。ペアを与えられた時、プリミティブな手續 `car` と `cdr` を用いてその部分を抽出することができます。<sup>4</sup> 従って、`cons`, `car`, `cdr` を以下のように使用できます。

```
(define x (cons 1 2))
(car x)
1
(cdr x)
2
```

Notice that a pair is a data object that can be given a name and manipulated, just like a primitive data object. Moreover, `cons` can be used to form pairs whose elements are pairs, and so on:

---

<sup>3</sup>The name `cons` stands for “construct.” The names `car` and `cdr` derive from the original implementation of Lisp on the IBM 704. That machine had an addressing scheme that allowed one to reference the “address” and “decrement” parts of a memory location. `Car` stands for “Contents of Address part of Register” and `cdr` (pronounced “could-er”) stands for “Contents of Decrement part of Register.”

<sup>4</sup>`cons` という名前は “construct” によります。`car` と `cdr` という名前は IBM 704 上でのオリジナルの Lisp 実装に由来します。このマシンはアドレッシングの仕組みとしてメモリロケーションの “アドレス” と “デクリメント” の部分を参照可能でした。`car` は “Contents of Address part of Register”(レジスタのアドレス部分の中身) を表し、`cdr`(“クダー” と読みます) は “Contents of Decrement part of Register.”(レジスタのデクリメント部分の中身) を表します。

ペアは名前を与えることができ、プリミティブなデータオブジェクトと同様に扱うことができるデータオブジェクトです。さらに `cons` はその要素がペアであるペアや、その繰り返しを作ることも可能です。

```
(define x (cons 1 2))
(define y (cons 3 4))
(define z (cons x y))
(car (car z))
1
(car (cdr z))
3
```

In Section 2.2 we will see how this ability to combine pairs means that pairs can be used as general-purpose building blocks to create all sorts of complex data structures. The single compound-data primitive `pair`, implemented by the procedures `cons`, `car`, and `cdr`, is the only glue we need. Data objects constructed from pairs are called *list-structured* data.

Section 2.2において、ペアを組み立てるこの能力が、全ての種類の複雑なデータ構造を作成するために汎用目的構築ブロックとしてペアが利用可能であることに対し、どのような意味を持つのかについて学びます。`cons`, `car`, `cdr` により実装された単一の複合データプリミティブペアが私達が必要とするただ1つの糊です。ペアから構築されたデータオブジェクトは *list-structured*(リスト構造化) データと呼ばれます。

## Representing rational numbers

Pairs offer a natural way to complete the rational-number system. Simply represent a rational number as a pair of two integers: a numerator and a denominator. Then `make-rat`, `numer`, and `denom` are readily implemented as follows:

ペアは分数システムを仕上げるための自然な方法を提供します。単純に分数を2つの整数、分子と分母のペアとして表現します。そして `make-rat`, `numer`, `denom` は簡単に次のように実装することができます。<sup>5</sup>

---

<sup>5</sup>Another way to define the selectors and constructor is

セレクタとコンストラクタを定義するもう1つの実装として次が上げられます。

```
(define make-rat cons)
(define numer car)
```

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
```

Also, in order to display the results of our computations, we can print rational numbers by printing the numerator, a slash, and the denominator:<sup>6</sup>

また演算結果を表示するために、分数を分子、スラッシュ、分母で表示することにします。<sup>7</sup>

```
(define (print-rat x)
  (newline)
  _____
(define denom cdr)
```

The first definition associates the name `make-rat` with the value of the expression `cons`, which is the primitive procedure that constructs pairs. Thus `make-rat` and `cons` are names for the same primitive constructor.

最初の定義は名前 `make-rat` を式 `cons` の値に関連付けます。それはペアを構築するプリミティブな手続です。従って `make-rat` と `cons` は同じプリミティブなコンストラクタになります。

Defining selectors and constructors in this way is efficient: Instead of `make-rat` calling `cons`, `make-rat` is `cons`, so there is only one procedure called, not two, when `make-rat` is called. On the other hand, doing this defeats debugging aids that trace procedure calls or put breakpoints on procedure calls: You may want to watch `make-rat` being called, but you certainly don't want to watch every call to `cons`.

セレクタとコンストラクタをこのように定義するのは効率が良いです。`make-rat` が `cons` を *calling*(呼び出す) 代わりに、`make-rat` が `cons` で *is*(ある) ためです。そのため `make-rat` が呼ばれた時に 2つではなく、1つの手続が呼ばれるだけになります。しかし一方で、これを行うことは手続呼出のトレースや手続呼出に対するブレイクポイントの設定に対するデバッグ上の手助けを無効にしてしまいます。あなたは `make-rat` の呼出を見たくなるのであって、`cons` への全ての呼出を見たい訳ではないからです。

We have chosen not to use this style of definition in this book.

この本ではこの定義スタイルを使用しないことにしました。

<sup>6</sup>`Display` is the Scheme primitive for printing data. The Scheme primitive `newline` starts a new line for printing. Neither of these procedures returns a useful value, so in the uses of `print-rat` below, we show only what `print-rat` prints, not what the interpreter prints as the value returned by `print-rat`.

<sup>7</sup>`display` はデータを表示する Scheme のプリミティブです。Scheme のプリミティブである `newline` は表示を新しい行から始めます。これらの手続のどちらも意味のある値は返しません。そのため下記の `print-rat` 内での使用においては `print-rat` が表示する物のみを示し、インタプリタが `print-rat` の返り値として表示する物は示していません。

```
(display (numer x))
(display "/")
(display (denom x)))
```

Now we can try our rational-number procedures:

これで分数手続を試すことができます。

```
(define one-half (make-rat 1 2))
(print-rat one-half)
1/2
(define one-third (make-rat 1 3))
(print-rat (add-rat one-half one-third))
5/6
(print-rat (mul-rat one-half one-third))
1/6
(print-rat (add-rat one-third one-third))
6/9
```

As the final example shows, our rational-number implementation does not reduce rational numbers to lowest terms. We can remedy this by changing `make-rat`. If we have a `gcd` procedure like the one in Section 1.2.5 that produces the greatest common divisor of two integers, we can use `gcd` to reduce the numerator and the denominator to lowest terms before constructing the pair:

最後の例が示すとおり、私達の分数実装は分数を最も小さな項に約分しません。これを `make-rat` を変更することで改良できます。もしSection 1.2.5で扱った 2 つの整数の最大公約数を生成する `gcd` 手続を持っていれば、`gcd` を用いて分子と分母を最小の項に、ペアを構築する前に縮小することができます。

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
```

Now we have これで次の希望した結果を得ます。

```
(print-rat (add-rat one-third one-third))
2/3
```

as desired. This modification was accomplished by changing the constructor `make-rat` without changing any of the procedures (such as `add-rat` and `mul-rat`) that implement the actual operations.

この変更は (`add-rat` や `mul-rat` のような) 実際の命令を実装する他の手続の変更無しに、コンストラクタ `make-rat` を変更することで達成されました。

**Exercise 2.1:** Define a better version of `make-rat` that handles both positive and negative arguments. `Make-rat` should normalize the sign so that if the rational number is positive, both the numerator and denominator are positive, and if the rational number is negative, only the numerator is negative.

正と負の両方の引数を扱えるより良い版の `make-rat` を定義せよ。`make-rat` は符号の正常化を行わなければならない。従ってもし分数が正であれば分子と分母の両方が正であるし、もし分数が負であれば分子のみが負でなければならぬ。

### 2.1.2 Abstraction Barriers

Before continuing with more examples of compound data and data abstraction, let us consider some of the issues raised by the rational-number example. We defined the rational-number operations in terms of a constructor `make-rat` and selectors `numer` and `denom`. In general, the underlying idea of data abstraction is to identify for each type of data object a basic set of operations in terms of which all manipulations of data objects of that type will be expressed, and then to use only those operations in manipulating the data.

さらなる複合データとデータ抽象化の例を続ける前に、分数の例にて持ち上がったいくつかの問題について考えてみましょう。私達は分数操作をコンストラクタ `make-rat` とセレクタ `numer` と `denom` を用いて定義しました。一般的にデータ抽象化の基となる考えはデータオブジェクトの各型に対し、その型のデータオブジェクトの全ての操作が表される命令を用いて、命令の基本的な集合を判断し、そのデータを操作する時にそれらの命令のみを用いることです。

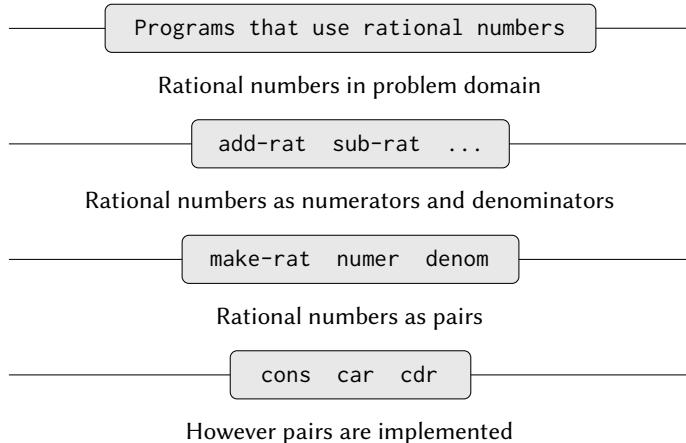
We can envision the structure of the rational-number system as shown in [Figure 2.1](#). The horizontal lines represent *abstraction barriers* that isolate different “levels” of the system. At each level, the barrier separates the programs (above) that use the data abstraction from the programs (below) that implement the data abstraction. Programs that use rational numbers manipulate them solely in terms of the procedures supplied “for public use” by the rational-number package: `add-rat`, `sub-rat`, `mul-rat`, `div-rat`, and `equal-rat?`. These, in turn, are implemented solely in terms of the constructor and

selectors `make-rat`, `numer`, and `denom`, which themselves are implemented in terms of pairs. The details of how pairs are implemented are irrelevant to the rest of the rational-number package so long as pairs can be manipulated by the use of `cons`, `car`, and `cdr`. In effect, procedures at each level are the interfaces that define the abstraction barriers and connect the different levels.

私達はFigure 2.1にて示された分数システムの構造を想像することができます。水平線は*abstraction barriers*(抽象化バリア)を表現し、システムの異なる“レベル”を分離します。各レベルではバリアはデータ抽象化を利用する(上側の)プログラムをデータ抽象化を実装する(下側の)プログラムから分離します。分数を利用するプログラムはもっぱら分数パッケージにより“公用向け”に提供された手続を用いて分数を操作します。それら手続とは `add-rat`, `sub-rat`, `mul-rat`, `div-rat`, それに `equal-rat?` です。これらは順に、もっぱらコンストラクタとセレクタである `make-rat`, `numer`, `denom` を用いて実装されます。この3つはペアを用いて実装されます。ペアがどのように実装されているかの詳細はペアが `cons`, `car`, `cdr` の使用により操作できる限りにおいては分数パッケージの他の物に取っては重要ではありません。実質的に、各レベルにおける手続は抽象化バリアを定義するインターフェイスであり、異なるレベルを接続します。

This simple idea has many advantages. One advantage is that it makes programs much easier to maintain and to modify. Any complex data structure can be represented in a variety of ways with the primitive data structures provided by a programming language. Of course, the choice of representation influences the programs that operate on it; thus, if the representation were to be changed at some later time, all such programs might have to be modified accordingly. This task could be time-consuming and expensive in the case of large programs unless the dependence on the representation were to be confined by design to a very few program modules.

この単純な考えは多くの利点を持ちます。1つの利点はプログラムの保守と変更をより簡単にすることです。任意の複雑なデータ構造が、プログラミング言語により提供されるプリミティブなデータ構造を用いて多彩な方法で表現されます。もちろん、表現の選択がその上で操作を行うプログラムに影響を与えます。従ってもし表現がある程度後に変更された場合、全ての当該プログラムはそれに応じて変更されなければなりません。この作業は大きなプログラムの場合においては表現上の依存が設計によりとても少ないプログラムモジュールに対してのみに制限されていなければ時間のかかる高コストな物に成り得ます。



**Figure 2.1:** Data-abstraction barriers in the rational-number package.

For example, an alternate way to address the problem of reducing rational numbers to lowest terms is to perform the reduction whenever we access the parts of a rational number, rather than when we construct it. This leads to different constructor and selector procedures:

例として、分数を最小の項へと約分する問題の解法の代替法には、分数を組み立てた時でなく、分数のパーツにアクセスする度に約分を実行する方法があります。これは異なるコンストラクタとセレクタ手続に導きます。

```
(define (make-rat n d) (cons n d))
(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g)))
(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g)))
```

The difference between this implementation and the previous one lies in when

we compute the `gcd`. If in our typical use of rational numbers we access the numerators and denominators of the same rational numbers many times, it would be preferable to compute the `gcd` when the rational numbers are constructed. If not, we may be better off waiting until access time to compute the `gcd`. In any case, when we change from one representation to the other, the procedures `add-rat`, `sub-rat`, and so on do not have to be modified at all.

この実装と依然の実装との間の違いはいつ `gcd` を求めるかにあります。私達の典型的な分数の使用においては、同じ分数の分子と分母に何度もアクセスする場合、分数が組み立てられる時に `gcd` を求めるほうが好ましいです。そうでなければ `gcd` を求めるのはアクセスする時まで待ったほうが良いかもしれません。どちらの場合でも、一方の表現からもう一方の表現へと変更する場合、手続、`add-rat`, `sub-rat`, その他は全く変更する必要がありません。

Constraining the dependence on the representation to a few interface procedures helps us design programs as well as modify them, because it allows us to maintain the flexibility to consider alternate implementations. To continue with our simple example, suppose we are designing a rational-number package and we can't decide initially whether to perform the `gcd` at construction time or at selection time. The data-abstraction methodology gives us a way to defer that decision without losing the ability to make progress on the rest of the system.

表現上の依存対象を少ないインターフェイス手続に制約することはプログラムの設計と共にそれらの変更をも手助けします。なぜなら代替的な実装を考えるための柔軟性を保つことを可能にするためです。私達の簡単な例で続けるために、私達は分数パッケージを設計中で、早期に `gcd` を構築時と選択時のどちらで実行するか決められないと想像して下さい。データ抽象化メソドロジはその決定をシステムの他の部分上の進行の可能性を失わせずに決定を遅らせる方法を与えます。

**Exercise 2.2:** Consider the problem of representing line segments in a plane. Each segment is represented as a pair of points: a starting point and an ending point. Define a constructor `make-segment` and selectors `start-segment` and `end-segment` that define the representation of segments in terms of points. Furthermore, a point can be represented as a pair of numbers: the *x* coordinate and the *y* coordinate. Accordingly, specify a constructor `make-point` and selectors `x-point` and `y-point` that define this representation. Finally, using your selectors and constructors, define a procedure

`midpoint-segment` that takes a line segment as argument and returns its midpoint (the point whose coordinates are the average of the coordinates of the endpoints). To try your procedures, you'll need a way to print points:

平面上の線分を表現する問題について考える。各線分は点のペアにて表現する。始点と終点である。コンストラクタ `make-segment` とセレクタ `start-segment` と `end-segment` を定義せよ。それらは点を用いて線分の表現を定義する。さらに点は数値のペアにて表現できる。 $x$  座標と  $y$  座標である。それに沿ってこの表現を定義するコンストラクタ `make-point` とセレクタ `x-point` を `y-point` を定めよ。最後に、セレクタとコンストラクタを用いて引数として線分を取りその中点(その座標が両端点の座標の平均である点)を返す手続 `midpoint-segment` を定義せよ。あなたの手続をテストするためには以下の点を表示する方法が必要だろう。

```
(define (print-point p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display ")"))
```

**Exercise 2.3:** Implement a representation for rectangles in a plane.  
(Hint: You may want to make use of [Exercise 2.2](#).) In terms of your constructors and selectors, create procedures that compute the perimeter and the area of a given rectangle. Now implement a different representation for rectangles. Can you design your system with suitable abstraction barriers, so that the same perimeter and area procedures will work using either representation?

平面上の長方形のための表現を実装せよ。(ヒント：[Exercise 2.2](#)を利用したいだろう。) コンストラクタとセレクタを利用して、与えられた長方形の周辺の長さと面積を求める手続を作れ。適切な抽象化バリアーを用いてどんな表現を用いても同じ周辺長と面積の手続が働くよう、あなたのシステムを設計できるだろうか？

### 2.1.3 What Is Meant by Data?

We began the rational-number implementation in [Section 2.1.1](#) by implementing the rational-number operations `add-rat`, `sub-rat`, and so on in terms of three unspecified procedures: `make-rat`, `numer`, and `denom`. At that point, we could think of the operations as being defined in terms of data objects—numerators, denominators, and rational numbers—whose behavior was specified by the latter three procedures.

[Section 2.1.1](#)にて分数実装を分数演算 `add-rat`, `sub-rat`, その他を 3 つの定められていない手続、`make-rat`, `numer`, `denom` を用いて実装することから始めました。その時点では命令はデータオブジェクト—分子、分母と分数を用いて定義されると考えることができました。データオブジェクトの振舞は後者の 3 つの手続により指定されました。

But exactly what is meant by *data*? It is not enough to say “whatever is implemented by the given selectors and constructors.” Clearly, not every arbitrary set of three procedures can serve as an appropriate basis for the rational-number implementation. We need to guarantee that, if we construct a rational number  $x$  from a pair of integers  $n$  and  $d$ , then extracting the `numer` and the `denom` of  $x$  and dividing them should yield the same result as dividing  $n$  by  $d$ . In other words, `make-rat`, `numer`, and `denom` must satisfy the condition that, for any integer  $n$  and any non-zero integer  $d$ , if  $x$  is `(make-rat n d)`, then

しかし *data*(データ) とは正確には何を意味するのでしょうか。“与えられたセレクタとコンストラクタにより実装された物全て”と言うのみでは十分ではありません。明かに 3 つの手続の任意の集合全てが分数実装に対する適切な基準としての役割を果せる訳ではありません。もし分数  $x$  を整数のペア  $n$  と  $d$  から組み立てた場合、 $x$  の `numer` と `denom` の抽出しそれらを割ることは、 $n$  を  $d$  で割るのと同じ結果になることを保証せねばなりません。言い替えれば、`make-rat`, `numer`, `denom` は任意の整数  $n$  と零でない整数  $d$  に対しもし  $x$  が `(make-rat n d)` である時、その場合以下の条件を満たさなければなりません。

$$\frac{(\text{numer } x)}{(\text{denom } x)} = \frac{n}{d}.$$

In fact, this is the only condition `make-rat`, `numer`, and `denom` must fulfill in order to form a suitable basis for a rational-number representation. In general, we can think of data as defined by some collection of selectors and constructors, together with specified conditions that these procedures must fulfill in order to

be a valid representation.<sup>8</sup>

実際にこれが `make-rat`, `numer`, `denom` が分数表現のための適切な基準を形成するために満たさなければならないただ 1 つの条件です。一般的に、私達はデータをセレクタとコンストラクタのある集合と共に、これらの手続が有効な表現となるために満たさなければならない制約により定義されると考えることができます。<sup>9</sup>

This point of view can serve to define not only “high-level” data objects, such as rational numbers, but lower-level objects as well. Consider the notion of a pair, which we used in order to define our rational numbers. We never actually said what a pair was, only that the language supplied procedures `cons`, `car`,

---

<sup>8</sup>Surprisingly, this idea is very difficult to formulate rigorously. There are two approaches to giving such a formulation. One, pioneered by C. A. R. Hoare (1972), is known as the method of *abstract models*. It formalizes the “procedures plus conditions” specification as outlined in the rational-number example above. Note that the condition on the rational-number representation was stated in terms of facts about integers (equality and division). In general, abstract models define new kinds of data objects in terms of previously defined types of data objects. Assertions about data objects can therefore be checked by reducing them to assertions about previously defined data objects. Another approach, introduced by Zilles at MIT, by Goguen, Thatcher, Wagner, and Wright at IBM (see Thatcher et al. 1978), and by Guttag at Toronto (see Guttag 1977), is called *algebraic specification*. It regards the “procedures” as elements of an abstract algebraic system whose behavior is specified by axioms that correspond to our “conditions,” and uses the techniques of abstract algebra to check assertions about data objects. Both methods are surveyed in the paper by Liskov and Zilles (1975).

<sup>9</sup>意外にもこの考えは厳格に形式化することがとても難しいのです。そのような形式化を与える試みは 2 つあります。1 つは C. A. R. Hoare (1972) により開拓され、*abstract models*(抽象モデル) として知られています。“手続プラス制約”的仕様を上の分数の例内で概説されたように形式化します。分数表現上の条件は整数に関する事実(等値関係と除算)を用いて規定されています。一般的に抽象モデルは新しい種類のデータオブジェクトを以前に定義されたデータオブジェクトの型を用いて定義します。従ってデータオブジェクトに関する成立条件はそれらを以前に定義されたデータオブジェクトに関する成立条件へと還元していくことでチェックできます。もう 1 つの試みは MIT の Zilles と IBM の Goguen, Thatcher, Wagner, Wright により紹介され (Thatcher et al. 1978 を参照)、またトロント大学の Guttag により紹介されました。(Guttag 1977 を参照)。その試みは“手続”を抽象代数システムの要素と見做し、その振舞は“条件”に相当する公理により指定されました。そして抽象代数のテクニックを用いてデータオブジェクトに関する成立条件をチェックしました。両者の手法が Liskov and Zilles (1975) により論文として調査されています。

and `cdr` for operating on pairs. But the only thing we need to know about these three operations is that if we glue two objects together using `cons` we can retrieve the objects using `car` and `cdr`. That is, the operations satisfy the condition that, for any objects `x` and `y`, if `z` is  $(\text{cons } x \ y)$  then  $(\text{car } z)$  is `x` and  $(\text{cdr } z)$  is `y`. Indeed, we mentioned that these three procedures are included as primitives in our language. However, any triple of procedures that satisfies the above condition can be used as the basis for implementing pairs. This point is illustrated strikingly by the fact that we could implement `cons`, `car`, and `cdr` without using any data structures at all but only using procedures. Here are the definitions:

この視点は分数のような“高階データオブジェクト”のみを定義するのではなく、より低いレベルのオブジェクトの定義も提供することができます。私達が分数を定義するために使用したペアの概念について考えてみます。私達はまだペアとは実際には何であるのか述べていません。言語が手続 `cons`, `car`, `cdr` をペア上の命令として提供するとのみ説明しています。しかしこれら 3 つの命令について知らなければいけないことはもし私達が 2 つのオブジェクトを `cons` を用いて貼り合わせた時、`car` と `cdr` を用いてそれらのオブジェクトを取得することができることのみです。つまり、それらの命令は任意のオブジェクト `x` と `y` に対し、もし `z` が  $(\text{cons } x \ y)$  であるなら  $(\text{car } z)$  は `x` であり、 $(\text{cdr } z)$  は `y` であるという制約を満たしています。実際に、これらの 3 つの手続は言語にプリミティブとして含まれていることについて既に述べました。しかし、上記の制約を満たす任意の 3 つの手続ならペアを実装するための基盤として使用することができます。この点は私達が `cons`, `car`, `cdr` をどんなデータ構造も全く利用せずに、しかし手続のみを用いて実装できることにより、著しく説明されます。これがその定義です。

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1: CONS" m))))
  dispatch)
(define (car z) (z 0))
(define (cdr z) (z 1))
```

This use of procedures corresponds to nothing like our intuitive notion of what data should be. Nevertheless, all we need to do to show that this is a valid way

to represent pairs is to verify that these procedures satisfy the condition given above.

この手続の使用はデータが何であるべきかという私達の直感的概念のような物には全く関係しません。それでもなお、これがペアを表現するのに有効な方法であると示すのに必要なこと全てはこれらの手続が上で与えられた制約を満たすことです。

The subtle point to notice is that the value returned by `(cons x y)` is a procedure—namely the internally defined procedure `dispatch`, which takes one argument and returns either `x` or `y` depending on whether the argument is 0 or 1. Correspondingly, `(car z)` is defined to apply `z` to 0. Hence, if `z` is the procedure formed by `(cons x y)`, then `z` applied to 0 will yield `x`. Thus, we have shown that `(car (cons x y))` yields `x`, as desired. Similarly, `(cdr (cons x y))` applies the procedure returned by `(cons x y)` to 1, which returns `y`. Therefore, this procedural implementation of pairs is a valid implementation, and if we access pairs using only `cons`, `car`, and `cdr` we cannot distinguish this implementation from one that uses “real” data structures.

注意すべき微細な点は `(cons x y)` により返される値は手続—すなわち内部で定義された手続 `dispatch` であることです。それが1つの引数を取り `x` か `y` のどちらかを引数が 0 であるか 1 であるかに従って返します。相応して、`(car z)` は `z` を 0 に適用します。故にもし `z` が `(cons x y)` により作られた手続であるのなら、`z` を 0 に適用すれば `x` を返します。従って、`(car (cons x y))` が希望通りに `x` を返すことを示しました。同様に `(cdr (cons x y))` は `(cons x y)` の返り値としての手続を 1 に適用し、`y` を返します。従ってこのペアの手続としての実装は有効な実装であり、もし私達が `cons`, `car`, `cdr` のみを用いてペアにアクセスする場合、この実装を“本物の”データ構造を用いる実装と区別することはできません。

The point of exhibiting the procedural representation of pairs is not that our language works this way (Scheme, and Lisp systems in general, implement pairs directly, for efficiency reasons) but that it could work this way. The procedural representation, although obscure, is a perfectly adequate way to represent pairs, since it fulfills the only conditions that pairs need to fulfill. This example also demonstrates that the ability to manipulate procedures as objects automatically provides the ability to represent compound data. This may seem a curiosity now, but procedural representations of data will play a central role in our programming repertoire. This style of programming is often called *message passing*, and we will be using it as a basic tool in Chapter 3 when we address

the issues of modeling and simulation.

ペアの手続による表現を提示することのポイントは私達の言語がこのように働いているということではなく (Scheme や一般的な Lisp システムは効率上の理由からペアを直接的に実装します)、しかしそれがこのように働くことができるということです。手続による表現は曖昧ですが、ペアを表現するのに完璧に適切な方法です。ペアが満たすべき必要な条件を満たすからです。この例はまた手続をオブジェクトとして操作する能力が自動的に複合データを表現する能力を提供することを実演しました。これは今は珍しく見えるかもしれません、しかし手続によるデータの表現は私達のプログラミングレパートリの中核的役割を演じます。このプログラミングスタイルは時折 *message passing*(メッセージパッシング) と呼ばれ、私達はこれを [Chapter 3](#) にてモデリングとミュレーションの問題を解決する時に基本的なツールとして用います。

**Exercise 2.4:** Here is an alternative procedural representation of pairs. For this representation, verify that `(car (cons x y))` yields `x` for any objects `x` and `y`.

ここにペアの代替的な手続上の表現がある。この表現に対して `(car (cons x y))` が任意のオブジェクト `x` と `y` に対して `x` を返すか確認せよ。

```
(define (cons x y)
  (lambda (m) (m x y)))
(define (car z)
  (z (lambda (p q) p)))
```

What is the corresponding definition of `cdr?` (Hint: To verify that this works, make use of the substitution model of [Section 1.1.5](#).)

対応する `cdr` の定義はどうなるか? (ヒント: これが正しく働くか確認するには [Section 1.1.5](#) の置換モデルを使用せよ)

**Exercise 2.5:** Show that we can represent pairs of nonnegative integers using only numbers and arithmetic operations if we represent the pair  $a$  and  $b$  as the integer that is the product  $2^a 3^b$ . Give the corresponding definitions of the procedures `cons`, `car`, and `cdr`.

負ではない整数のペアを数値と数値演算命令のみを用いて表現できることを、もし  $a$  と  $b$  のペアを積  $2^a 3^b$  の整数で表現すれば可能であることにより示せ。対応する手続 `cons`, `car`, `cdr` の定義を与えよ。

**Exercise 2.6:** In case representing pairs as procedures wasn't mind-boggling enough, consider that, in a language that can manipulate procedures, we can get by without numbers (at least insofar as nonnegative integers are concerned) by implementing 0 and the operation of adding 1 as

ペアを手続として表現することが十分に驚かせるに値するものでない場合、手続を操作可能なある言語においては 0 と 1 を足すことを以下のように実装することで数値が無くてもやっていける(少なくとも負ではない整数のみを考える場合においては)ことを考えてみよ。

```
(define zero (lambda (f) (lambda (x) x)))
(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

This representation is known as *Church numerals*, after its inventor, Alonzo Church, the logician who invented the  $\lambda$ -calculus.

この表現はその開発者に因んで *Church numerals*(チャーチ数)として知られる。Alonzo Church は  $\lambda$ -演算を発明した論理学者である。

Define `one` and `two` directly (not in terms of `zero` and `add-1`). (Hint: Use substitution to evaluate `(add-1 zero)`). Give a direct definition of the addition procedure `+` (not in terms of repeated application of `add-1`).

`one` と `two` を直接 (`zero` と `add-1` を用いずに) 定義せよ。(ヒント: 加算手続の直接的な定義 `+` を与えよ。`add-1` の繰り返し適用は用いない)

## 2.1.4 Extended Exercise: Interval Arithmetic

Alyssa P. Hacker is designing a system to help people solve engineering problems. One feature she wants to provide in her system is the ability to manipulate inexact quantities (such as measured parameters of physical devices) with known precision, so that when computations are done with such approximate quantities the results will be numbers of known precision.

Alyssa P. Hacker は人々が工学上の問題を解くのを手助けするシステムを設計しています。システムにおいて彼女が提供したい 1 つの機能は(物理機器

の測定されたパラメータのような) 不正確な量を既知の精度にて扱う能力です。演算がそのような近似量にて行われた時、結果が既知の精度の値になるようにするためです。

Electrical engineers will be using Alyssa's system to compute electrical quantities. It is sometimes necessary for them to compute the value of a parallel equivalent resistance  $R_p$  of two resistors  $R_1, R_2$  using the formula

電気技術者達が Alyssa のシステムを電気の量を計算するために使用します。彼らは時折 2 つの抵抗  $R_1, R_2$  の並列に等価な抵抗値  $R_p$  を次の式を用いて計算する必要があります。

$$R_p = \frac{1}{1/R_1 + 1/R_2}.$$

Resistance values are usually known only up to some tolerance guaranteed by the manufacturer of the resistor. For example, if you buy a resistor labeled "6.8 ohms with 10% tolerance" you can only be sure that the resistor has a resistance between  $6.8 - 0.68 = 6.12$  and  $6.8 + 0.68 = 7.48$  ohms. Thus, if you have a 6.8-ohm 10% resistor in parallel with a 4.7-ohm 5% resistor, the resistance of the combination can range from about 2.58 ohms (if the two resistors are at the lower bounds) to about 2.97 ohms (if the two resistors are at the upper bounds).

抵抗値は通常抵抗の生産者により保証されるいくつかの許容誤差未満であることが知られています。例えばもしあなたが "10% の許容誤差で  $6.8\Omega$ " とラベリングされた抵抗を買ったとしたら、確かなのはその抵抗は  $6.8 - 0.68 = 6.12$  と  $6.8 + 0.68 = 7.48\Omega$  の間の抵抗を持つことのみです。従って、もし  $6.8\Omega$  10% の抵抗と並列に  $4.7\Omega$  5% の抵抗を接続した場合に、組み合わせの抵抗は約  $2.58\Omega$ (2 つの抵抗が低限である場合) から約  $2.97\Omega$ (2 つの抵抗が上限である場合) の区間になります。

Alyssa's idea is to implement "interval arithmetic" as a set of arithmetic operations for combining "intervals" (objects that represent the range of possible values of an inexact quantity). The result of adding, subtracting, multiplying, or dividing two intervals is itself an interval, representing the range of the result.

Alyssa のアイデアは "interval arithmetic"(区間演算) を "区間"(不正確な量の取り得る値の区間を表現するオブジェクト) を連結する演算命令の集合として実装することです。2 つの区間の加算、減算、乗算、除算の結果はそれ自身が区間であり、結果の範囲を表します。

Alyssa postulates the existence of an abstract object called an "interval" that has two endpoints: a lower bound and an upper bound. She also presumes

that, given the endpoints of an interval, she can construct the interval using the data constructor `make-interval`. Alyssa first writes a procedure for adding two intervals. She reasons that the minimum value the sum could be is the sum of the two lower bounds and the maximum value it could be is the sum of the two upper bounds:

Alyssa は 2 つの終端、下限と上限を持つ“区間”と呼ばれる抽象オブジェクトの存在を仮定しました。彼女はまた区間の終端を与えられた時、データコンストラクタ `make-interval` を用いて区間の構築ができると仮定しました。Alyssa は最初に 2 つの区間を足す手続を書きました。彼女は和の最小値は 2 つの下限の和であり、最大値は 2 つの上限の和になるだろうと推測しました。

```
(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                 (+ (upper-bound x) (upper-bound y))))
```

Alyssa also works out the product of two intervals by finding the minimum and the maximum of the products of the bounds and using them as the bounds of the resulting interval. (`Min` and `max` are primitives that find the minimum or maximum of any number of arguments.)

Alyssa はまた 2 つの区間の積を限界値の積の最小値と最大値を見つけることで算出し、そしてそれらを結果区間の限界値として用いました。`(min` と `max` は任意の数の引数の最小値と最大値を見つけるプリミティブです)。

```
(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y))))
    (p2 (* (lower-bound x) (upper-bound y))))
    (p3 (* (upper-bound x) (lower-bound y))))
    (p4 (* (upper-bound x) (upper-bound y))))
  (make-interval (min p1 p2 p3 p4)
                 (max p1 p2 p3 p4)))
```

To divide two intervals, Alyssa multiplies the first by the reciprocal of the second. Note that the bounds of the reciprocal interval are the reciprocal of the upper bound and the reciprocal of the lower bound, in that order.

2 つの区間を割るために、Alyssa は一つ目に 2 つ目の逆数を掛けました。区間の逆数の限界値は上限の逆数と下限の逆数をその順で用いることに注意して下さい。

```
(define (div-interval x y)
```

```
(mul-interval
  x
  (make-interval (/ 1.0 (upper-bound y))
    (/ 1.0 (lower-bound y)))))
```

**Exercise 2.7:** Alyssa's program is incomplete because she has not specified the implementation of the interval abstraction. Here is a definition of the interval constructor:

Alyssa のプログラムは未完成である。なぜなら彼女は区間の抽象の実装を特定していない。ここに区間のコンストラクタの定義を置く。

```
(define (make-interval a b) (cons a b))
```

Define selectors `upper-bound` and `lower-bound` to complete the implementation.

セレクタ `upper-bound` と `lower-bound` を定義し実装を完成させよ。

**Exercise 2.8:** Using reasoning analogous to Alyssa's, describe how the difference of two intervals may be computed. Define a corresponding subtraction procedure, called `sub-interval`.

Alyssa の考え方と同様の推論を用いて、2つの区間の差がどのように計算されるかを説明せよ。対応する減算手続 `sub-interval` を定義せよ。

**Exercise 2.9:** The *width* of an interval is half of the difference between its upper and lower bounds. The width is a measure of the uncertainty of the number specified by the interval. For some arithmetic operations the width of the result of combining two intervals is a function only of the widths of the argument intervals, whereas for others the width of the combination is not a function of the widths of the argument intervals. Show that the width of the sum (or difference) of two intervals is a function only of the widths of the intervals being added (or subtracted). Give examples to show that this is not true for multiplication or division.

区間の *width*(幅) は上限と下限の差の半値である。幅は区間で指定された数値の不確かさの基準である。いくつかの数値演算に対しては、2つの区間を結合した結果の幅は引数区間の幅のみによる関

数である。一方で他の演算においては結合の幅は引数の幅の関数ではない。2つの区間の和、または差の幅は足される、または引かれる区間の幅の関数であることを示せ。これが乗算と除算においては正しくないことを例をもって示せ。

**Exercise 2.10:** Ben Bitdiddle, an expert systems programmer, looks over Alyssa's shoulder and comments that it is not clear what it means to divide by an interval that spans zero. Modify Alyssa's code to check for this condition and to signal an error if it occurs.

エキスパートシステムプログラマの Ben Bitdiddle は Alyssa の肩越しに覗いて、区間の長さが 0 の時に割ったらどうなるのか不明だよとコメントした。Alyssa のコードを変更し、この条件をチェックしてもしそれが起こればエラーを返すようにせよ。

**Exercise 2.11:** In passing, Ben also cryptically comments: “By testing the signs of the endpoints of the intervals, it is possible to break `mul-interval` into nine cases, only one of which requires more than two multiplications.” Rewrite this procedure using Ben’s suggestion.

通り過ぎながら Ben はまた曖昧なコメントを残した。“区間の終端の符号をテストすることで `mul-interval` を 7 つに場合分けできる。その 1 つのみが 2 つ以上の乗算を必要とする。”この手続を Ben の提案に従い書き直せ。

After debugging her program, Alyssa shows it to a potential user, who complains that her program solves the wrong problem. He wants a program that can deal with numbers represented as a center value and an additive tolerance; for example, he wants to work with intervals such as  $3.5 \pm 0.15$  rather than  $[3.35, 3.65]$ . Alyssa returns to her desk and fixes this problem by supplying an alternate constructor and alternate selectors:

プログラムをデバッグした後で、Alyssa はユーザ候補の一人に見せた。彼は彼女のプログラムは間違った問題を解いていると文句を言った。彼が欲しいのは中央値として表現された数値と追加の許容誤差を扱えるプログラムだ。例えば彼は  $3.5 \pm 0.15$  のような区間を扱いたく、 $[3.35, 3.65]$  ではない。Alyssa は彼女の机に戻りこの問題を代替となるコンストラクタとセレクタを提供することに直した。

```
(define (make-center-width c w)
```

```
(make-interval (- c w) (+ c w)))
(define (center i)
  (/ (+ (lower-bound i) (upper-bound i)) 2))
(define (width i)
  (/ (- (upper-bound i) (lower-bound i)) 2))
```

Unfortunately, most of Alyssa's users are engineers. Real engineering situations usually involve measurements with only a small uncertainty, measured as the ratio of the width of the interval to the midpoint of the interval. Engineers usually specify percentage tolerances on the parameters of devices, as in the resistor specifications given earlier.

不運なことに、Alyssa のユーザの多くはエンジニアです。実際の工学の場では通常、小さな不確かさを伴なう計測を伴い、区間の中央値に対する区間の幅の割合として測定されます。エンジニアは通常パーセンテージにて許容誤差を端末のパラメータ上に、以前に与えた抵抗の仕様のように指定します。

**Exercise 2.12:** Define a constructor `make-center-percent` that takes a center and a percentage tolerance and produces the desired interval. You must also define a selector `percent` that produces the percentage tolerance for a given interval. The `center` selector is the same as the one shown above.

コンストラクタ `make-center-percent` を中央値とパーセンテージ許容誤差を取り望まれた区間を返すように定義せよ。セレクタ `percent` を与えられた区間にに対するパーセンテージ許容誤差を返すように定義することも行うこと。`center` セレクタは上で見たものと同じである。

**Exercise 2.13:** Show that under the assumption of small percentage tolerances there is a simple formula for the approximate percentage tolerance of the product of two intervals in terms of the tolerances of the factors. You may simplify the problem by assuming that all numbers are positive.

小さなパーセンテージ許容誤差の前提の下では、2つの区間の積のパーセンテージ許容誤差を因数の許容誤差を用いて近似するための簡単な式が存在することを示せ。全ての数値は正であると前提して問題を簡単にして良い。

After considerable work, Alyssa P. Hacker delivers her finished system. Several years later, after she has forgotten all about it, she gets a frenzied call from an

irate user, Lem E. Tweakit. It seems that Lem has noticed that the formula for parallel resistors can be written in two algebraically equivalent ways:

大変な仕事を終え、Alyssa P. Hacker は完了したシステムを受け渡しました。何年か後、彼女が全てを忘れた頃に、彼女は興奮した電話を、怒ったユーザ、Lem E. Tweakit から受けました。どうやら Lem は並列接続の抵抗の式が 2 つの代数的に等価な方法で書くことができることに気付いたようです。

$$\frac{R_1 R_2}{R_1 + R_2}$$

and

$$\frac{1}{1/R_1 + 1/R_2}.$$

He has written the following two programs, each of which computes the parallel-resistors formula differently:

彼は以下の 2 つのプログラムを書きました。それぞれが並列接続の抵抗値を異なる式で計算します。

```
(define (par1 r1 r2)
  (div-interval (mul-interval r1 r2)
                (add-interval r1 r2)))

(define (par2 r1 r2)
  (let ((one (make-interval 1 1)))
    (div-interval
      one (add-interval (div-interval one r1)
                        (div-interval one r2)))))
```

Lem complains that Alyssa's program gives different answers for the two ways of computing. This is a serious complaint.

Lem は Alyssa のプログラムは 2 つの方法の演算にて異なる値を返すと抗議しました。これは深刻な苦情です。

**Exercise 2.14:** Demonstrate that Lem is right. Investigate the behavior of the system on a variety of arithmetic expressions. Make some intervals  $A$  and  $B$ , and use them in computing the expressions  $A/A$  and  $A/B$ . You will get the most insight by using intervals whose width is a small percentage of the center value. Examine

the results of the computation in center-percent form (see [Exercise 2.12](#)).

Lem が正しいことを確認せよ。様々な数値演算にてシステムの挙動を調べよ。ある区間  $A$  と  $B$  を作成し、式  $A/A$  と  $A/B$  の計算においてそれらを用いよ。幅が中央値の小さなパーセンテージである区間を用いることで多くの実態を掴むことができるだろう。center-percent 形式 ([Exercise 2.12](#)参照) の演算の結果を調査せよ。

**Exercise 2.15:** Eva Lu Ator, another user, has also noticed the different intervals computed by different but algebraically equivalent expressions. She says that a formula to compute with intervals using Alyssa's system will produce tighter error bounds if it can be written in such a form that no variable that represents an uncertain number is repeated. Thus, she says, `par2` is a “better” program for parallel resistances than `par1`. Is she right? Why?

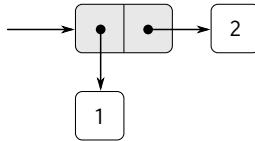
Eva Lu Ator はもう一人のユーザで、彼女もまた異なるが代数的には等価な式により異なる区間が算出されることに気付いた。彼女は Alyssa のシステムを用いて区間の計算をする式が、もし式が不確かな値を表現する変数がどれも繰り返されない形であれば、より厳しいエラーの限界を算出すると言う。従って彼女は抵抗の並列に対し、`par2`の方が `par1` より“より良い”プログラムであると述べた。彼女は正しいだろうか？それは何故か？

**Exercise 2.16:** Explain, in general, why equivalent algebraic expressions may lead to different answers. Can you devise an interval arithmetic package that does not have this shortcoming, or is this task impossible? (Warning: This problem is very difficult.)

一般的に、なぜ等価な代数式が異なる答に導くのか説明せよ。この欠点を持たない区間演算パッケージを開発することは可能だろうか。または不可能だろうか。(警告：この問題はとても難しい)

## 2.2 Hierarchical Data and the Closure Property

As we have seen, pairs provide a primitive “glue” that we can use to construct compound data objects. [Figure 2.2](#) shows a standard way to visualize



**Figure 2.2:** Box-and-pointer representation of `(cons 1 2)`.

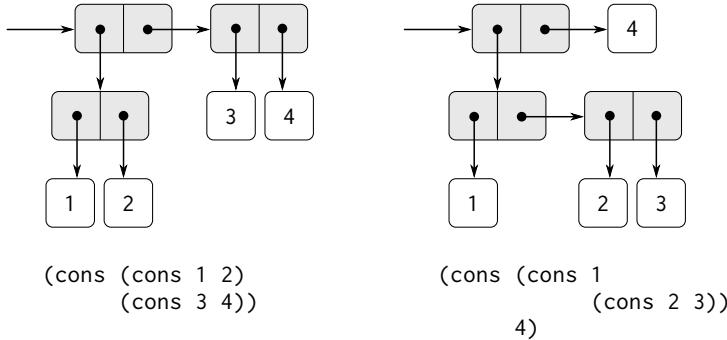
a pair—in this case, the pair formed by `(cons 1 2)`. In this representation, which is called *box-and-pointer notation*, each object is shown as a *pointer* to a box. The box for a primitive object contains a representation of the object. For example, the box for a number contains a numeral. The box for a pair is actually a double box, the left part containing (a pointer to) the `car` of the pair and the right part containing the `cdr`.

ここまで学んだように、ペアは私達が複合データオブジェクトを構築するのに利用可能なプリミティブな“糊”を提供する。[Figure 2.2](#)はペア—この場合は`(cons 1 2)`にて形成されたペアを図示する標準的な方法を示している。この*box-and-pointer notation*(箱と点表記法)と呼ばれる表現において、各オブジェクトは箱への*pointer*(ポインタ)として表わされている。プリミティブオブジェクトの箱はオブジェクトの表現を持っている。例えば数値の箱は数字を持っている。ペアの箱は実際には二重の箱で、左部分はペアの `car`(へのポインタ)を持っており、右部分は `cdr` を持っている。

We have already seen that `cons` can be used to combine not only numbers but pairs as well. (You made use of this fact, or should have, in doing [Exercise 2.2](#) and [Exercise 2.3](#).) As a consequence, pairs provide a universal building block from which we can construct all sorts of data structures. [Figure 2.3](#) shows two ways to use pairs to combine the numbers 1, 2, 3, and 4.

私達は既に `cons` が数値のみでなくペアもまた組み合わせられることについて学びました。[\(Exercise 2.2 と Exercise 2.3 で\)](#)あなたはこの事実を用いたか、または用いざるを得なかつたでしょう。結果としてペアは全ての種類のデータ構造を構築可能な普遍的な構築ブロックを提供します。[Figure 2.3](#)は数値 1, 2, 3, 4 を組み合わせるためにペアを用いる 2 つの方法を示しています。

The ability to create pairs whose elements are pairs is the essence of list structure's importance as a representational tool. We refer to this ability as the *closure property* of `cons`. In general, an operation for combining data objects



**Figure 2.3:** Two ways to combine 1, 2, 3, and 4 using pairs.

satisfies the closure property if the results of combining things with that operation can themselves be combined using the same operation.<sup>10</sup> Closure is the key to power in any means of combination because it permits us to create *hierarchical* structures—structures made up of parts, which themselves are made up of parts, and so on.

要素がペアであるペアを作成する能力は表現ツールとしてのリスト構造の重要な本質です。私達はこの能力を `cons` の閉包性 (closure property) と呼びます。一般的に、データオブジェクトを組み合わせる操作はもしその命令による組み合わせの結果それが自身が同じ命令を用いて組み合わせることが可能ならば閉包性を満たします。<sup>11</sup> 閉包はどのような目的の組み合わせをも強力にする

<sup>10</sup>The use of the word “closure” here comes from abstract algebra, where a set of elements is said to be closed under an operation if applying the operation to elements in the set produces an element that is again an element of the set. The Lisp community also (unfortunately) uses the word “closure” to describe a totally unrelated concept: A closure is an implementation technique for representing procedures with free variables. We do not use the word “closure” in this second sense in this book.

<sup>11</sup>ここで“closure”(閉包)という用語の使用は抽象代数から来ており、もし操作の集合の要素への適用により生成される要素が再び同じ集合の要素である場合、要素の集合が操作の下において閉じられていると呼ばれます。Lisp コミュニティでは(残念なことに)用語“closure”を全く関係のない概念にも使用しています。closure とは自由変数を持つ手続きを表現するための実装テクニックです。私達は“closure”をこの本の中では2つ目の意味ではありません。

鍵となります。なぜなら *hierarchical*(階層) 構造 — 複数のパーツから成る構造であり、パーツ自身も複数のパーツから成るような構造を作成することが可能になるためです。

From the outset of **Chapter 1**, we've made essential use of closure in dealing with procedures, because all but the very simplest programs rely on the fact that the elements of a combination can themselves be combinations. In this section, we take up the consequences of closure for compound data. We describe some conventional techniques for using pairs to represent sequences and trees, and we exhibit a graphics language that illustrates closure in a vivid way.<sup>12</sup>

**Chapter 1** の始めから、手続の取扱において閉包を本質的に利用してきました。とても簡単なプログラムを除けば全てのプログラムは組み合わせの要素はそれ自身もまた組み合わせであるという事実に依存しているためです。この節では複合データにとっての閉包の重要性を取り上げます。ペアを使用して列と木を表現するための、いくつかの便利なテクニックを説明します。そして鮮烈な方法でクロージャを図示するグラフィック言語を提示します。<sup>13</sup>

---

<sup>12</sup>The notion that a means of combination should satisfy closure is a straightforward idea. Unfortunately, the data combiners provided in many popular programming languages do not satisfy closure, or make closure cumbersome to exploit. In Fortran or Basic, one typically combines data elements by assembling them into arrays—but one cannot form arrays whose elements are themselves arrays. Pascal and C admit structures whose elements are structures. However, this requires that the programmer manipulate pointers explicitly, and adhere to the restriction that each field of a structure can contain only elements of a prespecified form. Unlike Lisp with its pairs, these languages have no built-in general-purpose glue that makes it easy to manipulate compound data in a uniform way. This limitation lies behind Alan Perlis's comment in his foreword to this book: "In Pascal the plethora of declarable data structures induces a specialization within functions that inhibits and penalizes casual cooperation. It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures."

<sup>13</sup>組み合わせは閉包であるべきだという手段の概念は単純なアイデアです。残念なことに多くの人気のあるプログラミング言語が提供するデータの組み合わせ手法は閉包性を満たしませんし、閉包性の活用が面倒です。Fortran や BASIC ではデータ要素を組合せる典型的な 1 つの手段は配列にそれらをまとめることです。しかし配列の要素自身が配列である配列を形成できません。Pascal と C は構造体の要素が構造体であることを認めます。しかしこれはプログラマが明示的にポインタを取り扱うことを要求し、構造体の各フィールドが事前に指定された形式の要素のみを保管できるという制約に帰着します。Lisp のペアとは異なりこれらの言語は複合データを統一的な方法で扱うことを簡単にする組み込みの汎用目的な糊を持っています。この制約がこの本の前書きにおける Alan Perlis のコメントの背景にあります。“Pascal における過剰な宣言可能なデータ構

## 2.2.1 Representing Sequences

One of the useful structures we can build with pairs is a *sequence*—an ordered collection of data objects. There are, of course, many ways to represent sequences in terms of pairs. One particularly straightforward representation is illustrated in Figure 2.4, where the sequence 1, 2, 3, 4 is represented as a chain of pairs. The `car` of each pair is the corresponding item in the chain, and the `cdr` of the pair is the next pair in the chain. The `cdr` of the final pair signals the end of the sequence by pointing to a distinguished value that is not a pair, represented in box-and-pointer diagrams as a diagonal line and in programs as the value of the variable `nil`. The entire sequence is constructed by nested `cons` operations:

ペアを用いて構築可能な便利な構造の 1 つが *sequence*(列)一順に並べたデータオブジェクトの集合です。もちろんペアを用いて列を表現する方法は数多く存在します。特に簡単な表現方法の 1 つを Figure 2.4 に示します。列 1, 2, 3, 4 がペアの連鎖として表わされています。各ペアの `car` は鎖内で対応するアイテムであり、各ペアの `cdr` は鎖内での次のペアです。最後のペアの `cdr` は列の終端をペアではないことを識別する値を指し示すことで合図します。箱とポインタの図では斜めの線にて表現され、プログラムでは変数 `nil` の値にて示されます。列全体は入れ子の `cons` 命令にて構築されます。

```
(cons 1
      (cons 2
          (cons 3
              (cons 4 nil))))
```

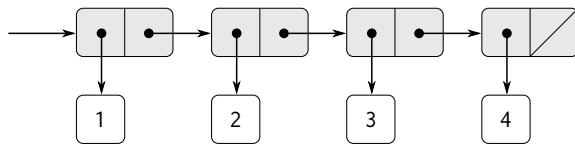
Such a sequence of pairs, formed by nested `conses`, is called a *list*, and Scheme provides a primitive called `list` to help in constructing lists.<sup>14</sup> The above sequence could be produced by `(list 1 2 3 4)`. In general,

そのようなペアの列は入れ子の `cons` にて形成され、*list*(リスト)と呼ばれます。そして Scheme は `list` と呼ばれるプリミティブを提供しリストの構築

---

造は関数内にて特殊化を引き起こし、カジュアルな連携を不利にし、抑制してしまう。1つのデータ構造を操作する 100 の関数を持つほうが、10 のデータ構造を操作する 10 の関数を持つよりも良い。”

<sup>14</sup>In this book, we use *list* to mean a chain of pairs terminated by the end-of-list marker. In contrast, the term *list structure* refers to any data structure made out of pairs, not just to lists.



**Figure 2.4:** The sequence 1, 2, 3, 4 represented as a chain of pairs.

を手助けします。<sup>15</sup> 上の列は (list 1 2 3 4) により生成可能です。

(`list` ⟨ $a_1a_2a_n$

is equivalent to

は以下と等価です。

```
(cons ⟨ $a_1cons ⟨ $a_2cons ...
      (cons ⟨ $a_n$$$ 
```

Lisp systems conventionally print lists by printing the sequence of elements, enclosed in parentheses. Thus, the data object in Figure 2.4 is printed as (1 2 3 4):

Lisp システムは慣習としてリストを括弧で括られた要素の列を表示することで表します。従ってFigure 2.4のデータオブジェクトは (1 2 3 4) の様に表示されます。

```
(define one-through-four (list 1 2 3 4))
one-through-four
(1 2 3 4)
```

Be careful not to confuse the expression (`list` 1 2 3 4) with the list (1 2 3 4), which is the result obtained when the expression is evaluated. Attempting to

---

<sup>15</sup>この本では *list* をリスト終端マーカーにて終端化されたペアの鎖を意味するように使用します。一方で用語 *list structure*(リスト構造) はペアから作り上げられた任意のデータ構造を参照し、ただのリストは意味しません。

evaluate the expression (1 2 3 4) will signal an error when the interpreter tries to apply the procedure 1 to arguments 2, 3, and 4.

式 (list 1 2 3 4) とリスト (1 2 3 4) を取り違えないよう気をつけて下さい。リストは式が評価された時に得られた結果です。式 (1 2 3 4) を評価しようとする試みはインタプリタが手続 1 を引数 2, 3, 4 に適用しようとした時にエラーを発します。

We can think of `car` as selecting the first item in the list, and of `cdr` as selecting the sublist consisting of all but the first item. Nested applications of `car` and `cdr` can be used to extract the second, third, and subsequent items in the list.<sup>16</sup>

`car` をリスト内の最初のアイテムを選択すると考えることもでき、`cdr` を最初のアイテム以外の全てにより成り立つサブ(副)リストを選択すると考えることも可能です。`car` と `cdr` の入れ子の適用はリスト内の 2 つ目、3 つ目、そしてその後に続く複数のアイテムを抽出するために利用可能です。<sup>17</sup>

The constructor `cons` makes a list like the original one, but with an additional item at the beginning.

コンストラクタ `cons` は元のリストと同様のリストを作りますが、最初に追加のアイテムを入れます。

```
(car one-through-four)  
1  
(cdr one-through-four)
```

---

<sup>16</sup>Since nested applications of `car` and `cdr` are cumbersome to write, Lisp dialects provide abbreviations for them—for instance,

```
(cadr <arg>) = (car (cdr <arg>))
```

The names of all such procedures start with `c` and end with `r`. Each `a` between them stands for a `car` operation and each `d` for a `cdr` operation, to be applied in the same order in which they appear in the name. The names `car` and `cdr` persist because simple combinations like `cadr` are pronounceable.

<sup>17</sup>`car` と `cdr` の入れ子の適用は書くのが面倒なため Lisp の各種方言はそれらに対する略記法を提供しています。例えば、

```
(cadr <arg>) = (car (cdr <arg>))
```

そのような手続全ての名前は `c` で始まり `r` で終わります。それらの間の各 `a` は `car` 命令を意図し、`d` は `cdr` 命令を意図し、その名前に現れた順と同じ順にて適用されます。`car` と `cdr` の名前は存続します。なぜなら `cadr` のような単純な組み合わせが発音可能だからです。

```
(2 3 4)
(car (cdr one-through-four))
2
(cons 10 one-through-four)
(10 1 2 3 4)
(cons 5 one-through-four)
(5 1 2 3 4)
```

The value of `nil`, used to terminate the chain of pairs, can be thought of as a sequence of no elements, the *empty list*. The word *nil* is a contraction of the Latin word *nihil*, which means “nothing.”<sup>18</sup>

`nil` の値はペアの鎖を終了するために使用されますが、全く要素が無い列 *empty list* (空リスト) として考えることもできます。`nil` という単語はラテン語の単語 *nihil* の省略形で、“無”を意味します。<sup>19</sup>

## List operations

The use of pairs to represent sequences of elements as lists is accompanied by conventional programming techniques for manipulating lists by successively “cdring down” the lists. For example, the procedure `list-ref` takes as arguments a list and a number  $n$  and returns the  $n^{\text{th}}$  item of the list. It is customary

---

<sup>18</sup>It's remarkable how much energy in the standardization of Lisp dialects has been dissipated in arguments that are literally over nothing: Should `nil` be an ordinary name? Should the value of `nil` be a symbol? Should it be a list? Should it be a pair? In Scheme, `nil` is an ordinary name, which we use in this section as a variable whose value is the end-of-list marker (just as `true` is an ordinary variable that has a true value). Other dialects of Lisp, including Common Lisp, treat `nil` as a special symbol. The authors of this book, who have endured too many language standardization brawls, would like to avoid the entire issue. Once we have introduced quotation in Section 2.3, we will denote the empty list as '`()`' and dispense with the variable `nil` entirely.

<sup>19</sup>どれだけのエネルギーが Lisp 方言の標準化において文字通り意味の無い議論に浪費されたかについては特筆に値します。`nil` は普通の名前であるべきか? `nil` の値は記号であるべきか? それはリストであるべきか? それはペアであるべきか? Scheme では `nil` は普通の名前でありこの節では変数として扱いその値はリスト終端マークです。`(true)` が普通の変数であり、真の値を持つと同様です。Common Lisp を含む他の Lisp 方言は `nil` を特別な記号として扱います。この本の著者達は、言語の標準化における数多くの乱闘に耐えてきたので、この問題全体を避けたいと思います。Section 2.3にて quote を紹介した後には空リストに '`()`' という名前を付け、全体的に変数 `nil` を免除します。

to number the elements of the list beginning with 0. The method for computing `list-ref` is the following:

複数のペアを使用して要素の列をリストのように表現することは、慣習的なプログラミングテクニックである連続してリストを“`cdr`で縮小する”ことによりリストを操作するのと同時に生じます。例えば手続 `list-ref` は引数としてリストと数値  $n$  を取り、リストの  $n$  番目の項目を返します。リストの要素を数えるのに 0 から始めるのが慣習です。`list-ref` を計算する方法は以下の通りです。

- For  $n = 0$ , `list-ref` should return the `car` of the list.  
 $n = 0$  の場合、`list-ref` はリストの `car` を返す。
- Otherwise, `list-ref` should return the  $(n - 1)$ -st item of the `cdr` of the list.  
そうでなければ、`list-ref` はリストの `cdr` の  $(n - 1)$  番目の項目を返す。

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
(define squares (list 1 4 9 16 25))
(list-ref squares 3)
16
```

Often we `cdr` down the whole list. To aid in this, Scheme includes a primitive predicate `null?`, which tests whether its argument is the empty list. The procedure `length`, which returns the number of items in a list, illustrates this typical pattern of use:

時折、私達はリスト全体を `cdr` で下ります。これを手助けするために、Scheme はプリミティブな手続 `null?` を持っています。その引数が空リストであるかどうかを試験します。手続 `length` はリスト内の要素数を返しますが、`null?` の使用の典型的なパターンを説明します。

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))
(define odds (list 1 3 5 7))
```

```
(length odds)
```

```
4
```

The `length` procedure implements a simple recursive plan. The reduction step is:

手続 `length` は単純な最近計画を実装します。集約ステップは以下の通りです。

- The `length` of any list is 1 plus the `length` of the `cdr` of the list.  
任意のリストの `length` はリストの `cdr` の `length` に 1 を足した値

This is applied successively until we reach the base case:

これが底となるケースに到達するまで繰り返し適用される

- The `length` of the empty list is 0.  
空リストの `length` は 0

We could also compute `length` in an iterative style:

また `length` は反復スタイルでも計算可能です。

```
(define (length items)
  (define (length-iter a count)
    (if (null? a)
        count
        (length-iter (cdr a) (+ 1 count))))
  (length-iter items 0))
```

Another conventional programming technique is to “cons up” an answer list while cdring down a list, as in the procedure `append`, which takes two lists as arguments and combines their elements to make a new list:

もう 1 つの慣習的なプログラミングテクニックは `cdr` を繰り返し利用しリストを下る間に、答のリストを“`cons` で積み上げ”ることです。これは手続 `append` にて利用され、`append` は 2 つのリストを引数として取り、それらの要素を結合し、新しいリストを作ります。

```
(append squares odds)
(1 4 9 16 25 1 3 5 7)
(append odds squares)
(1 3 5 7 1 4 9 16 25)
```

`Append` is also implemented using a recursive plan. To `append` lists `list1` and `list2`, do the following:

`append` もまた再帰計画を用いて実装されます。リスト `list1` と `list2` を `append` するためには以下の通りに行います。

- If `list1` is the empty list, then the result is just `list2`.  
もし `list1` が空リストであれば、結果は単に `list2`
- Otherwise, `append` the `cdr` of `list1` and `list2`, and `cons` the `car` of `list1` onto the result:  
そうでない場合、`list1` の `cdr` と `list2` を `append` し、その結果の上に `list1` の `car` を `cons` する

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1) (append (cdr list1) list2))))
```

**Exercise 2.17:** Define a procedure `last-pair` that returns the list that contains only the last element of a given (nonempty) list:  
与えられた(空でない)リストの最初の要素のみを持つリストを返す手続 `last-pair` を定義せよ。

```
(last-pair (list 23 72 149 34))
(34)
```

**Exercise 2.18:** Define a procedure `reverse` that takes a list as argument and returns a list of the same elements in reverse order:  
リストを引数として取り、同じ要素を逆順に持つリストを返す手続 `reverse` を定義せよ。

```
(reverse (list 1 4 9 16 25))
(25 16 9 4 1)
```

**Exercise 2.19:** Consider the change-counting program of [Section 1.2.2](#). It would be nice to be able to easily change the currency used by the program, so that we could compute the number of ways to change a British pound, for example. As the program is

written, the knowledge of the currency is distributed partly into the procedure `first-denomination` and partly into the procedure `count-change` (which knows that there are five kinds of U.S. coins). It would be nicer to be able to supply a list of coins to be used for making change.

Section 1.2.2の両替数え上げプログラムについて考える。プログラムにて用いられる通貨を容易に変更できるようになればとても良いだろう。そうすることで例えばイギリスのポンドの両替方法の数を計算できるようになるだろう。プログラムが書かれた時には、通貨の知識はある部分は手続 `first-denomination` の中に、またある部分は手続 `count-change` の中に存在した。`(count-change` は米国の貨幣には 5 種類あることを知っていた)。両替を行うため利用される貨幣のリストが提供できるようになればより良くなるだろう。

We want to rewrite the procedure `cc` so that its second argument is a list of the values of the coins to use rather than an integer specifying which coins to use. We could then have lists that defined each kind of currency:

`cc` を変更することで、その 2 つ目の引数がどの貨幣を使用するかを指定する整数ではなく、使用する貨幣の値のリストとなるようしたいと考える。そして通貨の各種類を定義するリストを持つことにする。

```
(define us-coins (list 50 25 10 5 1))  
(define uk-coins (list 100 50 20 10 5 2 1 0.5))
```

We could then call `cc` as follows:

次に `cc` を以下のように呼び出す。

```
(cc 100 us-coins)
```

292

To do this will require changing the program `cc` somewhat. It will still have the same form, but it will access its second argument differently, as follows:

これを行なうためにはプログラム `cc` に何らかの変更が必要だ。同じ形態を保つが、2 つ目の引数に異なる方法でアクセスする。以下のようになる。

```

(define (cc amount coin-values)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (no-more? coin-values)) 0)
        (else
          (+ (cc amount
                  (except-first-denomination
                   coin-values))
              (cc (- amount
                      (first-denomination
                       coin-values))
                  coin-values)))))))

```

Define the procedures `first-denomination`, `except-first-de-nomination`, and `no-more?` in terms of primitive operations on list structures. Does the order of the list `coin-values` affect the answer produced by `cc`? Why or why not?

リスト構造に対するプリミティブな命令を用いて手続 `first-denomination`, `except-first-de-nomination`, and `no-more?` を定義せよ。リスト `coin-values` の順は `cc` により生成される解答に影響を与えるか? それは何故か? または何故そうでないのか?

**Exercise 2.20:** The procedures `+`, `*`, and `list` take arbitrary numbers of arguments. One way to define such procedures is to use `define` with *dotted-tail notation*. In a procedure definition, a parameter list that has a dot before the last parameter name indicates that, when the procedure is called, the initial parameters (if any) will have as values the initial arguments, as usual, but the final parameter's value will be a *list* of any remaining arguments. For instance, given the definition

手続 `+`, `*`, `list` は任意の数の引数を取る。そのような手続を定義する1つの方法として *dotted-tail notation*(ドット付き末尾記法)と共に `define` を使用することが上げられる。手続定義において、最後のパラメータ名の前にドットがあるパラメータリストは手続が呼び出された時に、最初以下のパラメータが(もし存在したら)初期引数の値を通常通りに持つが、最後のパラメータの値は残りの引数全てのリストとなる。例えば、以下の定義を与えられた時に、

```
(define (f x y . z) (body))
```

the procedure **f** can be called with two or more arguments. If we evaluate

手続 **f** は 2 つ以上の引数で呼び出すことが可能だ。もし次を評価すれば、

```
(f 1 2 3 4 5 6)
```

then in the body of **f**, **x** will be 1, **y** will be 2, and **z** will be the list (3 4 5 6). Given the definition

**f** のボディでは **x** が 1、**y** が 2、そして **z** はリスト (3 4 5 6) となる。以下の定義を与えられた時、

```
(define (g . w) (body))
```

the procedure **g** can be called with zero or more arguments. If we evaluate

手続 **g** はゼロ個以上の引数にて呼び出し可能となる。次を評価すれば、

```
(g 1 2 3 4 5 6)
```

then in the body of **g**, **w** will be the list (1 2 3 4 5 6).

**g** のボディでは **w** はリスト (1 2 3 4 5 6) となる。<sup>20</sup>

Use this notation to write a procedure **same-parity** that takes one or more integers and returns a list of all the arguments that have the same even-odd parity as the first argument. For example,

この記法を用いて手続 **same-parity** を書け。**same-parity** は 1 つまたはそれ以上の整数を引数として取り、最初の引数と同じ偶奇性を持つ全ての引数のリストを返す。例えば、

```
(same-parity 1 2 3 4 5 6 7)
(1 3 5 7)
(same-parity 2 3 4 5 6 7)
(2 4 6)
```

---

<sup>20</sup>To define **f** and **g** using **lambda** we would write

**f** と **g** を **lambda** を用いて定義するには、以下のように記述する。

```
(define f (lambda (x y . z) (body)))
(define g (lambda w (body)))
```

## Mapping over lists

One extremely useful operation is to apply some transformation to each element in a list and generate the list of results. For instance, the following procedure scales each number in a list by a given factor:

あるとても便利な命令は、ある変換をリストの各要素に適用し、結果のリストを返します。例えば以下の手続はリストの各数値を与えられた因数で拡大します。

```
(define (scale-list items factor)
  (if (null? items)
      nil
      (cons (* (car items) factor)
            (scale-list (cdr items)
                        factor)))))

(scale-list (list 1 2 3 4 5) 10)
(10 20 30 40 50)
```

We can abstract this general idea and capture it as a common pattern expressed as a higher-order procedure, just as in [Section 1.3](#). The higher-order procedure here is called `map`. `Map` takes as arguments a procedure of one argument and a list, and returns a list of the results produced by applying the procedure to each element in the list.<sup>21</sup>

私達は[Section 1.3](#)のように、この一般的な考え方を抽象化し、高階手続にて表現された共通なパターンとして捉えることができます。ここでの高階手続は `map` と呼びれます。`map` は引数として 1 引数の手続とリストを取り、返り値と

---

<sup>21</sup> Scheme standardly provides a `map` procedure that is more general than the one described here. This more general `map` takes a procedure of  $n$  arguments, together with  $n$  lists, and applies the procedure to all the first elements of the lists, all the second elements of the lists, and so on, returning a list of the results. For example:

```
(map + (list 1 2 3) (list 40 50 60) (list 700 800 900))
(741 852 963)
(map (lambda (x y) (+ x (* 2 y)))
     (list 1 2 3)
     (list 4 5 6))
(9 12 15)
```

してその手続をリストの各要素に適用することで得られた結果のリストを返します。<sup>22</sup>

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))

(map abs (list -10 2.5 -11.6 17))
(10 2.5 11.6 17)

(map (lambda (x) (* x x)) (list 1 2 3 4))
(1 4 9 16)
```

Now we can give a new definition of **scale-list** in terms of **map**:

これで **map** を用いた新しい **scale-list** の定義を与えられる。

```
(define (scale-list items factor)
  (map (lambda (x) (* x factor))
       items))
```

**Map** is an important construct, not only because it captures a common pattern, but because it establishes a higher level of abstraction in dealing with lists. In the original definition of **scale-list**, the recursive structure of the program draws attention to the element-by-element processing of the list. Defining **scale-list** in terms of **map** suppresses that level of detail and emphasizes that scaling transforms a list of elements to a list of results. The difference between the two definitions is not that the computer is performing a different process

---

<sup>22</sup> Scheme は標準としてここで説明される物よりもより汎用的な **map** 手続を提供します。このより汎用的な **map** は  $n$  引数の手続を、 $n$  個のリストと共に取り、全てのリストの最初の要素を手続に適用し、次に全ての 2 つ目の要素を適用し、以下それを繰り返し、結果のリストを返します。例えば、

```
(map + (list 1 2 3) (list 40 50 60) (list 700 800 900))
(741 852 963)

(map (lambda (x y) (+ x (* 2 y)))
     (list 1 2 3)
     (list 4 5 6))
(9 12 15)
```

(it isn't) but that we think about the process differently. In effect, `map` helps establish an abstraction barrier that isolates the implementation of procedures that transform lists from the details of how the elements of the list are extracted and combined. Like the barriers shown in Figure 2.1, this abstraction gives us the flexibility to change the low-level details of how sequences are implemented, while preserving the conceptual framework of operations that transform sequences to sequences. Section 2.2.3 expands on this use of sequences as a framework for organizing programs.

`map` は重要な構造です。それが共通なパターンを掴むからだけでなく、リストを扱うより高いレベルの抽象化を確立させるためです。`scale-list` の元の定義ではプログラムの再帰構造はリストのエレメント毎の処理に注意を引きました。`map` を用いた `scale-list` の定義はそのレベルの詳細を抑制し、要素のリストから結果のリストへの拡大変換を強調します。2つの定義の間の違いはコンピュータが異なる処理を行うことではなく（異なりますが）、私達が過程について異って考えていることです。実際に、`map` はリストの要素がどのように抽出され、また結合されるかの詳細からリストを変換する手続の実装を分離する抽象化バリアを強化することを手助けします。Figure 2.1にて示されるバリアのように、この抽象化は私達に列がどのように実装されるかの低レベルの詳細を変更する柔軟性を提供し、その上で列から列へと変換する操作の概念上のフレームワークを保っている。Section 2.2.3はこのプログラムを構成するためのフレームワークとしての列の利用を拡張している。

**Exercise 2.21:** The procedure `square-list` takes a list of numbers as argument and returns a list of the squares of those numbers.

手続 `square-list` は数値のリストを引数として取りそれらの数値の二乗のリストを返す。

```
(square-list (list 1 2 3 4))  
(1 4 9 16)
```

Here are two different definitions of `square-list`. Complete both of them by filling in the missing expressions:

ここに2つの異なる `square-list` がある。失なわれた式を埋めることで両者を完成させよ。

```
(define (square-list items)  
  (if (null? items)  
      nil
```

```
(cons ??) ??)))
(define (square-list items)
  (map ??) ??)))
```

**Exercise 2.22:** Louis Reasoner tries to rewrite the first `square-list` procedure of [Exercise 2.21](#) so that it evolves an iterative process: Louis Reasoner は [Exercise 2.21](#) の最初の `square-list` 手続を書き直し、反復プロセスを展開させようと試みている。

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons (square (car things))
                    answer)))))

(iter items nil))
```

Unfortunately, defining `square-list` this way produces the answer list in the reverse order of the one desired. Why?

残念なことに、`square-list` をこのように定義しては解答のリストは希望の逆順になってしまふ。何故か?

Louis then tries to fix his bug by interchanging the arguments to `cons`:

Louis はそこで彼のバグを `cons` への引数を逆順にすることで直そうと試みた。

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons answer
                    (square (car things)))))))

(iter items nil))
```

This doesn't work either. Explain. これもまたうまく行かない。説明せよ。

**Exercise 2.23:** The procedure `for-each` is similar to `map`. It takes as arguments a procedure and a list of elements. However, rather than forming a list of the results, `for-each` just applies the procedure to each of the elements in turn, from left to right. The values returned by applying the procedure to the elements are not used at all—`for-each` is used with procedures that perform an action, such as printing. For example,

手続 `for-each` は `map` に似ている。手続と要素のリストを引数として取る。しかし結果のリストを形成するのではなく、`for-each` はただ手続を左から右へと毎回各要素に適用する。手続を要素に適用し返された値は全く利用しない—`for-each` は表示のような行動を起こす手続と共に利用される。例えば、

```
(for-each (lambda (x)
  (newline)
  (display x))
  (list 57 321 88))
57
321
88
```

The value returned by the call to `for-each` (not illustrated above) can be something arbitrary, such as true. Give an implementation of `for-each`.

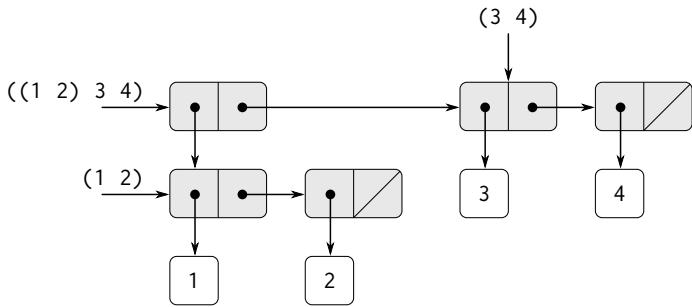
(上では示されていない)`for-each` 呼出による返り値は真のような不定な何かである。`for-each` の実装を与えよ。

## 2.2.2 Hierarchical Structures

The representation of sequences in terms of lists generalizes naturally to represent sequences whose elements may themselves be sequences. For example, we can regard the object `((1 2) 3 4)` constructed by

リストを用いた列の表現は要素が列自身である列を表現することを自然に一般化する。例えば以下の様に構築されたオブジェクト `((1 2) 3 4)` を

```
(cons (list 1 2) (list 3 4))
```



**Figure 2.5:** Structure formed by `(cons (list 1 2) (list 3 4))`.

as a list of three items, the first of which is itself a list, `(1 2)`. Indeed, this is suggested by the form in which the result is printed by the interpreter. Figure 2.5 shows the representation of this structure in terms of pairs.

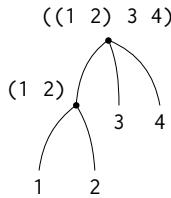
最初の項目はそれ自身がリスト `(1 2)` である、3 つの項目のリストであると見做すことができる。実際に、インタプリタにより表示される結果の形式によりこれは推奨されている。Figure 2.5がペアを用いたこの構造の表現を示している。

Another way to think of sequences whose elements are sequences is as *trees*. The elements of the sequence are the branches of the tree, and elements that are themselves sequences are subtrees. Figure 2.6 shows the structure in Figure 2.5 viewed as a tree.

要素それ自身が列である列のもう 1 つの考え方は木としての考え方である。列の要素は木の枝であり、それ自身が列であるelements(複数の要素) は部分木である。Figure 2.6は木として見た場合のFigure 2.5を示している。

Recursion is a natural tool for dealing with tree structures, since we can often reduce operations on trees to operations on their branches, which reduce in turn to operations on the branches of the branches, and so on, until we reach the leaves of the tree. As an example, compare the `length` procedure of Section 2.2.1 with the `count-leaves` procedure, which returns the total number of leaves of a tree:

再帰は木構造を扱うのに自然なツールです。良く木に対する操作を枝に対



**Figure 2.6:** The list structure in Figure 2.5 viewed as a tree.

する操作へと還元でき、それは順に枝の枝への操作へと還元され、木の葉に辿り着くまで繰り返されます。例として、Section 2.2.1の `length` 手続を木の葉の総数を求める `count-leaves` 手続と比べてみましょう。

```
(define x (cons (list 1 2) (list 3 4)))
(length x)
3
(count-leaves x)
4
(list x x)
(((1 2) 3 4) ((1 2) 3 4))
(length (list x x))
2
(count-leaves (list x x))
8
```

To implement `count-leaves`, recall the recursive plan for computing `length`:  
`count-leaves` を実装するには `length` を求める再帰計画を思い出します。

- Length of a list `x` is 1 plus length of the `cdr` of `x`.  
 リスト `x` の `length` は `x` の `cdr` の `length` に 1 を足した物
- Length of the empty list is 0.  
 空リストの `length` は 0

`Count-leaves` is similar. The value for the empty list is the same:  
`count-leaves` も同様えす。空リストの値は同じで

- Count-leaves of the empty list is 0.

空リストの count-leaves は 0

But in the reduction step, where we strip off the `car` of the list, we must take into account that the `car` may itself be a tree whose leaves we need to count. Thus, the appropriate reduction step is

しかし集約ステップにおいて、リストの `car` を取り除く時、`car` はそれ自身が後で数えねばならない木である可能性があることを計算に入れねばなりません。従って適切な集約ステップは

- Count-leaves of a tree  $x$  is count-leaves of the `car` of  $x$  plus count-leaves of the `cdr` of  $x$ .

木  $x$  の count-leaves は  $x$  の `car` の count-leaves と、 $x$  の `cdr` の count-leaves の和

Finally, by taking `cars` we reach actual leaves, so we need another base case:  
最終的に `car` を取ることにより実際の葉に届くので別の規範を必要とする。

- Count-leaves of a leaf is 1.

葉の count-leaves は 1

To aid in writing recursive procedures on trees, Scheme provides the primitive predicate `pair?`, which tests whether its argument is a pair. Here is the complete procedure:<sup>23</sup>

木に対する再帰手続を書くのを助けるために、Scheme はプリミティブな手続 `pair?` を提供します。`pair?` は引数がペアであるかをテストします。以下に完成した手続を置きます。<sup>24</sup>

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                  (count-leaves (cdr x)))))))
```

<sup>23</sup>The order of the first two clauses in the `cond` matters, since the empty list satisfies `null?` and also is not a pair.

<sup>24</sup>`cond` の最初の 2 つの項の順が大事です。空リストは `null?` を満たし、その上でペアでもありません

**Exercise 2.24:** Suppose we evaluate the expression (list 1 (list 2 (list 3 4))). Give the result printed by the interpreter, the corresponding box-and-pointer structure, and the interpretation of this as a tree (as in Figure 2.6).

式 (list 1 (list 2 (list 3 4))) を評価したとする。インタプリタの表示する結果、対応する箱と点構造、木としての解釈 (Figure 2.6相当) を示せ。

**Exercise 2.25:** Give combinations of `cars` and `cdrs` that will pick 7 from each of the following lists:

以下の各リストから 7 を抽出する `car` と `cdr` の組み合わせを与えよ。

```
(1 3 (5 7) 9)
((7))
(1 (2 (3 (4 (5 (6 7))))))
```

**Exercise 2.26:** Suppose we define `x` and `y` to be two lists:

2つのリスト `x` と `y` を定義したとする。

```
(define x (list 1 2 3))
(define y (list 4 5 6))
```

What result is printed by the interpreter in response to evaluating each of the following expressions:

以下の各式を評価した場合にレスポンスとしてインタプリタがどのような結果を表示するか?

```
(append x y)
(cons x y)
(list x y)
```

**Exercise 2.27:** Modify your `reverse` procedure of Exercise 2.18 to produce a `deep-reverse` procedure that takes a list as argument and returns as its value the list with its elements reversed and with all sublists deep-reversed as well. For example,

Exercise 2.18 `reverse` 手続を変更してリストを引数として取り、全ての要素が逆順に、さらに全てのサブリストも同様に逆順にされたリストをその値として返す手続 `deep-reverse` を作れ。例として、

```
(define x (list (list 1 2) (list 3 4)))
x
((1 2) (3 4))
(reverse x)
((3 4) (1 2))
(deep-reverse x)
((4 3) (2 1))
```

**Exercise 2.28:** Write a procedure `fringe` that takes as argument a tree (represented as a list) and returns a list whose elements are all the leaves of the tree arranged in left-to-right order. For example, リストとして表現された木を引数に取り、その木の全ての葉を左から右への順で要素としたリストを返す手続 `fringe` を書け。

```
(define x (list (list 1 2) (list 3 4)))
(fringe x)
(1 2 3 4)
(fringe (list x x))
(1 2 3 4 1 2 3 4)
```

**Exercise 2.29:** A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. We can represent a binary mobile using compound data by constructing it from two branches (for example, using `list`):

バイナリモバイル (binary mobile)<sup>25</sup> は左の枝と右の枝の 2 つの枝で構成される。各枝はある長さを持つ棒であり、そこから重りか別のバイナリモバイルをぶら下げる。バイナリモバイルを複合データを用いて 2 つの枝から組み立てることで表現できる。(例えば `list` を用いる。)

```
(define (make-mobile left right)
  (list left right))
```

---

<sup>25</sup> 訳注：天井から糸で釣ってあり、絶妙なバランスで揺れ、回る数々の棒のインテリア。枝が必ず 2 つに分かれるのでバイナリ (二進) と名付けられている。Google Images で `mobile` を検索すると実物が見られる。名詞 `mobile` は米語でモービル、英語でモバイルと発音するが前者は乗り物のイメージが強いので後者を採用した

A branch is constructed from a `length` (which must be a number) together with a `structure`, which may be either a number (representing a simple weight) or another mobile:

枝は `length`(数値であること) と `structure` から組み立てられる。`structure` は数値(簡単に重りを表わす) かまたは他のモバイルである。

```
(define (make-branch length structure)
  (list length structure))
```

- a Write the corresponding selectors `left-branch` and `right-branch`, which return the branches of a mobile, and `branch-length` and `branch-structure`, which return the components of a branch.

対応するセレクタ `left-branch` を `right-branch` を書け。このセレクタはモバイルの複数の枝を返す。また `branch-length` と `branch-structure` は枝のそれぞれのコンポーネント(構成要素)を返す。

- b Using your selectors, define a procedure `total-weight` that returns the total weight of a mobile.

セレクタを用いて手続 `total-weight` を定義せよ。それはモバイルの総重量を返す。

- c A mobile is said to be *balanced* if the torque applied by its top-left branch is equal to that applied by its top-right branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. Design a predicate that tests whether a binary mobile is balanced.

モバイルは一番上の左枝にかかるトルク(回転力)が一番上の右の枝にかかるトルクと等しい時(これはつまり、もし左の棒の長さとその棒にかかる重さを掛けた値が、相対する右側の積の値と同じ場合である)、かつ各部分モバイルも全て同様である場合に限り、*balanced*(バランスが取れた状態)であると言う。あるバイナリモバイルがバランスが取れているかテストする述語を設計せよ。

- d Suppose we change the representation of mobiles so that the constructors are

モバイルの表現をコンストラクタが以下になるように変更するを考える。

```
(define (make-mobile left right) (cons left right))
(define (make-branch length structure)
  (cons length structure))
```

How much do you need to change your programs to convert to the new representation?

新しい表現へとあなたのプログラムを変更するのにどれほどが必要か?

## Mapping over trees

Just as `map` is a powerful abstraction for dealing with sequences, `map` together with recursion is a powerful abstraction for dealing with trees. For instance, the `scale-tree` procedure, analogous to `scale-list` of [Section 2.2.1](#), takes as arguments a numeric factor and a tree whose leaves are numbers. It returns a tree of the same shape, where each number is multiplied by the factor. The recursive plan for `scale-tree` is similar to the one for `count-leaves`:

`map` が列を扱うのに強力な抽象化であるのと同様に、再帰を伴なう `map` は木を扱うのに強力な抽象化です。例えば[Section 2.2.1](#)の `scale-list` に同類な `scale-tree` 手続は引数として因数と葉が数値である木を取ります。これは同じ形の木を返しますが、各数値は因数により乗算されます。`scale-tree` の再帰計画は `count-leaves` に対する物に似ています。

```
(define (scale-tree tree factor)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (* tree factor))
        (else (cons (scale-tree (car tree) factor)
                     (scale-tree (cdr tree) factor))))))
(scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7)) 10)
(10 (20 (30 40) 50) (60 70))
```

Another way to implement `scale-tree` is to regard the tree as a sequence of sub-trees and use `map`. We map over the sequence, scaling each sub-tree in turn,

and return the list of results. In the base case, where the tree is a leaf, we simply multiply by the factor:

`scale-tree` を実装するもう 1 つの方法は木を部分木の列と見做し `map` を使用します。列全体に `map` をかけ、各部分木を順に拡大し、結果のリストを返します。その木が葉である場合には単純に因数を掛けます。

```
(define (scale-tree tree factor)
  (map (lambda (sub-tree)
    (if (pair? sub-tree)
        (scale-tree sub-tree factor)
        (* sub-tree factor)))
    tree))
```

Many tree operations can be implemented by similar combinations of sequence operations and recursion.

多くの木の操作が同様な列操作と再帰の組み合わせにて実装可能です。

**Exercise 2.30:** Define a procedure `square-tree` analogous to the `square-list` procedure of [Exercise 2.21](#). That is, `square-tree` should behave as follows:

[Exercise 2.21](#) の `square-list` と同様の手続 `square-tree` を定義せよ。`square-tree` は以下の振舞を行う。

```
(square-tree
  (list 1
    (list 2 (list 3 4) 5)
    (list 6 7))
  (1 (4 (9 16) 25) (36 49)))
```

Define `square-tree` both directly (i.e., without using any higher-order procedures) and also by using `map` and recursion.

`square-tree` を直接な (つまり高階関数を全く用いない) 方法と `map` と再帰を用いる方法の両者を定義せよ。

**Exercise 2.31:** Abstract your answer to [Exercise 2.30](#) to produce a procedure `tree-map` with the property that `square-tree` could be defined as

あなたの [Exercise 2.30](#)への解答を抽象化し、手続 `tree-map` を作れ。`tree-map` を用いて `square-tree` は以下のように定義できる。

```
(define (square-tree tree) (tree-map square tree))
```

**Exercise 2.32:** We can represent a set as a list of distinct elements, and we can represent the set of all subsets of the set as a list of lists. For example, if the set is (1 2 3), then the set of all subsets is (() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3)). Complete the following definition of a procedure that generates the set of subsets of a set and give a clear explanation of why it works:

集合は識別可能な要素のリストとして表現できる。そして集合の全ての部分集合集合をリストのリストとして表わせられる。例えば、集合が (1 2 3) である時、全ての部分集合の集合は (() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3)) だ。以下の集合の全ての部分集合の集合を生成する手続の定義を完成し、なぜうまくいくのかを明確に説明せよ。

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map (lambda (?) (cons (car s) ?)) rest))))))
```

### 2.2.3 Sequences as Conventional Interfaces

In working with compound data, we've stressed how data abstraction permits us to design programs without becoming enmeshed in the details of data representations, and how abstraction preserves for us the flexibility to experiment with alternative representations. In this section, we introduce another powerful design principle for working with data structures—the use of *conventional interfaces*.

複合データを用いて働く場合、これまでデータ抽象化がどれだけプログラムの設計をデータ表現の詳細に陥らざに行えるか、また抽象化がどれだけ代替的な表現方法を試みる柔軟性を保つかについて強調してきました。この節ではもう 1 つの強力なデータ構造を用いる設計原則を紹介します。*conventional interfaces*(慣習的インターフェイス) の使用です。

In Section 1.3 we saw how program abstractions, implemented as higher-order procedures, can capture common patterns in programs that deal with

numerical data. Our ability to formulate analogous operations for working with compound data depends crucially on the style in which we manipulate our data structures. Consider, for example, the following procedure, analogous to the `count-leaves` procedure of [Section 2.2.2](#), which takes a tree as argument and computes the sum of the squares of the leaves that are odd:

[Section 1.3](#)においてプログラム抽象化、高階手続としての実装がどのようにして数値データを取り扱うプログラムの共通パターンを掴むことができるのかを学んできました。複合データを取り扱う類似の操作を形式化する能力は決定的にデータ構造を取り扱うスタイルに依存します。例えば次の手続について考えてみて下さい。[Section 2.2.2](#)の `count-leaves` 手續に類似しており、木を引数として取り、奇数の葉の二乗の合計を求めます。

```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                  (sum-odd-squares (cdr tree)))))))
```

On the surface, this procedure is very different from the following one, which constructs a list of all the even Fibonacci numbers  $\text{Fib}(k)$ , where  $k$  is less than or equal to a given integer  $n$ :

表面上では、この手続は以下の物ととても異なっています。以下では全ての偶数のフィボナッチ数  $\text{Fib}(k)$  のリストを、 $k$  が与えられた  $n$  以下の範囲にて作成しています。

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

Despite the fact that these two procedures are structurally very different, a more abstract description of the two computations reveals a great deal of similarity. The first program

これらの 2 つの手続は構造的にとても異なっているという事実にも係らず、2 つの計算のより抽象的な記述は大きな類似性を明らかにします。最初のプログラムは

- enumerates the leaves of a tree; 木の葉を列挙する
- filters them, selecting the odd ones; フィルタを通して奇数のみを選ぶ
- squares each of the selected ones; and 選択された数の二乗を求める
- accumulates the results using `+`, starting with 0. 初期値 0 にて `+` を用いて集積する。

The second program

2 つ目のプログラムは

- enumerates the integers from 0 to  $n$ ; 0 から  $n$  を列挙する
- computes the Fibonacci number for each integer; 各整数のフィボナッチ数を求める
- filters them, selecting the even ones; and フィルタを通して偶数を選択する
- accumulates the results using `cons`, starting with the empty list.

初期値は空リストにて `cons` を用いて結果を集積する

A signal-processing engineer would find it natural to conceptualize these processes in terms of signals flowing through a cascade of stages, each of which implements part of the program plan, as shown in Figure 2.7. In `sum-odd-squares`, we begin with an *enumerator*, which generates a “signal” consisting of the leaves of a given tree. This signal is passed through a *filter*, which eliminates all but the odd elements. The resulting signal is in turn passed through a *map*, which is a “transducer” that applies the `square` procedure to each element. The output of the map is then fed to an *accumulator*, which combines the elements using `+`, starting from an initial 0. The plan for `even-fibs` is analogous.

信号処理のエンジニアはこれらの処理をステージのカスケードを通して流れる信号を用いて処理するという概念的説明を自然だと思われるでしょう。各ステージはプログラム設計の部分を Figure 2.7 に示すように実装しています。`sum-odd-squares` では *enumerator*(エニュメレータ) にて始めました。それは与えられた木の葉から成る“信号”を生成します。この信号は *filter*(フィルタ)

を通して奇数要素以外を全て取り除きます。残った信号は順に“変換器”である `map` を通し、それが `square` 手続を各要素に適用します。`map` の出力は次に `accumulator`(集積機) に与えられ、それが要素を初期値 0 と + を用いて連結します。`even-fibs` の設計も同様です。

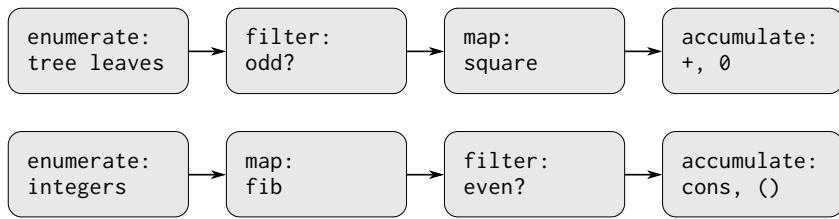
Unfortunately, the two procedure definitions above fail to exhibit this signal-flow structure. For instance, if we examine the `sum-odd-squares` procedure, we find that the enumeration is implemented partly by the `null?` and `pair?` tests and partly by the tree-recursive structure of the procedure. Similarly, the accumulation is found partly in the tests and partly in the addition used in the recursion. In general, there are no distinct parts of either procedure that correspond to the elements in the signal-flow description. Our two procedures decompose the computations in a different way, spreading the enumeration over the program and mingling it with the map, the filter, and the accumulation. If we could organize our programs to make the signal-flow structure manifest in the procedures we write, this would increase the conceptual clarity of the resulting code.

残念ながら上記の 2 つの手続の定義はこの信号の流れの構造を提示するには失敗しています。例えば `sum-odd-squares` を調べてみると `enumeration`(列挙) は部分的に `null?` の `pair?` のテストにて実装され、別の部分では手続の木再帰構造により実装されています。同様に集積は部分的にテストの中に見つかり、また部分的に再帰中で使用される足し算に見つかります。全体的にどちらの手続も信号の流れの記述内の要素に関連する明確な部分は存在しません。2 つの手続は演算を異なる方法で分解し、列挙をプログラム全体に広げて `map`, `filter`, `accumulation` に混ぜました。もしプログラムを手続中に信号処理構造の宣言を作るように構成できるのであれば結果としてのコードの概念の明快さを増すことができるでしょう。

## Sequence Operations

The key to organizing programs so as to more clearly reflect the signal-flow structure is to concentrate on the “signals” that flow from one stage in the process to the next. If we represent these signals as lists, then we can use list operations to implement the processing at each of the stages. For instance, we can implement the mapping stages of the signal-flow diagrams using the `map` procedure from Section 2.2.1:

プログラムを体系化し信号伝達構造をより明確に反映する鍵はある段階の



**Figure 2.7:** The signal-flow plans for the

処理から次へと流れる“信号”に集中することです。もしこれらの信号をリストとして表現するなら、各段階の処理をリスト操作を用いて実装できます。例えば信号伝達図の map の段階を [Section 2.2.1](#) の map 手続を用いて実装できます。

```
(map square (list 1 2 3 4 5))
(1 4 9 16 25)
```

Filtering a sequence to select only those elements that satisfy a given predicate is accomplished by

列をフィルタリングして与えられた述語を満足する要素のみを選択することは以下の様に達成できます。

```
(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence)))))
        (else (filter predicate (cdr sequence)))))
```

For example,

例として、

```
(filter odd? (list 1 2 3 4 5))
(1 3 5)
```

Accumulations can be implemented by

集積は次のように実装します。

```
(define (accumulate op initial sequence)
```

```

(if (null? sequence)
    initial
    (op (car sequence)
        (accumulate op initial (cdr sequence))))))
(accumulate + 0 (list 1 2 3 4 5))
15
(accumulate * 1 (list 1 2 3 4 5))
120
(accumulate cons nil (list 1 2 3 4 5))
(1 2 3 4 5)

```

All that remains to implement signal-flow diagrams is to enumerate the sequence of elements to be processed. For `even-fibs`, we need to generate the sequence of integers in a given range, which we can do as follows:

信号伝達図を実装するのに残っているもの全ては処理すべき要素の列を列挙することです。`even-fibs` のためには与えられた区間の整数の列を生成しなければならず、以下のように行うことができます。

```

(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low (enumerate-interval (+ low 1) high))))
(enumerate-interval 2 7)
(2 3 4 5 6 7)

```

To enumerate the leaves of a tree, we can use<sup>26</sup>

木の葉を列挙するには、以下の様にして可能です。<sup>27</sup>

```

(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree)))))))

```

---

<sup>26</sup>This is, in fact, precisely the `fringe` procedure from [Exercise 2.28](#). Here we've renamed it to emphasize that it is part of a family of general sequence-manipulation procedures.

<sup>27</sup>これは実際には [Exercise 2.28](#)の `fringe` 手続そのものです。ここではその名を変えて列操作手続一般に属するパートであることを強調しています。

```
(enumerate-tree (list 1 (list 2 (list 3 4)) 5))
(1 2 3 4 5)
```

Now we can reformulate `sum-odd-squares` and `even-fibs` as in the signal-flow diagrams. For `sum-odd-squares`, we enumerate the sequence of leaves of the tree, filter this to keep only the odd numbers in the sequence, square each element, and sum the results:

これで `sum-odd-squares` と `even-fibs` を信号伝達図の様に再形式化することができます。`sum-odd-squares` のためには木の葉の列を列挙し、これをフィルタにかけ列の奇数のみを保持し、各要素を二乗し、結果の合計を求めます。

```
(define (sum-odd-squares tree)
  (accumulate
    + 0 (map square (filter odd? (enumerate-tree tree)))))
```

For `even-fibs`, we enumerate the integers from 0 to  $n$ , generate the Fibonacci number for each of these integers, filter the resulting sequence to keep only the even elements, and accumulate the results into a list:

`even-fibs` に対しては 0 から  $n$  の整数を列挙し、これらの整数のそれぞれに対するフィボナッチ数を生成し、結果列をフィルタにかけ偶数の要素のみを保持し、結果をリストの中に集積します。

```
(define (even-fibs n)
  (accumulate
    cons
    nil
    (filter even? (map fib (enumerate-interval 0 n)))))
```

The value of expressing programs as sequence operations is that this helps us make program designs that are modular, that is, designs that are constructed by combining relatively independent pieces. We can encourage modular design by providing a library of standard components together with a conventional interface for connecting the components in flexible ways.

列操作としての伝達プログラムの価値はこれがモジュラ形式のプログラムデザインを行うことを手助けしてくれることにあります。モジュラであるとは相対的に独立した部品を組み立てることで構築される設計です。柔軟な形でコンポーネントを接続するための慣習的なインターフェイスと共に、標準コンポーネントのライブラリを提供することで、モジュラ設計を推進することができます。

Modular construction is a powerful strategy for controlling complexity in engineering design. In real signal-processing applications, for example, designers regularly build systems by cascading elements selected from standardized families of filters and transducers. Similarly, sequence operations provide a library of standard program elements that we can mix and match. For instance, we can reuse pieces from the `sum-odd-squares` and `even-fibs` procedures in a program that constructs a list of the squares of the first  $n + 1$  Fibonacci numbers:

モジュラ構築は複雑性を工学的設計において複雑性をコントロールすることに対して強力な戦略です。例えば現実の信号処理アプリケーションでは、設計者は恒常にフィルタと変換器の標準化されたグループから選択された要素を繋げることでシステムを構築します。同様に列操作は標準的プログラム要素を様々な組合せたライブラリを提供します。実例として私達は `sum-odd-squares` と `even-fibs` の手続の部品を用いて、フィボナッチ数の最初から  $n + 1$  個の二乗のリストを作成できます。

```
(define (list-fib-squares n)
  (accumulate
    cons
    nil
    (map square (map fib (enumerate-interval 0 n)))))

(list-fib-squares 10)
(0 1 1 4 9 25 64 169 441 1156 3025)
```

We can rearrange the pieces and use them in computing the product of the squares of the odd integers in a sequence:

部品を再配置し、列の奇数の二乗の積を計算するのに使うことも可能です。

```
(define (product-of-squares-of-odd-elements sequence)
  (accumulate * 1 (map square (filter odd? sequence))))
(product-of-squares-of-odd-elements (list 1 2 3 4 5))
225
```

We can also formulate conventional data-processing applications in terms of sequence operations. Suppose we have a sequence of personnel records and we want to find the salary of the highest-paid programmer. Assume that we have a selector `salary` that returns the salary of a record, and a predicate `programmer?` that tests if a record is for a programmer. Then we can write

慣習的なデータ処理アプリケーションを列操作を用いて説明することもできます。個人の記録の列があるとし、最も高給なプログラマの給料を見つけていとします。記録の給料を返すセレクタ `salary` と記録がプログラマの物であるかを判定する述語 `programmer?` があるとします。すると以下のように書けます。

```
(define (salary-of-highest-paid-programmer records)
  (accumulate
    max 0 (map salary (filter programmer? records))))
```

These examples give just a hint of the vast range of operations that can be expressed as sequence operations.<sup>28</sup>

この例は列操作として表わすことができる広範囲な処理のヒントを与えたにすぎません。<sup>29</sup>

Sequences, implemented here as lists, serve as a conventional interface that permits us to combine processing modules. Additionally, when we uniformly represent structures as sequences, we have localized the data-structure dependencies in our programs to a small number of sequence operations. By changing these, we can experiment with alternative representations of sequences, while leaving the overall design of our programs intact. We will exploit this capability in Section 3.5, when we generalize the sequence-processing paradigm to admit infinite sequences.

---

<sup>28</sup>Richard Waters (1979) developed a program that automatically analyzes traditional Fortran programs, viewing them in terms of maps, filters, and accumulations. He found that fully 90 percent of the code in the Fortran Scientific Subroutine Package fits neatly into this paradigm. One of the reasons for the success of Lisp as a programming language is that lists provide a standard medium for expressing ordered collections so that they can be manipulated using higher-order operations. The programming language APL owes much of its power and appeal to a similar choice. In APL all data are represented as arrays, and there is a universal and convenient set of generic operators for all sorts of array operations.

<sup>29</sup>Richard Waters (1979)は伝統的な Fortran プログラムを自動的に解析し、`map`, フィルタ, 集積を用いてそらを俯瞰するプログラムを開発しました。彼は Fortran の科学サブルーチンパッケージのコードの実に 90% がこのパラダイムにうまくはまるこことを発見しました。Lisp がプログラミング言語として成功した理由の 1 つにリストが順序有り集合を表すのに標準的な手段を提供したことがあり、そのため高階手続を用いて操作することが可能になりました。プログラミング言語 APL はその力と魅力の多くを同様の選択のおかげで得ました。APL では全てのデータは配列として表現され、統一的、かつ便利な全ての種類の配列操作のための包括的な命令集合が存在します。

ここではリストとして実装された列は処理モジュールを接続することを可能にする慣習的インターフェイスとしての役割を行います。その上、私達が構造を列として統一的に表現した時、私達はプログラム中のデータ構造依存性を少ない数の列操作へと局所化しました。これらを変更することで、プログラム設計を全体的に保存したまま列の代替的表現方法を試みることができます。私達はこの能力をSection 3.5にて列処理パラダイムを無限列を許可するよう一般化する時に利用します。

**Exercise 2.33:** Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as accumulations:

欠けた式を埋めて次の集積としてのいくつかの基本的なリスト操作命令の定義を完成させよ。

```
(define (map p sequence)
  (accumulate (lambda (x y) (??)) nil sequence))
(define (append seq1 seq2)
  (accumulate cons (??) (??)))
(define (length sequence)
  (accumulate (??) 0 sequence))
```

**Exercise 2.34:** Evaluating a polynomial in  $x$  at a given value of  $x$  can be formulated as an accumulation. We evaluate the polynomial  $x$  の多項式を  $x$  の与えられた値にて評価することは集積として表すことができる。以下の多項式を良く知られたHorner's rule(ホーナー法)と呼ばれるアルゴリズムを用いて評価する。

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

using a well-known algorithm called *Horner's rule*, which structures the computation as

ホーナー法は上記の計算を以下のような構造にする。

$$(\dots (a_n x + a_{n-1}) x + \dots + a_1) x + a_0.$$

In other words, we start with  $a_n$ , multiply by  $x$ , add  $a_{n-1}$ , multiply by  $x$ , and so on, until we reach  $a_0$ .<sup>30</sup>

---

<sup>30</sup>According to Knuth 1981, this rule was formulated by W. G. Horner early in the

すなわち  $a_n$  で始め、 $x$  を掛け、 $a_{n-1}$  を足し、 $x$  を掛け、を  $a_0$  に到達するまで繰り返す。<sup>31</sup>

Fill in the following template to produce a procedure that evaluates a polynomial using Horner's rule. Assume that the coefficients of the polynomial are arranged in a sequence, from  $a_0$  through  $a_n$ .

以下のテンプレートを埋めホーナー法を用いて多項式を評価する手続を作り出せ。多項式の係数  $a_0$  から  $a_n$  は列で用意されると想定せよ。

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms) (?))
              0
              coefficient-sequence))
```

For example, to compute  $1 + 3x + 5x^3 + x^5$  at  $x = 2$  you would evaluate

例えば  $1 + 3x + 5x^3 + x^5$  を  $x = 2$  の時の値を求める場合、次のように評価を行う。

---

nineteenth century, but the method was actually used by Newton over a hundred years earlier. Horner's rule evaluates the polynomial using fewer additions and multiplications than does the straightforward method of first computing  $a_n x^n$ , then adding  $a_{n-1} x^{n-1}$ , and so on. In fact, it is possible to prove that any algorithm for evaluating arbitrary polynomials must use at least as many additions and multiplications as does Horner's rule, and thus Horner's rule is an optimal algorithm for polynomial evaluation. This was proved (for the number of additions) by A. M. Ostrowski in a 1954 paper that essentially founded the modern study of optimal algorithms. The analogous statement for multiplications was proved by V. Y. Pan in 1966. The book by Borodin and Munro (1975) provides an overview of these and other results about optimal algorithms.

<sup>31</sup>Knuth 1981によるところこの方法は W. G. Horner により 19 世紀始めに考案された。しかしその手法は実際にはニュートンにより 100 年を越えた前に使用されていた。ホーナー法は多項式を最初に  $a_n x^n$  を計算し、 $a_{n-1} x^{n-1}$  を足すを繰り返す直接的な方法より少ない回数の和と積を用いて評価する。実際に任意の多項式を評価するためのどんなアルゴリズムもホーナー法が必要な数と同じ数の和と積を使用する必要があることを証明することが可能である。従ってホーナー法は多項式評価において最適なアルゴリズムである。これは(和の数において)A. M. Ostrowski による 1954 年の論文にて証明され、これが現代の最適アルゴリズム研究の基礎を築いた。同様の説明が積の数について V. Y. Pan により 1966 年に証明された。Borodin and Munro (1975)による本がこれらと他の最適アルゴリズムについての結果について概観している。

```
(horner-eval 2 (list 1 3 0 5 0 1))
```

**Exercise 2.35:** Redefine `count-leaves` from Section 2.2.2 as an accumulation:

Section 2.2.2の `count-leaves` を集積として再定義せよ。

```
(define (count-leaves t)
  (accumulate ?? ?? (map ?? ??)))
```

**Exercise 2.36:** The procedure `accumulate-n` is similar to `accumulate` except that it takes as its third argument a sequence of sequences, which are all assumed to have the same number of elements. It applies the designated accumulation procedure to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a sequence of the results. For instance, if `s` is a sequence containing four sequences, ((1 2 3) (4 5 6) (7 8 9) (10 11 12)), then the value of (`accumulate-n + 0 s`) should be the sequence (22 26 30). Fill in the missing expressions in the following definition of `accumulate-n`:

手続 `accumulate-n` は `accumulate` に似ているが 3 番目の引数として列の列を取り、その要素の列の長さは全て一定である。指定された集積手続を複数の列の最初の要素、二番目の要素、以下繰り返し、を全て連結するため適用し、結果の列を返す。例えばもし `s` が 4 つの列を含む列、((1 2 3) (4 5 6) (7 8 9) (10 11 12)) である時、(`accumulate-n + 0 s`) の値は列 (22 26 30) にならなければならない。以下の `accumulate-n` の定義の欠けた式を補え。

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init ??)
            (accumulate-n op init ??))))
```

**Exercise 2.37:** Suppose we represent vectors  $\mathbf{v} = (v_i)$  as sequences of numbers, and matrices  $\mathbf{m} = (m_{ij})$  as sequences of vectors (the rows of the matrix). For example, the matrix

ベクトル  $\mathbf{v} = (v_i)$  を数値の列として表現し、行列  $\mathbf{m} = (m_{ij})$  をベクトル(行列の行)の列として表現するとする。例えば以下の行列は、

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{pmatrix}$$

is represented as the sequence ((1 2 3 4) (4 5 6 6) (6 7 8 9)). With this representation, we can use sequence operations to concisely express the basic matrix and vector operations. These operations (which are described in any book on matrix algebra) are the following:

列 ((1 2 3 4) (4 5 6 6) (6 7 8 9)) として表現される。この表現と共に列操作を用いることで簡潔に基本的な行列とベクトルの操作を表現することができる。これらの操作は(行列演算のどんな本にも記述されている)次のものである。

<code>(dot-product v w)</code>	returns the sum $\sum_i v_i w_i$ ,
<code>(matrix-*-vector m v)</code>	returns the vector $\mathbf{t}$ , where $t_i = \sum_j m_{ij} v_j$ ,
<code>(matrix-*-matrix m n)</code>	returns the matrix $\mathbf{p}$ , where $p_{ij} = \sum_k m_{ik} n_{kj}$ ,
<code>(transpose m)</code>	returns the matrix $\mathbf{n}$ , where $n_{ij} = m_{ji}$ .

We can define the dot product as<sup>32</sup>

ドット積を次のように定義できる。<sup>33</sup>

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

Fill in the missing expressions in the following procedures for computing the other matrix operations. (The procedure `accumulate-n` is defined in [Exercise 2.36](#).)

以下の他の行列操作を演算するための手続の欠けた式を補え。(手続 `accumulate-n` は [Exercise 2.36](#)で定義されている。)

---

<sup>32</sup>This definition uses the extended version of `map` described in [Footnote 12](#).

<sup>33</sup>この定義は[Footnote 12](#)にて説明した `map` の拡張バージョンを使用する

```

(define (matrix-*-vector m v)
  (map ?? m))
(define (transpose mat)
  (accumulate-n ?? ?? mat))
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map ?? m)))

```

**Exercise 2.38:** The `accumulate` procedure is also known as `fold-right`, because it combines the first element of the sequence with the result of combining all the elements to the right. There is also a `fold-left`, which is similar to `fold-right`, except that it combines elements working in the opposite direction:

`accumulate` 手続はまた `fold-right` としても知られている。それが列の最初の要素と右側の要素全てを結合した結果とを結合するためである。`fold-left` も存在し、`fold-right` と似ているが、要素の結合を逆の向きに行う。

```

(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))

```

What are the values of  
以下の式の値はいくらか。

```

(fold-right / 1 (list 1 2 3))
(fold-left / 1 (list 1 2 3))
(fold-right list nil (list 1 2 3))
(fold-left list nil (list 1 2 3))

```

Give a property that `op` should satisfy to guarantee that `fold-right` and `fold-left` will produce the same values for any sequence.

`op` が `fold-right` と `fold-left` にて同じ値を任意の列に対し生成することを保証するのに必要な特性を答えよ。

**Exercise 2.39:** Complete the following definitions of `reverse` (Exercise 2.18) in terms of `fold-right` and `fold-left` from Exercise 2.38:

以下の `reverse`(Exercise 2.18) の `fold-right` と `fold-left`(Exercise 2.38) を用いた定義を完成させよ。

```
(define (reverse sequence)
  (fold-right (lambda (x y) (??)) nil sequence))
(define (reverse sequence)
  (fold-left (lambda (x y) (??)) nil sequence))
```

## Nested Mappings

We can extend the sequence paradigm to include many computations that are commonly expressed using nested loops.<sup>34</sup> Consider this problem: Given a positive integer  $n$ , find all ordered pairs of distinct positive integers  $i$  and  $j$ , where  $1 \leq j < i \leq n$ , such that  $i + j$  is prime. For example, if  $n$  is 6, then the pairs are the following:

列のパラダイムを拡張し、一般的に入れ子ループを用いて表現される多くの演算を含めてみます。<sup>35</sup> 次の問題について考えてみて下さい：正の整数  $n$  を与えられた時、異なる正の整数  $i$  と  $j$  の全ての順序付けペアを見つけよ。条件として  $1 \leq j < i \leq n$ 、かつ  $i + j$  は素数である。例として、もし  $n$  が 6 ならばペアは以下の通りである。

$i$	2	3	4	4	5	6	6
$j$	1	2	1	3	2	1	5
$i + j$	3	5	5	7	7	7	11

<sup>34</sup>This approach to nested mappings was shown to us by David Turner, whose languages KRC and Miranda provide elegant formalisms for dealing with these constructs. The examples in this section (see also Exercise 2.42) are adapted from Turner 1981. In Section 3.5.3, we'll see how this approach generalizes to infinite sequences.

<sup>35</sup>この入れ子マッピングへの取り組み方は David Turner により示されました。彼の言語である KRC と Miranda はこれらの構成概念を取り扱うための洗練された形式主義を与えました。この節の例（また Exercise 2.42 も参照）は Turner 1981 より翻案されました。Section 3.5.3 ではこのやり方が無限長列に対しどのように一般化されるかを学びます。

A natural way to organize this computation is to generate the sequence of all ordered pairs of positive integers less than or equal to  $n$ , filter to select those pairs whose sum is prime, and then, for each pair  $(i, j)$  that passes through the filter, produce the triple  $(i, j, i + j)$ .

この演算を体系化する自然な方法は全ての順序付けられた  $n$  以下の正の整数のペアを生成し、フィルタを通してその合計が素数であるもののみを選択し、フィルタを通った各ペア  $(i, j)$  に対し三つ組  $(i, j, i + j)$  を生成することです。

Here is a way to generate the sequence of pairs: For each integer  $i \leq n$ , enumerate the integers  $j < i$ , and for each such  $i$  and  $j$  generate the pair  $(i, j)$ . In terms of sequence operations, we map along the sequence (`enumerate-interval 1 n`). For each  $i$  in this sequence, we map along the sequence (`enumerate-interval 1 (- i 1)`). For each  $j$  in this latter sequence, we generate the pair `(list i j)`. This gives us a sequence of pairs for each  $i$ . Combining all the sequences for all the  $i$  (by accumulating with `append`) produces the required sequence of pairs:<sup>36</sup>

ここでペアの列を生成する方法を上げます：全ての整数  $i \leq n$  に対し、整数  $j < i$  を列挙し、全てのそのような  $i$  と  $j$  に対し、ペア  $(i, j)$  を生成します。列操作を用いて、列 `(enumerate-interval 1 n)` に沿って `map` を行います。この列の各  $i$  に対し、列 `(enumerate-interval 1 (- i 1))` に沿って `map` を行います。この後者の列の  $j$  に対し、ペア `(list i j)` を生成します。これが全ての  $i$  に対するペアの列を与えます。全ての  $i$  に対する全ての列を `(append` を用いて集積することにより) 接続することで要求されたペアの列を生成します。<sup>37</sup>

```
(accumulate
  append nil (map (lambda (i)
    (map (lambda (j) (list i j))
      (enumerate-interval 1 (- i 1))))
    (enumerate-interval 1 n)))
```

The combination of mapping and accumulating with `append` is so common in this sort of program that we will isolate it as a separate procedure:

`map` と集積の組み合わせを `append` と共に用いるのはこの種のプログラムにおいてとても一般的ですので、これを分離したプログラムとして分けます。

---

<sup>36</sup>We're representing a pair here as a list of two elements rather than as a Lisp pair. Thus, the "pair"  $(i, j)$  is represented as `(list i j)`, not `(cons i j)`.

<sup>37</sup>ここではペアを 2 つの要素のリストとして表現しており、Lisp のペアとしてではありません。従って "ペア"  $(i, j)$  は `(list i j)` であり、`(cons i j)` ではありません。

```
(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))
```

Now filter this sequence of pairs to find those whose sum is prime. The filter predicate is called for each element of the sequence; its argument is a pair and it must extract the integers from the pair. Thus, the predicate to apply to each element in the sequence is

これでこのペアの列をフィルタにかけ和が素数であるものを探します。フィルタの述語が各要素に対して呼ばれます。その引数はペアであり、ペアから整数を抽出せねばなりません。従って列の各要素に適用される述語は以下のようになります。

```
(define (prime-sum? pair)
  (prime? (+ (car pair) (cadr pair))))
```

Finally, generate the sequence of results by mapping over the filtered pairs using the following procedure, which constructs a triple consisting of the two elements of the pair along with their sum:

最後に、フィルタを通ったペア全体に以下の手続を用いて map をかけた結果の列を生成します。以下の手続は 2 つの要素のペアとそれらの和を用いて 3 つ組を構築します。

```
(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))
```

Combining all these steps yields the complete procedure:

これらのステップ全てを接続すれば手続は完了です。

```
(define (prime-sum-pairs n)
  (map make-pair-sum
    (filter prime-sum?
      (flatmap
        (lambda (i)
          (map (lambda (j) (list i j))
            (enumerate-interval 1 (- i 1))))
        (enumerate-interval 1 n)))))
```

Nested mappings are also useful for sequences other than those that enumerate intervals. Suppose we wish to generate all the permutations of a set  $S$ ; that is, all the ways of ordering the items in the set. For instance, the permutations of  $\{1, 2, 3\}$  are  $\{1, 2, 3\}$ ,  $\{1, 3, 2\}$ ,  $\{2, 1, 3\}$ ,  $\{2, 3, 1\}$ ,  $\{3, 1, 2\}$ , and  $\{3, 2, 1\}$ . Here is

a plan for generating the permutations of  $S$ : For each item  $x$  in  $S$ , recursively generate the sequence of permutations of  $S - x$ ,<sup>38</sup> and adjoin  $x$  to the front of each one. This yields, for each  $x$  in  $S$ , the sequence of permutations of  $S$  that begin with  $x$ . Combining these sequences for all  $x$  gives all the permutations of  $S$ :<sup>39</sup>

入れ子の map は区間を列举するもの以外の列に対しても便利です。ある集合  $S$  の全ての順列を生成したいとします。つまり集合内の項目の全ての並べ方です。例えば  $\{1, 2, 3\}$  の順列は  $\{1, 2, 3\}$ ,  $\{1, 3, 2\}$ ,  $\{2, 1, 3\}$ ,  $\{2, 3, 1\}$ ,  $\{3, 1, 2\}$ , and  $\{3, 2, 1\}$  です。ここに集合  $S$  の順列を生成するための計画を上げます:  $S$  中の全て項目  $x$  に対し再帰的に  $S - x$  の順列の列を生成し、<sup>40</sup>次に  $x$  をそれぞれの先頭に置く。これは  $S$  の全ての  $x$  に対し  $S$  の  $x$  で始まる順列の列を生成する。これらの全ての  $x$  に対する列を接続すると  $S$  の全ての順列が与えられる。<sup>41</sup>

```
(define (permutations s)
  (if (null? s) ; 集合は空か?
      (list nil) ; 空集合を持つ列
      (flatmap (lambda (x)
                  (map (lambda (p) (cons x p))
                       (permutations (remove x s))))
               s)))
```

Notice how this strategy reduces the problem of generating permutations of  $S$  to the problem of generating the permutations of sets with fewer elements than  $S$ . In the terminal case, we work our way down to the empty list, which represents a set of no elements. For this, we generate (list nil), which is a sequence with one item, namely the set with no elements. The remove procedure used in permutations returns all the items in a given sequence except for a

<sup>38</sup>The set  $S - x$  is the set of all elements of  $S$ , excluding  $x$ .

<sup>39</sup>Semicolons in Scheme code are used to introduce *comments*. Everything from the semicolon to the end of the line is ignored by the interpreter. In this book we don't use many comments; we try to make our programs self-documenting by using descriptive names.

<sup>40</sup>集合  $S - x$  は  $S$  の全ての要素から  $x$  を除いた集合

<sup>41</sup>Scheme のコードではセミコロンは *comments*(コメント) を書く場合に利用されます。セミコロンから始まり行末までの全てはインタプリタに無視されます。この本ではあまり多くのコメントを使用していません。私達はプログラムに対し説明的な名前を付けることでそれ自身がドキュメントであるかのように作るよう努力しています。

given item. This can be expressed as a simple filter:

この戦略がどのように  $S$  の順列を生成する問題から  $S$  よりもより少ない要素の集合の順列生成の問題へと縮小しているかに注意して下さい。境界条件に関しては要素無しの集合を表す空リストまで順に処理を繰り返します。空リストに対して (`list nil`) を生成しました。これは 1 要素の列であり、要素無しの集合を表します。`permutations` 内で利用される `remove` 手続は与えられた式から与えられた項目以外の全ての要素を返します。これは簡単なフィルタにて表すことができます。

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item)))
          sequence))
```

**Exercise 2.40:** Define a procedure `unique-pairs` that, given an integer  $n$ , generates the sequence of pairs  $(i, j)$  with  $1 \leq j < i \leq n$ . Use `unique-pairs` to simplify the definition of `prime-sum-pairs` given above.

整数  $n$  を与えられ、ペア  $(i, j)$  を  $1 \leq j < i \leq n$  の条件で生成する手続 `unique-pairs` を定義せよ。`unique-pairs` を用いて上で与えられた `prime-sum-pairs` の定義をより簡単にせよ。

**Exercise 2.41:** Write a procedure to find all ordered triples of distinct positive integers  $i$ ,  $j$ , and  $k$  less than or equal to a given integer  $n$  that sum to a given integer  $s$ .

与えられた整数  $n$  以下でかつ合計が与えられた整数  $s$  である、全ての異なる正の整数  $i$ ,  $j$ ,  $k$  の順序有りの 3 つ組を求める手続を書け。

**Exercise 2.42:** The “eight-queens puzzle” asks how to place eight queens on a chessboard so that no queen is in check from any other (i.e., no two queens are in the same row, column, or diagonal). One possible solution is shown in Figure 2.8. One way to solve the puzzle is to work across the board, placing a queen in each column. Once we have placed  $k - 1$  queens, we must place the  $k^{\text{th}}$  queen in a position where it does not check any of the queens already on the board. We can formulate this approach recursively: Assume that we have already generated the sequence of all possible ways to

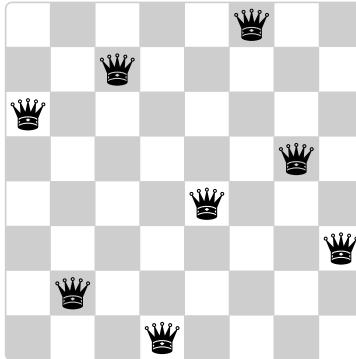
place  $k - 1$  queens in the first  $k - 1$  columns of the board. For each of these ways, generate an extended set of positions by placing a queen in each row of the  $k^{\text{th}}$  column. Now filter these, keeping only the positions for which the queen in the  $k^{\text{th}}$  column is safe with respect to the other queens. This produces the sequence of all ways to place  $k$  queens in the first  $k$  columns. By continuing this process, we will produce not only one solution, but all solutions to the puzzle.

“8 クイーンパズル”は8つのクイーンをチェス盤の上に、どのクイーンも他のクイーンを取ることができないようにするにはどのように置くかを尋ねる。(これはつまりどの2つのクイーンも同じ列、行、または斜めの線上に有ってはならないということである)。考えられる解の1つをFigure 2.8に示す。このパズルを解く1つの方法は盤上に渡って各列にクイーンを置く。 $k - 1$  個のクイーンを置いたら  $k$  番目のクイーンは既に盤上に置いてあるどのクイーンも取れない場所に置かなければならぬ。この取り組み方を再帰的に形式化できる：盤上の最初の  $k - 1$  列内の  $k - 1$  個のクイーンの可能な置き方全ての列を既に生成したと想定する。これら全ての方法に対し拡張した位置の拡張集合を  $k$  番目の列の各行にクイーンを置くことで生成する。次にこれらをフィルタにかけて  $k$  番目の列のクイーンが他のクイーンを考慮しても安全な位置のみを保持する。これは  $k$  個のクイーンを最初の  $k$  列内に置く全ての方法を生成する。この過程を繰り返すことで1つの解答のみでなく、このパズルの全ての解答を生成できる。

We implement this solution as a procedure `queens`, which returns a sequence of all solutions to the problem of placing  $n$  queens on an  $n \times n$  chessboard. `Queens` has an internal procedure `queen-cols` that returns the sequence of all ways to place queens in the first  $k$  columns of the board.

この解法を手続 `queens` として実装した。 $n$  個のクイーンを  $n \times n$  のチェス盤上に置く問題に対する全ての解の列を返す。`queens` は内部手続 `queen-cols` を持ち、それは盤の最初の  $k$  列中にクイーンを置く全ての方法の列を返す。

```
(define (queens board-size)
  (define (queen-cols k)
```



**Figure 2.8:** A solution to the eight-queens puzzle.

```
(if (= k 0)
  (list empty-board)
  (filter
    (lambda (positions) (safe? k positions))
    (flatmap
      (lambda (rest-of-queens)
        (map (lambda (new-row)
          (adjoin-position new-row
            k
            rest-of-queens))
          (enumerate-interval 1 board-size)))
      (queen-cols (- k 1))))))
  (queen-cols board-size))
```

In this procedure `rest-of-queens` is a way to place  $k - 1$  queens in the first  $k - 1$  columns, and `new-row` is a proposed row in which to place the queen for the  $k^{\text{th}}$  column. Complete the program by implementing the representation for sets of board positions, including the procedure `adjoin-position`, which adjoins a new row-column position to a set of positions, and `empty-board`, which represents an empty set of positions. You must also write the procedure `safe?`,

which determines for a set of positions, whether the queen in the  $k^{\text{th}}$  column is safe with respect to the others. (Note that we need only check whether the new queen is safe—the other queens are already guaranteed safe with respect to each other.)

この手続の中で、`rest-of-queens` は最初の  $k - 1$  列内に  $k - 1$  個のクイーンを置く方法であり、`new-row` は  $k$  番目の列に対してクイーンを置くように提案された行である。盤上の位置の集合に対する表現を、新しい列の位置を位置の集合に付け足す手続 `adjoin-position` と位置の空集合を表す `empty-board` を含めて実装することでプログラムを完成させよ。あなたは  $k$  番目の列にあるクイーンが他に対して安全であるかどうかを位置の集合に対して決定する `safe?` もまた書かなければならない。(新しいクイーンが安全であるかどうかのみをチェックする必要であることに注意すること—他のクイーンは既にお互いに安全であることが保証されている)。

**Exercise 2.43:** Louis Reasoner is having a terrible time doing [Exercise 2.42](#). His `queens` procedure seems to work, but it runs extremely slowly. (Louis never does manage to wait long enough for it to solve even the  $6 \times 6$  case.) When Louis asks Eva Lu Ator for help, she points out that he has interchanged the order of the nested mappings in the `flatmap`, writing it as

Louis Reasoner は [Exercise 2.42](#)を行なうことで酷い時間を過している。彼の `queens` 手続は動いているように見える。しかし実行がとても遅い。(Louis は  $6 \times 6$  の場合でさえそれを解くのにかかる長い時間を待つことができなかった)。Louis が Eva Lu Ator に助けを求めた時、彼女は Louis が `flatmap` 内の入れ子のマッピングの順を交換してしまったことを指摘した。以下のように書いていた。

```
(flatmap
  (lambda (new-row)
    (map (lambda (rest-of-queens)
            (adjoin-position new-row k rest-of-queens))
          (queen-cols (- k 1))))
  (enumerate-interval 1 board-size))
```

Explain why this interchange makes the program run slowly. Estimate how long it will take Louis's program to solve the eight-

queens puzzle, assuming that the program in Exercise 2.42 solves the puzzle in time  $T$ .

この交換がなぜプログラムの実行を遅くするのか説明せよ。Louis のプログラムが 8 クイーンパズルを解くのにどれだけの時間がかかるか推察せよ。Exercise 2.42 のプログラムが同じパズルを解くのに必要な時間が  $T$  であるとの前提で行え。

## 2.2.4 Example: A Picture Language

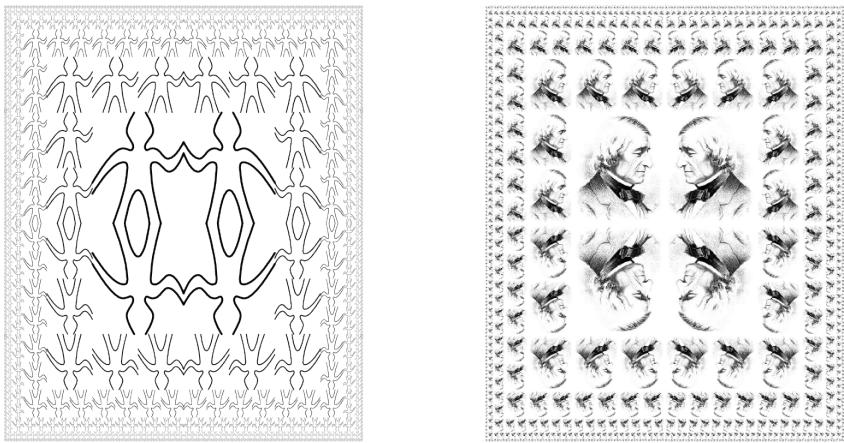
This section presents a simple language for drawing pictures that illustrates the power of data abstraction and closure, and also exploits higher-order procedures in an essential way. The language is designed to make it easy to experiment with patterns such as the ones in Figure 2.9, which are composed of repeated elements that are shifted and scaled.<sup>42</sup> In this language, the data objects being combined are represented as procedures rather than as list structure. Just as `cons`, which satisfies the closure property, allowed us to easily build arbitrarily complicated list structure, the operations in this language, which also satisfy the closure property, allow us to easily build arbitrarily complicated patterns.

この節では絵を描く簡単な言語をお見せします。これがデータ抽象と閉包の力を図示し、また高階手続を本質的な方法で利用します。この言語はFigure 2.9のようなパターンを試験することを簡単にするために設計されております。この図は要素が移動し、縮小しを繰り返しながら組み立てられています。<sup>43</sup> この言語内では、組み立てられるデータオブジェクトはリスト構造ではなく手続として表現されます。閉包の特性を見たす `cons` が簡単に自由に複雑なリスト構造を構築できるように、この言語内の命令もまた閉包の特性を満たし、簡単に自由に複雑なパターンを構築できます。

---

<sup>42</sup>The picture language is based on the language Peter Henderson created to construct images like M.C. Escher’s “Square Limit” woodcut (see Henderson 1982). The woodcut incorporates a repeated scaled pattern, similar to the arrangements drawn using the `square-limit` procedure in this section.

<sup>43</sup>ピクチャー言語は Peter Henderson が作成した言語を基にしており、この言語は M.C. Escher の木版画 “Square Limit”(Henderson 1982 参照) のようなイメージを構築するために作成されました。その木版画は繰り返しサイズが変更されたパターンが組込まれており、この節の `square-limit` 手續を用いて描かれた配置と似ております。



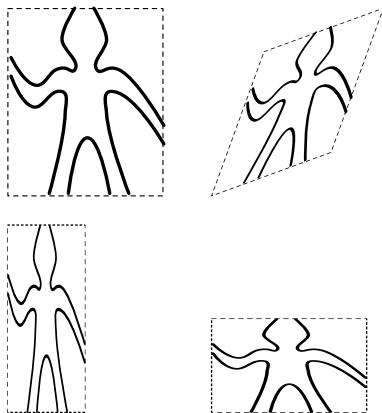
**Figure 2.9:** Designs generated with the picture language.

## The picture language

When we began our study of programming in [Section 1.1](#), we emphasized the importance of describing a language by focusing on the language's primitives, its means of combination, and its means of abstraction. We'll follow that framework here.

[Section 1.1](#)でプログラミングの学習を始めたとき、言語のプリミティブ、その組み合わせの手段、抽象化の手段に集中することが言語の説明の重要性だと強調しました。ここではその枠組みに従います。

Part of the elegance of this picture language is that there is only one kind of element, called a *painter*. A painter draws an image that is shifted and scaled to fit within a designated parallelogram-shaped frame. For example, there's a primitive painter we'll call `wave` that makes a crude line drawing, as shown in [Figure 2.10](#). The actual shape of the drawing depends on the frame—all four images in figure 2.10 are produced by the same `wave` painter, but with respect to four different frames. Painters can be more elaborate than this: The primitive painter called `rogers` paints a picture of MIT's founder, William Barton Rogers, as shown in [Figure 2.11](#).



**Figure 2.10:** Images produced by the `wave`

このピクチャー言語の優雅さの部分は要素の種類が `painter`(ペインタ) と呼ばれるものたった 1 つしかないことです。ペインタは指定された平行四辺形の枠の中にイメージを移動し、拡大縮小して描きます。例えば `wave` と呼ぶペインタがありそれは Figure 2.10 に見られるような粗野な線の絵を描きます。実際の絵の下腿はフレームに依存します—Figure 2.10 の 4 つの絵全ては同じ `wave` ペインタにより生成されていますが、4 つの異なるフレームを考慮しています。ペインタはこれよりもより複雑にすることが可能です。`rogers` と呼ばれるプリミティブなペインタは MIT の創始者である William Barton Rogers の絵を Figure 2.11 に示されるように描きます。<sup>44</sup> The four images in figure 2.11 are

---

<sup>44</sup>

William Barton Rogers (1804-1882) was the founder and first president of MIT. A geologist and talented teacher, he taught at William and Mary College and at the University of Virginia. In 1859 he moved to Boston, where he had more time for research, worked on a plan for establishing a “polytechnic institute,” and served as Massachusetts’s first State Inspector of Gas Meters.

William Barton Rogers (1804-1882) は MIT の創始者であり、かつ初代学長です。地質学者であり、才能溢れる教師である彼は William and Mary College と University of Virginia にて教鞭を取りました。1859 年に彼はボストンへ移りそこでより研究に打ち込み、“技術専門の研究所”を設立する計画を進めました。またマサチューセッツ州の最初のガスマータの州検査官も務めました。

---

When MIT was established in 1861, Rogers was elected its first president. Rogers espoused an ideal of “useful learning” that was different from the university education of the time, with its overemphasis on the classics, which, as he wrote, “stand in the way of the broader, higher and more practical instruction and discipline of the natural and social sciences.” This education was likewise to be different from narrow trade-school education. In Rogers’s words:

MIT が 1861 年に創設された時、Rogers は最初の学長に選ばれました。Rogers は “useful learning”(実用的学習) の活用を信奉しました。これは当時の大学教育からは異なるものでした。彼は依れば古典の過度の強調が“より幅広く、高く、より現実的な教育と自然科学、及び社会科学の前に立ち塞がっている”と書いています。同様に彼の教育は職業専門学校の狭い教育からも異なるものになろうとしていました。

The world-enforced distinction between the practical and the scientific worker is utterly futile, and the whole experience of modern times has demonstrated its utter worthlessness.

実利的であることと科学実務者の間の区別を強制する世界は全く無益だ。  
現代の経験全てがその完全な無益さを示している。

Rogers served as president of MIT until 1870, when he resigned due to ill health. In 1878 the second president of MIT, John Runkle, resigned under the pressure of a financial crisis brought on by the Panic of 1873 and strain of fighting off attempts by Harvard to take over MIT. Rogers returned to hold the office of president until 1881.

Rogers は MIT の学長を健康上の理由で辞任する 1870 年まで務めました。1878 年に二代目の MIT 学長 John Runkle は 1873 年からの大不況によりもたらされた財政危機のプレッシャーと Harvard による MIT の買収の試みに対する抵抗の緊張により辞任しました。Roger は学長のオフィスを支えるため 1881 年まで戻りました。

Rogers collapsed and died while addressing MIT’s graduating class at the commencement exercises of 1882. Runkle quoted Rogers’s last words in a memorial address delivered that same year:

Rogers は 1882 年、MIT の大学院の卒業試験に取り組む最中に倒れ、亡くなられました。Runkle が Roger の最後の言葉を同年に送られた弔事から引用しています。

“As I stand here today and see what the Institute is, . . . I call to mind the beginnings of science. I remember one hundred and fifty years ago Stephen Hales published a pamphlet on the subject of illuminating gas, in which he stated that his researches had demonstrated that 128 grains of bituminous coal – ” “Bituminous coal,” these were his last words on earth. Here he bent forward, as if consulting some notes on the table before him, then slowly regaining an erect position, threw up his hands, and was translated from the scene of his earthly labors and triumphs to “the tomorrow of death,” where the mysteries of life are solved, and the disembodied spirit finds unending satisfaction in contemplating the new and still unfathomable mysteries of the infinite future.

drawn with respect to the same four frames as the `wave` images in figure 2.10.

Figure 2.11 の 4 つのイメージは Figure 2.10 の `wave` のイメージと同じ 4 つのフレームを考慮して描かれています。

To combine images, we use various operations that construct new painters from given painters. For example, the `beside` operation takes two painters and produces a new, compound painter that draws the first painter's image in the left half of the frame and the second painter's image in the right half of the frame. Similarly, `below` takes two painters and produces a compound painter that draws the first painter's image below the second painter's image. Some operations transform a single painter to produce a new painter. For example, `flip-vert` takes a painter and produces a painter that draws its image upside-down, and `flip-horiz` produces a painter that draws the original painter's image left-to-right reversed.

イメージを結合するには与えられたペインタから新しいペインタを構築する種々の命令を使用します。例えば `beside` 命令は 2 つのペインタを取り新しい最初のペインタのイメージをフレームの左半分に、2 つ目のペインタのイメージをフレームの右半分に描く複合ペインタを生成します。同様に `below` は 2 つのペインタを取り、1 つ目のペインタのイメージを 2 つ目のペインタのイメ

---

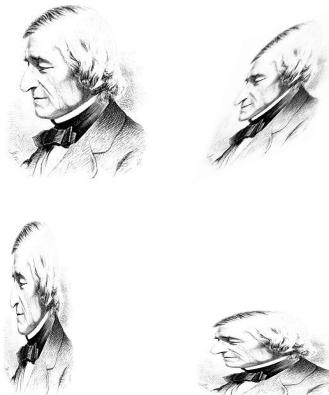
“本日ここに立ち本校とは何であるかを考えると … 科学の始まりを思い受かべます。150 年前に Stephen Hales は灯用のガスを主題にした小論文を発表しました。その中で彼は彼の研究が 128 グレインの瀝青炭 – ” “瀝青炭” これが地上での彼の最後の言葉でした。ここで彼は体を前に曲げ、彼の前のテーブルの上にある端書を確認するよういて、そしてゆっくりと直立した体制を取り戻し、両腕を上げ、そして彼の地上の労働と業績の場面から “死の明日” へと形を変えたのです。そこでは人生の謎は解決され、肉体から解放された魂は新しく、未だ測りしれない無限の未来の謎を熟考することに終りの無い充足を見つけるのです。

In the words of Francis A. Walker (MIT's third president):

Francis A. Walker(MIT の三代目の学長)の言葉では

All his life he had borne himself most faithfully and heroically, and he died as so good a knight would surely have wished, in harness, at his post, and in the very part and act of public duty.

彼自身が負うた彼の人生全ては最も誠実で雄々しく、そして彼は騎士が心から望んだかのごとく、仕事中に、その役職のまま、公務の行いの最中に亡くなった。



**Figure 2.11:** Images of William Barton Rogers,

ージの下に描きます。いくつかの命令は単一のペインタを変換し新しいペインタを生成します。例えば `flip-vert` はペインタを取りそのイメージを上下逆さに描くペインタを生成し、`flip-horiz` は元のペインタのイメージを左右逆に描くペインタを生成します。

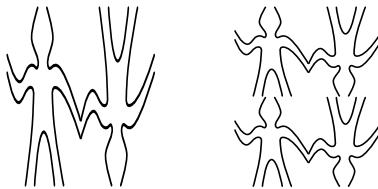
Figure 2.12 shows the drawing of a painter called `wave4` that is built up in two stages starting from `wave`:

Figure 2.12は `wave4` を呼んだペインタの描画を見せており、これは `wave` で始め 2段階を経て構築されています。

```
(define wave2 (beside wave (flip-vert wave)))
(define wave4 (below wave2 wave2))
```

In building up a complex image in this manner we are exploiting the fact that painters are closed under the language's means of combination. The `beside` or `below` of two painters is itself a painter; therefore, we can use it as an element in making more complex painters. As with building up list structure using `cons`, the closure of our data under the means of combination is crucial to the ability to create complex structures while using only a few operations.

複雑なイメージをこの様式で構築する場合はペインタが言語の接続手段の下で閉じているという事実を利用しています。2つのペインタの `beside` や



**Figure 2.12:** Creating a complex figure, starting

`below` はそれ自身がペインタです。従ってそれをより複雑なペインタを作るための要素として使用できます。`cons` を用いてリスト構造を構築するのと同様に、結合手段の下のデータの閉包はほんのわずかな命令を用いて複雑な構造を作成する能力にとって重大です。

Once we can combine painters, we would like to be able to abstract typical patterns of combining painters. We will implement the painter operations as Scheme procedures. This means that we don't need a special abstraction mechanism in the picture language: Since the means of combination are ordinary Scheme procedures, we automatically have the capability to do anything with painter operations that we can do with procedures. For example, we can abstract the pattern in `wave4` as

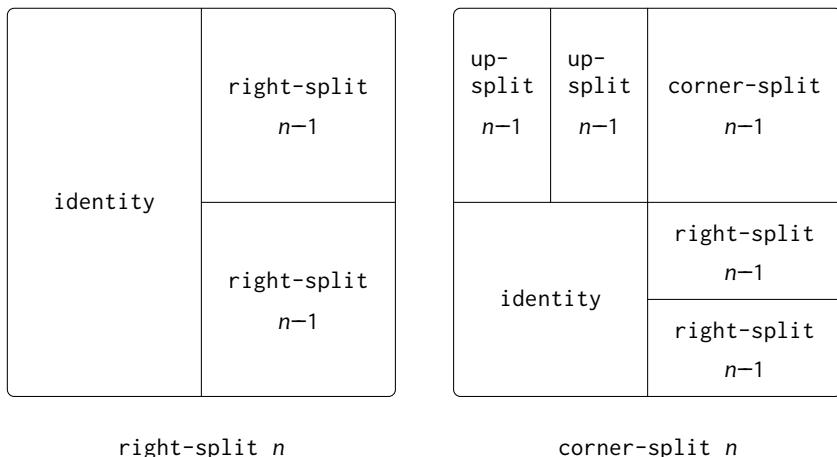
ペインタを結合できれば直ぐに、ペインタを接続する典型的なパターンを抽象化できるようになりたいと願うでしょう。ペインタ操作を Scheme の手続として実装することにします。それは私達がピクチャ言語内のメカニズムとして専用の抽象化を必要としないことを意味します。接続の手段が普通の Scheme の手続ですから、手続の範囲で行えるペインタの操作を用いて、何でもできる能力が自動的に得られます。例えば `wave4` 内のパターンを抽象化できます。

```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))
```

and define `wave4` as an instance of this pattern:

そして `wave4` をこのパターンのインスタンスとして定義します。

```
(define wave4 (flipped-pairs wave))
```



**Figure 2.13:** Recursive plans for `right-split` and `corner-split`.

We can also define recursive operations. Here's one that makes painters split and branch towards the right as shown in Figure 2.13 and Figure 2.14:

また再帰命令を定義することも可能です。以下はペインタを分割し、Figure 2.13に示すように右へ向けて枝分かれします。

```
(define (right-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (right-split painter (- n 1))))
        (beside painter (below smaller smaller)))))
```

We can produce balanced patterns by branching upwards as well as towards the right (see exercise Exercise 2.44 and figures Figure 2.13 and Figure 2.14):

右に向けてと同じように上方向にも枝分かれすることでバランスの取れたパターンを生成することも可能です。(課題Exercise 2.44と図Figure 2.13とFigure 2.14を参照して下さい)。

```
(define (corner-split painter n)
```

```
(if (= n 0)
    painter
    (let ((up (up-split painter (- n 1)))
          (right (right-split painter (- n 1))))
      (let ((top-left (beside up up))
            (bottom-right (below right right))
            (corner (corner-split painter (- n 1))))
        (beside (below painter top-left)
                (below bottom-right corner)))))))
```

By placing four copies of a `corner-split` appropriately, we obtain a pattern called `square-limit`, whose application to `wave` and `rogers` is shown in Figure 2.9:

`corner-split` の 4 つのコピーを置くことで `square-limit` と呼ばれるパターンを獲得することができ、`wave` と `rogers` に対する適用が Figure 2.9 に示されます。

```
(define (square-limit painter n)
  (let ((quarter (corner-split painter n)))
    (let ((half (beside (flip-horiz quarter) quarter)))
      (below (flip-vert half) half))))
```

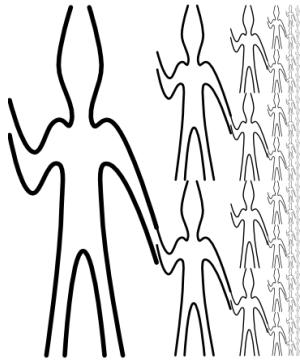
**Exercise 2.44:** Define the procedure `up-split` used by `corner-split`. It is similar to `right-split`, except that it switches the roles of `below` and `beside`.

`corner-split` にて使用された手続 `up-split` を定義せよ。`right-split` に似ているが、`below` と `beside` の役割を入れ替える。

## Higher-order operations

In addition to abstracting patterns of combining painters, we can work at a higher level, abstracting patterns of combining painter operations. That is, we can view the painter operations as elements to manipulate and can write means of combination for these elements—procedures that take painter operations as arguments and create new painter operations.

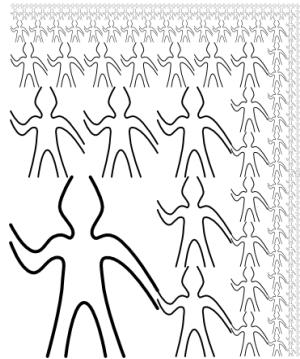
ペイント命令のパターンを抽象化するのに加えて、より高いレベルのペイント命令接続の抽象化パターンについて取り組むことができます。それはペイ



(right-split wave 4)



(right-split rogers 4)



(corner-split wave 4)



(corner-split rogers 4)

**Figure 2.14:** The recursive operations `right-split` and `corner-split` applied to the painters `wave` and `rogers`. Combining four `corner-split` figures produces symmetric square-limit designs as shown in [Figure 2.9](#).

ンタ命令を操作を行うための要素 —ペインタ命令を引数として取り新しいペインタ命令を作成する手続と見做し、そしてこれらの要素のための組み合わせの手段の記述が可能だということです。

For example, `flipped-pairs` and `square-limit` each arrange four copies of a painter's image in a square pattern; they differ only in how they orient the copies. One way to abstract this pattern of painter combination is with the following procedure, which takes four one-argument painter operations and produces a painter operation that transforms a given painter with those four operations and arranges the results in a square. `Tl`, `tr`, `bl`, and `br` are the transformations to apply to the top left copy, the top right copy, the bottom left copy, and the bottom right copy, respectively.

例として、`flipped-pairs` と `square-limit` はペインタのイメージを四角のパターン内にて 4 つのコピーをそれぞれが準備します。それらはどのような位置と向きに置くかということのみにおいて異なります。このペインタ接続のパターンを抽象化する 1 つの方法は以下のプロジェクトを用いて、4 つの 1 引数ペインタ命令を取り与えられたペインタをそれら 4 つの命令で変換する命令を生成し、結果を四角の中に配置します。`tl`, `tr`, `bl`, and `br` は左上、右上、左下、右下のコピーに対応する変換です。

```
(define (square-of-four tl tr bl br)
  (lambda (painter)
    (let ((top (beside (tl painter) (tr painter)))
          (bottom (beside (bl painter) (br painter))))
      (below bottom top))))
```

Then `flipped-pairs` can be defined in terms of `square-of-four` as follows:<sup>45</sup>  
すると `flipped-pairs` は `square-of-four` を以下のように用いて定義可能です。<sup>46</sup>

---

<sup>45</sup>Equivalently, we could write

```
(define flipped-pairs
  (square-of-four identity flip-vert identity flip-vert))
```

<sup>46</sup> 同等地に、こうも書けます。

```
(define flipped-pairs
  (square-of-four identity flip-vert identity flip-vert))
```

```
(define (flipped-pairs painter)
  (let ((combine4 (square-of-four identity flip-vert
                                    identity flip-vert)))
    (combine4 painter)))
```

and `square-limit` can be expressed as<sup>47</sup>

そして `square-limit` は以下の様に表現可能です。<sup>48</sup>

```
(define (square-limit painter n)
  (let ((combine4 (square-of-four flip-horiz identity
                                    rotate180 flip-vert)))
    (combine4 (corner-split painter n))))
```

**Exercise 2.45:** `Right-split` and `up-split` can be expressed as instances of a general splitting operation. Define a procedure `split` with the property that evaluating

`right-split` と `up-split` は一般的な分割命令のインスタンスだと言うことができる。手続 `split` を以下の式を評価する場合に、

```
(define right-split (split beside below))
(define up-split (split below beside))
```

produces procedures `right-split` and `up-split` with the same behaviors as the ones already defined.

既に定義住みのものと全く同じ振舞を行う手続 `right-split` と `up-split` を生成するよう定義せよ。

## Frames

Before we can show how to implement painters and their means of combination, we must first consider frames. A frame can be described by three vectors—an origin vector and two edge vectors. The origin vector specifies the

---

<sup>47</sup> `Rotate180` rotates a painter by 180 degrees (see [Exercise 2.50](#)). Instead of `rotate180` we could say `(compose flip-vert flip-horiz)`, using the `compose` procedure from [Exercise 1.42](#).

<sup>48</sup> `Rotate180` はペインタを 180 度回転します ([Exercise 2.50 参照](#))。`rotate180` の代わりに `(compose flip-vert flip-horiz)` と言うこともできます。`compose` 手続は [Exercise 1.42](#) から使用しました。

offset of the frame's origin from some absolute origin in the plane, and the edge vectors specify the offsets of the frame's corners from its origin. If the edges are perpendicular, the frame will be rectangular. Otherwise the frame will be a more general parallelogram.

ペインタとどのように実装しその接続手段を示す前に、始めにフレームについて考えなければなりません。フレームは3つのベクトル—始点ベクトルと2つの辺ベクトルで説明できます。始点ベクトルは平面上においてある絶対的な始点からフレームの始点までのオフセットを指定します。そして辺ベクトルは始点から角までのオフセットを指定します。もし2つの辺が垂直であればフレームは長方形になります。それ以外ではフレームはより一般的な平行四辺形になります。

[Figure 2.15](#) shows a frame and its associated vectors. In accordance with data abstraction, we need not be specific yet about how frames are represented, other than to say that there is a constructor `make-frame`, which takes three vectors and produces a frame, and three corresponding selectors `origin-frame`, `edge1-frame`, and `edge2-frame` (see [Exercise 2.47](#)).

[Figure 2.15](#)はフレームとその対応するベクトルを示します。データ抽象化に従い、まだフレームがどのように表現されるかについては、3つのベクトルを取りフレームを生成するコンストラクタ `make-frame` と関連する3つのセレクタ `origin-frame`, `edge1-frame`, `edge2-frame` が存在すること以外を特定する必要がありません。[\(Exercise 2.47\) を参照して下さい。](#)

We will use coordinates in the unit square ( $0 \leq x, y \leq 1$ ) to specify images. With each frame, we associate a *frame coordinate map*, which will be used to shift and scale images to fit the frame. The map transforms the unit square into the frame by mapping the vector  $\mathbf{v} = (x, y)$  to the vector sum

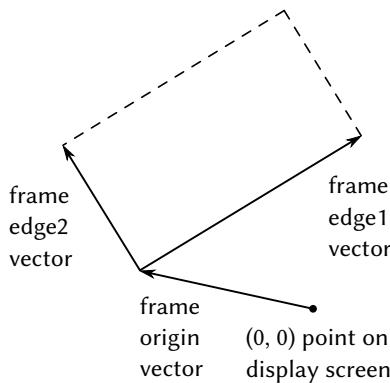
私達は単位正方形 ( $0 \leq x, y \leq 1$ ) 内の座標をイメージを指定するのに用いることにします。各フレームは、フレームに適合するようにイメージの移動と拡大縮小をするのに使われる *frame coordinate map*(フレーム座標マップ) に関連付けられます。マップは単位正方形をベクトル  $\mathbf{v} = (x, y)$  を次のベクトルの和にマッピングすることで変換します。

$$\text{Origin(Frame)} + x \cdot \text{Edge}_1(\text{Frame}) + y \cdot \text{Edge}_2(\text{Frame}).$$

For example,  $(0, 0)$  is mapped to the origin of the frame,  $(1, 1)$  to the vertex diagonally opposite the origin, and  $(0.5, 0.5)$  to the center of the frame. We can create a frame's coordinate map with the following procedure:<sup>49</sup>

---

<sup>49</sup>[Frame-coord-map](#) uses the vector operations described in [Exercise 2.46](#) below,



**Figure 2.15:** A frame is described by three vectors — an origin and two edges.

例えば、 $(0, 0)$  はフレームの始点に、 $(1, 1)$  は対角線上に始点の反対の頂点へ、そして  $(0.5, 0.5)$  はフレームの中心点にマッピングされます。フレーム座標マップは以下の手続により作成できます。<sup>50</sup>

```
(define (frame-coord-map frame)
  (lambda (v)
    (add-vect
      (origin-frame frame)
      (add-vect (scale-vect (xcor-vect v)
                            (edge1-frame frame))
                (scale-vect (ycor-vect v)
                            (edge2-frame frame))))))
```

---

which we assume have been implemented using some representation for vectors. Because of data abstraction, it doesn't matter what this vector representation is, so long as the vector operations behave correctly.

<sup>50</sup> `frame-coord-map` はこの先の [Exercise 2.46](#) にて説明されるベクトル操作を用います。ここでは何らかのベクトルの表現を用いて実装済みと仮定します。データ抽象化のおかげでこのベクトルの表現がどんなものかは、ベクトル操作が正しく振る舞われる限りにおいて問題にはなりません。

Observe that applying `frame-coord-map` to a frame returns a procedure that, given a vector, returns a vector. If the argument vector is in the unit square, the result vector will be in the frame. For example,

`frame-coord-map` をフレームに適用すると、ベクトルを取りベクトルを返す手続を返すことに注意して下さい。もし引数ベクトルが単位正方形の中なら、結果のベクトルはフレームの範囲内になります。例として、

```
((frame-coord-map a-frame) (make-vect 0 0))
```

returns the same vector as

は以下のベクトルと同じものを返します。

```
(origin-frame a-frame)
```

**Exercise 2.46:** A two-dimensional vector  $\mathbf{v}$  running from the origin to a point can be represented as a pair consisting of an  $x$ -coordinate and a  $y$ -coordinate. Implement a data abstraction for vectors by giving a constructor `make-vect` and corresponding selectors `xcor-vec` and `ycor-vec`. In terms of your selectors and constructor, implement procedures `add-vect`, `sub-vec`, and `scale-vec` that perform the operations vector addition, vector subtraction, and multiplying a vector by a scalar:

始点からある点へと走る 2 次元ベクトル  $\mathbf{v}$  は  $x$ -座標と  $y$ -座標から成るペアにより表現できる。ベクトルに対するデータ抽象をコンストラクタ `make-vec` と関連するセレクタ `xcor-vec` と `ycor-vec` を与えることにより実装せよ。セレクタとコンストラクタを用いてベクトルの足し算、引き算、スカラによる乗算を求める操作を実行する手続 `add-vec`, `sub-vec`, `scale-vec` を実装せよ。

$$\begin{aligned}(x_1, y_1) + (x_2, y_2) &= (x_1 + x_2, y_1 + y_2), \\(x_1, y_1) - (x_2, y_2) &= (x_1 - x_2, y_1 - y_2), \\s \cdot (x, y) &= (sx, sy).\end{aligned}$$

**Exercise 2.47:** Here are two possible constructors for frames:

ここに 2 つの有り得そうなフレームのコンストラクタがある

```
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))
(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
```

For each constructor supply the appropriate selectors to produce an implementation for frames.

各コンストラクタに適切な、フレームに対応する実装を生成するセレクタを提供せよ。

## Painters

A painter is represented as a procedure that, given a frame as argument, draws a particular image shifted and scaled to fit the frame. That is to say, if  $p$  is a painter and  $f$  is a frame, then we produce  $p$ 's image in  $f$  by calling  $p$  with  $f$  as argument.

ペインタはフレームを引数として与えられ特定のイメージをフレームにはまるように移動、拡大縮小して描く手続として表現される。すなわちもし  $p$  がペインタで  $f$  がフレームである場合、 $f$  を引数として  $p$  を呼び出すことで  $f$  の中に  $p$  のイメージを生成する。

The details of how primitive painters are implemented depend on the particular characteristics of the graphics system and the type of image to be drawn. For instance, suppose we have a procedure `draw-line` that draws a line on the screen between two specified points. Then we can create painters for line drawings, such as the `wave` painter in Figure 2.10, from lists of line segments as follows:<sup>51</sup>

プリミティブなペインタがどのように実装されているかの詳細は特定のグラフィックシステムの特質と描画されるイメージのタイプに依存します。例えばスクリーン上の2つの指定された点の間に線を引く手続 `draw-line` があると想定します。すると線分のリストから線を引くためのペインタ、例えばFigure 2.10の `wave` ペインタのようなものを以下のように作ることができます。<sup>52</sup>

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
      (lambda (segment)
        (draw-line
```

---

<sup>51</sup> `Segments->painter` uses the representation for line segments described in Exercise 2.48 below. It also uses the `for-each` procedure described in Exercise 2.23.

<sup>52</sup> `Segments->painter` は線分の表現に下記のExercise 2.48で説明されたものを使っています。またExercise 2.23で説明された `for-each` を使っています。

```
((frame-coord-map frame)
 (start-segment segment))
((frame-coord-map frame)
 (end-segment segment))))
segment-list)))
```

The segments are given using coordinates with respect to the unit square. For each segment in the list, the painter transforms the segment endpoints with the frame coordinate map and draws a line between the transformed points.

線分は単位正方形に対しての座標を用いて与えられます。リスト中の各線分に対してペインタは線分の終端をフレーム座標マップを用いて変換し、変換後の点の間に線を引きます。

Representing painters as procedures erects a powerful abstraction barrier in the picture language. We can create and intermix all sorts of primitive painters, based on a variety of graphics capabilities. The details of their implementation do not matter. Any procedure can serve as a painter, provided that it takes a frame as argument and draws something scaled to fit the frame.<sup>53</sup>

手続としてペインタを表現することは強力な抽象化バリアをピクチャーランゲージの中に確立します。私達は全ての種類のプリミティブなペインタを種々のグラフィック機能の基盤の上に作り、混ぜることができます。それらの実装の詳細は問題ではありません。フレームを引数として取りフレームに適切なサイズにスケールして何かを描く任意の手続がペインタの役を演じることができます。<sup>54</sup>

---

<sup>53</sup>For example, the `rogers` painter of Figure 2.11 was constructed from a gray-level image. For each point in a given frame, the `rogers` painter determines the point in the image that is mapped to it under the frame coordinate map, and shades it accordingly. By allowing different types of painters, we are capitalizing on the abstract data idea discussed in Section 2.1.3, where we argued that a rational-number representation could be anything at all that satisfies an appropriate condition. Here we're using the fact that a painter can be implemented in any way at all, so long as it draws something in the designated frame. Section 2.1.3 also showed how pairs could be implemented as procedures. Painters are our second example of a procedural representation for data.

<sup>54</sup>例えばFigure 2.11の`rogers`ペインタはグレーレベルのイメージから構築されています。与えられたフレームの中の各点に対し`rogers`ペインタはイメージ中のマッピングされる位置をフレーム座標マップの下に決定し、適切に影を付けます。異なるタイプのペインタを許可することで、Section 2.1.3で議論された分数表現は適切な条件を満たせば全く任意でかまわないという抽象データの考え方からより大きな利点を得ています。ここではペインタは指定されたフレーム内に何かを描くのであれば全くどのように実装さ

**Exercise 2.48:** A directed line segment in the plane can be represented as a pair of vectors—the vector running from the origin to the start-point of the segment, and the vector running from the origin to the end-point of the segment. Use your vector representation from Exercise 2.46 to define a representation for segments with a constructor `make-segment` and selectors `start-segment` and `end-segment`.

平面上で方向を持つ線分はベクトルのペア—原点から線分の始点へと向かうベクトルと原点から線分の終点へと向かうベクトルとして表現可能だ。Exercise 2.46のベクトル表現を用いて、コンストラクタ `make-segment` とセレクタ `start-segment` と `end-segment` を持つ線分表現を定義せよ。

**Exercise 2.49:** Use `segments->painter` to define the following primitive painters:

`segments->painter` を用いて以下のプリミティブなペインタを定義せよ。

- a The painter that draws the outline of the designated frame.  
指定したフレームの外枠を描くペインタ
- b The painter that draws an “X” by connecting opposite corners of the frame.  
フレームの反対の角を繋いで“X”を描くペインタ
- c The painter that draws a diamond shape by connecting the midpoints of the sides of the frame.  
フレームの辺の中点を結んでダイヤモンドの形を描くペインタ
- d The `wave` painter.  
`wave` ペインタ

---

れても構わないという事実を用いています。Section 2.1.3はまたペアがどのように手続として実装され得るかということも示しました。ペインタはデータに対する手続表現の二つ目の例です。

## Transforming and combining painters

An operation on painters (such as `flip-vert` or `beside`) works by creating a painter that invokes the original painters with respect to frames derived from the argument frame. Thus, for example, `flip-vert` doesn't have to know how a painter works in order to flip it—it just has to know how to turn a frame upside down: The flipped painter just uses the original painter, but in the inverted frame.

ペインタに対する操作(例えば `flip-vert` や `beside`)はフレーム引数に由来するフレームに対して元のペインタを実行するペインタを作成することで働いています。従って例えば `flip-vert` は引っくり返す場合にもそれがどのように描かれるのかは知る必要がありません—ただフレームをどのように引っくり返すのか知る必要があります。逆転したペインタはただ元のペインタを用いますが、フレームは逆転されているのです。

Painter operations are based on the procedure `transform-painter`, which takes as arguments a painter and information on how to transform a frame and produces a new painter. The transformed painter, when called on a frame, transforms the frame and calls the original painter on the transformed frame. The arguments to `transform-painter` are points (represented as vectors) that specify the corners of the new frame: When mapped into the frame, the first point specifies the new frame's origin and the other two specify the ends of its edge vectors. Thus, arguments within the unit square specify a frame contained within the original frame.

ペインタ操作は `transform-painter` 手続を基にしており、それはペインタとどのようにフレームを変換するかの情報を引数に取り、新しいペインタを生成します。変換されたペインタはフレーム上にて呼ばれた時に、フレームを変換して基のペインタを変換済みのフレーム上で呼び出します。`transform-painter` に対する引数は新しいフレームの角を指定する(ベクトルとして表現される)複数の点です。フレームにマッピングされる時、最初の点は新しいフレームの始点を指定し、他の2つのは辺ベクトルの終点を指定します。従って、単位正方形内の引数は元のフレームの中に含まれるフレームを指定します。

```
(define (transform-painter painter origin corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (painter (make-frame
```

```

new-origin
  (sub-vect (m corner1) new-origin)
  (sub-vect (m corner2) new-origin)))))))

```

Here's how to flip painter images vertically:

次はどのようにペインタのイメージを縦方向に逆向きにするかです。

```

(define (flip-vert painter)
  (transform-painter painter
    (make-vect 0.0 1.0) ; new origin
    (make-vect 1.0 1.0) ; new end of edge1
    (make-vect 0.0 0.0))) ; new end of edge2

```

Using `transform-painter`, we can easily define new transformations. For example, we can define a painter that shrinks its image to the upper-right quarter of the frame it is given:

`transform-painter` を用いることで簡単に新しい変換を定義することができます。右上 4 分の 1 のフレームは次のようにして与えられます。

```

(define (shrink-to-upper-right painter)
  (transform-painter
    painter (make-vect 0.5 0.5)
    (make-vect 1.0 0.5) (make-vect 0.5 1.0)))

```

Other transformations rotate images counterclockwise by 90 degrees<sup>55</sup>

他の変換はイメージを時計回りの逆に 90 度回転したり、<sup>56</sup>

```

(define (rotate90 painter)
  (transform-painter painter
    (make-vect 1.0 0.0)
    (make-vect 1.0 1.0)
    (make-vect 0.0 0.0)))

```

or squash images towards the center of the frame:<sup>57</sup>

<sup>55</sup>Rotate90 is a pure rotation only for square frames, because it also stretches and shrinks the image to fit into the rotated frame.

<sup>56</sup>rotate90 は四角形のフレームに対してのみの純粋な回転です。イメージもまた拡大縮小して回転したフレームに合わせられるためです。

<sup>57</sup>The diamond-shaped images in Figure 2.10 and Figure 2.11 were created with `squash-inwards` applied to `wave` and `rogers`.

イメージをフレームの中心に向けて潰したりします。<sup>58</sup>

```
(define (squash-inwards painter)
  (transform-painter painter
    (make-vect 0.0 0.0)
    (make-vect 0.65 0.35)
    (make-vect 0.35 0.65)))
```

Frame transformation is also the key to defining means of combining two or more painters. The `beside` procedure, for example, takes two painters, transforms them to paint in the left and right halves of an argument frame respectively, and produces a new, compound painter. When the compound painter is given a frame, it calls the first transformed painter to paint in the left half of the frame and calls the second transformed painter to paint in the right half of the frame:

フレーム変換は2つ以上のペインタを接続する手段を定義するための鍵でもあります。例えば`beside`手続は2つのペインタを取りそれらを引数のフレームの左半分と右半分にそれぞれ描画するように変換する新しい複合ペインタを生成します。複合ペインタがフレームを与えられ時、1つ目の変換済みペインタを呼びフレームの左半分に描き、次に二つ目の変換済みペインタを呼びフレームの右半分を描きます。

```
(define (beside painter1 painter2)
  (let ((split-point (make-vect 0.5 0.0)))
    (let ((paint-left
           (transform-painter
             painter1
             (make-vect 0.0 0.0)
             split-point
             (make-vect 0.0 1.0)))
          (paint-right
            (transform-painter
              painter2
              split-point
              (make-vect 1.0 0.0)))
```

---

<sup>58</sup>Figure 2.10とFigure 2.11内のひし形のイメージは`squash-inwards`を`wave`と`rogers`に適用することで作成されました。

```

        (make-vect 0.5 1.0))))
(lambda (frame)
  (paint-left frame)
  (paint-right frame))))))

```

Observe how the painter data abstraction, and in particular the representation of painters as procedures, makes `beside` easy to implement. The `beside` procedure need not know anything about the details of the component painters other than that each painter will draw something in its designated frame.

ペインタのデータ抽象化と、特にペインタの手続としての表現がどのように `beside` の実装を簡単にしているのか注目して下さい。`beside` 手続はコンポーネントのペイントの詳細について各ペインタが指定されたフレームに何かを描くこと以外は一切知る必要がありません。

**Exercise 2.50:** Define the transformation `flip-horiz`, which flips painters horizontally, and transformations that rotate painters counterclockwise by 180 degrees and 270 degrees.

ペインタを水平方向に引っくり返す変換 `flip-horiz` を定義せよ。またペインタを時計と逆回りに 180 度と 270 度回す変換を定義せよ。

**Exercise 2.51:** Define the `below` operation for painters. `Below` takes two painters as arguments. The resulting painter, given a frame, draws with the first painter in the bottom of the frame and with the second painter in the top. Define `below` in two different ways—first by writing a procedure that is analogous to the `beside` procedure given above, and again in terms of `beside` and suitable rotation operations (from [Exercise 2.50](#)).

ペインタに対する `below` 命令を定義せよ。`below` は 2 つのペインタを引数に取る。結果のペインタはフレームを与えられ、1 つ目のペインタにてフレームの底部を描き、2 つ目のペインタにて上部を描く。`below` を 2 つの異なる方法で定義せよ。1 つは上で与えた `beside` と同様な方法で、2 つ目は `beside` と適切な ([Exercise 2.50](#) の) 回転命令を利用せよ。

## Levels of language for robust design

The picture language exercises some of the critical ideas we've introduced about abstraction with procedures and data. The fundamental data abstractions, painters, are implemented using procedural representations, which enables the language to handle different basic drawing capabilities in a uniform way. The means of combination satisfy the closure property, which permits us to easily build up complex designs. Finally, all the tools for abstracting procedures are available to us for abstracting means of combination for painters.

ピクチャー言語は私達が紹介した手続とデータによる抽象化についての重要なアイデアのいくつかを訓練しました。基本的なデータ抽象化であるペインタは手続表現を用いて実装され言語に異なる基礎的な描画能力を統一した方法で扱うことを可能にしました。接続手段は閉包の性質を満たし簡単に複雑な設計を組み上げることを可能にしました。最後に、抽象化手続に対する全てのツールはペインタに対する接続手段の抽象化にとって有効でした。

We have also obtained a glimpse of another crucial idea about languages and program design. This is the approach of *stratified design*, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail.

私達はまた言語とプログラム設計に関する素晴らしい考えを垣間見ることができました。これは *stratified design*(階層化設計) の方法で、複雑なシステムは一連の言語を用いて記述される一連のレベルとして構造化されるべきであるという概念です。各レベルはパーツをパーツを接続して構築され、それらは次のレベルではプリミティブとして参照されます。そして各レベルで構築されたパーツは次のレベルにてプリミティブとして使用されます。階層化された設計の各レベルで使用される言語はプリミティブ、接続手段、そしてそのレベルの詳細に適切な抽象化手段を持っています。

Stratified design pervades the engineering of complex systems. For example, in computer engineering, resistors and transistors are combined (and described using a language of analog circuits) to produce parts such as and-gates and or-gates, which form the primitives of a language for digital-circuit design.<sup>59</sup> These

---

<sup>59</sup>Section 3.3.4 describes one such language.

parts are combined to build processors, bus structures, and memory systems, which are in turn combined to form computers, using languages appropriate to computer architecture. Computers are combined to form distributed systems, using languages appropriate for describing network interconnections, and so on.

階層化された設計は複雑なシステムの設計において普及しています。例えば計算機設計では抵抗とトランジスタは接続され(そしてアナログ回路言語を用いて記述され)AND ゲートや OR ゲートのようなパーツを生じ、それらがデジタル回路の言語のプリミティブを形成します。<sup>60</sup> このようなパーツはプロセッサ、バス構造、メモリシステムを構築するために接続され、それらはコンピュータを形成するために接続され、コンピューターアーキテクチャに相応しい言語を用います。コンピュータは分散システムを形成するために接続され、ネットワーク相互接続その他を記述するに適切な言語を用います。

As a tiny example of stratification, our picture language uses primitive elements (primitive painters) that are created using a language that specifies points and lines to provide the lists of line segments for `segments->painter`, or the shading details for a painter like `rogers`. The bulk of our description of the picture language focused on combining these primitives, using geometric combiners such as `beside` and `below`. We also worked at a higher level, regarding `beside` and `below` as primitives to be manipulated in a language whose operations, such as `square-of-four`, capture common patterns of combining geometric combiners.

階層化の簡単な例として、ピクチャー言語はプリミティブな要素(プリミティブペインタ)を用い、それらは点と線を指定し `segments->painter` のための線分のリストを提供したり、`rogers` のようなペインタに対するシェーディングの詳細を提供したりする言語を用いて作成されました。私達のピクチャー言語の説明の大部分がこれらのプリミティブを接続し `beside` や `below` のような幾何学的なコンバイナ(結合器)に充てられました。私達はまたより高階なレベルにおいて `beside` と `below` をプリミティブとして見做し `square-of-four` のような命令を持つ言語において幾何学的結合器を接続する共通のパターンを獲得することに努めました。

Stratified design helps make programs *robust*, that is, it makes it likely that small changes in a specification will require correspondingly small changes in the program. For instance, suppose we wanted to change the image based on `wave` shown in Figure 2.9. We could work at the lowest level to change the detailed appearance of the `wave` element; we could work at the middle level to

<sup>60</sup>Section 3.3.4がそのような言語について記述します。

change the way `corner-split` replicates the `wave`; we could work at the highest level to change how `square-limit` arranges the four copies of the corner. In general, each level of a stratified design provides a different vocabulary for expressing the characteristics of the system, and a different kind of ability to change it.

階層化設計はプログラムを *robust*(堅牢) にすることを手助けします。それはつまりプログラムにおける仕様上の小さな変更が相応した小さな変更を要求することを意味します。例えば Figure 2.9 で示された `wave` のイメージを変更したいとします。`wave` 要素の詳細な表現を変更する最も低レベルで行うことも可能ですが、中間のレベルにおいて `corner-split` が `wave` をどのように複製するかについて行うことも可能ですし、最高レベルにおいて `square-limit` がどのように角の 4 つのコピーを配置するかについて変更することも可能です。一般的に階層設計の各レベルは異なる語彙をシステムの特徴を表すのに提供します。そして異なる種類の変更方法をも提供します。

**Exercise 2.52:** Make changes to the square limit of `wave` shown in Figure 2.9 by working at each of the levels described above. In particular:

Figure 2.9 の `wave` の `square-limit` に、上で説明された各レベルで働くことで変更を加えよ。より詳細には、

- a Add some segments to the primitive `wave` painter of Exercise 2.49 (to add a smile, for example).

Exercise 2.49 のプリミティブな `wave` ペイントにいくつか線分を加えよ。(例えば笑顔を追加せよ)

- b Change the pattern constructed by `corner-split` (for example, by using only one copy of the `up-split` and `right-split` images instead of two).

`corner-split` により構築されるパターンを変更せよ(例えば `up-split` や `right-split` のイメージを 2 つでなく 1 つにせよ)

- c Modify the version of `square-limit` that uses `square-of-four` so as to assemble the corners in a different pattern. (For example, you might make the big Mr. Rogers look outward from each corner of the square.)

`square-of-four` を用いる `square-limit` のバージョンを変更し角を異なるパターンで組み立てるようにせよ。(例えば正方形の各角にて Mr. Rogers をそれぞれ外に向けよ)

## 2.3 Symbolic Data

All the compound data objects we have used so far were constructed ultimately from numbers. In this section we extend the representational capability of our language by introducing the ability to work with arbitrary symbols as data.

私達がここまで使用した全ての複合データオブジェクトは最終的には数値から構築されていました。この節では任意のシンボル(記号)をデータとして用いる能力を紹介することで、言語の表現力を拡張します。

### 2.3.1 Quotation

If we can form compound data using symbols, we can have lists such asもし記号を用いて複合データを形成できれば以下のようなリストを持つことができます。

```
(a b c d)
(23 45 17)
((Norah 12) (Molly 9) (Anna 7) (Lauren 6) (Charlotte 4))
```

Lists containing symbols can look just like the expressions of our language:  
記号を含むリストは言語の式と全く同じように見えます。

```
(* (+ 23 45)
  (+ x 9))
(define (fact n)
  (if (= n 1) 1 (* n (fact (- n 1)))))
```

In order to manipulate symbols we need a new element in our language: the ability to *quote* a data object. Suppose we want to construct the list `(a b)`. We can't accomplish this with `(list a b)`, because this expression constructs a list of the *values* of `a` and `b` rather than the symbols themselves. This issue is well known in the context of natural languages, where words and sentences may be

regarded either as semantic entities or as character strings (syntactic entities). The common practice in natural languages is to use quotation marks to indicate that a word or a sentence is to be treated literally as a string of characters. For instance, the first letter of “John” is clearly “J.” If we tell somebody “say your name aloud,” we expect to hear that person’s name. However, if we tell somebody “say ‘your name’ aloud,” we expect to hear the words “your name.” Note that we are forced to nest quotation marks to describe what somebody else might say.<sup>61</sup>

記号を扱う目的のためには言語に新しい要素を必要とします。データオブジェクトを *quote(引用)* する能力です。例えばリスト (a b) を構築したいとします。私達はこれを (list a b) を用いては達成できません。なぜならこの式は a と b の値からリストを構築する式であり、記号それ自体ではないからです。この問題は自然言語の文脈では良く知られていて、単語と文が意味上の要素として見做されているか、または文字列（文法上の要素）として見做されているかの場合が有り得ます。自然言語での共通な慣例は単語や文が文字通りに扱われることを示すためにクオーテーションマークを用いることです。例えば “John” の最初の文字は明らかに “J” です。もし私達が誰かに “貴方の名前を大きな声で言って” と伝えれば、その人の名前を聞くことを期待します。しかもし誰かに “‘貴方の名前’ と大きな声で言って” と伝えれば “貴方の名前” という語を聞くことを期待するでしょう。私達がクオーテーションマークを入れ子にすることを第三者が何を言うだろうかを説明するために強制されていることに注意して下さい。<sup>62</sup>

---

<sup>61</sup> Allowing quotation in a language wreaks havoc with the ability to reason about the language in simple terms, because it destroys the notion that equals can be substituted for equals. For example, three is one plus two, but the word “three” is not the phrase “one plus two.” Quotation is powerful because it gives us a way to build expressions that manipulate other expressions (as we will see when we write an interpreter in Chapter 4). But allowing statements in a language that talk about other statements in that language makes it very difficult to maintain any coherent principle of what “equals can be substituted for equals” should mean. For example, if we know that the evening star is the morning star, then from the statement “the evening star is Venus” we can deduce “the morning star is Venus.” However, given that “John knows that the evening star is Venus” we cannot infer that “John knows that the morning star is Venus.”

<sup>62</sup> 言語の中でクオーテーションを許可することが簡単な語で言語について推論する能力を与えると共に大きな破壊をもたらしています。それが等値な物は等値な物と置換できるという概念を破壊するためです。例えば 1 足す 2 は 3 ですが “3” という語は “1 足す 2” という語句ではありません。クオーテーションは他の表現を操作する表現を構築する

We can follow this same practice to identify lists and symbols that are to be treated as data objects rather than as expressions to be evaluated. However, our format for quoting differs from that of natural languages in that we place a quotation mark (traditionally, the single quote symbol ') only at the beginning of the object to be quoted. We can get away with this in Scheme syntax because we rely on blanks and parentheses to delimit objects. Thus, the meaning of the single quote character is to quote the next object.<sup>63</sup>

データオブジェクトとして扱われるべきであり、式として評価されるべきではないリストとシンボルを区別するために、これと同じ習慣に従うことができます。しかしクオートする形式は自然言語のそれとは異なり、クオーテーションマーク（伝統的にシングルクオートの記号 '）はクオートされるべきオブジェクトの先頭にのみ置かれます。Scheme の文法でこの様に逃れられるのはオブジェクトを区切るのに空白と括弧を信頼することができるためです。従ってシングルクオート文字の意味は次のオブジェクトをクオートすることになります。<sup>64</sup>

Now we can distinguish between symbols and their values:  
これでシンボルとその値を区別することが可能です。

```
(define a 1)  
(define b 2)  
(list a b)  
(1 2)  
(list 'a 'b)
```

---

手段を提供するため強力です。（Chapter 4でインタプリタを書く時に学びます）。しかし言語の中で、同じ言語の他の文について話す文を許すことは“等値な物は等値な物と交換できる”が何を意味すべきかという任意の一貫性を保守することをとても難しくします。例えばもし私達が宵の明星が明けの明星と同じであることを知っている場合、“宵の明星は金星”という文から“明けの明星は金星”であることを推論できます。しかし“John は宵の明星が金星であることを知っている”を与えられても“John は明けの明星が金星であることを知っている”とは推論することはできません。

<sup>63</sup>The single quote is different from the double quote we have been using to enclose character strings to be printed. Whereas the single quote can be used to denote lists or symbols, the double quote is used only with character strings. In this book, the only use for character strings is as items to be printed.

<sup>64</sup>シングルクオートは表示される文字列を囲むのに使用してきたダブルクオートとは異なります。シングルクオートがリストやシンボルを示すのに対し、ダブルクオートは文字列と共にのみ利用されます。この本では文字列の使用方法は表示されるための項目としてのみです。

```
(a b)
(list 'a b)
(a 2)
```

Quotation also allows us to type in compound objects, using the conventional printed representation for lists:<sup>65</sup>

クオーテーションはまた慣習的なリストに対する印字された表現を用いて複合オブジェクトの入力も可能にします。<sup>66</sup>

```
(car '(a b c))
a
(cdr '(a b c))
(b c)
```

In keeping with this, we can obtain the empty list by evaluating '(), and thus dispense with the variable nil.

これを守ることで、空リストを'()を評価して得ることができます。従って、変数 nil の使用を止められます。

---

<sup>65</sup>Strictly, our use of the quotation mark violates the general rule that all compound expressions in our language should be delimited by parentheses and look like lists. We can recover this consistency by introducing a special form quote, which serves the same purpose as the quotation mark. Thus, we would type (quote a) instead of 'a, and we would type (quote (a b c)) instead of '(a b c). This is precisely how the interpreter works. The quotation mark is just a single-character abbreviation for wrapping the next complete expression with quote to form (quote <expression>). This is important because it maintains the principle that any expression seen by the interpreter can be manipulated as a data object. For instance, we could construct the expression (car '(a b c)), which is the same as (car (quote (a b c))), by evaluating (list 'car (list 'quote '(a b c))).

<sup>66</sup>厳密には私達のクオーテーションマークの使用方法は言語における全ての複合式は括弧で区切られリストのように見えるという全体のルールを破ります。この整合性に対しては特殊形式 quote を紹介することで回復することが可能です。これはクオーテーションマークと同じ目的を演じます。従って'aの代わりに (quote a)と入力できますし、「(a b c)の代わりに (quote (a b c))と入力できます。これはインタプリタは正確にはどのように働くかということです。クオーテーションマークは单一文字による省略形に過ぎず次の完全な式を quote でラッピングすることで (quote <expression>) を形成します。これは重要なことです。なぜならインタプリタに読まれる任意の式がデータオブジェクトとして扱うことができるという原則を保持するからです。例えば (car '(a b c)) という式は (car (quote (a b c)))と同じで、(list 'car (list 'quote '(a b c)))を評価することで構築できます。

One additional primitive used in manipulating symbols is `eq?`, which takes two symbols as arguments and tests whether they are the same.<sup>67</sup> Using `eq?`, we can implement a useful procedure called `memq`. This takes two arguments, a symbol and a list. If the symbol is not contained in the list (i.e., is not `eq?` to any item in the list), then `memq` returns false. Otherwise, it returns the sublist of the list beginning with the first occurrence of the symbol:

記号を操作するのに使われるもう 1 つ追加のプリミティブとして `eq?` があります。これは 2 つのシンボルを引数として取りそれらが同じであるかテストします。<sup>68</sup> `eq?` を用いることで `memq` と呼ばれる便利な手続を実装できます。これは 2 つの引数、シンボルとリストを取ります。もしシンボルがリストに含まれていない場合(つまりリスト中のどの項目にも `eq?` でない場合)`memq` は `false` を返します。そうでなければリスト中のそのシンボルが最初に出現する場所からのサブリストを返します。

```
(define (memq item x)
  (cond ((null? x) false)
        ((eq? item (car x)) x)
        (else (memq item (cdr x)))))
```

For example, the value of 例えば、次の式の値は

```
(memq 'apple '(pear banana prune))
```

is `false`, whereas the value of

`false` になります。そして次の式の値は

```
(memq 'apple '(x (apple sauce) y apple pear))
```

is `(apple pear)`.

`(apple pear)` です。

**Exercise 2.53:** What would the interpreter print in response to evaluating each of the following expressions?

<sup>67</sup>We can consider two symbols to be “the same” if they consist of the same characters in the same order. Such a definition skirts a deep issue that we are not yet ready to address: the meaning of “sameness” in a programming language. We will return to this in [Chapter 3 \(Section 3.1.3\)](#).

<sup>68</sup>2 つのシンボルが同じ文字で同じ順に構成されている場合にそれらが“同じ”であると考えることができます。そのような定義はまだ私達が解決するには準備の足りない深い問題を回避しています。プログラミング言語における“同一性”的意味です。私達はこの問題に [Chapter 3 \(Section 3.1.3\)](#) にて戻ります。

以下の式のそれぞれを評価した応答としてインタプリタは何を表示するか?

```
(list 'a 'b 'c)
(list (list 'george))
(cdr '((x1 x2) (y1 y2)))
(cadr '((x1 x2) (y1 y2)))
(pair? (car '(a short list)))
(memq 'red '((red shoes) (blue socks)))
(memq 'red '(red shoes blue socks))
```

**Exercise 2.54:** Two lists are said to be `equal?` if they contain equal elements arranged in the same order. For example,

2つのリストはそれが同じ要素と同じ順で持っている場合に `equal?` と言える。例えば

```
(equal? '(this is a list) '(this is a list))
```

is true, but

は真であるが

```
(equal? '(this is a list) '(this (is a) list))
```

is false. To be more precise, we can define `equal?` recursively in terms of the basic `eq?` equality of symbols by saying that `a` and `b` are `equal?` if they are both symbols and the symbols are `eq?`, or if they are both lists such that `(car a)` is `equal?` to `(car b)` and `(cdr a)` is `equal?` to `(cdr b)`. Using this idea, implement `equal?` as a procedure.<sup>69</sup>

は偽である。具体的には基本となる `eq?` の記号の等価性を再帰的に用いて `equal?` を定義できる。`a` と `b` が `equal?` であるとはそれらが両方とも記号である場合、かつ記号が `eq?` である場合、または

---

<sup>69</sup>In practice, programmers use `equal?` to compare lists that contain numbers as well as symbols. Numbers are not considered to be symbols. The question of whether two numerically equal numbers (as tested by `=`) are also `eq?` is highly implementation-dependent. A better definition of `equal?` (such as the one that comes as a primitive in Scheme) would also stipulate that if `a` and `b` are both numbers, then `a` and `b` are `equal?` if they are numerically equal.

両方ともリストであり (car a) が (car b) に equal? であり、かつ (cdr a) が (cdr b) に equal? であるような場合である。この考えを用いて equal? を手続として実装せよ。<sup>70</sup>

**Exercise 2.55:** Eva Lu Ator types to the interpreter the expression  
Eva Lu Ator はインタプリタに次の式を入力した。

```
(car ''abracadabra)
```

To her surprise, the interpreter prints back quote. Explain.

驚いたことにインタプリタは quote を応答として表示した。説明せよ。

### 2.3.2 Example: Symbolic Differentiation

As an illustration of symbol manipulation and a further illustration of data abstraction, consider the design of a procedure that performs symbolic differentiation of algebraic expressions. We would like the procedure to take as arguments an algebraic expression and a variable and to return the derivative of the expression with respect to the variable. For example, if the arguments to the procedure are  $ax^2 + bx + c$  and  $x$ , the procedure should return  $2ax + b$ . Symbolic differentiation is of special historical significance in Lisp. It was one of the motivating examples behind the development of a computer language for symbol manipulation. Furthermore, it marked the beginning of the line of research that led to the development of powerful systems for symbolic mathematical work, which are currently being used by a growing number of applied mathematicians and physicists.

記号操作の説明として、及びより一層のデータ抽象の説明として、代数式の記号微分を行う手続の設計について考えてみましょう。手続は引数として代数式と変数を取り変数に対する式の導関数を返すことにします。例えば手続に対する引数が  $ax^2 + bx + c$  と  $x$  の時、手続は  $2ax + b$  を返さなければいけません。記号微分は Lisp にとって歴史的に特別な意味があります。記号操作のため

---

<sup>70</sup> 実際には、プログラマは equal? を数値と同じくシンボルを含むリストの比較に用いる。数値は記号とは認識されない。数の上で等しい二つの数値 (= でテストした場合の様に) が eq? でもそうであるかという問題は高度に実装依存である。equal? の (Scheme にプリミティブとして提供されているような) より良い定義でも、もし a と b が数値であるなら、それらが数値として等しい場合に equal? であると明記するだろう。

のコンピュータ言語の開発の裏にこれがその動機の一例として存在しました。さらにこれが現在増大する応用数学者及び科学者に用いられている記号数理の成果のための強力なシステムの開発へと導く一連の研究の始まりに跡を残しました。

In developing the symbolic-differentiation program, we will follow the same strategy of data abstraction that we followed in developing the rational-number system of [Section 2.1.1](#). That is, we will first define a differentiation algorithm that operates on abstract objects such as “sums,” “products,” and “variables” without worrying about how these are to be represented. Only afterward will we address the representation problem.

記号微分プログラムの開発においても [Section 2.1.1](#) の分数システムの開発と同じデータ抽象化の戦略に従います。最初に “sums,” “products,” “variables” のような抽象オブジェクトを操作する微分アルゴリズムを定義します。これらがどのように表現されているのかについて心配することありません。後程、表現上の問題については解決しましょう。

### The differentiation program with abstract data

In order to keep things simple, we will consider a very simple symbolic-differentiation program that handles expressions that are built up using only the operations of addition and multiplication with two arguments. Differentiation of any such expression can be carried out by applying the following reduction rules:

問題を簡単にするために、二引数の足し算とかけ算の命令のみから構築される式のみを扱うとしても簡単な記号微分プログラムについて考えましょう。任意のそのような式の微分は以下の簡約ルールを適用することで実行されます。

#### 2.3.3 Example: Symbolic Differentiation

As an illustration of symbol manipulation and a further illustration of data abstraction, consider the design of a procedure that performs symbolic differentiation of algebraic expressions. We would like the procedure to take as arguments an algebraic expression and a variable and to return the derivative of the expression with respect to the variable. For example, if the arguments to the procedure are  $ax^2 + bx + c$  and  $x$ , the procedure should return  $2ax + b$ . Symbolic differentiation is of special historical significance in Lisp. It was one

of the motivating examples behind the development of a computer language for symbol manipulation. Furthermore, it marked the beginning of the line of research that led to the development of powerful systems for symbolic mathematical work, which are currently being used by a growing number of applied mathematicians and physicists.

記号操作の説明として、及びより一層のデータ抽象の説明として、代数式の記号微分を行う手続の設計について考えてみましょう。手続は引数として代数式と変数を取り変数に対する式の導関数を返すことにします。例えば手続に対する引数が  $ax^2 + bx + c$  と  $x$  の時、手続は  $2ax + b$  を返さなければいけません。記号微分は Lisp にとって歴史的に特別な意味があります。記号操作のためのコンピュータ言語の開発の裏にこれがその動機の一例として存在しました。さらにこれが現在増大する応用数学者及び科学者に用いられている記号数理の成果のための強力なシステムの開発へと導く一連の研究の始まりに跡を残しました。

In developing the symbolic-differentiation program, we will follow the same strategy of data abstraction that we followed in developing the rational-number system of [Section 2.1.1](#). That is, we will first define a differentiation algorithm that operates on abstract objects such as “sums,” “products,” and “variables” without worrying about how these are to be represented. Only afterward will we address the representation problem.

記号微分プログラムの開発においても [Section 2.1.1](#) の分数システムの開発と同じデータ抽象化の戦略に従います。最初に “sums,” “products,” “variables” のような抽象オブジェクトを操作する微分アルゴリズムを定義します。これらがどのように表現されているのかについて心配することありません。後程、表現上の問題については解決しましょう。

## The differentiation program with abstract data

In order to keep things simple, we will consider a very simple symbolic-differentiation program that handles expressions that are built up using only the operations of addition and multiplication with two arguments. Differentiation of any such expression can be carried out by applying the following reduction rules:

問題を簡単にするために、二引数の足し算とかけ算の命令のみから構築される式のみを扱うとしても簡単な記号微分プログラムについて考えましょう。任意のそのような式の微分は以下の簡約ルールを適用することで実行されます。

$$\frac{dc}{dx} = 0, \quad c \text{ は定数か、 } x \text{ と異なる変数},$$

$$\frac{dx}{dx} = 1,$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx},$$

$$\frac{d(uv)}{dx} = u \frac{dv}{dx} + v \frac{du}{dx}.$$

Observe that the latter two rules are recursive in nature. That is, to obtain the derivative of a sum we first find the derivatives of the terms and add them. Each of the terms may in turn be an expression that needs to be decomposed. Decomposing into smaller and smaller pieces will eventually produce pieces that are either constants or variables, whose derivatives will be either 0 or 1.

後者の 2 つのルールは自然に再帰的であることに注意して下さい。つまり和の導関数を得るためにには最初に項の導関数を求めそれらを足す必要があります。各項は順に分解が必要な式に成り得ます。順により小さな部分へと分解していくことはやがて定数か、その導関数が 0 か 1 のどちらかになる変数になります。

To embody these rules in a procedure we indulge in a little wishful thinking, as we did in designing the rational-number implementation. If we had a means for representing algebraic expressions, we should be able to tell whether an expression is a sum, a product, a constant, or a variable. We should be able to extract the parts of an expression. For a sum, for example we want to be able to extract the addend (first term) and the augend (second term). We should also be able to construct expressions from parts. Let us assume that we already have procedures to implement the following selectors, constructors, and predicates:

これらのルールを手続で具体化するために、分数実装の設計で行ったように少し希望的観測に耽ります。もし代数式を表現するための手段があるのなら式が和か積か定数であるかを判別することができるはずです。式のパートを抽出することができるはずです。例えば足し算に対して加数(第一項)と被加数(第二項)を抽出できるはずです。またパートから式を構築することもできるはずです。既に以下のセレクタ、コンストラクタ、述語を実装するための手続を持っていると仮定しましょう。

(variable? e)

e は変数であるか?

(same-variable? v1 v2)

v1 と v2 は同じ変数であるか?

(sum? e)	e は和か?
(addend e)	和 e の加数.
(augend e)	和 e の被加数.
(make-sum a1 a2)	a1 と a2 の和を構築する
(product? e)	e は積か?
(multiplier e)	積 e の乗数
(multiplicand e)	積 e の被乗数
(make-product m1 m2)	m1 と m2 の積を構築する

Using these, and the primitive predicate `number?`, which identifies numbers, we can express the differentiation rules as the following procedure:

これらと数値であるかを判断するプリミティブな述語 `number?` を用いて、以下の手続の様に微分のルールを表現できます。

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        ((sum? exp) (make-sum (deriv (addend exp) var)
                               (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
          (make-product (multiplier exp)
                        (deriv (multiplicand exp) var))
          (make-product (deriv (multiplier exp) var)
                        (multiplicand exp)))))

        (else
         (error "unknown expression type: DERIV" exp))))
```

This `deriv` procedure incorporates the complete differentiation algorithm. Since it is expressed in terms of abstract data, it will work no matter how we choose to represent algebraic expressions, as long as we design a proper set of selectors and constructors. This is the issue we must address next.

この `deriv` 手続は完全な微分アルゴリズムに立脚しています。代数データの項により表現されているため、適切なセレクタとコンストラクタを設計する限りにおいて、どのように代数式を表現しても動きます。この条件の部分が次に解決すべき問題です。

## Representing algebraic expressions

We can imagine many ways to use list structure to represent algebraic expressions. For example, we could use lists of symbols that mirror the usual algebraic notation, representing  $ax + b$  as the list `(a * x + b)`. However, one especially straightforward choice is to use the same parenthesized prefix notation that Lisp uses for combinations; that is, to represent  $ax + b$  as `(+ (* a x) b)`. Then our data representation for the differentiation problem is as follows:

代数式を表現するリスト構造を使用する手法は数多く想像できます。例えば通常の代数記法を真似する記号のリストを用い、 $ax + b$  をリスト `(a * x + b)` の様に表現することもできるでしょう。しかし特に直接的な1つの選択は Lisp が複合式に用いるのと同じく括弧で括った接頭辞記述法です。つまり  $ax + b$  は `(+ (* a x) b)` と表現されます。従って微分問題に対するデータ表現は以下のとおりです。

- The variables are symbols. They are identified by the primitive predicate `symbol?`:

変数はシンボルである。プリミティブな述語 `symbol?` で判別される。

```
(define (variable? x) (symbol? x))
```

- Two variables are the same if the symbols representing them are `eq?`:  
2つの変数はそれらを表現するシンボルが `eq?` である時同じだ。

```
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
```

- Sums and products are constructed as lists:  
和と積はリストとして構築される。

```
(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))
```

- A sum is a list whose first element is the symbol `+`:  
和は最初の要素がシンボル `+` のリストだ。

```
(define (sum? x) (and (pair? x) (eq? (car x) '+)))
```

- The addend is the second item of the sum list:  
加数は和のリストの二つ目の項だ。

```
(define (addend s) (cadr s))
```

- The augend is the third item of the sum list:  
被加数は和のリストの三つめの項だ。

```
(define (augend s) (caddr s))
```

- A product is a list whose first element is the symbol \*:  
積は最初の要素がシンボル \* のリストだ。

```
(define (product? x) (and (pair? x) (eq? (car x) '*)))
```

- The multiplier is the second item of the product list:  
乗数は積のリストの二つ目の項だ。

```
(define (multiplier p) (cadr p))
```

- The multiplicand is the third item of the product list:  
被乗数は積のリストの三つめの項だ。

```
(define (multiplicand p) (caddr p))
```

Thus, we need only combine these with the algorithm as embodied by `deriv` in order to have a working symbolic-differentiation program. Let us look at some examples of its behavior:

従って記号微分プログラムを得るために、これらを `deriv` により組込まれたアルゴリズムを用いて組み立てることのみが必要でです。いくつかの例とその振舞を見てみましょう。

```
(deriv '(+ x 3) 'x)
(+ 1 0)
(deriv '(* x y) 'x)
(+ (* x 0) (* 1 y))
(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* x y) (+ 1 0))
  (* (+ (* x 0) (* 1 y))
     (+ x 3)))
```

The program produces answers that are correct; however, they are unsimplified. It is true that

プログラムは正しい解答を生成します。しかし、それらは簡略化されていません。

$$\frac{d(xy)}{dx} = x \cdot 0 + 1 \cdot y,$$

but we would like the program to know that  $x \cdot 0 = 0$ ,  $1 \cdot y = y$ , and  $0 + y = y$ . The answer for the second example should have been simply  $y$ . As the third example shows, this becomes a serious issue when the expressions are complex.

しかし私達はこのプログラムに  $x \cdot 0 = 0$ ,  $1 \cdot y = y$ ,  $0 + y = y$  を理解して欲しいと望みます。二つ目の例の解答は単純に  $y$  となるべきです。三つめの例が示すように、これは式が複雑な場合には深刻な問題となります。

Our difficulty is much like the one we encountered with the rational-number implementation: we haven't reduced answers to simplest form. To accomplish the rational-number reduction, we needed to change only the constructors and the selectors of the implementation. We can adopt a similar strategy here. We won't change `deriv` at all. Instead, we will change `make-sum` so that if both summands are numbers, `make-sum` will add them and return their sum. Also, if one of the summands is 0, then `make-sum` will return the other summand.

この困難は分数実装において出くわしたものととても似ています。最も単純な形式に解答を約分していませんでした。分数を約分するためには実装のコンストラクタとセレクタのみを変更する必要がありました。ここでも同様の戦略を受け入れることができます。`deriv` には全く変更は加えません。その代わりに `make-sum` を変更し両方の加数が数値である場合、`make-sum` はそれらを足してその和を返します。また加数の 1 つが 0 ならば `make-sum` はもう一方の加数を返します。

```
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2))
         (+ a1 a2))
        (else (list '+ a1 a2))))
```

This uses the procedure `=number?`, which checks whether an expression is equal to a given number:

これには手続 `=number?` を用いました。式が与えられた数値と等しいかチェックします。

```
(define (=number? exp num) (and (number? exp) (= exp num)))
```

Similarly, we will change `make-product` to build in the rules that 0 times anything is 0 and 1 times anything is the thing itself:

同様に `make-product` を変更し任意の項に 0 を掛ければ 0、任意の項に 1 を掛ければそれ自身にするルールを構築します。

```
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
```

Here is how this version works on our three examples:

これがこのバージョンが先程の 3 つの例でどのように動くかです

```
(deriv '(+ x 3) 'x)
1
(deriv '(* x y) 'x)
y
(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* x y) (* y (+ x 3)))
```

Although this is quite an improvement, the third example shows that there is still a long way to go before we get a program that puts expressions into a form that we might agree is “simplest.” The problem of algebraic simplification is complex because, among other reasons, a form that may be simplest for one purpose may not be for another.

これはとても改善が見られますが、三つめの例は式を“最も単純”だと同意を得られる形式に変形するプログラムを得るまでには今だ長い道程があることを示します。代数の簡約の問題は複雑です。他の理由の中でも、ある目的にとって最も単純な形式が他の目的にとってはそうではないことが有り得るためです。

**Exercise 2.56:** Show how to extend the basic differentiator to handle more kinds of expressions. For instance, implement the differentiation rule

より多くの種類の式を扱うために基本的な微分をどのように拡張すべきか示せ。例として、以下の微分ルールを実装せよ。

$$\frac{d(u^n)}{dx} = nu^{n-1} \frac{du}{dx}$$

by adding a new clause to the `deriv` program and defining appropriate procedures `exponentiation?`, `base`, `exponent`, and `make-exponentiation`. (You may use the symbol `**` to denote exponentiation.) Build in the rules that anything raised to the power 0 is 1 and anything raised to the power 1 is the thing itself.

`deriv` プログラムに新しい節を追加し、適切な手続 `exponentiation?`, `base`, `exponent`, `make-exponentiation` を定義せよ。(シンボル `**` を指数演算の表記に用いても良い)。任意の数の 0 乗は 1 であり、任意の数の 1 乗はそれ自身であるというルールを構築せよ。

**Exercise 2.57:** Extend the differentiation program to handle sums and products of arbitrary numbers of (two or more) terms. Then the last example above could be expressed as

微分プログラムを拡張し、(2 以上の) 任意の数の項の和と積を扱えるようにせよ。すると上の最後の例は以下のように表現できる。

```
(deriv '(* x y (+ x 3)) 'x)
```

Try to do this by changing only the representation for sums and products, without changing the `deriv` procedure at all. For example, the `addend` of a sum would be the first term, and the `augend` would be the sum of the rest of the terms.

この問題を和と積の表現のみを変更することで行え。`deriv` 手続には全く変更を加えない。例えば和の `addend`(加数) は最初の項になり、`augend`(被加数) は残りの項の和となるであろう。

**Exercise 2.58:** Suppose we want to modify the differentiation program so that it works with ordinary mathematical notation, in which `+` and `*` are infix rather than prefix operators. Since the differentiation program is defined in terms of abstract data, we can modify it to work with different representations of expressions solely by changing the predicates, selectors, and constructors that

define the representation of the algebraic expressions on which the differentiator is to operate.

微分プログラムを変更し通常の数学の記法を扱えるようにしたいとする。 $+$ と $*$ は接中辞となり接頭辞演算子ではなくなる。微分プログラムは抽象データを用いて定義されているので、もっぱら微分プログラムが操作する代数式を表現する述語、セレクタ、コンストラクタを変更することで式の異なる表現を対応するように変更することができる。

- a Show how to do this in order to differentiate algebraic expressions presented in infix form, such as  $(x + (3 * (x + (y + 2))))$ . To simplify the task, assume that  $+$  and  $*$  always take two arguments and that expressions are fully parenthesized.  
 $(x + (3 * (x + (y + 2))))$  の様な接中辞で表される代数式を微分することを行なうのか示せ。作業を簡単にするために $+$ と $*$ は常に 2 つの引数を取り式は完全に括弧で括られていると仮定せよ。
- b The problem becomes substantially harder if we allow standard algebraic notation, such as  $(x + 3 * (x + y + 2))$ , which drops unnecessary parentheses and assumes that multiplication is done before addition. Can you design appropriate predicates, selectors, and constructors for this notation such that our derivative program still works?  
 $(x + 3 * (x + y + 2))$  のような標準的な代数記法を認めることで問題は大幅に難しくなる。これは必要の無い括弧を省略し、乗算は加算の前に行われると仮定している。私達の微分プログラムがそれでも働くこの記法に対する適切な述語、セレクタ、コンストラクタを設計できるだろうか？

### 2.3.4 Example: Representing Sets

In the previous examples we built representations for two kinds of compound data objects: rational numbers and algebraic expressions. In one of these examples we had the choice of simplifying (reducing) the expressions at either construction time or selection time, but other than that the choice of a representation for these structures in terms of lists was straightforward. When we

turn to the representation of sets, the choice of a representation is not so obvious. Indeed, there are a number of possible representations, and they differ significantly from one another in several ways.

以前の例において 2 つ種類の複合データオブジェクトの表現を構築しました。分数と代数式です。これらの例の 1 つでは組立時と選択時のどちらかで式を単純化(簡約化)を行うか選択肢がありました。しかしそれ以外ではリストを用いたこれらの構造に対する表現の選択肢は直接的なものでした。私達が集合の表現に向かう時、表現の選択肢はあまり明白ではありません。本当に数多くの可能な表現が存在し、それらはお互いからいくつかの点において著しく異なります。

Informally, a set is simply a collection of distinct objects. To give a more precise definition we can employ the method of data abstraction. That is, we define “set” by specifying the operations that are to be used on sets. These are `union-set`, `intersection-set`, `element-of-set?`, and `adjoin-set`. `Element-of-set?` is a predicate that determines whether a given element is a member of a set. `Adjoin-set` takes an object and a set as arguments and returns a set that contains the elements of the original set and also the adjoined element. `Union-set` computes the union of two sets, which is the set containing each element that appears in either argument. `Intersection-set` computes the intersection of two sets, which is the set containing only elements that appear in both arguments. From the viewpoint of data abstraction, we are free to design any representation that implements these operations in a way consistent with the interpretations given above.

非公式には集合は異なる要素の単純な集まりです。より正確な定義を与るために私達はデータ抽象の手法を用いることができます。それは“集合”を集合上で用いられる操作を特定することで定義することです。これらは `union-set`, `intersection-set`, `element-of-set?`, `adjoin-set` です。`element-of-set?` は与えられた要素が集合のメンバーであるかを判定する述語です。`adjoin-set` はオブジェクトと集合を引数として取り、元の集合の要素と挿入された要素をも含む集合を返します。`union-set` はどちらかの引数に現れる全ての要素含む集合である、2つの集合の和集合を計算します。`intersection-set` は両方の引数の中に現れる要素のみを含む、2つの集合の共通集合を計算します。データ抽象の視点から見れば、私達は上で与えられた解釈と一致する方法であれば、これらの命令を実装するどんな表現を設計することも自由です。<sup>71</sup>

---

<sup>71</sup>If we want to be more formal, we can specify “consistent with the interpretations given above” to mean that the operations satisfy a collection of rules such as these:

## Sets as unordered lists

One way to represent a set is as a list of its elements in which no element appears more than once. The empty set is represented by the empty list. In this representation, `element-of-set?` is similar to the procedure `memq` of Section 2.3.1. It uses `equal?` instead of `eq?` so that the set elements need not be symbols:

集合を表現するための 1 つの方法は、どの要素も一度より多くは現れない要素のリストとします。空集合は空リストとして表現されます。この表現では `element-of-set?` は Section 2.3.1 の手続 `memq` と似ています。`eq?` の代わりに `equal?` を用いているので集合要素はシンボルである必要はありません。

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set))))))
```

Using this, we can write `adjoin-set`. If the object to be adjoined is already in the set, we just return the set. Otherwise, we use `cons` to add the object to the list that represents the set:

これを用いて `adjoin-set` を書けます。`adjoin` されるオブジェクトが既に集合に存在する場合、単に元の集合を返します。そうでなければ `cons` を用いてオブジェクトを集合を表すリストに追加します。

---

もしより正式でありたければ、“上で与えられた解釈と一致する”の部分を、命令群が以下のようなルールの集合を満たすと指定することができます。

- For any set  $S$  and any object  $x$ ,  $(\text{element-of-set? } x (\text{adjoin-set } x S))$  is true (informally: “Adjoining an object to a set produces a set that contains the object”).

任意の集合  $S$  と任意のオブジェクト  $x$  に対し、 $(\text{element-of-set? } x (\text{adjoin-set } x S))$  は真（非公式には“オブジェクトを集合に `adjoin` すればそのオブジェクトを含む集合を生成する”）

- For any sets  $S$  and  $T$  and any object  $x$ ,  $(\text{element-of-set? } x (\text{union-set } S T))$  is equal to  $(\text{or } (\text{element-of-set? } x S) (\text{element-of-set? } x T))$  (informally: “The elements of  $(\text{union } S T)$  are the elements that are in  $S$  or in  $T$ ”).

任意の集合  $S$  と  $T$  と任意のオブジェクト  $x$  に対し、 $(\text{element-of-set? } x (\text{union-set } S T))$  は  $(\text{or } (\text{element-of-set? } x S) (\text{element-of-set? } x T))$  に等しい（非公式には“ $(\text{union } S T)$  の要素は  $S$  または  $T$  に存在する要素”）

- For any object  $x$ ,  $(\text{element-of-set? } x '())$  is false (informally: “No object is an element of the empty set”).

任意のオブジェクト  $x$  に対し  $(\text{element-of-set? } x '())$  は偽（非公式には“どのオブジェクトも空集合の要素ではない”）

```
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))
```

For `intersection-set` we can use a recursive strategy. If we know how to form the intersection of `set2` and the `cdr` of `set1`, we only need to decide whether to include the `car` of `set1` in this. But this depends on whether `(car set1)` is also in `set2`. Here is the resulting procedure:

`intersection-set` に対して再帰の戦略を使用できます。もし `set2` と `set1` の `cdr` の共通集合の求める方がわかれば、これに `set1` の `car` を含めるかどうか決定することのみが必要です。しかしこれは `(car set1)` が `set2` にも存在するかに依存します。以下に結果の手続を示します。

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

In designing a representation, one of the issues we should be concerned with is efficiency. Consider the number of steps required by our set operations. Since they all use `element-of-set?`, the speed of this operation has a major impact on the efficiency of the set implementation as a whole. Now, in order to check whether an object is a member of a set, `element-of-set?` may have to scan the entire set. (In the worst case, the object turns out not to be in the set.) Hence, if the set has  $n$  elements, `element-of-set?` might take up to  $n$  steps. Thus, the number of steps required grows as  $\Theta(n)$ . The number of steps required by `adjoin-set`, which uses this operation, also grows as  $\Theta(n)$ . For `intersection-set`, which does an `element-of-set?` check for each element of `set1`, the number of steps required grows as the product of the sizes of the sets involved, or  $\Theta(n^2)$  for two sets of size  $n$ . The same will be true of `union-set`.

表現の設計において、私達が考慮しなければならない問題の1つは効率です。集合操作により必要とされるステップ数を考えて下さい。それら全てが `element-of-set?` を使用するので、この命令のスピードは総じて集合実装の効率上に主要な影響を与えます。ここで、あるオブジェクトが集合の要素であるかをチェックするために、`element-of-set?` は集合全体をスキャンしなければ

ならないかもしれません。(最悪の場合、そのオブジェクトがその集合の中に存在しないことが分かるかもしれません)。それ故にもしその集合が  $n$  要素を持つ場合、`element-of-set?` は最大  $n$  ステップかかるかもしれません。従って必要とされるステップ数は  $\Theta(n)$  で増加します。`adjoin-set` により必要とされるステップ数は、それがこの命令を用いるので、これもまた  $\Theta(n)$  で増加します。`intersection-set` は、`set1` の各要素に対し `element-of-set?` のチェックを行うため、必要とされるステップ数は関係する集合のサイズの積か、またはサイズ  $n$  の 2 つの集合に対し  $\Theta(n^2)$  で増加します。`union-set` に対しても同じことが言えます。

**Exercise 2.59:** Implement the `union-set` operation for the unordered-list representation of sets.

集合の順序無しリスト表現に対する `union-set` 命令を実装せよ。

**Exercise 2.60:** We specified that a set would be represented as a list with no duplicates. Now suppose we allow duplicates. For instance, the set  $\{1, 2, 3\}$  could be represented as the list  $(2 \ 3 \ 2 \ 1 \ 3 \ 2 \ 2)$ . Design procedures `element-of-set?`, `adjoin-set`, `union-set`, and `intersection-set` that operate on this representation. How does the efficiency of each compare with the corresponding procedure for the non-duplicate representation? Are there applications for which you would use this representation in preference to the non-duplicate one?

集合は重複無しのリストとして表現されると指示した。ここで重複を許可すると仮定してみる。例として集合  $\{1, 2, 3\}$  はリスト  $(2 \ 3 \ 2 \ 1 \ 3 \ 2 \ 2)$  として表現できるだろう。この表現上で操作を行う手続き `element-of-set?`, `adjoin-set`, `union-set`, `intersection-set` を設計せよ。それぞれの効率は対応する重複無し表現に対する手続きに比べてどれ程だろうか？重複無しの集合に優先してこの表現を用いるだろうアプリケーションはあるだろうか？

## Sets as ordered lists

One way to speed up our set operations is to change the representation so that the set elements are listed in increasing order. To do this, we need some way to compare two objects so that we can say which is bigger. For example, we could compare symbols lexicographically, or we could agree on some method

for assigning a unique number to an object and then compare the elements by comparing the corresponding numbers. To keep our discussion simple, we will consider only the case where the set elements are numbers, so that we can compare elements using `>` and `<`. We will represent a set of numbers by listing its elements in increasing order. Whereas our first representation above allowed us to represent the set  $\{1, 3, 6, 10\}$  by listing the elements in any order, our new representation allows only the list  $(1 \ 3 \ 6 \ 10)$ .

私達の集合操作を高速化するための 1 つの方法として表現を変更することで集合要素を昇順に並べる方法があります。これを行うには 2 つのオブジェクトを比較する何らかの方法が必要です。それによりどちらが大きいを言うことができます。例えはシンボルを辞書順で比較したり、オブジェクトに一意の番号を付けその後要素を対応する番号で比較するための何らかの方法について同意できるでしょう。議論を単純にするため私達は集合要素が数値である場合のみについて考えます。それにより要素を `>` と `<` を用いて比較することができます。数値の集合をその要素を昇順に並べることで表現しましょう。上の最初の表現は集合  $\{1, 3, 6, 10\}$  を要素を任意の順で並べることで表現できる一方で、新しい表現はリスト  $(1 \ 3 \ 6 \ 10)$  のみを許します。

One advantage of ordering shows up in `element-of-set?`: In checking for the presence of an item, we no longer have to scan the entire set. If we reach a set element that is larger than the item we are looking for, then we know that the item is not in the set:

順序付けの 1 つの利点は `element-of-set?` にて現れます。項目の存在をチェックする場合において、集合全体をスキャンする必要がありません。もし探している項目よりも大きな要素に出会ったならばその集合にこの項目が無いことがわかります。

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-set? x (cdr set)))))
```

How many steps does this save? In the worst case, the item we are looking for may be the largest one in the set, so the number of steps is the same as for the unordered representation. On the other hand, if we search for items of many different sizes we can expect that sometimes we will be able to stop searching at a point near the beginning of the list and that other times we will still need to

examine most of the list. On the average we should expect to have to examine about half of the items in the set. Thus, the average number of steps required will be about  $n/2$ . This is still  $\Theta(n)$  growth, but it does save us, on the average, a factor of 2 in number of steps over the previous implementation.

これがどれだけのステップを割引くでしょうか? 最悪の場合、探している項目は集合の中で一番大きい物かもしれません。その場合ステップ数は順序無し表現と同じです。しかし一方でも多くの異なるサイズの項目を探している場合、時々はリストの先頭近くの点で検索を停止することができることを期待できます。そして他の場合にはやはりリストのほとんどを試験しなければいけません。平均では集合の項目数の半分近くを試験しなければいけないことが期待できるはずです。従って必要とされる平均のステップ数は約  $n/2$  になります。これはそれでも  $\Theta(n)$  で増加しますが、以前の実装に対して平均的にはステップ数において半分に節約します。

We obtain a more impressive speedup with `intersection-set`. In the unordered representation this operation required  $\Theta(n^2)$  steps, because we performed a complete scan of `set2` for each element of `set1`. But with the ordered representation, we can use a more clever method. Begin by comparing the initial elements, `x1` and `x2`, of the two sets. If `x1` equals `x2`, then that gives an element of the intersection, and the rest of the intersection is the intersection of the `cdr`-s of the two sets. Suppose, however, that `x1` is less than `x2`. Since `x2` is the smallest element in `set2`, we can immediately conclude that `x1` cannot appear anywhere in `set2` and hence is not in the intersection. Hence, the intersection is equal to the intersection of `set2` with the `cdr` of `set1`. Similarly, if `x2` is less than `x1`, then the intersection is given by the intersection of `set1` with the `cdr` of `set2`. Here is the procedure:

`intersection-set` ではより目覚ましい高速化を得ます。順序無し表現ではこの命令は  $\Theta(n^2)$  ステップを必要としました。`set1` の各要素に対して `set2` の完全なスキャンを実行していたためです。しかし順序有り表現ではより賢い方法を用いることができます。二つの集合の最初の要素 `x1` と `x2` を比較することで始め、もし `x1` と `x2` が等しい場合にはそれらは共通集合の要素です。そして共通集合の残りは 2 つの集合の `cdr` の共通集合です。そうでなく `x1` が `x2` より小さい場合を考えます。`x2` は `set2` の最小の要素ですから直ぐに `x1` は `set2` のどこにも現れず、従って共通集合には有り得ません。従って共通集合は `set2` と `set1` の `cdr` の共通集合に等しいとなります。同様にもし `x2` が `x1` より小さい場合、共通集合は `set1` と `set2` の `cdr` の共通集合にて与えられます。以下に手続を与えます。

```

(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((= x1 x2)
               (cons x1 (intersection-set (cdr set1)
                                           (cdr set2))))
              ((< x1 x2)
               (intersection-set (cdr set1) set2))
              ((< x2 x1)
               (intersection-set set1 (cdr set2)))))))

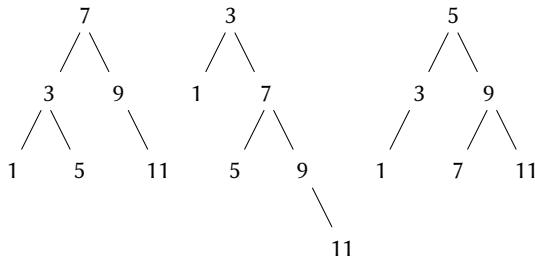
```

To estimate the number of steps required by this process, observe that at each step we reduce the intersection problem to computing intersections of smaller sets—removing the first element from `set1` or `set2` or both. Thus, the number of steps required is at most the sum of the sizes of `set1` and `set2`, rather than the product of the sizes as with the unordered representation. This is  $\Theta(n)$  growth rather than  $\Theta(n^2)$ —a considerable speedup, even for sets of moderate size.

この処理により必要とされるステップ数を推定するために、各ステップにて共通集合問題は縮小され、`set1` か `set2`、又はその両方の最初の要素を削除することで、より小さな集合の共通部分を求める問題になっていることに注意して下さい。従って必要とされるステップ数は最大でも `set1` と `set2` のサイズの合計であり、順序無し表現におけるサイズの積とはなりません。これは  $\Theta(n^2)$  でなく、 $\Theta(n)$  で増加するため、例え中程度のサイズの集合に対してでも考慮に値する高速化です。

**Exercise 2.61:** Give an implementation of `adjoin-set` using the ordered representation. By analogy with `element-of-set?` show how to take advantage of the ordering to produce a procedure that requires on the average about half as many steps as with the unordered representation.

順序有り表現を用いた `adjoin-set` の実装を与えよ。`element-of-set?` との類似点にて、どのように順序の利点を用いて順序無し表現に対し平均で約半分のステップを必要とする手続を生成するかを示せ。



**Figure 2.16:** Various binary trees that represent the set  $\{1, 3, 5, 7, 9, 11\}$ .

**Exercise 2.62:** Give a  $\Theta(n)$  implementation of `union-set` for sets represented as ordered lists.

順序付きリストとして表現された集合に対する `union-set` の実装を  $\Theta(n)$  の範囲で行え。

## Sets as binary trees

We can do better than the ordered-list representation by arranging the set elements in the form of a tree. Each node of the tree holds one element of the set, called the “entry” at that node, and a link to each of two other (possibly empty) nodes. The “left” link points to elements smaller than the one at the node, and the “right” link to elements greater than the one at the node. Figure 2.16 shows some trees that represent the set  $\{1, 3, 5, 7, 9, 11\}$ . The same set may be represented by a tree in a number of different ways. The only thing we require for a valid representation is that all elements in the left subtree be smaller than the node entry and that all elements in the right subtree be larger.

集合要素を木の形式にて準備することで順序有りリスト表現よりも良く行なうことができます。木の各ノードはそのノードにおける“エントリ”と呼ばれる集合の1つの要素と他の2つの(空にも有り得る)ノードへのリンクを持ちます。“左”的リンクはそのノードよりも小さな値を差し、“右”的リンクはそのノードの値より大きな値のノードを差します。Figure 2.16は集合  $\{1, 3, 5, 7, 9, 11\}$  を表現するいくつかの木を示しています。

The advantage of the tree representation is this: Suppose we want to check whether a number  $x$  is contained in a set. We begin by comparing  $x$  with the entry in the top node. If  $x$  is less than this, we know that we need only search the left subtree; if  $x$  is greater, we need only search the right subtree. Now, if the tree is “balanced,” each of these subtrees will be about half the size of the original. Thus, in one step we have reduced the problem of searching a tree of size  $n$  to searching a tree of size  $n/2$ . Since the size of the tree is halved at each step, we should expect that the number of steps needed to search a tree of size  $n$  grows as  $\Theta(\log n)$ .<sup>72</sup> For large sets, this will be a significant speedup over the previous representations.

木表現の優位点は次のとおりです。ある数値  $x$  がある集合に含まれているかどうかをチェックしたいと想定します。 $x$  をトップノードのエントリと比較することから始めます。もし  $x$  がこれよりも小さければ、左の部分木のみを探索すれば良いことがわかります。もし  $x$  が大きければ、右の部分木のみを探索する必要があります。ここで、木が“バランスが取れた”状態であるとは各部分木のサイズが元の約半分であるということです。従って一度のステップにおいてサイズ  $n$  の木の探索問題を、サイズ  $n/2$  の木の探索問題に縮小したことになります。各ステップにより木の探索に必要なステップは半分になるのでサイズ  $n$  の木の探索に必要なステップ数は  $\Theta(\log n)$  で増加することが期待されます。<sup>73</sup> 大きな集合に対しては以前の表現に比べこれは著しい高速化になるでしょう。

We can represent trees by using lists. Each node will be a list of three items: the entry at the node, the left subtree, and the right subtree. A left or a right subtree of the empty list will indicate that there is no subtree connected there. We can describe this representation by the following procedures:<sup>74</sup>

木はリストを用いて表現できます。各ノードは 3 つの項目のリストになります。ノードのエントリ、左部分木、右部分木です。左、または右部分木が空

---

<sup>72</sup>Halving the size of the problem at each step is the distinguishing characteristic of logarithmic growth, as we saw with the fast-exponentiation algorithm of Section 1.2.4 and the half-interval search method of Section 1.3.3.

<sup>73</sup>Section 1.2.4の高速指數アルゴリズムやSection 1.2.4の半区間検索手法で学んだように各ステップにて問題のサイズを半分にすることは対数增加の特徴的な性質です。

<sup>74</sup>We are representing sets in terms of trees, and trees in terms of lists—in effect, a data abstraction built upon a data abstraction. We can regard the procedures `entry`, `left-branch`, `right-branch`, and `make-tree` as a way of isolating the abstraction of a “binary tree” from the particular way we might wish to represent such a tree in terms of list structure.

リストの場合はそこに接続された部分木が存在しないことを示します。この表現を以下の手続にて説明できます。<sup>75</sup>

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry left right)
  (list entry left right))
```

Now we can write the `element-of-set?` procedure using the strategy described above:

これで `element-of-set?` 手続を上で説明された戦略を用いて書くことができます。

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set))
         (element-of-set? x (left-branch set)))
        ((> x (entry set))
         (element-of-set? x (right-branch set)))))
```

Adjoining an item to a set is implemented similarly and also requires  $\Theta(\log n)$  steps. To adjoin an item  $x$ , we compare  $x$  with the node entry to determine whether  $x$  should be added to the right or to the left branch, and having adjoined  $x$  to the appropriate branch we piece this newly constructed branch together with the original entry and the other branch. If  $x$  is equal to the entry, we just return the node. If we are asked to adjoin  $x$  to an empty tree, we generate a tree that has  $x$  as the entry and empty right and left branches. Here is the procedure:

集合に項目を付加することは同様に実装され、そしてまた  $\Theta(\log n)$  ステップを必要とします。項目  $x$  を付加するためには、 $x$  をノードのエントリと比較し  $x$  が右か左のどちらの枝に追加されるべきを判断し、 $x$  を適切な枝に追加し、この新しく構築された枝を元のエントリともう一方の枝と共に接続します。も

---

<sup>75</sup>私達は集合を木を用いて表現しており、そして木はリストを用いています—事実上、データ抽象化がデータ抽象化の上に構築されています。手続 `entry`, `left-branch`, `right-branch`, `make-tree` を私達がそのような木をリスト構造を用いて表現することを望んだ特定の方法から“二分木”的抽象化を分離する方法として見做すことができます。

し  $x$  を空の木に付加するよう求められたらエントリに  $x$ を持ち、右と左の枝は空である木を生成します。以下がこの手続です。

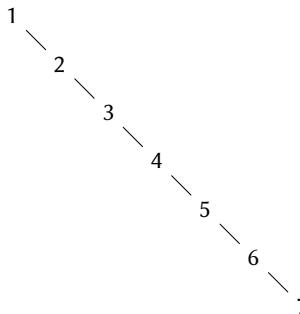
```
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
                    (right-branch set)))
        ((> x (entry set))
         (make-tree (entry set) (left-branch set)
                    (adjoin-set x (right-branch set)))))))
```

The above claim that searching the tree can be performed in a logarithmic number of steps rests on the assumption that the tree is “balanced,” i.e., that the left and the right subtree of every tree have approximately the same number of elements, so that each subtree contains about half the elements of its parent. But how can we be certain that the trees we construct will be balanced? Even if we start with a balanced tree, adding elements with `adjoin-set` may produce an unbalanced result. Since the position of a newly adjoined element depends on how the element compares with the items already in the set, we can expect that if we add elements “randomly” the tree will tend to be balanced on the average. But this is not a guarantee. For example, if we start with an empty set and adjoin the numbers 1 through 7 in sequence we end up with the highly unbalanced tree shown in Figure 2.17. In this tree all the left subtrees are empty, so it has no advantage over a simple ordered list. One way to solve this problem is to define an operation that transforms an arbitrary tree into a balanced tree with the same elements. Then we can perform this transformation after every few `adjoin-set` operations to keep our set in balance. There are also other ways to solve this problem, most of which involve designing new data structures for which searching and insertion both can be done in  $\Theta(\log n)$  steps.<sup>76</sup>

上の木の検索は対数ステップで実行可能であるとの主張は木は“バランスが取れている”という前提に依存しています。すなわち、全ての木の左と右の部分木は大体同じ要素の数を持っているため、各部分木はその親の約半分の要

---

<sup>76</sup> Examples of such structures include *B-trees* and *red-black trees*. There is a large literature on data structures devoted to this problem. See Cormen et al. 1990.



**Figure 2.17:** Unbalanced tree produced by adjoining 1 through 7 in sequence.

素持つてことになります。しかしどのようにすれば私達が構築した木がバランスが取れないと確信することができるのでしょうか。例えもしバランスの取れた木で開始したとしても、`adjoin-set` にて要素を足していくばバランスが取れていらない結果を生み出します。新しく付加される要素の位置は集合に既に存在する項目とどのように比較されるかに依存するために、もし要素を“ランダム”に追加すればその木が平均ではバランスが取れることが予想できます。しかしこれは保証されません。例えばもし空集合から始めて数値を 1 から 7 まで順番に追加していくば Figure 2.17 で示されるとてもアンバランスな木になってしまいます。この木では全ての左の部分木は空であり、単純な順序有りリストに対する優位点が存在しません。この問題を解く 1 つの方法として任意の木をバランスの取れた木に同じ要素を用いて変換する操作を定義することが上げられます。そうすれば数回毎の `adjoin-set` の後にこの変換を実行することで集合のバランスを保つことができます。この問題を解く他の方法もまた存在しますが、その多くは検索と挿入の両方が  $\Theta(\log n)$  ステップで行える新しいデータ構造を設計することを含みます。<sup>77</sup>

**Exercise 2.63:** Each of the following two procedures converts a binary tree to a list.

以下の 2 つの手続はそれぞれ二分木をリストに変換する。

---

<sup>77</sup> そのような構造の例には *B-trees* や *red-black trees* があります。この問題にささげられた巨大な文献が存在します。Cormen et al. 1990 を参照して下さい。

```

(define (tree->list-1 tree)
  (if (null? tree)
      '()
      (append (tree->list-1 (left-branch tree))
              (cons (entry tree)
                     (tree->list-1
                      (right-branch tree)))))))
(define (tree->list-2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                     (cons (entry tree)
                            (copy-to-list
                             (right-branch tree)
                             result-list))))))
  (copy-to-list tree '()))

```

a Do the two procedures produce the same result for every tree? If not, how do the results differ? What lists do the two procedures produce for the trees in Figure 2.16?

2つの手続は全ての木に対して同じ結果を生成するか? もしそうでなければどのように結果は異なるか? Figure 2.16の木に対して2つの手続はどんなリストを生成するか?

b Do the two procedures have the same order of growth in the number of steps required to convert a balanced tree with  $n$  elements to a list? If not, which one grows more slowly?

2つの手続は  $n$  要素のバランスの取れた木をリストに変換するのに同じステップ数増加のオーダであるか? もしそうでなければどちらがより遅く増加するか?

**Exercise 2.64:** The following procedure `list->tree` converts an ordered list to a balanced binary tree. The helper procedure `partial-tree` takes as arguments an integer  $n$  and list of at least  $n$  elements and constructs a balanced tree containing the first  $n$  elements of the list. The result returned by `partial-tree` is a pair (formed

with `cons`) whose `car` is the constructed tree and whose `cdr` is the list of elements not included in the tree.

以下の手続 `list->tree` は順序有りリストをバランスの取れた木に変換する。ヘルパ手続 `partial-tree` は引数として整数  $n$  と少なくとも  $n$  要素のリストを取り、リストの最初の  $n$  要素を含むバランスの取れた木を生成する。`partial-tree` の結果として返されるのは (`cons` で構築された) ペアであり、`car` が構築された木で `cdr` が木に含まれなかった要素のリストである、

```
(define (list->tree elements)
  (car (partial-tree elements (length elements))))
(define (partial-tree elts n)
  (if (= n 0)
      (cons '() elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result
              (partial-tree elts left-size)))
          (let ((left-tree (car left-result))
                (non-left-elts (cdr left-result))
                (right-size (- n (+ left-size 1)))))
            (let ((this-entry (car non-left-elts))
                  (right-result
                    (partial-tree
                      (cdr non-left-elts)
                      right-size)))
              (let ((right-tree (car right-result))
                    (remaining-elts
                      (cdr right-result)))
                (cons (make-tree this-entry
                                  left-tree
                                  right-tree)
                      remaining-elts))))))))
```

- a Write a short paragraph explaining as clearly as you can how `partial-tree` works. Draw the tree produced by `list->tree` for the list `(1 3 5 7 9 11)`.

できるだけ明確に `partial-tree` がどのように働くのか文章で答えよ。リスト (1 3 5 7 9 11) に対し `list->tree` により生成される木を描け。

- b What is the order of growth in the number of steps required by `list->tree` to convert a list of  $n$  elements?

`list->tree` が  $n$  要素のリストを変換するのに必要とされるステップ数の増加量オーダーはいくらか？

**Exercise 2.65:** Use the results of Exercise 2.63 and Exercise 2.64 to give  $\Theta(n)$  implementations of `union-set` and `intersection-set` for sets implemented as (balanced) binary trees.<sup>78</sup>

Exercise 2.63とExercise 2.64の結果を用いて(バランスの取れた)二分木として実装された集合の  $\Theta(n)$  における実装を与えよ。<sup>79</sup>

## Sets and information retrieval

We have examined options for using lists to represent sets and have seen how the choice of representation for a data object can have a large impact on the performance of the programs that use the data. Another reason for concentrating on sets is that the techniques discussed here appear again and again in applications involving information retrieval.

リストを集合を表現するのに使用する選択肢について調べ、データオブジェクトに対する表現の選択がどのようにそのデータを使用するプログラムのパフォーマンスに大きな影響を与えるかについて学びました。集合に専念するもう1つの理由としてここで議論されたテクニックが情報検索を含むアプリケーションにおいて何度も何度も現われることが上げられます。

Consider a data base containing a large number of individual records, such as the personnel files for a company or the transactions in an accounting system. A typical data-management system spends a large amount of time accessing or modifying the data in the records and therefore requires an efficient method for accessing records. This is done by identifying a part of each record to serve as an identifying *key*. A key can be anything that uniquely identifies the record. For a personnel file, it might be an employee's ID number. For an accounting system, it might be a transaction number. Whatever the key is, when we define

---

<sup>78</sup>Exercise 2.63 through Exercise 2.65 are due to Paul Hilfinger.

<sup>79</sup>Exercise 2.63からExercise 2.65はPaul Hilfingerによるものである。

the record as a data structure we should include a `key` selector procedure that retrieves the key associated with a given record.

企業が持つ個人情報や会計システムの取引等、大量の個人レコードを持つデータベースについて考えてみて下さい。典型的なデータ管理システムはレコードの中のデータへのアクセスや変更に多大な時間を過ごします。従ってレコードにアクセスする効率的な手法を必要とします。これは各レコードの一部に識別子である`key`(キー)としての役割を果たさせることで行われます。キーはレコードを一意に識別する任意の物でかまいません。個人情報に対しては従業員番号であったりします。会計システムにおいては取引番号であったりします。キーが何であれレコードをデータ構造として定義する時、与えられたレコードに関連するキーを取得する`key`セレクタ手続を含まなければなりません。

Now we represent the data base as a set of records. To locate the record with a given key we use a procedure `lookup`, which takes as arguments a key and a data base and which returns the record that has that key, or false if there is no such record. `Lookup` is implemented in almost the same way as `element-of-set?`. For example, if the set of records is implemented as an unordered list, we could use

さて、データベースをレコードの集合として表現します。与えられたキーでレコードを指し示すためには手続 `lookup` を用い、引数としてキーとデータベースを取り、そのキーを持つレコードを返すか、そのようなレコードが無ければ `false` を返します。例えばもしレコードの集合が順序無しリストで実装されていれば、以下を用いることができます。

```
(define (lookup given-key set-of-records)
  (cond ((null? set-of-records) false)
        ((equal? given-key (key (car set-of-records)))
         (car set-of-records))
        (else (lookup given-key (cdr set-of-records)))))
```

Of course, there are better ways to represent large sets than as unordered lists. Information-retrieval systems in which records have to be “randomly accessed” are typically implemented by a tree-based method, such as the binary-tree representation discussed previously. In designing such a system the methodology of data abstraction can be a great help. The designer can create an initial implementation using a simple, straightforward representation such as unordered lists. This will be unsuitable for the eventual system, but it can be useful in providing a “quick and dirty” data base with which to test the rest of the system.

Later on, the data representation can be modified to be more sophisticated. If the data base is accessed in terms of abstract selectors and constructors, this change in representation will not require any changes to the rest of the system.

もちろん、巨大な集合を表現するのには順序無しリストよりもより良い方法が存在します。レコードが“ランダムアクセス”される情報検索システムは一般的に木をベースにした手法で実装されます。以前に議論された二分木のような物です。そのようなシステムを設計する場合、データ抽象化の方法論はとても大きな手助けになります。設計者は順序無しリストの様な、簡単で直接的な表現を用いて初期実装を作成することができます。これは最終的なシステムには相応しくありません。しかし、“quick and dirty”(迅速だが汚い) データベースを残りのシステムをテストするために提供する目的には便利でしょう。後でデータ表現はより洗練された物に変更することができます。もしデータベースが抽象セレクタとコンストラクタによりアクセスされるのならば、この表現上の変更は残りのシステムに対し何の変更も要求しません。

**Exercise 2.66:** Implement the `lookup` procedure for the case where the set of records is structured as a binary tree, ordered by the numerical values of the keys.

レコードの集合が二分木として構造化され、キーの数値で順序付けられている場合の `lookup` 手続を実装せよ。

### 2.3.5 Example: Huffman Encoding Trees

This section provides practice in the use of list structure and data abstraction to manipulate sets and trees. The application is to methods for representing data as sequences of ones and zeros (bits). For example, the ASCII standard code used to represent text in computers encodes each character as a sequence of seven bits. Using seven bits allows us to distinguish  $2^7$ , or 128, possible different characters. In general, if we want to distinguish  $n$  different symbols, we will need to use  $\log_2 n$  bits per symbol. If all our messages are made up of the eight symbols A, B, C, D, E, F, G, and H, we can choose a code with three bits per character, for example

この節では集合と木を操作するためのリスト構造とデータ抽象化の使用のための練習を提供します。アプリケーションの狙いは1と0の(ビットの)列としてのデータを表現するための手法です。例えばASCII標準コードはコンピュータ内にて各文字を7ビットの列に符号化してテキストを表現するのに利用さ

れます。7ビットを用いることは $2^7$ 、または128の異なる文字を区別することができます。一般的に、もし $n$ 個の異なる記号を区別したい場合、記号当たりに $\log_2 n$ ビットの使用が必要となります。もし全てのメッセージが8つのシンボル、A, B, C, D, E, F, G, Hで作られている場合、一文字当たり3ビットのコードを選択することができます。以下に例を上げます。

A 000	C 010	E 100	G 110
B 001	D 011	F 101	H 111

With this code, the message

この符号を用いて、以下のメッセージは

BACADAEAFABBAAGAH

is encoded as the string of 54 bits

54ビットの列として符号化されます。

001000010000011000100000101000001001000000000110000111

Codes such as ASCII and the A-through-H code above are known as *fixed-length* codes, because they represent each symbol in the message with the same number of bits. It is sometimes advantageous to use *variable-length* codes, in which different symbols may be represented by different numbers of bits. For example, Morse code does not use the same number of dots and dashes for each letter of the alphabet. In particular, E, the most frequent letter, is represented by a single dot. In general, if our messages are such that some symbols appear very frequently and some very rarely, we can encode data more efficiently (i.e., using fewer bits per message) if we assign shorter codes to the frequent symbols. Consider the following alternative code for the letters A through H:

ASCIIや上記のAからHの符号は*fixed-length*(固定長)符号として知られています。それらがメッセージの各記号を同じ数のビットを用いて表現するためです。時には*variable-length*(可変長)符号を使用することが有利な場合もあります。異なるシンボルが異なる数のビットで表現され得るものです。例えばモールス符号はアルファベットの各文字に対して同じ数の点と長音を用いはしません。具体的にはEは最も頻繁に現われる文字ですので单一のドットで表現されます。一般的にはもしメッセージにおいてある記号がとても良く現れ、ある記号はとても稀に現れる場合、短い符号を頻出のシンボルに割り振ることでデータをより効率的に(つまりメッセージ当たりでより少ないビット数で)符号化することができます。

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111

With this code, the same message as above is encoded as the string

この符号では上の同じメッセージが以下の列として符号化されます。

100010100101101100011010100100000111001111

This string contains 42 bits, so it saves more than 20% in space in comparison with the fixed-length code shown above.

この列は 42 ビットですから上で示した固定長符号に比べ記憶域において 20% 以上節約できています。

One of the difficulties of using a variable-length code is knowing when you have reached the end of a symbol in reading a sequence of zeros and ones. Morse code solves this problem by using a special *separator code* (in this case, a pause) after the sequence of dots and dashes for each letter. Another solution is to design the code in such a way that no complete code for any symbol is the beginning (or *prefix*) of the code for another symbol. Such a code is called a *prefix code*. In the example above, A is encoded by 0 and B is encoded by 100, so no other symbol can have a code that begins with 0 or with 100.

可変長符号を用いる上での難点の 1つとして 0 と 1 の列を読んでいる時にいつシンボルの終わりに辿り着いたか知ることが上げられます。モールス符号はこの問題を各文字に対するトンとツーの列の後に特別な *separator code*(分離符号)(この場合には一息置くこと)を用いることで解決しました。他の解法としてはどの任意のシンボルに対する完全な符号も他のシンボルの符号の始め(または *prefix*(接頭辞))ではない様に符号を設計するという物があります。このような符号は *prefix code*(接頭符号)と呼ばれます。上の例では A は 0 で符号化され B は 100 で符号化されるので、他のどのシンボルも 0、または 100 で始まる符号を持つことができません。

In general, we can attain significant savings if we use variable-length prefix codes that take advantage of the relative frequencies of the symbols in the messages to be encoded. One particular scheme for doing this is called the Huffman encoding method, after its discoverer, David Huffman. A Huffman code can be represented as a binary tree whose leaves are the symbols that are encoded. At each non-leaf node of the tree there is a set containing all the symbols in the leaves that lie below the node. In addition, each symbol at a leaf is assigned a weight (which is its relative frequency), and each non-leaf node

contains a weight that is the sum of all the weights of the leaves lying below it. The weights are not used in the encoding or the decoding process. We will see below how they are used to help construct the tree.

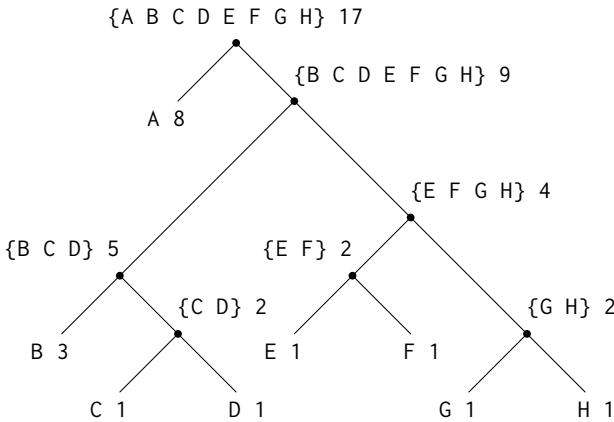
一般的に、もし符号化対象のメッセージ中のシンボルの相対頻度の利点を得られる可変長接頭符号を用いれば著しい儉約を達成することができます。これを行うための 1 つの特定な理論体系としてその発見者 David Huffman に因んでハフマン符号と呼ばれる手法があります。ハフマン符号は葉が符号化された記号である二分木として表現することができます。木の葉でないノードのそれぞれにはそのノードの下に位置する葉の中のシンボル全てを含む集合があります。加えて各葉のシンボルには重み(相対的な頻度)が割り振られており、葉でないノードのそれぞれはその下に位置する葉の重み全ての合計である重みを持っています。重みはエンコード、またはデコード処理では利用されません。以下では重みがどのように木の構築を手助けするかについて学びます。

Figure 2.18 shows the Huffman tree for the A-through-H code given above. The weights at the leaves indicate that the tree was designed for messages in which A appears with relative frequency 8, B with relative frequency 3, and the other letters each with relative frequency 1.

Figure 2.18 は上で与えられた A から H 符号に対するハフマン木を示しています。葉の重みはこの木が A は相対頻度 8、B は相対頻度 3、他の文字は相対頻度 1 で現われるメッセージに対し設計されたことを示しています。

Given a Huffman tree, we can find the encoding of any symbol by starting at the root and moving down until we reach the leaf that holds the symbol. Each time we move down a left branch we add a 0 to the code, and each time we move down a right branch we add a 1. (We decide which branch to follow by testing to see which branch either is the leaf node for the symbol or contains the symbol in its set.) For example, starting from the root of the tree in Figure 2.18, we arrive at the leaf for D by following a right branch, then a left branch, then a right branch, then a right branch; hence, the code for D is 1011.

ハフマン木を与えることで、任意のシンボルの符号を木の根から始めてそのシンボルを持つ葉に辿り着くまで降りていくことで見つけることができます。左の枝に降りる度に符号に 0 を追加し、右の枝に降りる度に 1 を追加します。(どの枝を降りるかはどの枝がそのシンボルに対する葉を含むか、つまりその集合にシンボルを含むかをテストして判断することで決定します)。例えば Figure 2.18 の木の根から始めて D に対する葉に辿り付くには右の枝を選択し、次に左の枝、次に右の枝、次に右の枝を辿ります。従って D に対する符号は 1011 になります。



**Figure 2.18:** A Huffman encoding tree.

To decode a bit sequence using a Huffman tree, we begin at the root and use the successive zeros and ones of the bit sequence to determine whether to move down the left or the right branch. Each time we come to a leaf, we have generated a new symbol in the message, at which point we start over from the root of the tree to find the next symbol. For example, suppose we are given the tree above and the sequence 10001010. Starting at the root, we move down the right branch, (since the first bit of the string is 1), then down the left branch (since the second bit is 0), then down the left branch (since the third bit is also 0). This brings us to the leaf for B, so the first symbol of the decoded message is B. Now we start again at the root, and we make a left move because the next bit in the string is 0. This brings us to the leaf for A. Then we start again at the root with the rest of the string 1010, so we move right, left, right, left and reach C. Thus, the entire message is BAC.

ビット列をハフマン木を用いて複合するには、根から始めてビット列の一連の0と1を用いて左か右の枝を下りるのか決定します。葉に着く度に、メッセージの新しいシンボルを生成し、その時点で木の根から再開し次のシンボルを見つけます。例えば上記の木と列10001010を与えられたとします。根から始めて右の枝へと下ります。(列の最初のビットが1だからです)。次に左の枝を下ります。(2つ目のビットが0だからです)。次に左の枝を下ります。(3つ

目のビットもまた 0 だからです)。この様にして B に対する葉に辿り着くので複合されたメッセージの最初のシンボルは B です。ここでまた根から再開し、次のビットが 0 なので左に移動します。これにより A の葉に辿り着きます。そしてまた根から残りの列 1010 と共に再開し、右、左、右左と動き C に辿り着きます。従つてメッセージ全体は BAC です。

## Generating Huffman trees

Given an “alphabet” of symbols and their relative frequencies, how do we construct the “best” code? (In other words, which tree will encode messages with the fewest bits?) Huffman gave an algorithm for doing this and showed that the resulting code is indeed the best variable-length code for messages where the relative frequency of the symbols matches the frequencies with which the code was constructed. We will not prove this optimality of Huffman codes here, but we will show how Huffman trees are constructed.<sup>80</sup>

シンボルの“アルファベット”とそれらに対応する頻度を与えられた時、どのように“最高の”符号を構築できるでしょうか? (言い替えれば、どの木がメッセージを最も少ないビット数で符号化するでしょうか?)。ハフマンはこれを行うアルゴリズムを与え、結果の符号が相対的なシンボルの頻度と符号が構築された時の頻度が合致した場合に、実際にメッセージに対する最良の可変長符号であることを示しました。このハフマン符号の最適性についてはここでは証明しません。しかしハフマン木がどのように構築されるかについては示します。<sup>81</sup>

The algorithm for generating a Huffman tree is very simple. The idea is to arrange the tree so that the symbols with the lowest frequency appear farthest away from the root. Begin with the set of leaf nodes, containing symbols and their frequencies, as determined by the initial data from which the code is to be constructed. Now find two leaves with the lowest weights and merge them to produce a node that has these two nodes as its left and right branches. The weight of the new node is the sum of the two weights. Remove the two leaves from the original set and replace them by this new node. Now continue this process. At each step, merge two nodes with the smallest weights, removing

---

<sup>80</sup>See Hamming 1980 for a discussion of the mathematical properties of Huffman codes.

<sup>81</sup>ハフマン符号の数学上の特性についての議論に対しては Hamming 1980 を参照して下さい。

them from the set and replacing them with a node that has these two as its left and right branches. The process stops when there is only one node left, which is the root of the entire tree. Here is how the Huffman tree of Figure 2.18 was generated:

ハフマン木を生成するためのアルゴリズムはとても簡単です。その考えは木を再配置することで最も低い頻度のシンボルが根から最も遠く現れるようになります。シンボルと頻度を含む葉のノードの集合と共に符号が構築される初期データにより決定されるに従はずみが最低の2つの葉を見つけ、それらをマージしてこの2つのノードを左と右の枝を持つノードを生成します。新しいノードの重みは2つの重みの和です。元の集合から2つの葉を削除しそれらをこの新しいノードで置き換えます。この処理を続けます。各ステップにて最も小さな重みを持つ2つのノードをマージし、集合から削除し、これらの2つを左と右の枝を持つノードで置き換えします。処理は1つのノードのみが残った時に停止し、それが木全体の根になります。以下にFigure 2.18のハフマン木がどのように生成されるかを示します。

```
Initial
leaves  {(A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)}
Merge   {(A 8) (B 3) ({C D} 2) (E 1) (F 1) (G 1) (H 1)}
Merge   {(A 8) (B 3) ({C D} 2) ({E F} 2) (G 1) (H 1)}
Merge   {(A 8) (B 3) ({C D} 2) ({E F} 2) ({G H} 2)}
Merge   {(A 8) (B 3) ({C D} 2) ({E F G H} 4)}
Merge   {(A 8) ({B C D} 5) ({E F G H} 4)}
Merge   {(A 8) ({B C D E F G H} 9)}
Final   {({A B C D E F G H} 17)}
merge
```

The algorithm does not always specify a unique tree, because there may not be unique smallest-weight nodes at each step. Also, the choice of the order in which the two nodes are merged (i.e., which will be the right branch and which will be the left branch) is arbitrary.

このアルゴリズムは常に同じ木を特定しません。各ステップにおいて重みが最小のノードのペアが一意となるとは限らないためです。またどの2つのノードがマージされるかの順も決定しません。(つまりどれが右になり、どれが左になるのかはわかりません)。

## Representing Huffman trees

In the exercises below we will work with a system that uses Huffman trees to encode and decode messages and generates Huffman trees according to the algorithm outlined above. We will begin by discussing how trees are represented.

以下ではハフマン木を用いてメッセージの符号化・複合化を行い、かつ上で概説したアルゴリズムに基づきハフマン木を作成します。木はどのように表現されるかの議論から始めます。

Leaves of the tree are represented by a list consisting of the symbol `leaf`, the symbol at the leaf, and the weight:

木の葉はシンボル `leaf`、葉に対するシンボル、そして重みから構成されるリストにて表現されます。

```
(define (make-leaf symbol weight)
  (list 'leaf symbol weight))
(define (leaf? object)
  (eq? (car object) 'leaf))
(define (symbol-leaf x) (cadr x))
(define (weight-leaf x) (caddr x))
```

A general tree will be a list of a left branch, a right branch, a set of symbols, and a weight. The set of symbols will be simply a list of the symbols, rather than some more sophisticated set representation. When we make a tree by merging two nodes, we obtain the weight of the tree as the sum of the weights of the nodes, and the set of symbols as the union of the sets of symbols for the nodes. Since our symbol sets are represented as lists, we can form the union by using the `append` procedure we defined in [Section 2.2.1](#):

一般的な木は左側の枝、右側の枝、シンボルの集合、そして重みのリストになります。シンボルの集合は単純にシンボルのリストとなり、より洗練された集合の表現を用いはしません。2つのノードをマージして木を作る時、木の重みを各ノードの重みの和として取得し、シンボルの集合は各ノードのシンボルの集合の和集合とします。シンボルの集合はリストとして表現されていますので和集合は[Section 2.2.1](#)で定義した `append` 手続を用いて形成することができます。

```
(define (make-code-tree left right)
  (list left
        right))
```

```
(append (symbols left)
        (symbols right))
(+ (weight left)
  (weight right))))
```

If we make a tree in this way, we have the following selectors:

もしこのように木を作るのなら以下のようなセレクタを持つことになります。

```
(define (left-branch tree) (car tree))
(define (right-branch tree) (cadr tree))

(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))
(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (caddr tree)))
```

The procedures `symbols` and `weight` must do something slightly different depending on whether they are called with a leaf or a general tree. These are simple examples of *generic procedures* (procedures that can handle more than one kind of data), which we will have much more to say about in [Section 2.4](#) and [Section 2.5](#).

手続 `symbols` と `weight` はそれらが葉と一般的な木のどちらと共に呼ばれたかにより少々異なったことをしなければなりません。これらは*generic procedures* (ジェネリック手続：二種類以上のデータを扱える手続) の簡単なサンプルであり、[Section 2.4](#)と [Section 2.5](#)にてより多くのことについて述べます。

## The decoding procedure

The following procedure implements the decoding algorithm. It takes as arguments a list of zeros and ones, together with a Huffman tree.

以下の手続は複合化アルゴリズムを実装します。0と1のリストをハフマン木と共に引数として取ります。

```

(define (decode bits tree)
  (define (decode-1 bits current-branch)
    (if (null? bits)
        '()
        (let ((next-branch
              (choose-branch
                (car bits) current-branch)))
          (if (leaf? next-branch)
              (cons (symbol-leaf next-branch)
                    (decode-1 (cdr bits) tree))
              (decode-1 (cdr bits) next-branch))))))
  (decode-1 bits tree))

(define (choose-branch bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))
        (else (error "bad bit: CHOOSE-BRANCH" bit))))

```

The procedure `decode-1` takes two arguments: the list of remaining bits and the current position in the tree. It keeps moving “down” the tree, choosing a left or a right branch according to whether the next bit in the list is a zero or a one. (This is done with the procedure `choose-branch`.) When it reaches a leaf, it returns the symbol at that leaf as the next symbol in the message by `consing` it onto the result of decoding the rest of the message, starting at the root of the tree. Note the error check in the final clause of `choose-branch`, which complains if the procedure finds something other than a zero or a one in the input data.

手続 `decode-1` は 2 つの引数を取ります。残りのビット列のリストと木における現在の位置です。木を“下り”続けるためリストの中の次のビットが 0 であるか 1 であるかに従って左、または右の枝を選択します。(これは手続 `choose-branch` と共に行われます)。葉に辿りついた時、その葉のシンボルをメッセージの次のシンボルとして、残りのメッセージを木の根から再開して複合した結果の頭に `cons` することで返します。`choose-branch` の最終条項のエラーチェックに注目して下さい。もし手続が 0 または 1 以外の物を入力データに見つけた場合にエラーを発します。

## Sets of weighted elements

In our representation of trees, each non-leaf node contains a set of symbols, which we have represented as a simple list. However, the tree-generating algorithm discussed above requires that we also work with sets of leaves and trees, successively merging the two smallest items. Since we will be required to repeatedly find the smallest item in a set, it is convenient to use an ordered representation for this kind of set.

私達の木の表現において、各葉ではないノードは簡単にリストとして表現したシンボルの集合を持ちます。しかし上で議論した木の生成アルゴリズムはまた葉と木の集合に対しても働き、2つの最小の項目のマージを続けなければなりません。集合の中の最も小さな項目を繰り返し見つけなければなりませんから、このような集合に対しては順序有りの表現を用いると便利です。

We will represent a set of leaves and trees as a list of elements, arranged in increasing order of weight. The following `adjoin-set` procedure for constructing sets is similar to the one described in [Exercise 2.61](#); however, items are compared by their weights, and the element being added to the set is never already in it.

葉と木の集合を重みの昇順で配置した要素のリストとして表現することにします。以下の集合を構築するための `adjoin-set` 手続は [Exercise 2.61](#) にて説明したものと似ています。しかし項目はその重みにて比較され、集合に追加される要素はその中に既に存在はしないとします。

```
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set)))
         (cons x set))
        (else (cons (car set)
                     (adjoin-set x (cdr set)))))))
```

The following procedure takes a list of symbol-frequency pairs such as ((A 4) (B 2) (C 1) (D 1)) and constructs an initial ordered set of leaves, ready to be merged according to the Huffman algorithm:

以下の手続は ((A 4) (B 2) (C 1) (D 1)) の様なシンボルと頻度のペアのリストを取り、葉の初期順序有り集合を構築し、ハフマンアルゴリズムに従いマージできるよう準備します。

```
(define (make-leaf-set pairs)
```

```
(if (null? pairs)
  '()
  (let ((pair (car pairs)))
    (adjoin-set (make-leaf (car pair)      ; symbol
                           (cadr pair))   ; frequency
                (make-leaf-set (cdr pairs)))))))
```

**Exercise 2.67:** Define an encoding tree and a sample message:

符号化木とサンプルのメッセージを定義する。

```
(define sample-tree
  (make-code-tree (make-leaf 'A 4)
                 (make-code-tree
                   (make-leaf 'B 2)
                   (make-code-tree (make-leaf 'D 1)
                                 (make-leaf 'C 1)))))
(define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))
```

Use the `decode` procedure to decode the message, and give the result.

`decode` 手続を用いてメッセージを複合し結果を与えるよ。

**Exercise 2.68:** The `encode` procedure takes as arguments a message and a tree and produces the list of bits that gives the encoded message.

`encode` 手続は引数としてメッセージと木を取り、符号化されメッセージを与えるビットのリストを生成する。

```
(define (encode message tree)
  (if (null? message)
    '()
    (append (encode-symbol (car message) tree)
            (encode (cdr message) tree))))
```

`Encode-symbol` is a procedure, which you must write, that returns the list of bits that encodes a given symbol according to a given tree. You should design `encode-symbol` so that it signals an error if the symbol is not in the tree at all. Test your procedure by encoding

the result you obtained in [Exercise 2.67](#) with the sample tree and seeing whether it is the same as the original sample message.

`encode-symbol` はあなたが書かねばならぬ手続である。与えられた木に従って与えられたシンボルの符号化を行いビットのリストを返す。`encode-symbol` をもしシンボルがその木に存在しない場合にエラーを発するように設計せよ。あなたの手続を [Exercise 2.67](#) で得られた結果とサンプルの木で符号化することでテストし、元のサンプルメッセージと同じであるか確認せよ。

**Exercise 2.69:** The following procedure takes as its argument a list of symbol-frequency pairs (where no symbol appears in more than one pair) and generates a Huffman encoding tree according to the Huffman algorithm.

以下の手続はその引数としてシンボルと頻度のペアのリストを取り(どのシンボルも1つより多くのペアには存在しない)、ハフマンアルゴリズムに従いハフマン符号化木を生成する。

```
(define (generate-huffman-tree pairs)
  (successive-merge (make-leaf-set pairs)))
```

`Make-leaf-set` is the procedure given above that transforms the list of pairs into an ordered set of leaves. `Successive-merge` is the procedure you must write, using `make-code-tree` to successively merge the smallest-weight elements of the set until there is only one element left, which is the desired Huffman tree. (This procedure is slightly tricky, but not really complicated. If you find yourself designing a complex procedure, then you are almost certainly doing something wrong. You can take significant advantage of the fact that we are using an ordered set representation.)

`make-leaf-set` は上で与えられた手続でペアのリストを葉の順序有り集合へと変換する。`successive-merge` はあなたが書かなければならぬ手続である。`make-code-tree` を用いて集合の重みが最小の要素を残り要素が1つになるまで繰り返しマージせよ。残った一要素こそが望まれたハフマン木である。(この手続は少しトリッキーであるがそんなに複雑ではない。もしあなたが自分の設計が複雑だと思うなら、ほとんど確実に何かを間違えている。順序有りの集合表現を用いているという事実から著しい利点を得ることができる。)

**Exercise 2.70:** The following eight-symbol alphabet with associated relative frequencies was designed to efficiently encode the lyrics of 1950s rock songs. (Note that the “symbols” of an “alphabet” need not be individual letters.)

以下の 8 つのシンボルによるアルファベット(文字体系)と関連する頻度は効率的に 1950 年代のロックの歌を符号化するために設計された。(“アルファベット”的“シンボル”は単体の文字である必要は無いことに注意せよ。)

A	2	GET	2	SHA	3	WAH	1
BOOM	1	JOB	2	NA	16	YIP	9

Use `generate-huffman-tree` (Exercise 2.69) to generate a corresponding Huffman tree, and use `encode` (Exercise 2.68) to encode the following message:

`generate-huffman-tree`(Exercise 2.69参照)を用いて対応するハフマン木を生成し、`encode`(Exercise 2.68参照)を用いて以下のメッセージを符号化せよ。

Get a job  
Sha na na na na na na na  
Get a job  
Sha na na na na na na na  
Wah yip yip yip yip yip yip yip yip  
Sha boom

How many bits are required for the encoding? What is the smallest number of bits that would be needed to encode this song if we used a fixed-length code for the eight-symbol alphabet?

符号化には何ビットが必要であるか? この歌をもし固定長符号を 8 つシンボルによるアルファベットに用いた時、最小で何ビットが必要であるか?

**Exercise 2.71:** Suppose we have a Huffman tree for an alphabet of  $n$  symbols, and that the relative frequencies of the symbols are  $1, 2, 4, \dots, 2^{n-1}$ . Sketch the tree for  $n = 5$ ; for  $n = 10$ . In such a tree (for general  $n$ ) how many bits are required to encode the most frequent symbol? The least frequent symbol?

$n$  個のシンボルのアルファベットに対するハフマン木を持っているとする。各シンボルに関連する頻度は  $1, 2, 4, \dots, 2^{n-1}$  だとしよう。 $n = 5, n = 10$  の場合の木をスケッチせよ。そのような（任意の  $n$  に対する）木において、最も頻度の高いシンボルを符号化するには何ビットが必要であるか？最も頻度の低いシンボルに対してはいくらか？

**Exercise 2.72:** Consider the encoding procedure that you designed in [Exercise 2.68](#). What is the order of growth in the number of steps needed to encode a symbol? Be sure to include the number of steps needed to search the symbol list at each node encountered. To answer this question in general is difficult. Consider the special case where the relative frequencies of the  $n$  symbols are as described in [Exercise 2.71](#), and give the order of growth (as a function of  $n$ ) of the number of steps needed to encode the most frequent and least frequent symbols in the alphabet.

あなたが[Exercise 2.68](#)にて設計した符号化手続について考える。シンボルを符号化するのに必要なステップ数の増加のオーダーはいくらか？各ノードに辿りついた時にシンボルリストを検索するのに必要なステップ数を含めることを確認するように。この質問的一般的な解答は難しい。 $n$  個のシンボルの相対頻度が[Exercise 2.71](#)で説明された特別な場合について考えよ。そしてアルファベットにおける最大頻度と最小頻度のシンボルを符号化するのに必要なステップ数の ( $n$  の関数としての) 増加のオーダーを与えよ。

## 2.4 Multiple Representations for Abstract Data

We have introduced data abstraction, a methodology for structuring systems in such a way that much of a program can be specified independent of the choices involved in implementing the data objects that the program manipulates. For example, we saw in [Section 2.1.1](#) how to separate the task of designing a program that uses rational numbers from the task of implementing rational numbers in terms of the computer language's primitive mechanisms for constructing compound data. The key idea was to erect an abstraction barrier – in this case, the selectors and constructors for rational numbers (`make-rat`,

`numer, denom`)—that isolates the way rational numbers are used from their underlying representation in terms of list structure. A similar abstraction barrier isolates the details of the procedures that perform rational arithmetic (`add-rat`, `sub-rat`, `mul-rat`, and `div-rat`) from the “higher-level” procedures that use rational numbers. The resulting program has the structure shown in Figure 2.1.

ここまでデータ抽象化について紹介してきました。これはプログラムが操作するデータオブジェクトの実装における選択に独立して多くのプログラムが指定され得る様な方法でシステムを構造化するメソドロジ(方法論)でした。例えばSection 2.1.1にてどのように分数を用いるプログラムの設計タスクを、複雑なデータを構築するためのコンピュータ言語のプリミティブなメカニズムを用いて分数を実装するタスクから分離するのかについて学びました。鍵となる考えは抽象化バリアを建てる—分数の使用法をそれらの根底にあるリスト構造を用いた表現から分離することでした。同様の抽象化バリアは分数演算(`add-rat`, `sub-rat`, `mul-rat`, `div-rat`)を実行する手続の詳細を分数を用いる“より高いレベル”的手續から分離します。結果のプログラムはFigure 2.1 で示される構造を持ちます。

These data-abstraction barriers are powerful tools for controlling complexity. By isolating the underlying representations of data objects, we can divide the task of designing a large program into smaller tasks that can be performed separately. But this kind of data abstraction is not yet powerful enough, because it may not always make sense to speak of “the underlying representation” for a data object.

これらのデータ抽象化バリアは複雑さをコントロールする強力なツールです。基礎をなすデータオブジェクトの表現を分離することで、大きなプログラムを設計するタスクを小さなタスクに分割し別々に実行することを可能にします。しかしこの種のデータ抽象化はまだ十分に強力ではありません。データオブジェクトに対して“基礎をなす表現”について話すことが常に意味があるとは限らないためです。

For one thing, there might be more than one useful representation for a data object, and we might like to design systems that can deal with multiple representations. To take a simple example, complex numbers may be represented in two almost equivalent ways: in rectangular form (real and imaginary parts) and in polar form (magnitude and angle). Sometimes rectangular form is more appropriate and sometimes polar form is more appropriate. Indeed, it is perfectly plausible to imagine a system in which complex numbers are rep-

resented in both ways, and in which the procedures for manipulating complex numbers work with either representation.

一例として、あるデータオブジェクトに対しては便利な表現が1つよりも多くあるかもしれません。そして私達はシステムを複数の表現を扱えるように設計したいと願うでしょう。簡単な例を得るために、複素数が2つのほとんど同様な方法で表現されるとしましょう。(実数と虚数から成る)直行形式と(大きさと角度から成る)極形式です。ある時は直行形式がより適切で、ある時には極形式がより適切になります。実際に複素数が両者の方法にて表現され、複素数を操作する手続がどちらの表現でも働くことができるシステムを考えることは完全に適切であろうと思われます。

More importantly, programming systems are often designed by many people working over extended periods of time, subject to requirements that change over time. In such an environment, it is simply not possible for everyone to agree in advance on choices of data representation. So in addition to the data-abstraction barriers that isolate representation from use, we need abstraction barriers that isolate different design choices from each other and permit different choices to coexist in a single program. Furthermore, since large programs are often created by combining pre-existing modules that were designed in isolation, we need conventions that permit programmers to incorporate modules into larger systems *additively*, that is, without having to redesign or reimplement these modules.

より重要なこととして、プログラミングシステムは時折、長期間に渡り多くの人々が働くことにより設計されるため、長い間仕様変更にさらされます。そのような環境では、データ表現の選択を前もって同意することが誰にとっても可能ということは単純に有り得ません。そのため使用から表現を分離するデータ抽象化バリアに加えて、お互いから異なる設計の選択を分離し、单一のプログラム内にて異なる選択の共存を許す抽象化バリアを必要とします。さらに、巨大なプログラムは時折、以前から存在する独立して設計されたモジュールを組み合わせることで作成されるため、プログラマにモジュールを *additively*(付加的)に組み合わせて巨大なシステムにすることを許可するための約束事が必要です。それはつまりこれらのモジュールを再設計、または再実装する必要が無いようにするためです。

In this section, we will learn how to cope with data that may be represented in different ways by different parts of a program. This requires constructing *generic procedures*—procedures that can operate on data that may be represented in more than one way. Our main technique for building generic

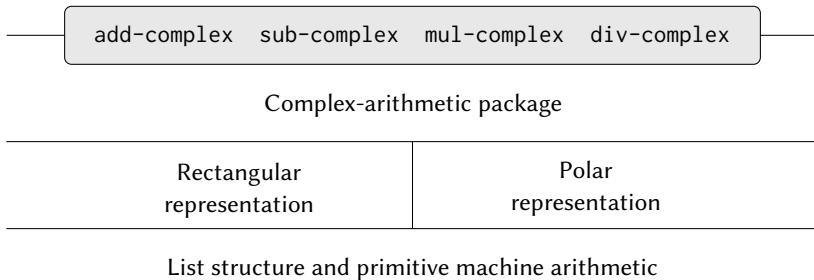
procedures will be to work in terms of data objects that have *type tags*, that is, data objects that include explicit information about how they are to be processed. We will also discuss *data-directed* programming, a powerful and convenient implementation strategy for additively assembling systems with generic operations.

この節ではプログラムの異なるパートにより異なる方法で表現され得るデータをどのように処理するかについて学びます。これは*generic procedures*(ジェネリック手続)——二種類以上の方法で表現され得るデータを処理可能な手続の構築が必要となります。ジェネリック手続を構築する主なテクニックは*type tags*(タイプタグ)を持つデータオブジェクトを利用して処理することになります。それはつまりデータオブジェクト自身がどのように処理されるべきであるかについての情報を明示的に含むことです。また*data-directed*(データ適従) プログラミングについても議論します。これは強力、かつ便利なジェネリック命令を用いて付加的にシステムを組み立てる実装戦略です。

We begin with the simple complex-number example. We will see how type tags and data-directed style enable us to design separate rectangular and polar representations for complex numbers while maintaining the notion of an abstract “complex-number” data object. We will accomplish this by defining arithmetic procedures for complex numbers (`add-complex`, `sub-complex`, `mul-complex`, and `div-complex`) in terms of generic selectors that access parts of a complex number independent of how the number is represented. The resulting complex-number system, as shown in Figure 2.19, contains two different kinds of abstraction barriers. The “horizontal” abstraction barriers play the same role as the ones in Figure 2.1. They isolate “higher-level” operations from “lower-level” representations. In addition, there is a “vertical” barrier that gives us the ability to separately design and install alternative representations.

簡単な複素数の例から始めます。どのようにタイプタグとデータに従うスタイルが抽象的な“複素数”データオブジェクトの概念を維持しながら複素数の直交形式と極形式の表現を分けて設計を行うことを可能にしているかについて学びます。私達はこれを、複素数に対する数値演算手続(`add-complex`, `sub-complex`, `mul-complex`, `div-complex`)を、複素数がどのように表現されるのかから独立して複素数の部分にアクセスするジェネリックなセレクタを用いて定義することで達成します。結果としての複素数システムはFigure 2.19に示されるとおり、2つの異なる種類の抽象化バリアを含みます。“水平方向”的抽象化バリアはFigure 2.1と同じ役割を演じます。それらは“高レベル”的命令を“低レベル”的表現から分離します。それに加えて“垂直方向”的バリアが存在し、

## Programs that use complex numbers



**Figure 2.19:** Data-abstraction barriers in the complex-number system.

代替的な表現を分離して設計しインストールする能力を与えます。

In Section 2.5 we will show how to use type tags and data-directed style to develop a generic arithmetic package. This provides procedures (`add`, `mul`, and so on) that can be used to manipulate all sorts of “numbers” and can be easily extended when a new kind of number is needed. In Section 2.5.3, we’ll show how to use generic arithmetic in a system that performs symbolic algebra.

Section 2.5にてどのようにタイプタグとデータ適従スタイルを用いてジェネリックな数値演算パッケージを開発するかについて示します。これは全ての種類の“数値”を操作するのに用いることができる手続 (`add`, `mul`, その他) を提供します。Section 2.5.3ではどのようにして記号代数を実行するジェネリックな数値演算をシステム内に用いるかについて示します。

### 2.4.1 Representations for Complex Numbers

We will develop a system that performs arithmetic operations on complex numbers as a simple but unrealistic example of a program that uses generic operations. We begin by discussing two plausible representations for complex numbers as ordered pairs: rectangular form (real part and imaginary part)

and polar form (magnitude and angle).<sup>82</sup> Section 2.4.2 will show how both representations can be made to coexist in a single system through the use of type tags and generic operations.

ジェネリックな命令を用いる単純な代わりに非現実的なプログラムの例として複素数上で数値演算命令を実行するシステムを開発します。順序有りペアとしての複素数に対する 2 つのもともとらしい表現について議論することから始めます。直行形式(実数部と虚数部)と極形式(大きさと角度)です。<sup>83</sup> Section 2.4.2 がどのようにして両方の表現がタイプタグとジェネリック命令の使用を通して单一のシステム内にて共存できるように作成され得るのかについてを示します。

Like rational numbers, complex numbers are naturally represented as ordered pairs. The set of complex numbers can be thought of as a two-dimensional space with two orthogonal axes, the “real” axis and the “imaginary” axis. (See Figure 2.20.) From this point of view, the complex number  $z = x + iy$  (where  $i^2 = -1$ ) can be thought of as the point in the plane whose real coordinate is  $x$  and whose imaginary coordinate is  $y$ . Addition of complex numbers reduces in this representation to addition of coordinates:

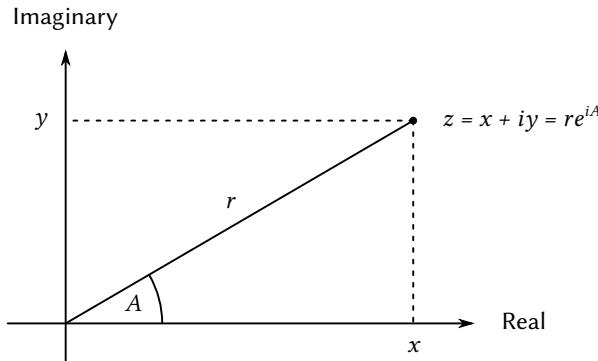
分数と同様に、複素数は自然に順序有りペアとして表現されます。複素数の集合は 2 つの直行する軸を持つ二次元空間として考えることができます。この視点から複素数  $z = x + iy$  (ここで  $i^2 = -1$ ) はその平面中の実数座標が  $x$  かつ、虚数座標が  $y$  の点として考えることができます。複素数の和はこの表現において座標の和と還元できます。

$$\begin{aligned}\text{Real-part}(z_1 + z_2) &= \text{Real-part}(z_1) + \text{Real-part}(z_2), \\ \text{Imaginary-part}(z_1 + z_2) &= \text{Imaginary-part}(z_1) + \text{Imaginary-part}(z_2).\end{aligned}$$

---

<sup>82</sup>In actual computational systems, rectangular form is preferable to polar form most of the time because of roundoff errors in conversion between rectangular and polar form. This is why the complex-number example is unrealistic. Nevertheless, it provides a clear illustration of the design of a system using generic operations and a good introduction to the more substantial systems to be developed later in this chapter.

<sup>83</sup> 実際の計算システムにおいては直行形式のほうが極形式よりも多くの場合には好まれます。直行形式と極形式の間の変換における丸め誤差のためです。これがなぜ複素数システムのサンプルが非現実的であるかの理由です。それにもかかわらず、この例はジェネリック命令を用いたシステムの設計の明確な説明を提供し、またこの章の中で後に開発されるより実質的なシステムに対する良い導入部であります。



**Figure 2.20:** Complex numbers as points in the plane.

When multiplying complex numbers, it is more natural to think in terms of representing a complex number in polar form, as a magnitude and an angle ( $r$  and  $A$  in Figure 2.20). The product of two complex numbers is the vector obtained by stretching one complex number by the length of the other and then rotating it through the angle of the other:

複素数をかけ算する場合、複素数を大きさと角度 (Figure 2.20内の  $r$  と  $A$ ) としての極形式の表現を用いて考える方がより自然です。2つの複素数の積は一方の複素数をもう一方の長さで延し次にもう一方の角度の分、回転することで得られるベクトルになります。

$$\begin{aligned}\text{Magnitude}(z_1 \cdot z_2) &= \text{Magnitude}(z_1) \cdot \text{Magnitude}(z_2), \\ \text{Angle}(z_1 \cdot z_2) &= \text{Angle}(z_1) + \text{Angle}(z_2).\end{aligned}$$

$$\begin{aligned}\text{大きさ}(z_1 \cdot z_2) &= \text{大きさ}(z_1) \cdot \text{大きさ}(z_2), \\ \text{角度}(z_1 \cdot z_2) &= \text{角度}(z_1) + \text{角度}(z_2).\end{aligned}$$

Thus, there are two different representations for complex numbers, which are appropriate for different operations. Yet, from the viewpoint of someone writing a program that uses complex numbers, the principle of data abstraction suggests that all the operations for manipulating complex numbers should be available regardless of which representation is used by the computer. For example, it is often useful to be able to find the magnitude of a complex number

that is specified by rectangular coordinates. Similarly, it is often useful to be able to determine the real part of a complex number that is specified by polar coordinates.

従って複素数には 2 つの異なる表現が存在し、それぞれは異なる操作に適しています。けれども、複素数を用いるプログラムを書いている誰かさんの視点からは、データ抽象化の主義が複素数を操作するための全ての命令はどの表現がコンピュータにより用いられるかに係らず存在するべきだと提案します。例えば直行形式にて指定される複素数の大きさを求められることはしばしば便利であります。同様に極形式にて指定される複素数の実数部を決定できることも時折便利であります。

To design such a system, we can follow the same data-abstraction strategy we followed in designing the rational-number package in [Section 2.1.1](#). Assume that the operations on complex numbers are implemented in terms of four selectors: `real-part`, `imag-part`, `magnitude` and `angle`. Also assume that we have two procedures for constructing complex numbers: `make-from-real-imag` returns a complex number with specified real and imaginary parts, and `make-from-mag-ang` returns a complex number with specified magnitude and angle. These procedures have the property that, for any complex number `z`, both

そのようなシステムを設計するために、[Section 2.1.1](#)にて分数パッケージの設計において従ったのと同じデータ抽象化戦略に従えます。複素数上の命令が 4 つのセレクタを使用して実装されると想定します。`real-part`, `imag-part`, `magnitude`, `angle` です。また複素数を構築する 2 つの手続を持っているとも想定します。`make-from-real-imag` は指定された実数部と虚数部を持つ複素数を返し、`make-from-mag-ang` は指定された大きさと角度を持つ複素数を返します。これらの手続は任意の複素数に対し同じ特性を持ちます。

```
(make-from-real-imag (real-part z) (imag-part z))
```

と

```
(make-from-mag-ang (magnitude z) (angle z))
```

produce complex numbers that are equal to `z`.

の両方が `z` に等しい複素数を生成します。

Using these constructors and selectors, we can implement arithmetic on complex numbers using the “abstract data” specified by the constructors and selectors, just as we did for rational numbers in [Section 2.1.1](#). As shown in the formulas above, we can add and subtract complex numbers in terms of real and

imaginary parts while multiplying and dividing complex numbers in terms of magnitudes and angles:

これらのコンストラクタとセレクタを用いて、Section 2.1.1で分数に対して行ったのと全く同様に、コンストラクタとセレクタにて指定された“抽象データ”を用いて複素数上での数値演算を実装可能です。前述の式にて示されたように、複素数の和と差は実数部と虚数部を用いることで、また複素数の積と商は大きさと角度を用いることで実装できます。

```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2)))))

(define (sub-complex z1 z2)
  (make-from-real-imag (- (real-part z1) (real-part z2))
                       (- (imag-part z1) (imag-part z2))))

(define (mul-complex z1 z2)
  (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                     (+ (angle z1) (angle z2)))))

(define (div-complex z1 z2)
  (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                     (- (angle z1) (angle z2))))
```

To complete the complex-number package, we must choose a representation and we must implement the constructors and selectors in terms of primitive numbers and primitive list structure. There are two obvious ways to do this: We can represent a complex number in “rectangular form” as a pair (real part, imaginary part) or in “polar form” as a pair (magnitude, angle). Which shall we choose?

複素数パッケージを完了させるためには表現を選択し、コンストラクタとセレクタをプリミティブな数値とリスト構造を用いて実装しなければなりません。これを行うために 2 つ明らかな方法があります。“直行形式”の複素数はペア(実数部, 虚数部)として表現し、また極形式はペア(大きさ, 角度)にて表現します。どちらを選択すべきでしょうか?

In order to make the different choices concrete, imagine that there are two programmers, Ben Bitdiddle and Alyssa P. Hacker, who are independently de-

signing representations for the complex-number system. Ben chooses to represent complex numbers in rectangular form. With this choice, selecting the real and imaginary parts of a complex number is straightforward, as is constructing a complex number with given real and imaginary parts. To find the magnitude and the angle, or to construct a complex number with a given magnitude and angle, he uses the trigonometric relations

異なる選択を具体的にするために、二人のプログラマ、Ben Bitdiddle と Alyssa P. Hacker がいると想像して下さい。二人は複素数システムのための表現を独立して設計します。Ben は複素数を直行形式にて表現することを選択しました。この選択により現状として複素数を与えられた実数部と虚数部から構築するため、実数部と虚数部を複素数から選択するのは直接的です。大きさと角度を求めるためには、または複素数を与えられた大きさと角度から構築するために彼は三角法の関係を用いました。

$$\begin{aligned} x &= r \cos A, & r &= \sqrt{x^2 + y^2}, \\ y &= r \sin A, & A &= \arctan(y, x), \end{aligned}$$

which relate the real and imaginary parts  $(x, y)$  to the magnitude and the angle  $(r, A)$ .<sup>84</sup> Ben's representation is therefore given by the following selectors and constructors:

これは実数部と虚数部  $(x, y)$  を大きさと角度  $(r, A)$  へと関係づけます。<sup>85</sup> Ben の表現は従つて以下のセレクタとコンストラクタにより与えられます。

```
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (magnitude z)
  (sqrt (+ (square (real-part z))
            (square (imag-part z)))))

(define (angle z)
  (atan (imag-part z) (real-part z)))
(define (make-from-real-imag x y) (cons x y))
(define (make-from-mag-ang r a))
```

---

<sup>84</sup>The arctangent function referred to here, computed by Scheme's `atan` procedure, is defined so as to take two arguments  $y$  and  $x$  and to return the angle whose tangent is  $y/x$ . The signs of the arguments determine the quadrant of the angle.

<sup>85</sup>ここで参照されたアークタンジェント関数は、Scheme の `atan` 手続にて計算されますが、2つの引数  $y$  と  $x$  を取りタンジェントが  $y/x$  となる角度を返すように定義されました。引数の符号が角度の象限を決定します。

```
(cons (* r (cos a)) (* r (sin a))))
```

Alyssa, in contrast, chooses to represent complex numbers in polar form. For her, selecting the magnitude and angle is straightforward, but she has to use the trigonometric relations to obtain the real and imaginary parts. Alyssa's representation is:

一方、Alyssa は複素数を極形式にて表現することを選択しました。彼女にとつては大きさと角度を選択するのは直接的です。しかし実数部と虚数部を得るために三角法の関係を用いねばなりません。Alyssa の表現は次のとおりです。

```
(define (real-part z) (* (magnitude z) (cos (angle z))))
(define (imag-part z) (* (magnitude z) (sin (angle z))))
(define (magnitude z) (car z))
(define (angle z) (cdr z))
(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))
(define (make-from-mag-ang r a) (cons r a))
```

The discipline of data abstraction ensures that the same implementation of `add-complex`, `sub-complex`, `mul-complex`, and `div-complex` will work with either Ben's representation or Alyssa's representation.

データ抽象化の規律は `add-complex`, `sub-complex`, `mul-complex`, `div-complex` の同じ実装が Ben の表現と Alyssa の表現のどちらに対してもうまくいくことを保証します。

## 2.4.2 Tagged data

One way to view data abstraction is as an application of the “principle of least commitment.” In implementing the complex-number system in Section 2.4.1, we can use either Ben's rectangular representation or Alyssa's polar representation. The abstraction barrier formed by the selectors and constructors permits us to defer to the last possible moment the choice of a concrete representation for our data objects and thus retain maximum flexibility in our system design.

データ抽象化を考え方の 1 つは“最小責務の原則”の適用としてです。Section 2.4.1 の複素数システムの実装において、私達は Ben の直行形式表現と

Alyssa の極形式表現のどちらも使用することができました。セレクタとコンストラクタにより形成された抽象化バリアが最後の可能な瞬間にデータオブジェクトに対する具体的な表現の選択に従うことを可能にしています。従ってシステム設計において最高の柔軟性を維持することができるのです。

The principle of least commitment can be carried to even further extremes. If we desire, we can maintain the ambiguity of representation even *after* we have designed the selectors and constructors, and elect to use both Ben's representation *and* Alyssa's representation. If both representations are included in a single system, however, we will need some way to distinguish data in polar form from data in rectangular form. Otherwise, if we were asked, for instance, to find the **magnitude** of the pair  $(3, 4)$ , we wouldn't know whether to answer 5 (interpreting the number in rectangular form) or 3 (interpreting the number in polar form). A straightforward way to accomplish this distinction is to include a **type tag**—the symbol **rectangular** or **polar**—as part of each complex number. Then when we need to manipulate a complex number we can use the tag to decide which selector to apply.

最小責務の原則はさらにもっと高みへと到達することができます。もし私達が望めば、セレクタとコンストラクタを設計した“後”にさえ表現の多義性を維持することが可能です。そして Ben の表現 “と” Alyssa の表現の両方の使用を選択できます。もし両方の表現が单一のシステムに含まれる場合、極形式のデータを直行形式のデータから識別するための何らかの方法が必要になります。そうでなければ、例えばペア  $(3, 4)$  の大きさを求めるよう尋ねられた場合に(数値を直行形式だと考えて)5と答えるべきか(数値が極形式であると考えて)3と答えるべきであるのか分かりません。この識別を直接的な方法で達成するために **type tag**(タイプタグ)—**rectangular** または **polar** のシンボル—を各複素数の部分として導入します。すると複素数を操作せねばならない時にタグを用いてどちらのセレクタを適用すべきか決定することができます。

In order to manipulate tagged data, we will assume that we have procedures **type-tag** and **contents** that extract from a data object the tag and the actual contents (the polar or rectangular coordinates, in the case of a complex number). We will also postulate a procedure **attach-tag** that takes a tag and contents and produces a tagged data object. A straightforward way to implement this is to use ordinary list structure:

タグ付きデータを操作するためにデータオブジェクトからタグと(複素数の場合には極形式、または直行形式の)実際のコンテンツを抽出する手続 **type-tag** と **contents** を持つと想定します。またタグとコンテンツを取りタグ付き

データオブジェクトを生成する手続 `attach-tag` を仮定します。これを実装する直接的な方法は普通のリスト構造を用いることです。

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum: TYPE-TAG" datum)))
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum: CONTENTS" datum)))
```

Using these procedures, we can define predicates `rectangular?` and `polar?`, which recognize rectangular and polar numbers, respectively:

これら手続を用いて述語 `rectangular?` と `polar?` を定義し、直行形式と極形式のそれぞれを認識することができます。

```
(define (rectangular? z) (eq? (type-tag z) 'rectangular))
(define (polar? z) (eq? (type-tag z) 'polar))
```

With type tags, Ben and Alyssa can now modify their code so that their two different representations can coexist in the same system. Whenever Ben constructs a complex number, he tags it as rectangular. Whenever Alyssa constructs a complex number, she tags it as polar. In addition, Ben and Alyssa must make sure that the names of their procedures do not conflict. One way to do this is for Ben to append the suffix `rectangular` to the name of each of his representation procedures and for Alyssa to append `polar` to the names of hers. Here is Ben's revised rectangular representation from Section 2.4.1:

タイプタグを用いて Ben と Alyssa はこれで彼等のコードを変更し 2 つの異なる表現が同じシステム内にて共存させることができるようになりました。Ben が複素数を構築する度に彼は直行形式であるとタグを付けます。Alyssa が複素数を構築する度に、彼女はそれを極形式であるとタグを付けます。加えて、Ben と Alyssa は手続の名前が衝突しないように確認しなければなりません。これを行う 1 つの方法として Ben は彼の各表現手続に接尾辞 `rectangular` を追加し、Alyssa は彼女の手続に対し `polar` を付け加えます。以下は Ben のSection 2.4.1から改正した直行形式表現です。

```

(define (real-part-rectangular z) (car z))
(define (imag-part-rectangular z) (cdr z))
(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z))
            (square (imag-part-rectangular z)))))

(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z)))

(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)))
(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
              (cons (* r (cos a)) (* r (sin a)))))


```

and here is Alyssa's revised polar representation:

そして以下は Alyssa の改訂版極形式表現です。

```

(define (real-part-polar z)
  (* (magnitude-polar z) (cos (angle-polar z))))
(define (imag-part-polar z)
  (* (magnitude-polar z) (sin (angle-polar z))))
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))

(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
              (cons (sqrt (+ (square x) (square y)))
                    (atan y x))))
(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)))


```

Each generic selector is implemented as a procedure that checks the tag of its argument and calls the appropriate procedure for handling data of that type. For example, to obtain the real part of a complex number, `real-part` examines the tag to determine whether to use Ben's `real-part-rectangular` or Alyssa's `real-part-polar`. In either case, we use `contents` to extract the bare, untagged datum and send this to the rectangular or polar procedure as required:

各ジェネリックなセレクタは引数のタグをチェックし、そのタイプのデータを扱うのに適切な手続を呼び出す様に実装されます。例として、複素数の実数部を得る場合、`real-part` はタグを確かめ Ben の `real-part-rectangular` か Alyssa の `real-part-polar` のどちらを使うのかを決定します。どちらの場合でも `contents` を用いて生のタグの無いデータを抽出し直交形式、または極形式の手続を必要に応じて呼び出します。

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type: REAL-PART" z)))))

(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))
        (else (error "Unknown type: IMAG-PART" z)))))

(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
         (magnitude-polar (contents z)))
        (else (error "Unknown type: MAGNITUDE" z)))))

(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
         (angle-polar (contents z)))
        (else (error "Unknown type: ANGLE" z))))
```

To implement the complex-number arithmetic operations, we can use the same procedures `add-complex`, `sub-complex`, `mul-complex`, and `div-complex` from Section 2.4.1, because the selectors they call are generic, and so will work with

either representation. For example, the procedure `add-complex` is still

複素数演算命令を実装するためにはSection 2.4.1から同じ手続 `add-complex`, `sub-complex`, `mul-complex`, `div-complex` を使うことができます。なぜならそれらが呼び出すセレクタはジェネリックであるためどちらの表現に対しても働くからです。例として手続 `add-complex` は今でも以下のとおりです。

```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                        (+ (imag-part z1) (imag-part z2))))
```

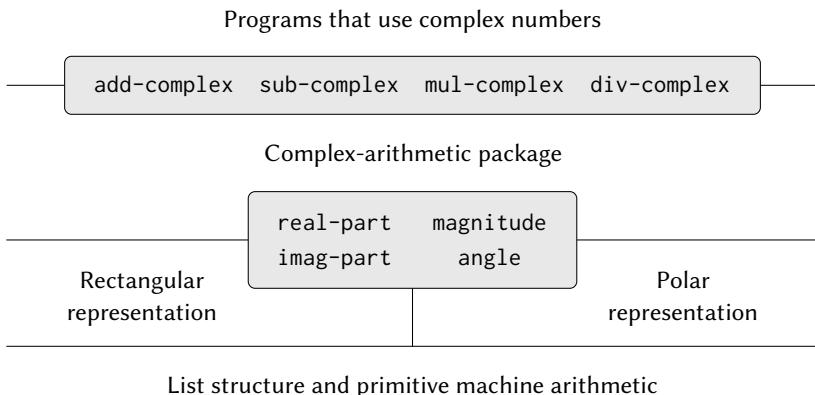
Finally, we must choose whether to construct complex numbers using Ben's representation or Alyssa's representation. One reasonable choice is to construct rectangular numbers whenever we have real and imaginary parts and to construct polar numbers whenever we have magnitudes and angles:

最後に、複素数を Ben の表現と Alyssa の表現のどちらを利用して構築するのか決定しなければなりません。妥当な選択として実数部と虚数部がある場合には直行形式を用い、大きさと角度がある場合には極形式を用いて構築します。

```
(define (make-from-real-imag x y)
  (make-from-real-imag-rectangular x y))
(define (make-from-mag-ang r a)
  (make-from-mag-ang-polar r a))
```

The resulting complex-number system has the structure shown in Figure 2.21. The system has been decomposed into three relatively independent parts: the complex-number-arithmetic operations, Alyssa's polar implementation, and Ben's rectangular implementation. The polar and rectangular implementations could have been written by Ben and Alyssa working separately, and both of these can be used as underlying representations by a third programmer implementing the complex-arithmetic procedures in terms of the abstract constructor/selector interface.

結果としての複素数システムはFigure 2.21にて表される構造を持ちます。システムは 3 つの関連する独立した部分に分離されます。複素数演算命令、Alyssa の極形式実装、そして Ben の直行形式実装です。極形式と直行形式の実装は Ben と Alyssa が別々に働きながら書かれることが可能でした。そして両者が抽象コンストラクタ、セレクタのインターフェイスを用いながら複素数演算手続を実装する第三者のプログラマにより基礎を成す表現として利用されることが可能です。



**Figure 2.21:** Structure of the generic complex-arithmetic system.

Since each data object is tagged with its type, the selectors operate on the data in a generic manner. That is, each selector is defined to have a behavior that depends upon the particular type of data it is applied to. Notice the general mechanism for interfacing the separate representations: Within a given representation implementation (say, Alyssa's polar package) a complex number is an untyped pair (magnitude, angle). When a generic selector operates on a number of `polar` type, it strips off the tag and passes the contents on to Alyssa's code. Conversely, when Alyssa constructs a number for general use, she tags it with a type so that it can be appropriately recognized by the higher-level procedures. This discipline of stripping off and attaching tags as data objects are passed from level to level can be an important organizational strategy, as we shall see in [Section 2.5](#).

各データオブジェクトはその型にてタグ付けられているので、セレクタはデータに対しジェネリックな方法で操作します。これは各セレクタがそれが適用される個々のデータの型に従う振舞を持つように定義されているということです。分けられた表現を結び付けるための一般的なメカニズムについて注意して下さい。与えられた表現実装(例えば Alyssa の極形式パッケージ)の中では複素数は型の無いペア(大きさ, 角度)です。ジェネリックなセレクタが極形式

の型(タイプ)の複素数を操作する時、タグを取り中身を Alyssa のコードに渡します。反対に Alyssa が通常の使用のために数値を構築する時、彼女が型でタグを付けることにより高いレベルの手続により適切に認識されることができます。データオブジェクトがあるレベルから別のレベルへと渡されるに従い、このタグの取り付けと除去の規律が重要な組織的戦略となります。[Section 2.5](#)にてこれについて学びます。

### 2.4.3 Data-Directed Programming and Additivity

The general strategy of checking the type of a datum and calling an appropriate procedure is called *dispatching on type*. This is a powerful strategy for obtaining modularity in system design. On the other hand, implementing the dispatch as in [Section 2.4.2](#) has two significant weaknesses. One weakness is that the generic interface procedures (`real-part`, `imag-part`, `magnitude`, and `angle`) must know about all the different representations. For instance, suppose we wanted to incorporate a new representation for complex numbers into our complex-number system. We would need to identify this new representation with a type, and then add a clause to each of the generic interface procedures to check for the new type and apply the appropriate selector for that representation.

データの型をチェックし適切な手続を呼ぶ一般的な戦略は *dispatching on type*(タイプ別処理) と呼ばれる。これはシステム設計においてモジュール方式を得るための強力な戦略です。一方で[Section 2.4.2](#)のような呼出の実装は 2 つの明らかな弱点が存在します。1 つはジェネリックインターフェイス手続(`real-part`, `imag-part`, `magnitude`, `angle`) は全ての異なる表現について知つていなければなりません。例えば複素数に対する新しい表現を複素数システムに組み入れたいとしましょう。この新しい表現を型にて識別し、次に全てのジェネリックインターフェイス手続に新しい型をチェックする条項を追加し、その表現に対する適切なセレクタを適用する必要が出てくるでしょう。

Another weakness of the technique is that even though the individual representations can be designed separately, we must guarantee that no two procedures in the entire system have the same name. This is why Ben and Alyssa had to change the names of their original procedures from [Section 2.4.1](#).

もう 1 つのこのテクニックの弱点は例え個々の表現が別々に設計できたとしても、システム全体の中でどの 2 つの手続も同じ名前を持たないことを保証せねばなりません。これがなぜ Ben と Alyssa が[Section 2.4.1](#)の彼等の元の手

続の名前を変更しなければいけないかの理由でした。

The issue underlying both of these weaknesses is that the technique for implementing generic interfaces is not *additive*. The person implementing the generic selector procedures must modify those procedures each time a new representation is installed, and the people interfacing the individual representations must modify their code to avoid name conflicts. In each of these cases, the changes that must be made to the code are straightforward, but they must be made nonetheless, and this is a source of inconvenience and error. This is not much of a problem for the complex-number system as it stands, but suppose there were not two but hundreds of different representations for complex numbers. And suppose that there were many generic selectors to be maintained in the abstract-data interface. Suppose, in fact, that no one programmer knew all the interface procedures or all the representations. The problem is real and must be addressed in such programs as large-scale data-base-management systems.

両者の弱点の根底にある問題はジェネリックインターフェイスを実装するためのテクニックが*additive*(付加的)でないことです。ジェネリックセレクタ手続を実装する人はこれらの手続を新しい表現がインストールされる度に変更せねばならず、また個々の表現を接続する人々は名前衝突が起こらぬ様に彼等のコードを変更せねばなりません。これらのケースのそれぞれでコードに対して加えられなければならない変更是簡単ですが、それでも必ず行わねばならず、不自由さと障害の原因となります。これは複素数システムに対しては現時点ではあまり大きな問題ではありません。しかしだた 2 つではなく数百もの異なる表現が複素数に対して存在すると仮定してみて下さい。その上どのプログラマも全てのインターフェイス手続や全ての表現について知らないと想定してみて下さい。問題は現実的であり大規模なデータベース管理システムのようなプログラムでは必ず解決される必要があります。

What we need is a means for modularizing the system design even further. This is provided by the programming technique known as *data-directed programming*. To understand how data-directed programming works, begin with the observation that whenever we deal with a set of generic operations that are common to a set of different types we are, in effect, dealing with a two-dimensional table that contains the possible operations on one axis and the possible types on the other axis. The entries in the table are the procedures that implement each operation for each type of argument presented. In the complex-number system developed in the previous section, the correspondence between

operation name, data type, and actual procedure was spread out among the various conditional clauses in the generic interface procedures. But the same information could have been organized in a table, as shown in Figure 2.22.

私達に必要なものはより一層のシステム設計のモジュール化のための手段です。これは*data-directed programming*(データ適従プログラミング)として知られるプログラミングテクニックにより提供されます。データ適従プログラミングがどのように働くかを理解するためには、異なる型の集合に対して共通なジェネリックな命令の集合を扱う度に、実際に予想される命令を 1 つの軸に、予想される型をもう一方の軸に持つ二次元の表に取り組み、その観察結果から始めます。表の項目には与えられた各引数の型に対する各命令を実装する手続です。前の章にて開発された複素数システムでは命令の名前、データタイプ、実際の手続の間の対応はジェネリックなインターフェイス手続の種々の条件節の間に広がっています。しかし同じ情報がFigure 2.22の中に示されるように 1 つのテーブルの中に組込まれることができたはずです。

Data-directed programming is the technique of designing programs to work with such a table directly. Previously, we implemented the mechanism that interfaces the complex-arithmetic code with the two representation packages as a set of procedures that each perform an explicit dispatch on type. Here we will implement the interface as a single procedure that looks up the combination of the operation name and argument type in the table to find the correct procedure to apply, and then applies it to the contents of the argument. If we do this, then to add a new representation package to the system we need not change any existing procedures; we need only add new entries to the table.

データ適従プログラミングはそのようなテーブルと直接連携するためのプログラム設計のテクニックです。以前は私達は複素数演算コードをそれぞれが明示的に型に従う呼び出しを行う手続の集合としての 2 つの表現パッケージと接続するメカニズムを実装しました。ここではインターフェイスを命令の名前と引数タイプの組み合わせをテーブルの中から調べ適用すべき正しい手続を見つける单一の手続として実装します。そして次にその手続を引数の中身に対して適用します。これを行なえば、システムに対して新しい表現パッケージの追加するために既存の手続に何の変更を行う必要もありません。必要なのは表に新しい項目を追加することです。

To implement this plan, assume that we have two procedures, `put` and `get`, for manipulating the operation-and-type table:

この計画を実装するために、2 つの手続 `put` と `get` を命令と型のテーブルを操作するために持っていると仮定します。

Operations	Types	
	Polar	Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

**Figure 2.22:** Table of operations for the complex-number system.

- `(put <op> <type> <item>)` installs the `<item>` in the table, indexed by the `<op>` and the `<type>`.
- `(get <op> <type>)` looks up the `<op>`, `<type>` entry in the table and returns the item found there. If no item is found, `get` returns false.
- `(put <op> <type> <item>)` は `<item>` をテーブルに挿入し、`<op>` と `<type>` で索引付けられる
- `(get <op> <type>)` は `<op>`, `<type>` の項目をテーブルから探し見つかった項目を返す。もし見つからない場合には `get` は false を返す

For now, we can assume that `put` and `get` are included in our language. In Chapter 3 (Section 3.3.3) we will see how to implement these and other operations for manipulating tables.

今のところは `put` と `get` が私達の言語に含まれていると仮定しましょう。Chapter 3 (Section 3.3.3)においてこれらと他のテーブル操作の命令をどのように実装するかについて学びます。

Here is how data-directed programming can be used in the complex-number system. Ben, who developed the rectangular representation, implements his code just as he did originally. He defines a collection of procedures, or a *package*, and interfaces these to the rest of the system by adding entries to the table that tell the system how to operate on rectangular numbers. This is accomplished by calling the following procedure:

ここからはデータ適従プログラミングが複素数システムにおいてどのように使用できるかについて示します。直行形式表現を開発した Ben は彼が元々行

ったとおりにコードを実装しました。彼は手続の集合、つまりは*package*(パッケージ)を定義し、システムにどのように直行形式の数値を取り扱うかを教えるテーブルに項目を追加することで、パッケージをシステムの残りに対して接続します。これは以下の手續を呼び出すことにより達成されます。

```
(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y) (cons x y))
  (define (magnitude z)
    (sqrt (+ (square (real-part z))
              (square (imag-part z)))))

  (define (angle z)
    (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a)
    (cons (* r (cos a)) (* r (sin a)))))

  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

Notice that the internal procedures here are the same procedures from Section 2.4.1 that Ben wrote when he was working in isolation. No changes are necessary in order to interface them to the rest of the system. Moreover, since these procedure definitions are internal to the installation procedure, Ben needn't worry about name conflicts with other procedures outside the rectangular package. To interface these to the rest of the system, Ben installs his **real-part** procedure under the operation name **real-part** and the type (**rectangular**), and

similarly for the other selectors.<sup>86</sup> The interface also defines the constructors to be used by the external system.<sup>87</sup> These are identical to Ben's internally defined constructors, except that they attach the tag.

この中の内部手続は Ben がSection 2.4.1にて分離を行った時に彼が書いたものと同じ手續であることに注意して下さい。これらをシステムの残りに接続するためには全く変更が必要ありません。さらに、これらの手続の定義はインストールを行う手続の内部であるため、Ben は直行形式パッケージの外部の他の手続に対して名前の衝突が起こることを全く心配する必要がありません。これらをシステムの残りに対し接続するために、Ben は彼の `real-part` 手続を命令名 `real-part` と型 (`rectangular`) の元にインストールしました。そして他のセレクタに対しても同様に行いました。<sup>88</sup> このインターフェイスはまた外部システムにより利用されるコンストラクタも定義します。<sup>89</sup> これらは Ben の内部定義コンストラクタと全く同じです。ただしタグを付加することが異なります。

Alyssa's polar package is analogous:  
Alyssa の極形式パッケージも同様です。

```
(define (install-polar-package)
  ;; internal procedures
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-mag-ang r a) (cons r a))
  (define (real-part z)
    (* (magnitude z) (cos (angle z))))
  (define (imag-part z)
    (* (magnitude z) (sin (angle z))))
  (define (make-from-real-imag x y)
    (cons (sqrt (+ (square x) (square y))))
```

<sup>86</sup>We use the list (`rectangular`) rather than the symbol `rectangular` to allow for the possibility of operations with multiple arguments, not all of the same type.

<sup>87</sup>The type the constructors are installed under needn't be a list because a constructor is always used to make an object of one particular type.

<sup>88</sup>私達はシンボル `rectangular` ではなくリスト (`rectangular`) を用いました。全てが同じ型ではない複数の引数を伴なう命令の可能性を考慮するためです。

<sup>89</sup>コンストラクタがその下にインストールされる型はリストである必要がありません。なぜならコンストラクタは常にある特定の型のオブジェクトを作成するために使用されるためです。

```

        (atan y x)))
;; interface to the rest of the system
(define (tag x) (attach-tag 'polar x))
(put 'real-part '(polar) real-part)
(put 'imag-part '(polar) imag-part)
(put 'magnitude '(polar) magnitude)
(put 'angle '(polar) angle)
(put 'make-from-real-imag 'polar
     (lambda (x y) (tag (make-from-real-imag x y))))
(put 'make-from-mag-ang 'polar
     (lambda (r a) (tag (make-from-mag-ang r a))))
'done)

```

Even though Ben and Alyssa both still use their original procedures defined with the same names as each other's (e.g., `real-part`), these definitions are now internal to different procedures (see [Section 1.1.8](#)), so there is no name conflict.

Ben と Alyssa の両者が今でも御互いに同じ名前 (例えば `real-part`) にて定義された彼等の元々の手続を使用しようとも、これらの定義は今では異なる手続の内部定義 ([Section 1.1.8](#) 参照) です。従って名前の衝突は起りません。

The complex-arithmetic selectors access the table by means of a general “operation” procedure called `apply-generic`, which applies a generic operation to some arguments. `Apply-generic` looks in the table under the name of the operation and the types of the arguments and applies the resulting procedure if one is present:

複素数演算のセレクタは `apply-generic` と呼ばれる普遍的な “operation” 手続を用いてテーブルにアクセスします。これはジェネリックな命令を引数に対して適用します。`apply-generic` は命令の名前と引数の型の下に表を調べ結果としての手続が存在すれば適用します。<sup>90</sup>

---

<sup>90</sup> `Apply-generic` uses the dotted-tail notation described in [Exercise 2.20](#), because different generic operations may take different numbers of arguments. In `apply-generic`, `op` has as its value the first argument to `apply-generic` and `args` has as its value a list of the remaining arguments.

`apply-generic` は [Exercise 2.20](#) で説明したドット付き末尾記法を用います。異なるジェネリック命令は異なる数の引数を取る場合が考えられるためです。`apply-generic` では `op` がその値として `apply-generic` の第一引数を持ち、`args` はその値として残りの引数のリストを持ちます。

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "No method for these types: APPLY-GENERIC"
            (list op type-tags))))))
```

Using `apply-generic`, we can define our generic selectors as follows:

`apply-generic` を用いることで、私達のジェネリックなセレクタを以下のように定義することができます。

```
(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))
```

Observe that these do not change at all if a new representation is added to the system.

もし新しい表現がシステムに追加されたとしてもこれらが全く変更されないことに注意して下さい。

We can also extract from the table the constructors to be used by the programs external to the packages in making complex numbers from real and imaginary parts and from magnitudes and angles. As in [Section 2.4.2](#), we construct rectangular numbers whenever we have real and imaginary parts, and polar numbers whenever we have magnitudes and angles:

またテーブルからコンストラクタを抽出することもできます。コンストラクタはパッケージの外部プログラムにより使用でき、実数部と虚数部が大きさ

---

`Apply-generic` also uses the primitive procedure `apply`, which takes two arguments, a procedure and a list. `Apply` applies the procedure, using the elements in the list as arguments. For example,

`apply-generic` はまたプリミティブな手続 `apply` を用います。これは 2 つの引数、手続とリストを取ります。`apply` はリストの要素を引数として手続を適用します。例えば、

```
(apply + (list 1 2 3 4))
```

returns 10.

は 10 を返します。

と角度から複素数を作ります。Section 2.4.2にあるとおり、実数部と虚数部がある場合には直交形式で構築し、大きさと角度がある場合には極形式にて構築します。

```
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag 'rectangular) x y))
(define (make-from-mag-ang r a)
  ((get 'make-from-mag-ang 'polar) r a))
```

**Exercise 2.73:** Section 2.3.2 described a program that performs symbolic differentiation:

Section 2.3.2は記号微分を行うプログラムについて説明した。

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var))))
        ((product? exp)
         (make-sum (make-product
                    (multiplier exp)
                    (deriv (multiplicand exp) var))
                   (make-product
                     (deriv (multiplier exp) var)
                     (multiplicand exp)))))
        ;<more rules can be added here>
        (else (error "unknown expression type:
                      DERIV" exp))))
```

We can regard this program as performing a dispatch on the type of the expression to be differentiated. In this situation the “type tag” of the datum is the algebraic operator symbol (such as +) and the operation being performed is `deriv`. We can transform this program into data-directed style by rewriting the basic derivative procedure as

このプログラムを微分する式のタイプにより呼出を行っていると解釈することもできる。このシチュエーションではデータの“タイプタグ”が代数演算子の記号(例えば+)であり実行される命令は `deriv` である。このプログラムを基本の微分手続を書き直すことでデータ適従プログラミングスタイルに変換することができる。

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp))
                (operands exp) var))))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
```

- a Explain what was done above. Why can't we assimilate the predicates `number?` and `variable?` into the data-directed dispatch?

上で何が行われたのか説明せよ。なぜ手続 `number?` と `variable?` をデータ適従呼出しに吸収することができないのか？

- b Write the procedures for derivatives of sums and products, and the auxiliary code required to install them in the table used by the program above.

和と積の微分のための手続とそれらを上記のプログラムで使用されたテーブルにインストールする補助コードを書け。

- c Choose any additional differentiation rule that you like, such as the one for exponents ([Exercise 2.56](#)), and install it in this data-directed system.

貴方の好きな追加の微分ルール、例えば指数に対する物([Exercise 2.56](#))を選択し、このデータ適従システムにインストールせよ。

- d In this simple algebraic manipulator the type of an expression is the algebraic operator that binds it together. Suppose, however, we indexed the procedures in the opposite way, so that the dispatch line in `deriv` looked like

この単純な代数操作において、式の型はそれを一緒に束縛する代数演算子である。しかし手続を逆の向きに索引付けし deriv の呼出行を以下のようにした場合、

```
((get (operator exp) 'deriv) (operands exp) var)
```

What corresponding changes to the derivative system are required?

微分システムへの対応する変更は何が必要か？

**Exercise 2.74:** Insatiable Enterprises, Inc., is a highly decentralized conglomerate company consisting of a large number of independent divisions located all over the world. The company's computer facilities have just been interconnected by means of a clever network-interfacing scheme that makes the entire network appear to any user to be a single computer. Insatiable's president, in her first attempt to exploit the ability of the network to extract administrative information from division files, is dismayed to discover that, although all the division files have been implemented as data structures in Scheme, the particular data structure used varies from division to division. A meeting of division managers is hastily called to search for a strategy to integrate the files that will satisfy headquarters' needs while preserving the existing autonomy of the divisions.

Insatiable Enterprises, Inc.(強欲エンタープライズ社)は高いレベルで非集中化された数多くの独立事業所を世界中に抱える複合企業である。社のコンピュータ施設は接続されたばかりであるが、賢いネットワーク接続計画を用いてネットワーク全体がどのユーザに対しても 1 台のコンピュータとして現れる。強欲社の社長は初めてネットワークの機能を用いて事業所ファイルから管理者情報を取得しようと試みたが、全ての事業所ファイルは Scheme のデータ構造として実装されているのにも係らず、使用されている個々のデータ構造は事業所の間で異なっていることに狼狽した。事業所長の会議が大急ぎで開催され既存の事業所の自立性を保ちつつ本社の要求を満足できるファイル統合の戦略を探すことになった。Show how such a strategy can be implemented with data-directed programming. As an example, suppose that each division's personnel records consist of a single file, which contains a set of records

keyed on employees' names. The structure of the set varies from division to division. Furthermore, each employee's record is itself a set (structured differently from division to division) that contains information keyed under identifiers such as **address** and **salary**. In particular:

そのような戦略がデータ適従戦略を用いてどのように実装できるか示せ。例として各事業所の職員記録は単一のファイルから成る従業員の名前をキーにしたレコードの集合であると想定せよ。集合の構造は事業所毎に変わる。さらに各従業員のレコードはそれ自身が集合（事業所毎で異なる構造）であり **address** と **salary** のような識別子の下で鍵付けられた情報を含んでいる。具体的には

- a Implement for headquarters a **get-record** procedure that retrieves a specified employee's record from a specified personnel file. The procedure should be applicable to any division's file. Explain how the individual divisions' files should be structured. In particular, what type information must be supplied?

本社のために指定された従業員のレコードを指定された職員記録ファイルから取得する **get-record** 手続を実装せよ。手続は任意の事業所のファイルに適用できなければならない。個々の事業所のファイルがどのように構造化されねばならないか説明せよ。具体的にはどんな型の情報が提供されねばならないか

- b Implement for headquarters a **get-salary** procedure that returns the salary information from a given employee's record from any division's personnel file. How should the record be structured in order to make this operation work?

本社のために任意の事業所の職員記録ファイルから与えられた職員記録から給与情報を返す **get-salary** 手続を実装せよ。記録はこの操作が動くようどのように構造化されねばならないか？

- c Implement for headquarters a **find-employee-record** procedure. This should search all the divisions' files for the record of a given employee and return the record. Assume that this

procedure takes as arguments an employee's name and a list of all the divisions' files.

本社のために `find-employee-record` 手続を実装せよ。これは全ての事業所のファイルに対し与えられた従業員のレコードを探し、レコードを返さねばならない。この手続が引数として従業員の名前と全ての事業所のファイルのリストを与えられると仮定せよ。

- d When Insatiable takes over a new company, what changes must be made in order to incorporate the new personnel information into the central system?

強欲社が新しい会社を吸収した時、どんな変更が新しい職員情報を中央システムに受け入れるために必要であるか？

## Message passing

The key idea of data-directed programming is to handle generic operations in programs by dealing explicitly with operation-and-type tables, such as the table in Figure 2.22. The style of programming we used in Section 2.4.2 organized the required dispatching on type by having each operation take care of its own dispatching. In effect, this decomposes the operation-and-type table into rows, with each generic operation procedure representing a row of the table.

データ適従プログラミングの鍵となる考えはプログラム中のジェネリックな命令を Figure 2.22 の様な命令と型のテーブルを明示的に処理することで扱うことです。Section 2.4.2で用いたプログラミングスタイル要求された型に基く呼出を各命令がそれ自身の呼出の世話をを行うことで組織化しました。実際にこれは命令と型のテーブルを、テーブルの行を表す各ジェネリックな操作手続を用いて行に分解します。

An alternative implementation strategy is to decompose the table into columns and, instead of using “intelligent operations” that dispatch on data types, to work with “intelligent data objects” that dispatch on operation names. We can do this by arranging things so that a data object, such as a rectangular number, is represented as a procedure that takes as input the required operation name and performs the operation indicated. In such a discipline, `make-from-real-imag` could be written as

代替的な実装戦略はテーブルを列に分解し、データ型に基づき呼び出しを行う“知的な命令”を用いる代わりに、命令名に基づき呼び出しを行う“知的な

データオブジェクト”を用いて動かすものです。直行形式の複素数の様なデータオブジェクトが入力として必要な命令名を取り指定された命令を実行するように準備を行うことで行うことができます。そのような規律の下では `make-from-real-imag` は以下のように書くことができます。

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude)
           (sqrt (+ (square x) (square y))))
          ((eq? op 'angle) (atan y x))
          (else (error "Unknown op:
                         MAKE-FROM-REAL-IMAG" op)))))

dispatch)
```

The corresponding `apply-generic` procedure, which applies a generic operation to an argument, now simply feeds the operation’s name to the data object and lets the object do the work:<sup>91</sup>

対応する `apply-generic` 手続はジェネリックな命令を引数に適用しますが、ここでは単純に命令の名前をデータオブジェクトに与えオブジェクトに仕事を行わせます。<sup>92</sup>

```
(define (apply-generic op arg) (arg op))
```

Note that the value returned by `make-from-real-imag` is a procedure—the internal `dispatch` procedure. This is the procedure that is invoked when `apply-generic` requests an operation to be performed.

`make-from-real-imag` により返される値は手続—内部手続 `dispatch` のであることに注意して下さい。これが `apply-generic` が命令に実行を要求した時に起動される手続です。

This style of programming is called *message passing*. The name comes from the image that a data object is an entity that receives the requested operation name as a “message.” We have already seen an example of message passing in [Section 2.1.3](#), where we saw how `cons`, `car`, and `cdr` could be defined

---

<sup>91</sup>One limitation of this organization is it permits only generic procedures of one argument.

<sup>92</sup>この構造の 1 つの制約は一引数のジェネリック手続のみを許容することです。

with no data objects but only procedures. Here we see that message passing is not a mathematical trick but a useful technique for organizing systems with generic operations. In the remainder of this chapter we will continue to use data-directed programming, rather than message passing, to discuss generic arithmetic operations. In [Chapter 3](#) we will return to message passing, and we will see that it can be a powerful tool for structuring simulation programs.

このプログラミングスタイルは *message passing*(メッセージパッシング)と呼ばれます。その名前はデータオブジェクトが要求された命令の名前を“メッセージ”として受け取った要素であるというイメージから来ています。私達は既にメッセージパッシングの例を [Section 2.1.3](#) にて見ていました。その時は `cons`, `car`, `cdr` がデータオブジェクト無し、手続のみでどのように定義され得るかを学びました。ここではメッセージパッシングは数学上のトリックではなくジェネリック命令を用いてシステムを構造化するのに便利なテクニックであることを学びます。この章の残りではメッセージパッシングではなくデータ適従プログラミングの使用を続け、全般的な数値演算操作について議論します。そしてそれがシミュレーションプログラムの構造化に対して強力なツールに成り得ることを学びます。

**Exercise 2.75:** Implement the constructor `make-from-mag-ang` in message-passing style. This procedure should be analogous to the `make-from-real-imag` procedure given above.

コンストラクタ `make-from-mag-ang` をメッセージパッシングスタイルにて実装せよ。この手続は上で与えられた `make-from-real-imag` と同様でなければならない。

**Exercise 2.76:** As a large system with generic operations evolves, new types of data objects or new operations may be needed. For each of the three strategies—generic operations with explicit dispatch, data-directed style, and message-passing-style—describe the changes that must be made to a system in order to add new types or new operations. Which organization would be most appropriate for a system in which new types must often be added? Which would be most appropriate for a system in which new operations must often be added?

ジェネリックな命令を用いた巨大システムが発展するにつれ、新しい型のデータオブジェクトや命令が必要となるかもしれない。3つの戦略—ジェネリック命令の明示的呼出、データ適従スタイル、

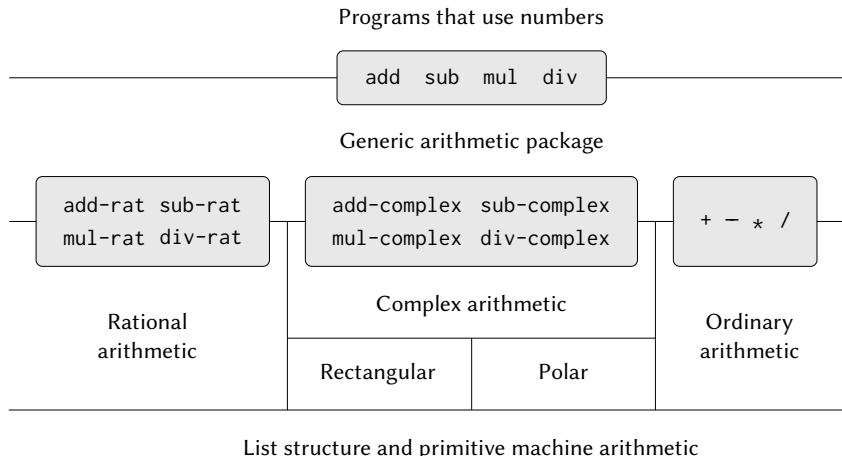
メッセージパッキングーのそれぞれに対して新しい型や命令を追加するために必要なシステムに対する変更について説明せよ。どの構造化が新しい型が良く追加されるシステムに対して最も適切であるか? どれが新しい命令が良く追加されねばならぬシステムに対して最も適切であるか?

## 2.5 Systems with Generic Operations

In the previous section, we saw how to design systems in which data objects can be represented in more than one way. The key idea is to link the code that specifies the data operations to the several representations by means of generic interface procedures. Now we will see how to use this same idea not only to define operations that are generic over different representations but also to define operations that are generic over different kinds of arguments. We have already seen several different packages of arithmetic operations: the primitive arithmetic ( $+$ ,  $-$ ,  $*$ ,  $/$ ) built into our language, the rational-number arithmetic (`add-rat`, `sub-rat`, `mul-rat`, `div-rat`) of [Section 2.1.1](#), and the complex-number arithmetic that we implemented in [Section 2.4.3](#). We will now use data-directed techniques to construct a package of arithmetic operations that incorporates all the arithmetic packages we have already constructed.

前の節ではデータオブジェクトが2つ以上の方で表現されるシステムをどのように設計するかについて学んだ。鍵となる考えはデータ操作を指定するコードをいくつかの表現に対しジェネリックなインターフェイス手続を用いてリンクすることでした。ここではこれと同じ考え方異なる表現上のジェネリックな命令の定義のみでなく、異なる種類の引数上のジェネリックな命令を定義するためにどのように用いるかについて学びます。私達は既にいくつかの数値演算命令の異なるパッケージを見てきました。言語内に構築されたプリミティブ数値演算( $+$ ,  $-$ ,  $*$ ,  $/$ )、[Section 2.1.1](#)の分数演算(`add-rat`, `sub-rat`, `mul-rat`, `div-rat`)、[Section 2.4.3](#)で実装した複素数演算です。ここではデータ適従のテクニックを用いて私達がこれまでに構築した全ての数値演算パッケージを内蔵する数値演算のパッケージを構築します。

[Figure 2.23](#) shows the structure of the system we shall build. Notice the abstraction barriers. From the perspective of someone using “numbers,” there is a single procedure `add` that operates on whatever numbers are supplied. `Add` is part of a generic interface that allows the separate ordinary-arithmetic,



**Figure 2.23:** Generic arithmetic system.

rational-arithmetic, and complex-arithmetic packages to be accessed uniformly by programs that use numbers. Any individual arithmetic package (such as the complex package) may itself be accessed through generic procedures (such as `add-complex`) that combine packages designed for different representations (such as rectangular and polar). Moreover, the structure of the system is additive, so that one can design the individual arithmetic packages separately and combine them to produce a generic arithmetic system.

Figure 2.23は私達が構築するシステムの構造を示しています。抽象化バリアに注目して下さい。“数値”を扱う第三者の視点からはそこにあるのはどの種類の数値が提供されても单一の手続 `add` です。`add` はジェネリックインターフェイスの部分で別々の実数演算、分数演算、複素数演算のパッケージに、数値を使用するプログラムから統一的なアクセスを可能にします。(複素数の様な) 任意の個別数値演算パッケージはそれ自身が(直行形式と極形式の様な)異なる表現のために設計されたパッケージを結合する(`add-complex`の様な) ジェネリックな手続を通してアクセスできます。さらに、システムの構造は付加的なため個々の数値演算パッケージは別々に設計することが可能で、それらを結合してジェネリックな数値演算システムを生成できます。

### 2.5.1 Generic Arithmetic Operations

The task of designing generic arithmetic operations is analogous to that of designing the generic complex-number operations. We would like, for instance, to have a generic addition procedure `add` that acts like ordinary primitive addition `+` on ordinary numbers, like `add-rat` on rational numbers, and like `add-complex` on complex numbers. We can implement `add`, and the other generic arithmetic operations, by following the same strategy we used in Section 2.4.3 to implement the generic selectors for complex numbers. We will attach a type tag to each kind of number and cause the generic procedure to dispatch to an appropriate package according to the data type of its arguments.

ジェネリックな数値演算命令の設計タスクはジェネリックな複素数命令を設計するのと同様です。例えば、実数上での通常の加算のプリミティブ`+`、分数上の`add-rat`や複素数上の`add-complex`のように振る舞うジェネリックな加算手続`add`を持ちたいとします。`add`と他のジェネリックな数値演算命令をSection 2.4.3にて複素数に対するジェネリックなセレクタを実装するのに用いたのと同じ戦略に従うことでの実装することが可能です。全ての種類の数値にタイプタグをアタッチすることでジェネリック手続にその引数のデータタイプに従って適切なパッケージを呼び出す理由とします。

The generic arithmetic procedures are defined as follows:

ジェネリックな数値演算は以下のように定義されます。

```
(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div x y))
```

We begin by installing a package for handling *ordinary* numbers, that is, the primitive numbers of our language. We will tag these with the symbol `scheme-number`. The arithmetic operations in this package are the primitive arithmetic procedures (so there is no need to define extra procedures to handle the untagged numbers). Since these operations each take two arguments, they are installed in the table keyed by the list (`scheme-number scheme-number`):

*ordinary*(通常の)数値を扱うためのパッケージをインストールすることで始めます。これは私達の言語のプリミティブな数値のことです。これらにシンボル`scheme-number`でタグを付けます。このパッケージ内の数値演算命令はプリミティブな数値演算手続です。(そのためタグの無い数値を扱うために拡張手続を定義する必要はありません)。これらの命令はそれぞれが2つの引数を

取るためリスト (scheme-number scheme-number) を鍵にしてテーブルにインストールされます。

```
(define (install-scheme-number-package)
  (define (tag x) (attach-tag 'scheme-number x))
  (put 'add '(scheme-number scheme-number)
       (lambda (x y) (tag (+ x y))))
  (put 'sub '(scheme-number scheme-number)
       (lambda (x y) (tag (- x y))))
  (put 'mul '(scheme-number scheme-number)
       (lambda (x y) (tag (* x y))))
  (put 'div '(scheme-number scheme-number)
       (lambda (x y) (tag (/ x y))))
  (put 'make 'scheme-number (lambda (x) (tag x)))
  'done)
```

Users of the Scheme-number package will create (tagged) ordinary numbers by means of the procedure:

scheme-number パッケージのユーザは (タグ付きの) 普通の数値を手続を用いて作成します。

```
(define (make-scheme-number n)
  ((get 'make 'scheme-number) n))
```

Now that the framework of the generic arithmetic system is in place, we can readily include new kinds of numbers. Here is a package that performs rational arithmetic. Notice that, as a benefit of additivity, we can use without modification the rational-number code from [Section 2.1.1](#) as the internal procedures in the package:

さてこのジェネリック数値演算システムのフレームワークが準備できたので新しい種類の数値も容易に含めることができます。ここに分数演算を実行するパッケージがあります。付加的あることの利点として[Section 2.1.1](#)の分数コードをパッケージ内の内部手続として変更無しに利用できることに注目して下さい。

```
(define (install-rational-package)
  ;; internal procedures
  (define (numer x) (car x))
  (define (denom x) (cdr x))
```

```

(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
;; interface to rest of the system
(define (tag x) (attach-tag 'rational x))
(put 'add '(rational rational)
     (lambda (x y) (tag (add-rat x y))))
(put 'sub '(rational rational)
     (lambda (x y) (tag (sub-rat x y))))
(put 'mul '(rational rational)
     (lambda (x y) (tag (mul-rat x y))))
(put 'div '(rational rational)
     (lambda (x y) (tag (div-rat x y))))
(put 'make 'rational
     (lambda (n d) (tag (make-rat n d))))
'done)
(define (make-rational n d)
  ((get 'make 'rational) n d))

```

We can install a similar package to handle complex numbers, using the tag `complex`. In creating the package, we extract from the table the operations `make-from-real-imag` and `make-from-mag-ang` that were defined by the rectangular and polar packages. Additivity permits us to use, as the internal oper-

ations, the same `add-complex`, `sub-complex`, `mul-complex`, and `div-complex` procedures from Section 2.4.1.

複素数を扱うために同様のパッケージをタグ `complex` を用いてインストールできます。パッケージを作る際に、直行形式と極形式のパッケージにて定義された `make-from-real-imag` と `make-from-mag-ang` の命令をテーブルから抽出します。付加性が内部命令として同じ Section 2.4.1 の手続 `add-complex`, `sub-complex`, `mul-complex`, `div-complex` を使用することを可能にします。

```
(define (install-complex-package)
  ;; imported procedures from rectangular and polar packages
  (define (make-from-real-imag x y)
    ((get 'make-from-real-imag 'rectangular) x y))
  (define (make-from-mag-ang r a)
    ((get 'make-from-mag-ang 'polar) r a))
  ;; internal procedures
  (define (add-complex z1 z2)
    (make-from-real-imag (+ (real-part z1) (real-part z2))
                         (+ (imag-part z1) (imag-part z2))))
  (define (sub-complex z1 z2)
    (make-from-real-imag (- (real-part z1) (real-part z2))
                         (- (imag-part z1) (imag-part z2))))
  (define (mul-complex z1 z2)
    (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                      (+ (angle z1) (angle z2))))
  (define (div-complex z1 z2)
    (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                      (- (angle z1) (angle z2))))
  ;; interface to rest of the system
  (define (tag z) (attach-tag 'complex z))
  (put 'add '(complex complex)
       (lambda (z1 z2) (tag (add-complex z1 z2))))
  (put 'sub '(complex complex)
       (lambda (z1 z2) (tag (sub-complex z1 z2))))
  (put 'mul '(complex complex)
       (lambda (z1 z2) (tag (mul-complex z1 z2))))
  (put 'div '(complex complex)
       (lambda (z1 z2) (tag (div-complex z1 z2))))
```

```
(put 'make-from-real-imag 'complex
      (lambda (x y) (tag (make-from-real-imag x y))))
(put 'make-from-mag-ang 'complex
      (lambda (r a) (tag (make-from-mag-ang r a))))
'done)
```

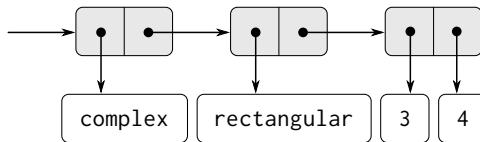
Programs outside the complex-number package can construct complex numbers either from real and imaginary parts or from magnitudes and angles. Notice how the underlying procedures, originally defined in the rectangular and polar packages, are exported to the complex package, and exported from there to the outside world.

複素数パッケージの外側のプログラムは複素数を実数部と虚数部からでも大きさと角度からでも構築することができます。元は直行形式と極形式のパッケージ内にて定義された内在する手続がどのように複素数パッケージにエクスポートされているか、そしてそこからどのようにして外部の世界へとエクスポートされているかについて注意して下さい。

```
(define (make-complex-from-real-imag x y)
  ((get 'make-from-real-imag 'complex) x y))
(define (make-complex-from-mag-ang r a)
  ((get 'make-from-mag-ang 'complex) r a))
```

What we have here is a two-level tag system. A typical complex number, such as  $3 + 4i$  in rectangular form, would be represented as shown in Figure 2.24. The outer tag (`complex`) is used to direct the number to the complex package. Once within the complex package, the next tag (`rectangular`) is used to direct the number to the rectangular package. In a large and complicated system there might be many levels, each interfaced with the next by means of generic operations. As a data object is passed “downward,” the outer tag that is used to direct it to the appropriate package is stripped off (by applying `contents`) and the next level of tag (if any) becomes visible to be used for further dispatching.

ここで私達が行ったのは2つのレベルのタグシステムです。典型的な複素数、直交形式で  $3 + 4i$  のような物はFigure 2.24で示されるように表現されます。外型のタグ (`complex`) は数値を複素数パッケージへと導きます。複素数パッケージに入れば、次のタグ (`rectangular`) が数値を直行形式パッケージへと導きます。巨大で複雑なシステムでは多くのレベルが存在するかもしれません、それぞれはジェネリックな命令を用いて次へと接続されます。データオブジェクトが“下方”へ渡されるにつれ、適切なパッケージへ導く外側のタグは(`contents`を



**Figure 2.24:** Representation of  $3 + 4i$  in rectangular form.

適用することで)取り去られ、次のレベルのタグ(もし存在すれば)がさらなる呼出のために使用されるため見えるようになります。

In the above packages, we used `add-rat`, `add-complex`, and the other arithmetic procedures exactly as originally written. Once these definitions are internal to different installation procedures, however, they no longer need names that are distinct from each other: we could simply name them `add`, `sub`, `mul`, and `div` in both packages.

上記のパッケージでは、`add-rat`, `add-complex`, それに他の数値演算手続を全く元々書かれた状態で利用しました。しかし、これらの定義が異なるインストール手続の内部となれば直ぐに、お互いから識別可能である名前にする必要は無くなります。単純に両者のパッケージにて `add`, `sub`, `mul`, `div` と名付けることが可能になります。

**Exercise 2.77:** Louis Reasoner tries to evaluate the expression (`magnitude z`) where `z` is the object shown in Figure 2.24. To his surprise, instead of the answer 5 he gets an error message from `apply-generic`, saying there is no method for the operation `magnitude` on the types (`complex`). He shows this interaction to Alyssa P. Hacker, who says “The problem is that the complex-number selectors were never defined for `complex` numbers, just for `polar` and `rectangular` numbers. All you have to do to make this work is add the following to the `complex` package:”

Louis Reasoner は `z` が Figure 2.24 で示されるオブジェクトである場合に式 (`magnitude z`) を評価しようとした。驚いたことに、答の 5 の代わりに彼が受け取ったのは `apply-generic` からのエラーメッセージで、型 (`complex`) 上に `magnitude` 命令の手段が存在しないと言う。彼はこの応答を Alyssa P. Hacker に見せた所、彼女は “問題は複素数セレクタが `complex` の数値に対して定義されて

いない、`polar` と `rectangular` の数値に対してのみ行われている。これを動かすためにしなければならないことは以下を `complex` パッケージに追加することだ。”と述べた。

```
(put 'real-part '(complex) real-part)
(put 'imag-part '(complex) imag-part)
(put 'magnitude '(complex) magnitude)
(put 'angle '(complex) angle)
```

Describe in detail why this works. As an example, trace through all the procedures called in evaluating the expression `(magnitude z)` where `z` is the object shown in Figure 2.24. In particular, how many times is `apply-generic` invoked? What procedure is dispatched to in each case?

これでなぜ動くのか詳細を説明せよ。例として式 `(magnitude z)` を `z` がFigure 2.24にて示されるデータオブジェクトの場合に評価する時、呼び出される全ての手続をトレースせよ具体的には、`apply-generic` は何回起動されるか？どの手續が各ケースに対して呼び出されるか？

**Exercise 2.78:** The internal procedures in the `scheme-number` package are essentially nothing more than calls to the primitive procedures `+`, `-`, etc. It was not possible to use the primitives of the language directly because our type-tag system requires that each data object have a type attached to it. In fact, however, all Lisp implementations do have a type system, which they use internally. Primitive predicates such as `symbol?` and `number?` determine whether data objects have particular types. Modify the definitions of `type-tag`, `contents`, and `attach-tag` from Section 2.4.2 so that our generic system takes advantage of Scheme's internal type system. That is to say, the system should work as before except that ordinary numbers should be represented simply as Scheme numbers rather than as pairs whose `car` is the symbol `scheme-number`.

`scheme-number` パッケージの内部手続は本質的にプリミティブな手続 `+`, `-`, その他の呼び出し以上の物ではない。言語のプリミティブを直接使用することはできない。私達のタイプタグシステムが各データオブジェクトに対し型付けられていることを要件とするためである。しかし実際には全ての Lisp 実装は型システムを持って

おり、内部にて使用している。`symbol?` や `number?` のようなプリミティブな述語はデータオブジェクトが特定の型を持つか決定する。Section 2.4.2 の `type-tag`, `contents`, and `attach-tag` の定義を変更し私達のジェネリックシステムが Scheme の内部型システムの利点を得るようにせよ。これは言い替えれば、システムは以前と同じように動作する必要があるが、ただし普通の数値はその `car` がシンボル `scheme-number` であるペアでなく、単純に Scheme の数値として表現されるようにせよ。

**Exercise 2.79:** Define a generic equality predicate `equ?` that tests the equality of two numbers, and install it in the generic arithmetic package. This operation should work for ordinary numbers, rational numbers, and complex numbers.

2つの数値の等値関係をテストするジェネリックな等値の述語 `equ?` を定義し、ジェネリック数値演算パッケージにインストールせよ。この命令は通常の数値、分数、複素数に対しても働くこと。

**Exercise 2.80:** Define a generic predicate `=zero?` that tests if its argument is zero, and install it in the generic arithmetic package. This operation should work for ordinary numbers, rational numbers, and complex numbers.

引数が 0 であるかテストするジェネリックな述語 `=zero?` を定義しジェネリック数値演算パッケージにインストールせよ。この命令は通常の数値、分数、複素数に対しても働くこと。

## 2.5.2 Combining Data of Different Types

We have seen how to define a unified arithmetic system that encompasses ordinary numbers, complex numbers, rational numbers, and any other type of number we might decide to invent, but we have ignored an important issue. The operations we have defined so far treat the different data types as being completely independent. Thus, there are separate packages for adding, say, two ordinary numbers, or two complex numbers. What we have not yet considered is the fact that it is meaningful to define operations that cross the type boundaries, such as the addition of a complex number to an ordinary number. We have gone to great pains to introduce barriers between parts of our programs so that they can be developed and understood separately. We would like to

introduce the cross-type operations in some carefully controlled way, so that we can support them without seriously violating our module boundaries.

通常の数値、複素数、分数、そして開発するだろう任意の他の型の数値を包括する統一数値演算システムをどのように定義するかについて学びました。しかし私達は重要な問題を無視してきました。今まで私達が定義した命令は異なるデータの型を完全に独立しているとして扱ってきました。従って追加すべき分かれたパッケージが、例えば2つの普通の数値や2つの複素数が存在します。私達がまだ考慮していないことは型の境界を渡る命令を定義することには意義があるという事実です。例えば複素数と実数の加算です。私達はこれまでプログラムの間にバリアを築くために大きな努力をしてきました。それが分離して開発、理解されることを可能にするためでした。私達は型を渡る命令をある程度注意深くコントロールされた手段にて導入したいと思います。そうすることで私達のモジュール境界を重大な侵害が起らないようにそれらをサポートすることができるようになります。

One way to handle cross-type operations is to design a different procedure for each possible combination of types for which the operation is valid. For example, we could extend the complex-number package so that it provides a procedure for adding complex numbers to ordinary numbers and installs this in the table using the tag (`complex scheme-number`):<sup>93</sup>

クロスタイプ(型を渡る)命令を扱う1つの方法は命令が有効な型の可能な組み合わせそれぞれに対して異なる手続を設計することです。例えば複素数パッケージを拡張し、それが複素数と実数の加算を提供し、タグ (`complex scheme-number`) を用いてテーブルにインストールするようにします。<sup>94</sup>

```
;; to be included in the complex package
(define (add-complex-to-schemenum z x)
  (make-from-real-imag (+ (real-part z) x) (imag-part z)))
(put 'add '(complex scheme-number)
  (lambda (z x) (tag (add-complex-to-schemenum z x))))
```

This technique works, but it is cumbersome. With such a system, the cost of introducing a new type is not just the construction of the package of procedures for that type but also the construction and installation of the procedures

---

<sup>93</sup>We also have to supply an almost identical procedure to handle the types (`scheme-number complex`).

<sup>94</sup>私達はまたほとんど同一の手続を型 (`scheme-number complex`) を扱うために提供しなければなりません。

that implement the cross-type operations. This can easily be much more code than is needed to define the operations on the type itself. The method also undermines our ability to combine separate packages additively, or at least to limit the extent to which the implementors of the individual packages need to take account of other packages. For instance, in the example above, it seems reasonable that handling mixed operations on complex numbers and ordinary numbers should be the responsibility of the complex-number package. Combining rational numbers and complex numbers, however, might be done by the complex package, by the rational package, or by some third package that uses operations extracted from these two packages. Formulating coherent policies on the division of responsibility among packages can be an overwhelming task in designing systems with many packages and many cross-type operations.

このテクニックはうまく行きますが、面倒です。このようなシステムでは新しい型を導入するコストはその型のための手続のパッケージを構築するだけでなく、クロスタイルの命令を実装する手續の構築とインストールに及びます。これは簡単にその型自身の命令を定義するために必要なものより多くのコードとなるでしょう。この手法はまた分かれたパッケージを付加的に接続する能力を弱めたり、最低でも個々のパッケージの実装者が他のパッケージの考慮をしなければならない範囲を制約する能力をダメにしてしまいます。例えば、上の例では複素数と実数上の混合命令の扱いが複素数パッケージの責任となるのは妥当に見えます。しかし分数と複素数の接続においては複素数パッケージで行われるかもしれないし、分数パッケージかもしれないし、これらの2つのパッケージから抽出した命令を用いる何らかの第三者パッケージかもしれません。パッケージ間の区分上における整合性のポリシーの形式化が、多くのパッケージと多くのクロスタイル命令を伴なうシステム設計において計り知れなくなってしまいます。

## Coercion

In the general situation of completely unrelated operations acting on completely unrelated types, implementing explicit cross-type operations, cumbersome though it may be, is the best that one can hope for. Fortunately, we can usually do better by taking advantage of additional structure that may be latent in our type system. Often the different data types are not completely independent, and there may be ways by which objects of one type may be viewed as being of another type. This process is called *coercion*. For example, if we are

asked to arithmetically combine an ordinary number with a complex number, we can view the ordinary number as a complex number whose imaginary part is zero. This transforms the problem to that of combining two complex numbers, which can be handled in the ordinary way by the complex-arithmetic package.

完全に依存しない型達上にて振る舞う完全に依存しない命令群が一般的な状況においては明示的にクロスタイプ命令を実装することは、面倒かもしれません、人が望む最高の物かもしれません。幸運なことに私達は通常、私達の型システム内の潜在的に存在するだろう付加的な構造の利点を用いることによりより良く行うことが可能です。時折、異なるデータの型は完全には独立しておらず、ある型のオブジェクトが他の型であるように見られる場合が複数存在するでしょう。この過程は*coercion*(強制)と呼ばれます。例えばもし私達が算術上、実数と複素数を合成するよう求められた場合に、私達は実数を虚数部が0の複素数だと見做すことができます。これはこの問題を2つの複素数の合成へと変換し、複素数パッケージにより通常の方法にて取り扱うことが可能になります。

In general, we can implement this idea by designing coercion procedures that transform an object of one type into an equivalent object of another type. Here is a typical coercion procedure, which transforms a given ordinary number to a complex number with that real part and zero imaginary part:

一般的に、ある型のオブジェクトを等価な他の型のオブジェクトに変換する強制手続を設計することでこの考えを実装することができます。以下は典型的な強制手続です。これは与えられた普通の数値(実数)を実数部とゼロである虚数部を持つ複素数に変換します。

```
(define (scheme-number->complex n)
  (make-complex-from-real-imag (contents n) 0))
```

We install these coercion procedures in a special coercion table, indexed under the names of the two types:

これらの強制手続を2つの型の名前により索引付けした特別な強制テーブルにインストールします。

```
(put-coercion 'scheme-number
              'complex
              scheme-number->complex)
```

(We assume that there are `put-coercion` and `get-coercion` procedures available for manipulating this table.) Generally some of the slots in the table will

be empty, because it is not generally possible to coerce an arbitrary data object of each type into all other types. For example, there is no way to coerce an arbitrary complex number to an ordinary number, so there will be no general `complex->scheme-number` procedure included in the table.

(このテーブルを操作するために手続 `put-coercion` と `get-coercion` が存在すると仮定します)。一般にこのテーブルの枠のいくつかは空になります。全ての型の任意のデータオブジェクトを全ての他の型へと強制することは一般には不可能です。例えば任意の複素数を実数に強制することはできません。そのため普遍的な `complex->scheme-number` 手続はテーブルに含まれることはできません。

Once the coercion table has been set up, we can handle coercion in a uniform manner by modifying the `apply-generic` procedure of Section 2.4.3. When asked to apply an operation, we first check whether the operation is defined for the arguments' types, just as before. If so, we dispatch to the procedure found in the operation-and-type table. Otherwise, we try coercion. For simplicity, we consider only the case where there are two arguments.<sup>95</sup> We check the coercion table to see if objects of the first type can be coerced to the second type. If so, we coerce the first argument and try the operation again. If objects of the first type cannot in general be coerced to the second type, we try the coercion the other way around to see if there is a way to coerce the second argument to the type of the first argument. Finally, if there is no known way to coerce either type to the other type, we give up. Here is the procedure:

強制テーブルが準備されれば、Section 2.4.3 の `apply-generic` 手続を変更することで統一的な作法で強制を取り扱うことができます。命令を適用するよう求められた時、最初にその命令が引数の型に対して定義されているかどうかを以前と同様にチェックします。もしそうであれば命令と型のテーブルで見つかった手続を呼び出します。そうでなければ強制を試みます。単純化のために、2つの引数を伴う場合のみについて考えることにします。<sup>96</sup> 強制テーブルをチェックし、最初の型のオブジェクトが2つ目の型に強制できるか確認します。もしそうであれば、最初の引数を強制し、命令の試行を再び行います。もし最初の型のオブジェクトが一般に2つ目の型に強制できない場合、逆に2つ目の引数を1つ目の引数の型に強制できるか試します。最後にどちらの型も他方の型に強制できない場合、諦めます。以下がこの手続です。

---

<sup>95</sup> See Exercise 2.82 for generalizations.

<sup>96</sup> 一般化についてはExercise 2.82を参照して下さい。

```

(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((type1 (car type-tags))
                    (type2 (cadr type-tags))
                    (a1 (car args))
                    (a2 (cadr args)))
                (let ((t1->t2 (get-coercion type1 type2))
                      (t2->t1 (get-coercion type2 type1)))
                  (cond (t1->t2
                           (apply-generic op (t1->t2 a1) a2))
                        (t2->t1
                           (apply-generic op a1 (t2->t1 a2)))
                        (else (error "No method for these types"
                                     (list op type-tags))))))
              (error "No method for these types"
                     (list op type-tags)))))))

```

This coercion scheme has many advantages over the method of defining explicit cross-type operations, as outlined above. Although we still need to write coercion procedures to relate the types (possibly  $n^2$  procedures for a system with  $n$  types), we need to write only one procedure for each pair of types rather than a different procedure for each collection of types and each generic operation.<sup>97</sup> What we are counting on here is the fact that the appropriate transformation between types depends only on the types themselves, not on the operation to be applied.

この強制スキームは上で概説された様に明示的なクロスタイプ命令の定義手法上に多くの利点を持ちます。私達は依然、型に関する強制手続を書かねばなりませんが( $n$  個の型のシステムに対し場合により  $n^2$  の手続)、全ての型

---

<sup>97</sup>If we are clever, we can usually get by with fewer than  $n^2$  coercion procedures. For instance, if we know how to convert from type 1 to type 2 and from type 2 to type 3, then we can use this knowledge to convert from type 1 to type 3. This can greatly decrease the number of coercion procedures we need to supply explicitly when we add a new type to the system. If we are willing to build the required amount of sophistication into our system, we can have it search the “graph” of relations among types and automatically generate those coercion procedures that can be inferred from the ones that are supplied explicitly.

の集合と各ジェネリック命令に対し異なる手続を書くのではなく、型のペア 1 組につき 1 つの手続を書くだけで済みます。<sup>98</sup> ここで私達が信頼しているものはタイプ間の適切な変換は型それ自身のみに依存し、適用される命令には依存しないという事実です。

On the other hand, there may be applications for which our coercion scheme is not general enough. Even when neither of the objects to be combined can be converted to the type of the other it may still be possible to perform the operation by converting both objects to a third type. In order to deal with such complexity and still preserve modularity in our programs, it is usually necessary to build systems that take advantage of still further structure in the relations among types, as we discuss next.

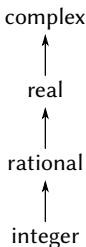
一方で、私達の強制スキームが十分に汎用ではないアプリケーションが存在するかもしれません。たとえ合成されるオブジェクトの両方ともが他方に変換できないとしても両者を第三の型に変換することで命令を実行することが可能になるかもしれません。そのような複雑さに対処するため、そしてそれでもプログラムのモジュール方式を維持するために、通常はより一層タイプ間の関係の構造の利点を得るシステムを構築することが、次で議論するように必要です。

## Hierarchies of types

The coercion scheme presented above relied on the existence of natural relations between pairs of types. Often there is more “global” structure in how the different types relate to each other. For instance, suppose we are building a generic arithmetic system to handle integers, rational numbers, real numbers, and complex numbers. In such a system, it is quite natural to regard an integer as a special kind of rational number, which is in turn a special kind of real number, which is in turn a special kind of complex number. What we actually have is a so-called *hierarchy of types*, in which, for example, integers are a *subtype* of rational numbers (i.e., any operation that can be applied to a rational

---

<sup>98</sup>もし私達が賢いならば普通は  $n^2$  よりも少ない強制手続で済みます。例えばもし型 1 から型 2 への変換方法と型 2 から型 3 への変換方法を知っている場合、この知識を用いて型 1 から型 3 へ変換することができます。これはシステムに新しい型を追加する時に明示的に提供せねばならない強制手続の数を劇的に減らします。もしシステムに必要なだけの洗練を組み入れたいのなら、システムにタイプ間の“グラフ”を検索させて自動的に明示的に提供された物から推論可能な強制手続を生成させることができます。



**Figure 2.25:** A tower of types.

number can automatically be applied to an integer). Conversely, we say that rational numbers form a *supertype* of integers. The particular hierarchy we have here is of a very simple kind, in which each type has at most one supertype and at most one subtype. Such a structure, called a *tower*, is illustrated in Figure 2.25.

上で展開された強制スキームは型のペアの間の自然な関係の存在に当てにしていました。より“一般的な”構造が、異なる型のお互いへの関係の仕方には良く存在します。例えば、私達が整数、分数、実数、複素数を扱う一般的な数値演算システムを構築していると仮定します。そのようなシステムにおいては整数を特別な種類の分数として見做すことはとても自然です。分数は同様に特別な種類の実数であり、実数は同様に特別な種類の複素数であります。私達が実際に手にしている物は*hierarchy of types*(型の階層) 呼ばれるもので、その中では例えば整数は分数の *subtype*(サブタイプ) です(すなわち分数に適用できる任意の命令は自動的に整数に適用できます)。逆に分数は整数の *supertype*(スーパー タイプ、親の型) と呼びます。今手にした階層はとても単純な種類で、各型はたかだか1つのスーパー タイプを持ち、たかだか1つのサブ タイプを持ちます。そのような構造は *tower*(タワー、塔) と呼ばれFigure 2.25で示されます。If we have a tower structure, then we can greatly simplify the problem of adding a new type to the hierarchy, for we need only specify how the new type is embedded in the next supertype above it and how it is the supertype of the type below it. For example, if we want to add an integer to a complex number, we need not explicitly define a special coercion procedure `integer->complex`. Instead, we define how an integer can be transformed into a rational number, how a rational number is transformed into a real number, and how a

real number is transformed into a complex number. We then allow the system to transform the integer into a complex number through these steps and then add the two complex numbers.

もしタワー構造を持つ場合、階層に新しい型を追加する問題を著しく単純化できます。新しい型がどのようにその上のスーパータイプの隣に組込まれるか、そしてどのようにその型がその下の型に対してスーパータイプであるかを指定するだけです。例えばもし複素数に対して整数を追加したい場合、明示的に特別な強制手続 `integer->complex` を定義する必要はありません。その代わりに整数がどのように分数に変換できるか、分数がどのように実数に変換できるか、実数がどのように複素数に変換できるかを定義します。そうしたらシステムに整数を複素数に変換することをこれらのステップを通して変換することを許可し、次に 2 つの複素数を加算します。

We can redesign our `apply-generic` procedure in the following way: For each type, we need to supply a `raise` procedure, which “raises” objects of that type one level in the tower. Then when the system is required to operate on objects of different types it can successively raise the lower types until all the objects are at the same level in the tower. ([Exercise 2.83](#) and [Exercise 2.84](#) concern the details of implementing such a strategy.)

`apply-generic` 手続を以下のように再設計することもできます。各型に対して `raise` 手続を与える必要があります。これはある型のオブジェクトをタワーにおいて 1 レベル上げます。そうすればシステムが異なる型のオブジェクト上にて操作する必要がある時、全てのオブジェクトがタワー内にて同じレベルになるまで連続して上げることができます ([Exercise 2.83](#)と[Exercise 2.84](#)がそのような戦略の実装の詳細について考察しています)。

Another advantage of a tower is that we can easily implement the notion that every type “inherits” all operations defined on a supertype. For instance, if we do not supply a special procedure for finding the real part of an integer, we should nevertheless expect that `real-part` will be defined for integers by virtue of the fact that integers are a subtype of complex numbers. In a tower, we can arrange for this to happen in a uniform way by modifying `apply-generic`. If the required operation is not directly defined for the type of the object given, we raise the object to its supertype and try again. We thus crawl up the tower, transforming our argument as we go, until we either find a level at which the desired operation can be performed or hit the top (in which case we give up).

タワーの別の利点には全ての型がスーパータイプ上に定義された全ての命令を“継承”する概念を簡単に実装できることが上げられます。例えばもし整

数の実数部を求めるための特別な手続を提供しない場合、それにもかかわらず整数は複素数のサブタイプであるという事実のおかげで、整数のための `real-part` が定義されることが期待できます。タワーでは `apply-generic` を変更するという統一的な方法でこの様なことが起こるよう準備することが可能です。もし必要な命令が与えられたオブジェクトの型のために直接定義されていない場合、オブジェクトをそのスーパーイプに上げることで再試行できます。従ってタワーを這い上がりながら望まれた命令が実行可能になるまで引数を変換するか、頂上まで辿りついてそこで諦めることができます。

Yet another advantage of a tower over a more general hierarchy is that it gives us a simple way to “lower” a data object to the simplest representation. For example, if we add  $2 + 3i$  to  $4 - 3i$ , it would be nice to obtain the answer as the integer 6 rather than as the complex number  $6 + 0i$ . Exercise 2.85 discusses a way to implement such a lowering operation. (The trick is that we need a general way to distinguish those objects that can be lowered, such as  $6 + 0i$ , from those that cannot, such as  $6 + 2i$ .)

別のより一般的な階層に比べた場合、もう1つタワーの利点はデータオブジェクトをより簡単な表現へ“下げる”簡単な方法を提供することです。例えば  $2 + 3i$  を  $4 - 3i$  に足した場合、その答は複素数  $6 + 0i$  よりも整数 6 で得るほうがより良いと言えるでしょう。Exercise 2.85はそのようなレベルを下げる命令の実装について議論します。(この仕掛けには  $6 + 0i$  のような階層のレベルを下げられるオブジェクトを  $6 + 2i$  のような下げられないオブジェクトから見分ける一般的な方法が必要です)。

## Inadequacies of hierarchies

If the data types in our system can be naturally arranged in a tower, this greatly simplifies the problems of dealing with generic operations on different types, as we have seen. Unfortunately, this is usually not the case. Figure 2.26 illustrates a more complex arrangement of mixed types, this one showing relations among different types of geometric figures. We see that, in general, a type may have more than one subtype. Triangles and quadrilaterals, for instance, are both subtypes of polygons. In addition, a type may have more than one supertype. For example, an isosceles right triangle may be regarded either as an isosceles triangle or as a right triangle. This multiple-supertypes issue is particularly thorny, since it means that there is no unique way to “raise” a type in the hierarchy. Finding the “correct” supertype in which to apply an

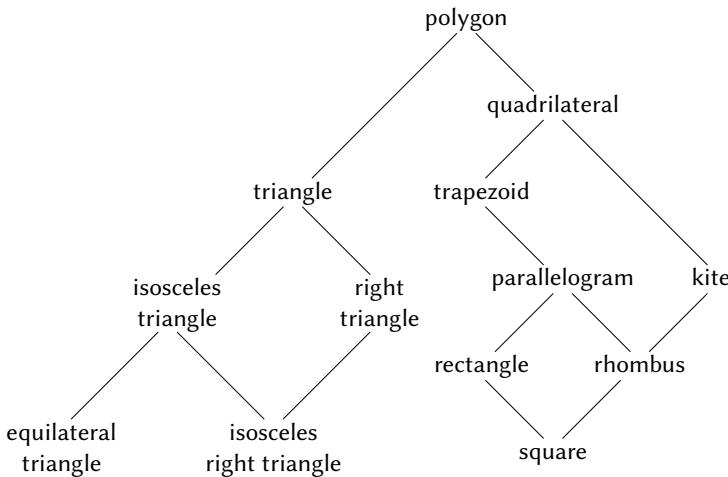
operation to an object may involve considerable searching through the entire type network on the part of a procedure such as `apply-generic`. Since there generally are multiple subtypes for a type, there is a similar problem in coercing a value “down” the type hierarchy. Dealing with large numbers of interrelated types while still preserving modularity in the design of large systems is very difficult, and is an area of much current research.<sup>99</sup>

もしシステムのデータの型が自然にタワーに配置できる場合、ここまで見てきた通りに、異なる型上のジェネリック命令の取扱の問題を著しく単純化できます。残念なことに、これは普通の場合ではありません。[Figure 2.26](#) は複数の型より複雑な配置を図示しています。この図は幾何学的図形の異なる型の間の関係を見せてています。一般的に1つの型が複数のサブタイプを持つことがわかります。例えば三角形と四角形は共に多角形のサブタイプです。加えてある型は複数のスーパー型を持つことがあります。例えば二等辺直角三角形は二等辺三角形、または直角三角形と見做すことができます。この複数スーパー型問題は特に困難で、階層内において型を“上げる”单一の方法が存在しません。オブジェクトに命令を適用するため“正しい”スーパー型を求めるることは `apply-generic` の様な手続に不可欠な型ネットワーク全体を通しての多大な検索を巻き起す可能性があります。一般的にある型に対して複数のサブタイプが存在するので値に対し型階層を“下げる”強制にも同様の問題が存在します。巨大システムの設計におけるモジュール化方式をそれでも維持しながら多くの数の相互に関係する型の取り扱うことはとても難しく、現在の多くの研究領域です。<sup>100</sup>

---

<sup>99</sup>This statement, which also appears in the first edition of this book, is just as true now as it was when we wrote it twelve years ago. Developing a useful, general framework for expressing the relations among different types of entities (what philosophers call “ontology”) seems intractably difficult. The main difference between the confusion that existed ten years ago and the confusion that exists now is that now a variety of inadequate ontological theories have been embodied in a plethora of correspondingly inadequate programming languages. For example, much of the complexity of object-oriented programming languages—and the subtle and confusing differences among contemporary object-oriented languages—centers on the treatment of generic operations on interrelated types. Our own discussion of computational objects in [Chapter 3](#) avoids these issues entirely. Readers familiar with object-oriented programming will notice that we have much to say in chapter 3 about local state, but we do not even mention “classes” or “inheritance.” In fact, we suspect that these problems cannot be adequately addressed in terms of computer-language design alone, without also drawing on work in knowledge representation and automated reasoning.

<sup>100</sup>第一版でも存在したこの文は12年前と同じく今も変わりません。実用的目的



**Figure 2.26:** Relations among types of geometric figures.

**Exercise 2.81:** Louis Reasoner has noticed that `apply-generic` may try to coerce the arguments to each other's type even if they already have the same type. Therefore, he reasons, we need to put procedures in the coercion table to *coerce* arguments of each type to their own type. For example, in addition to the `scheme-number-`

---

異なる型の要素間の関係の表現するフレームワーク（哲学者が“オントロジー”（存在論）と呼ぶもの）を開発することは不可能に見えるほど難しいことです。10年前に存在した混乱と現在に存在する混乱との違いは、種々の不適切な存在論上の理論が、相応して不適切なプログラミング言語に過剰に組込まれていることです。例えばオブジェクト指向言語の複雑性の多くは—そして現在のオブジェクト指向言語間の微妙で混乱させる違いは—相互に関係する型上のジェネリック命令の扱いを中心とします。[Chapter 3](#)での私達自身による計算オブジェクトの議論はこれらの問題を完全に避けます。オブジェクト指向言語に親しみのある読者は[Chapter 3](#)においてローカルの状態について多くの触れるべきことが存在すると気付くでしょう。しかし私達は“クラス”や“継承”についてさえ述べることはしません。実際に私達はこれらの問題が知識表現上の成果の利用と自動的な推論無しにコンピュータ言語設計のみで適切に解決されることは無いと疑っています。

>complex coercion shown above, he would do:

Louis Reasoner は apply-generic が引数に対しそれらが既に同じ型であってもお互いの型に強制を試行することに気付いた。そのため彼は強制テーブルに各型の引数をそれら自身の型に coerce(強制)するための手続を追加する必要があるのではないかと “reason”(推論) した。

```
(define (scheme-number->scheme-number n) n)
(define (complex->complex z) z)
(put-coercion 'scheme-number
              'scheme-number
              scheme-number->scheme-number)
(put-coercion 'complex 'complex complex->complex)
```

a With Louis's coercion procedures installed, what happens if apply-generic is called with two arguments of type scheme-number or two arguments of type complex for an operation that is not found in the table for those types? For example, assume that we've defined a generic exponentiation operation:

Louis の強制手続がインストールされると apply-generic が命令に対する 2 つの引数の型が scheme-number、または 2 つの引数の型が complex でありそれらの型に対する命令がテーブルに存在しない場合何が起こるだろうか? 例えばジェネリックな指数関数命令を定義したとしよう。

```
(define (exp x y) (apply-generic 'exp x y))
```

and have put a procedure for exponentiation in the Scheme-number package but not in any other package:

そして Scheme-number パッケージの指数関数に対する手続を追加したとする。ただし他の型に関しては全て行わない。

```
; following added to Scheme-number package
(put 'exp '(scheme-number scheme-number)
      (lambda (x y) (tag (expt x y))))
; using primitive expt
```

What happens if we call `exp` with two complex numbers as arguments?

2つの複素数引数により `exp` を呼び出した場合、何が起こるだろうか？

- b Is Louis correct that something had to be done about coercion with arguments of the same type, or does `apply-generic` work correctly as is?

同じ型の引数に伴なう強制に関して何かが行われるべきかについて Louis は正しいだろうか？それとも `apply-generic` はそのままで正しく動作するだろうか？

- c Modify `apply-generic` so that it doesn't try coercion if the two arguments have the same type.

`apply-generic` を変更し 2つの引数が同じ型である場合に強制を試行しないようにせよ。

**Exercise 2.82:** Show how to generalize `apply-generic` to handle coercion in the general case of multiple arguments. One strategy is to attempt to coerce all the arguments to the type of the first argument, then to the type of the second argument, and so on. Give an example of a situation where this strategy (and likewise the two-argument version given above) is not sufficiently general. (Hint: Consider the case where there are some suitable mixed-type operations present in the table that will not be tried.)

`apply-generic` を複数引数の全体的な場合に強制を扱わせるような一般化を行わせるにはどのように行うかを示せ。1つの戦略として全ての引数を最初の引数の型に強制するよう試行し、次に 2つ目、以降繰り返しが上げられる。この戦略（と上で与えられた 2引数版が同様に）全体には不十分である例を示せ。（ヒント：テーブルにいくつか適切な型が混ざった命令が存在し、それが試行されない場合について考えよ。）

**Exercise 2.83:** Suppose you are designing a generic arithmetic system for dealing with the tower of types shown in Figure 2.25: integer, rational, real, complex. For each type (except complex), design a procedure that raises objects of that type one level in the tower.

Show how to install a generic `raise` operation that will work for each type (except complex).

Figure 2.25で示される型のタワーを取り扱うジェネリックな数値演算システムの設計を行っているとする。整数、分数、実数、複素数に対応する。各型(複素数を除く)に対してその型のオブジェクトをタワー内にて1レベル上げる手続を設計せよ。(複素数を除く)各型に対し動作するジェネリックな `raise` 命令をどのようにインストールするか示せ。

**Exercise 2.84:** Using the `raise` operation of Exercise 2.83, modify the `apply-generic` procedure so that it coerces its arguments to have the same type by the method of successive raising, as discussed in this section. You will need to devise a way to test which of two types is higher in the tower. Do this in a manner that is “compatible” with the rest of the system and will not lead to problems in adding new levels to the tower.

Exercise 2.83の `raise` 命令を用いて `apply-generic` 手続を変更し、複数の引数が一連の“上げる”動作を行うことによりこの節で語られた様に同じ型を持つようにせよ。2つの型のどちらがタワー内にてより高いレベルであるかテストする手段を開発する必要がある。これを残りのシステムと“互換性”を保ち、タワーに新しいレベルを追加する場合にも問題が無いような手段で行え。

**Exercise 2.85:** This section mentioned a method for “simplifying” a data object by lowering it in the tower of types as far as possible. Design a procedure `drop` that accomplishes this for the tower described in Exercise 2.83. The key is to decide, in some general way, whether an object can be lowered. For example, the complex number  $1.5 + 0i$  can be lowered as far as `real`, the complex number  $1 + 0i$  can be lowered as far as `integer`, and the complex number  $2 + 3i$  cannot be lowered at all. Here is a plan for determining whether an object can be lowered: Begin by defining a generic operation `project` that “pushes” an object down in the tower. For example, projecting a complex number would involve throwing away the imaginary part. Then a number can be dropped if, when we `project` it and `raise` the result back to the type we started with, we end up with something equal to what we started

with. Show how to implement this idea in detail, by writing a `drop` procedure that drops an object as far as possible. You will need to design the various projection operations<sup>101</sup> and install `project` as a generic operation in the system. You will also need to make use of a generic equality predicate, such as described in [Exercise 2.79](#). Finally, use `drop` to rewrite `apply-generic` from [Exercise 2.84](#) so that it “simplifies” its answers.

この節では可能な限りタワー内の型レベルを下げるることによりデータオブジェクトの“単純化”を行う手段について説明した。[Exercise 2.83](#)に記述されたタワーに対しこれを達成する手続 `drop` を設計せよ。いくつかの一般的な方法の中から決定する鍵は、オブジェクトを下げることができるかどうかである。例えば複素数  $1.5 + 0i$  は `real(実数)` である限り下げられ、複素数  $1 + 0i$  は `integer(整数)` である限り下げることができ、複素数  $2 + 3i$  は下げることが絶対にできない。以下に、あるオブジェクトが下げることができるか決定する計画を示す。オブジェクトをタワー内にて“押し下げる”ジェネリックな命令 `project(射影)` を定義することから始める。例えば複素数の射影は虚数部を捨てることになる。すると数値は `project` した結果を元の型に `raise(上げ)` た時に開始した時点と同じ値になれば `drop(落とす)` ことができることになる。可能な場合にオブジェクトを落とす手続 `drop` を書くことで、この考えをどのように実装するか詳細に示せ。色々な射影命令を設計し、ジェネリックな命令として `project` をシステム内にインストールする必要がある。<sup>102</sup> また[Exercise 2.79](#)で説明した等値関係のジェネリックな述語を利用する必要もある。最後に `drop` を用いて[Exercise 2.84](#)の `apply-generic` を書き直し解答を“単純化”する。

**Exercise 2.86:** Suppose we want to handle complex numbers whose real parts, imaginary parts, magnitudes, and angles can be either ordinary numbers, rational numbers, or other numbers we might wish to add to the system. Describe and implement the changes to the system needed to accommodate this. You will have to define

---

<sup>101</sup>A real number can be projected to an integer using the `round` primitive, which returns the closest integer to its argument.

<sup>102</sup>実数は引数に最も近い整数を返すプリミティブ `round` を用いて整数に射影することができる。

operations such as `sine` and `cosine` that are generic over ordinary numbers and rational numbers.

実数部、虚数部、大きさ、角度が通常の数値、分数、またはシステムに追加したくなるかもしれない数のどれかを用いることができる複素数を扱えるようにしたいとする。これを達成するために必要なシステムに対する変更を説明し、実装せよ。普通の数と分数に対してジェネリックな `sine` や `cosine` のような命令を定義する必要が出てくるであろう。

### 2.5.3 Example: Symbolic Algebra

The manipulation of symbolic algebraic expressions is a complex process that illustrates many of the hardest problems that occur in the design of large-scale systems. An algebraic expression, in general, can be viewed as a hierarchical structure, a tree of operators applied to operands. We can construct algebraic expressions by starting with a set of primitive objects, such as constants and variables, and combining these by means of algebraic operators, such as addition and multiplication. As in other languages, we form abstractions that enable us to refer to compound objects in simple terms. Typical abstractions in symbolic algebra are ideas such as linear combination, polynomial, rational function, or trigonometric function. We can regard these as compound “types,” which are often useful for directing the processing of expressions. For example, we could describe the expression

記号代数表現の操作は巨大なスケールのシステムの設計において起こり得る最も困難な問題の多くを説明する複雑な処理です。代数表現は一般的に階層構造であると見ることができ、演算子の木がオペランドに適用されます。代数表現を定数と変数のようなプリミティブなオブジェクトの集合から始めて、これらを加算や乗算のような代数演算子を用いて接続することで構築することができます。他の言語と同様に、複合オブジェクトに簡単な用語で参照することを可能にするための抽象化を形式化します。典型的な記号代数における抽象化は線形結合、多項式、有理関数、三角関数のような考えです。これらを式の処理を方向付けするのによく便利である複合“型”と見做すことができます。例えば私達は以下の式を

$$x^2 \sin(y^2 + 1) + x \cos 2y + \cos(y^3 - 2y^2)$$

as a polynomial in  $x$  with coefficients that are trigonometric functions of polynomials in  $y$  whose coefficients are integers.

係数を伴なう  $x$  の多項式と係数が整数である  $y$  の三角関数として記述することができます。

We will not attempt to develop a complete algebraic-manipulation system here. Such systems are exceedingly complex programs, embodying deep algebraic knowledge and elegant algorithms. What we will do is look at a simple but important part of algebraic manipulation: the arithmetic of polynomials. We will illustrate the kinds of decisions the designer of such a system faces, and how to apply the ideas of abstract data and generic operations to help organize this effort.

私達は完全な代数操作システムをここで開発しようとはしません。そのようなシステムは非常に複雑なプログラムであり、深い代数学の知識と洗練されたアルゴリズムを具体化する必要があります。私達が行うのは代数操作の単純だが重要な部分について考えること、つまり多項式の演算です。そのようなシステムの設計者が直面する決定すべきことや、この試みのまとめを手助けするために抽象データやジェネリックな命令をどのようにして適用するかのような事柄について説明します。

## Arithmetic on polynomials

Our first task in designing a system for performing arithmetic on polynomials is to decide just what a polynomial is. Polynomials are normally defined relative to certain variables (the *indeterminates* of the polynomial). For simplicity, we will restrict ourselves to polynomials having just one indeterminate (*univariate polynomials*).<sup>103</sup> We will define a polynomial to be a sum of terms, each of which is either a coefficient, a power of the indeterminate, or a product of a coefficient and a power of the indeterminate. A coefficient is defined as an algebraic expression that is not dependent upon the indeterminate of the polynomial. For example,

多項式上の数値演算を実行するシステムの設計における最初のタスクは多項式とは何かを決定することです。多項式は通常いくつかの変数 (多項式

---

<sup>103</sup>On the other hand, we will allow polynomials whose coefficients are themselves polynomials in other variables. This will give us essentially the same representational power as a full multivariate system, although it does lead to coercion problems, as discussed below.

の*indeterminates*(不定元)) に関連して定義されます。簡単にするために多項式はただ 1 つの不定元 (*univariate polynomials*(一変数多項式)) に制約します。<sup>104</sup> 多項式とは項の和であり、各項は係数、不定元の累乗数、または係数と不定元の累乗数の積であると定義します。係数は多項式の不定元に依存しない代数表現であると定義します。例えば、

$$5x^2 + 3x + 7$$

is a simple polynomial in  $x$ , and

は簡単な  $x$  の多項式であり、

$$(y^2 + 1)x^3 + (2y)x + 1$$

is a polynomial in  $x$  whose coefficients are polynomials in  $y$ .

は係数が  $y$  の多項式である  $x$  の多項式です。

Already we are skirting some thorny issues. Is the first of these polynomials the same as the polynomial  $5y^2 + 3y + 7$ , or not? A reasonable answer might be “yes, if we are considering a polynomial purely as a mathematical function, but no, if we are considering a polynomial to be a syntactic form.” The second polynomial is algebraically equivalent to a polynomial in  $y$  whose coefficients are polynomials in  $x$ . Should our system recognize this, or not? Furthermore, there are other ways to represent a polynomial—for example, as a product of factors, or (for a univariate polynomial) as the set of roots, or as a listing of the values of the polynomial at a specified set of points.<sup>105</sup> We can finesse these questions by deciding that in our algebraic-manipulation system a “polynomial” will be a particular syntactic form, not its underlying mathematical meaning.

既にいくつかの困難な問題を回避しています。これらの多項式の最初の物は多項式  $5y^2 + 3y + 7$  と同じかそれとも異なるでしょうか? 妥当な答は“多項

---

<sup>104</sup>一方で係数は別の変数にてそれ自身が多項式であることを許可します。これにより本質的に完全に多変量システムと同じ表現力を得ますが、強制においてこの先で記述される問題が発生します。

<sup>105</sup>For univariate polynomials, giving the value of a polynomial at a given set of points can be a particularly good representation. This makes polynomial arithmetic extremely simple. To obtain, for example, the sum of two polynomials represented in this way, we need only add the values of the polynomials at corresponding points. To transform back to a more familiar representation, we can use the Lagrange interpolation formula, which shows how to recover the coefficients of a polynomial of degree  $n$  given the values of the polynomial at  $n + 1$  points.

式を純粹に数学の関数であると考えれば答は YES です。しかもしも多項式を文法上の形式であると考えれば答は NO です”となるでしょう。。2つ目の多項式は代数学的に係数が  $x$  の多項式である  $y$  の多項式に等価です。私達のシステムはこれを認識するべきでしようか？さらに他にも多項式を表現する方法は存在します—例えば因数の積としてや(1変数多項式に対しては)累乗根の集合として、また指定した点の集合における多項式の値の列挙として。<sup>106</sup> これらの問題を私達の数値演算操作システムにおいて、根底にある数学上の意味でなく、“多項式”が特定の文法形式であることを決定することでうまく行うことができます。

Now we must consider how to go about doing arithmetic on polynomials. In this simple system, we will consider only addition and multiplication. Moreover, we will insist that two polynomials to be combined must have the same indeterminate.

さて、多項式上で数値演算を行なうことについてどのように進めるか考えねばなりません。この簡単なシステムでは加算と乗算についてのみしか考えません。さらに接続される 2つの多項式は同じ不定元を持たなければならないとします。

We will approach the design of our system by following the familiar discipline of data abstraction. We will represent polynomials using a data structure called a *poly*, which consists of a variable and a collection of terms. We assume that we have selectors `variable` and `term-list` that extract those parts from a *poly* and a constructor `make-poly` that assembles a *poly* from a given variable and a term list. A variable will be just a symbol, so we can use the `same-variable?` procedure of Section 2.3.2 to compare variables. The following procedures define addition and multiplication of *polys*:

私達のシステムの設計はデータ抽象化にて馴染のある規律に従うことで取り組みます。多項式を *poly* と呼ぶ新しいデータ構造を用いて表現します。*poly* は変数と項の係数により構成されます。*poly* からそれらの部分を抽出するセレクタ `variable` と `term-list` と与えられた変数と項のリストから *poly* を組み上げるコンストラクタ `make-poly` が既にあると仮定します。変数はただのシン

<sup>106</sup> 1変数多項式に対しては与えられた点の集合における多項式の値を与えることは特に良い表現に成り得ます。これは多項式数値演算をとても簡単にすることができます。例としてこの方法で表現された 2つの多項式の和を求めるには相対する点の多項式の当たいを足すだけで済みます。より親しみ易い表現に戻すには  $n+1$  個の点における多項式の値を与えた場合に  $n$  次の多項式の係数を取り戻すラグランジュ補完公式を用いることができます。

ボルでありSection 2.3.2の `same-variable?` 手続を用いて変数の比較が可能です。以下の手続は `poly` の加算と乗算を定義します。

```
(define (add-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (add-terms (term-list p1)
                            (term-list p2)))
      (error "Polys not in same var: ADD-POLY"
            (list p1 p2)))))

(define (mul-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (mul-terms (term-list p1)
                            (term-list p2)))
      (error "Polys not in same var: MUL-POLY"
            (list p1 p2))))
```

To incorporate polynomials into our generic arithmetic system, we need to supply them with type tags. We'll use the tag `polynomial`, and install appropriate operations on tagged polynomials in the operation table. We'll embed all our code in an installation procedure for the polynomial package, similar to the ones in Section 2.5.1:

多項式を私達の数値演算システムに組込むためにはそれらをタイプタグと共に提供する必要があります。タグ `polynomial` を用いることにし、タグ付き多項式上の適切な命令を命令テーブルにインストールします。Section 2.5.1と同様に、多項式パッケージに対するインストール手続に私達の全てのコードを組み込んでしまうことにします。

```
(define (install-polynomial-package)
  ;; 内部手続
  ;; poly の表現
  (define (make-poly variable term-list)
    (cons variable term-list))
  (define (variable p) (car p))
  (define (term-list p) (cdr p))
  (procedures same-variable? and variable? from section 2.3.2)
```

```

;; 項と項のリストの表現
⟨procedures adjoin-term ... coeff from text below⟩

(define (add-poly p1 p2) ...)
⟨add-poly で使用される手続⟩
(define (mul-poly p1 p2) ...)
⟨mul-poly で使用される手続⟩

;; システムの残りへのインターフェイス
(define (tag p) (attach-tag 'polynomial p))
(put 'add '(polynomial polynomial)
     (lambda (p1 p2) (tag (add-poly p1 p2))))
(put 'mul '(polynomial polynomial)
     (lambda (p1 p2) (tag (mul-poly p1 p2))))
(put 'make 'polynomial
     (lambda (var terms)
       (tag (make-poly var terms))))
'done)

```

Polynomial addition is performed termwise. Terms of the same order (i.e., with the same power of the indeterminate) must be combined. This is done by forming a new term of the same order whose coefficient is the sum of the coefficients of the addends. Terms in one addend for which there are no terms of the same order in the other addend are simply accumulated into the sum polynomial being constructed.

多項式の加算は項別に実行されます。同じ次数の項(つまり同じ指数の不定元)が合成されねばなりません。これは係数は加数の係数の合計である同じ次数の新しい項を形成することにより行われます。ある加数の項の同じ次数の項がもう一方に無い場合には単純に構築される和の多項式に積み上げられます。

In order to manipulate term lists, we will assume that we have a constructor `the-empty-termlist` that returns an empty term list and a constructor `adjoin-term` that adjoins a new term to a term list. We will also assume that we have a predicate `empty-termlist?` that tells if a given term list is empty, a selector `first-term` that extracts the highest-order term from a term list, and a selector `rest-terms` that returns all but the highest-order term. To manipulate terms, we will suppose that we have a constructor `make-term` that constructs

a term with given order and coefficient, and selectors `order` and `coeff` that return, respectively, the order and the coefficient of the term. These operations allow us to consider both terms and term lists as data abstractions, whose concrete representations we can worry about separately.

項のリストを操作するために、空の項リストを返すコンストラクタ `the-empty-termlist` と新しい項を項リストに挿入するコンストラクタ `adjoin-term` を既に持っていると仮定します。また与えられた項リストが空であるか判断する述語 `empty-termlist?` と項リストから最大次数の項を抽出するセレクタ `first-term`、最大次数の項を除く全てを返すセレクタ `rest-terms` もまた持っていると仮定します。項を操作するために、与えられた次数と係数から項を構築するコンストラクタ `make-term` と項の次数と係数をそれぞれ返すセレクタ `order` と `coeff` を既に持っていると仮定します。これらの命令は項と項のリストの両方を実際の表現については分離して考えられるデータ抽象として捉えることを許します。

Here is the procedure that constructs the term list for the sum of two polynomials:<sup>107</sup>

以下は2つの多項式の和のために項リストを構築する手続です。<sup>108</sup>

```
(define (add-terms L1 L2)
  (cond ((empty-termlist? L1) L2)
        ((empty-termlist? L2) L1)
        (else
          (let ((t1 (first-term L1))
                (t2 (first-term L2)))
            (cond ((> (order t1) (order t2))
                  (adjoin-term
                    t1 (add-terms (rest-terms L1) L2)))
                  ((< (order t1) (order t2))
                  (adjoin-term
                    t2 (add-terms L1 (rest-terms L2)))))
```

---

<sup>107</sup>This operation is very much like the ordered `union-set` operation we developed in [Exercise 2.62](#). In fact, if we think of the terms of the polynomial as a set ordered according to the power of the indeterminate, then the program that produces the term list for a sum is almost identical to `union-set`.

<sup>108</sup>この命令は[Exercise 2.62](#)にて開発した `union-set` 命令にとても似ています。実際にもし多項式の項を不定元の指数に従い並べた集合だと考えるなら、和のために項リストを生成するプログラムは `union-set` とほとんど同じです。

```

(else
  (adjoin-term
    (make-term (order t1)
               (add (coeff t1) (coeff t2)))
    (add-terms (rest-terms L1)
               (rest-terms L2))))))))

```

The most important point to note here is that we used the generic addition procedure `add` to add together the coefficients of the terms being combined. This has powerful consequences, as we will see below.

ここで注意すべき最も重要な点はジェネリックな加算手続 `add` を用いて合成される 2 つの項の係数を一緒に足したことです。これは以下で見るように強力な帰結です。

In order to multiply two term lists, we multiply each term of the first list by all the terms of the other list, repeatedly using `mul-term-by-all-terms`, which multiplies a given term by all terms in a given term list. The resulting term lists (one for each term of the first list) are accumulated into a sum. Multiplying two terms forms a term whose order is the sum of the orders of the factors and whose coefficient is the product of the coefficients of the factors:

2 つの項リストを乗算するために最初のリストの各項をもう一方のリストの全ての項で乗算するのに繰り返し `mul-term-by-all-terms` を使用します。`mul-term-by-all-terms` は与えられた項を全ての与えられた項リストの項で乗算します。結果の項リスト (最初のリストの各項に対して 1 つ) は合計に積み上げられます。2 つの項の乗算は次数が乗数の次数の和で係数が乗数の係数の積となる項を形成します。

```

(define (mul-terms L1 L2)
  (if (empty-termlist? L1)
      (the-empty-termlist)
      (add-terms (mul-term-by-all-terms (first-term L1) L2)
                 (mul-terms (rest-terms L1) L2)))))

(define (mul-term-by-all-terms t1 L)
  (if (empty-termlist? L)
      (the-empty-termlist)
      (let ((t2 (first-term L)))
        (adjoin-term
          (make-term (+ (order t1) (order t2))

```

```
(mul (coeff t1) (coeff t2))
(mul-term-by-all-terms t1 (rest-terms L))))
```

This is really all there is to polynomial addition and multiplication. Notice that, since we operate on terms using the generic procedures `add` and `mul`, our polynomial package is automatically able to handle any type of coefficient that is known about by the generic arithmetic package. If we include a coercion mechanism such as one of those discussed in [Section 2.5.2](#), then we also are automatically able to handle operations on polynomials of different coefficient types, such as

これが本当に多項式の和と積のためにあるもの全てです。ジェネリック手続 `add` と `mul` を用いて項を操作するため、多項式パッケージはジェネリック数値演算パッケージにより知られている任意の型の係数を取り扱うことが可能であることに注意して下さい。もし [Section 2.5.2](#)で議論されたような強制メカニズムを含めていた場合、型の異なる係数の多項式上でも命令を取り扱うことが自動的にできます。

$$[3x^2 + (2 + 3i)x + 7] \cdot \left[ x^4 + \frac{2}{3}x^2 + (5 + 3i) \right].$$

Because we installed the polynomial addition and multiplication procedures `add-poly` and `mul-poly` in the generic arithmetic system as the `add` and `mul` operations for type `polynomial`, our system is also automatically able to handle polynomial operations such as

多項式の加算と乗算の手続、`add-poly` と `mul-poly` をジェネリックな数値演算システムに型 `polynomial` のための命令 `add` と `mul` としてインストールしたため、私達のシステムはまた自動的に以下のような多項式操作を取り扱うことが可能です。

$$[(y + 1)x^2 + (y^2 + 1)x + (y - 1)] \cdot [(y - 2)x + (y^3 + 7)].$$

The reason is that when the system tries to combine coefficients, it will dispatch through `add` and `mul`. Since the coefficients are themselves polynomials (in  $y$ ), these will be combined using `add-poly` and `mul-poly`. The result is a kind of “data-directed recursion” in which, for example, a call to `mul-poly` will result in recursive calls to `mul-poly` in order to multiply the coefficients. If the coefficients of the coefficients were themselves polynomials (as might be used to represent polynomials in three variables), the data direction would ensure

that the system would follow through another level of recursive calls, and so on through as many levels as the structure of the data dictates.

その理由はシステムが係数を合成しようと試す時、`add` と `mul` を通して呼出を行うためです。係数はそれ自身 ( $y$  の) 多項式ですから、これらは `add-poly` と `mul-poly` を用いて合成されます。結果は“データ適従再帰”的な物で、例えば `mul-poly` の呼出は係数の乗算のために `mul-poly` の再帰呼出に帰着します。もし係数の係数がそれ自身多項式（多項式を 3 变数で表現した場合）の場合、データ適従はシステムがまた別のレベルの再帰呼出に従うことを保証します。そしてデータの構造が指示するだけのより多くのレベルについてもまた同様です。<sup>109</sup>

## Representing term lists

Finally, we must confront the job of implementing a good representation for term lists. A term list is, in effect, a set of coefficients keyed by the order of the term. Hence, any of the methods for representing sets, as discussed in Section 2.3.3, can be applied to this task. On the other hand, our procedures `add-terms` and `mul-terms` always access term lists sequentially from highest to lowest order. Thus, we will use some kind of ordered list representation.

ようやく項リストに対する良い表現を実装する仕事に直面せねばなりません。項リストは実際には項の次数をキーにした係数の集合です。従って Section 2.3.3 にて議論したような任意の集合表現の手法がこのタスクに適用可能です。一方で手続 `add-terms` と `mul-terms` は常に高い次数から低い次数へと連続して項リストを常にアクセスします。従って何らかの順序付きリスト表現を用いることにしましょう。

---

<sup>109</sup>To make this work completely smoothly, we should also add to our generic arithmetic system the ability to coerce a “number” to a polynomial by regarding it as a polynomial of degree zero whose coefficient is the number. This is necessary if we are going to perform operations such as

これを完全に順調に行うには私達のジェネリック数値演算システムに“数値”を次数が 0 で係数がその数である多項式であると見做すことで多項式に強制する能力も追加する必要があります。これは以下のような式に対して実行を行いたい場合に必要です。

$$[x^2 + (y + 1)x + 5] + [x^2 + 2x + 1],$$

which requires adding the coefficient  $y + 1$  to the coefficient 2.

これは係数  $y + 1$  を係数 2 に対し足す必要があります。

How should we structure the list that represents a term list? One consideration is the “density” of the polynomials we intend to manipulate. A polynomial is said to be *dense* if it has nonzero coefficients in terms of most orders. If it has many zero terms it is said to be *sparse*. For example,

項リストを表現するリストをどのように構造化すべきでしょうか。1つの考慮点は私達が操作しようとする多項式の“濃度”です。多項式は多くの次数に関して0でない係数を持つ場合dense(密)と呼ばれます。もし多くの0の項を持つ場合にはsparse(疎)と呼ばれます。例えば、

$$A : \quad x^5 + 2x^4 + 3x^2 - 2x - 5$$

is a dense polynomial, whereas

は密多項式です。

$$B : \quad x^{100} + 2x^2 + 1$$

is sparse.

は疎です。

The term lists of dense polynomials are most efficiently represented as lists of the coefficients. For example, *A* above would be nicely represented as (1 2 0 3 -2 -5). The order of a term in this representation is the length of the sublist beginning with that term’s coefficient, decremented by 1.<sup>110</sup> This would be a terrible representation for a sparse polynomial such as *B*: There would be a giant list of zeros punctuated by a few lonely nonzero terms. A more reasonable representation of the term list of a sparse polynomial is as a list of the nonzero terms, where each term is a list containing the order of the term and the coefficient for that order. In such a scheme, polynomial *B* is efficiently represented as ((100 1) (2 2) (0 1)). As most polynomial manipulations are performed on sparse polynomials, we will use this method. We will assume that term lists are represented as lists of terms, arranged from highest-order to lowest-order term. Once we have made this decision, implementing the selectors and constructors for terms and term lists is straightforward.<sup>111</sup>

---

<sup>110</sup>In these polynomial examples, we assume that we have implemented the generic arithmetic system using the type mechanism suggested in [Exercise 2.78](#). Thus, coefficients that are ordinary numbers will be represented as the numbers themselves rather than as pairs whose `car` is the symbol `scheme-number`.

<sup>111</sup>Although we are assuming that term lists are ordered, we have implemented `adjoin-term` to simply `cons` the new term onto the existing term list. We can get away with this so long as we guarantee that the procedures (such as `add-terms`) that

密多項式の項リストは係数のリストとして最も効率良く表現されます。例えば上の A は (1 2 0 3 -2 -5) としてうまく表わされます。この表現の項の次数はその項の係数で始まるサブリストの長さから 1 を引いた数です。<sup>112</sup> これは B のような疎多項式には酷い表現に成り得ます。少なく孤立した非ゼロな項により中断される巨大なゼロのリストになるでしょう。疎多項式のより適切な項リストの表現は各項が項の次数とその次数に対する係数を含むリストである非ゼロ項のリストです。そのような仕組みでは多項式 B は効率的に ((100 1) (2 2) (0 1)) として表現されます。多くの多項式操作が疎多項式上にて実行されるため、私達はこちらの手法を用います。項リストは項のリストとして表現され高次から低次の項へと並べられます。これを決定すれば項と項リストに対するセレクタとコンストラクタの実装は簡単です。<sup>113</sup>

```
(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons term term-list)))

(define (the-empty-termlist) '())
(define (first-term term-list) (car term-list))
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))

(define (make-term order coeff) (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))
```

---

use `adjoin-term` always call it with a higher-order term than appears in the list. If we did not want to make such a guarantee, we could have implemented `adjoin-term` to be similar to the `adjoin-set` constructor for the ordered-list representation of sets ([Exercise 2.61](#)).

<sup>112</sup>これらの多項式の例では[Exercise 2.78](#)で提案された型メカニズムを用いてジェネリック数値演算システムを実装したと前提しています。従って普通の数値の係数は数値それ自身で表現され、`car` がシンボル `scheme-number` のペアではありません。

<sup>113</sup>項リストが順序有りだと想定していますが、`adjoin-term` を単純に新しい項を既存の項リスト上に `cons` するように実装しました。`adjoin-term` を用いる (`add-terms` のような) 手続が常にリスト内の物より高次な項と共にそれを呼ぶことを保証するならばこのままにしておくことができます。もしそのような保証を行うことが望ましくなかったならば `adjoin-term` を集合の順序付きリスト表現のための `adjoin-set` ([Exercise 2.61](#)) と同様に実装しておくべきだったでしょう。

where `=zero?` is as defined in Exercise 2.80. (See also Exercise 2.87 below.)

`=zero?` は Exercise 2.80 で定義されています。(下の Exercise 2.87 も参照して下さい)。

Users of the polynomial package will create (tagged) polynomials by means of the procedure:

多項式パッケージのユーザは(タグ付き)多項式を以下の手続で作成します。

```
(define (make-polynomial var terms)
  ((get 'make 'polynomial) var terms))
```

**Exercise 2.87:** Install `=zero?` for polynomials in the generic arithmetic package. This will allow `adjoin-term` to work for polynomials with coefficients that are themselves polynomials.

対抗式に対する `=zero?` をジェネリック数値演算パッケージにインストールせよ。これは `adjoin-term` に係数それ自身が多項式である多項式に対して動作を可能にする。

**Exercise 2.88:** Extend the polynomial system to include subtraction of polynomials. (Hint: You may find it helpful to define a generic negation operation.)

多項式システムを拡張し多項式の減算を含めよ。(ヒント: ジェネリックな単項算術否定演算子を定義することが手助けとなるだろう。)

**Exercise 2.89:** Define procedures that implement the term-list representation described above as appropriate for dense polynomials. 密多項式に対して適切だと上で説明された項リスト表現を実装する手続を定義せよ。

**Exercise 2.90:** Suppose we want to have a polynomial system that is efficient for both sparse and dense polynomials. One way to do this is to allow both kinds of term-list representations in our system. The situation is analogous to the complex-number example of Section 2.4, where we allowed both rectangular and polar representations. To do this we must distinguish different types of term lists and make the operations on term lists generic. Redesign the polynomial system to implement this generalization. This is a major effort, not a local change.

疎と密、両方の多項式に対して効率の良い多項式システムを得たいとする。これを行う1つの方法は両方の種類の項リスト表現をシステム内に許可することである。状況はSection 2.4の複素数の例と同様で、そこでは直行形式と極形式の両表現を許可した。これを行うため、異なる型の項リストを識別し、項リスト上の命令をジェネリックにせねばならない。多項式システムをこの汎化を行うために再設計せよ。これは局所的な変更ではなく大域的な変更になる。

**Exercise 2.91:** A univariate polynomial can be divided by another one to produce a polynomial quotient and a polynomial remainder. For example,

1変数多項式は別の1変数多項式により割ることができ、多項式の商と多項式の剰余を算出する。例えば、

$$\frac{x^5 - 1}{x^2 - 1} = x^3 + x, \text{ remainder } x - 1.$$

Division can be performed via long division. That is, divide the highest-order term of the dividend by the highest-order term of the divisor. The result is the first term of the quotient. Next, multiply the result by the divisor, subtract that from the dividend, and produce the rest of the answer by recursively dividing the difference by the divisor. Stop when the order of the divisor exceeds the order of the dividend and declare the dividend to be the remainder. Also, if the dividend ever becomes zero, return zero as both quotient and remainder.

除算は長除法を通して行うことができる。これは被除数の最高次の項を除数の最高次の項で割る。結果は商の最初の項である。次に結果に除数を掛け、被除数からその結果を引く。そして残りの答を再帰的に差を除数で割ることにより求める。除数の次数が被除数の次数を越えた時に停止し、その時の被除数を剰余であると宣言する。またもし被除数がゼロになった場合には商と剰余の両者をゼロとして返す。

We can design a `div-poly` procedure on the model of `add-poly` and `mul-poly`. The procedure checks to see if the two polys have the same variable. If so, `div-poly` strips off the variable and passes the problem to `div-terms`, which performs the division operation

on term lists. `Div-poly` finally reattaches the variable to the result supplied by `div-terms`. It is convenient to design `div-terms` to compute both the quotient and the remainder of a division. `Div-terms` can take two term lists as arguments and return a list of the quotient term list and the remainder term list.

`add-poly` と `mul-poly` のモデルの上に `div-poly` 手続を設計することが可能だ。この手続は 2 つの多項式が同じ変数を持つかチェックする。そうであれば `div-poly` は変数を取り去りその問題を `div-terms` に渡す。`div-terms` は除算命令を項リスト上にて実行する。`div-poly` は最終的に変数を再度 `div-terms` の結果に取り付ける。除算の商と剩余の両者を求める `div-terms` を設計することは便利だ。`div-terms` は 2 つの項リストを引数として取り商の項リストと剩余の項リストのリストを返す。

Complete the following definition of `div-terms` by filling in the missing expressions. Use this to implement `div-poly`, which takes two polys as arguments and returns a list of the quotient and remainder polys.

以下の `div-terms` の定義を欠けた式を埋めることにより完成させよ。これを用いて `div-poly` を実装せよ。`div-poly` は 2 つの多項式を引数として取り商と剩余の多項式のリストを返す。

```
(define (div-terms L1 L2)
  (if (empty-term-list? L1)
      (list (the-empty-term-list) (the-empty-term-list))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-term-list) L1)
            (let ((new-c (div (coeff t1) (coeff t2)))
                  (new-o (- (order t1) (order t2))))
              (let ((rest-of-result
                     (compute rest of result recursively)
                     )))
                (form complete result)
                ))))))
```

## Hierarchies of types in symbolic algebra

Our polynomial system illustrates how objects of one type (polynomials) may in fact be complex objects that have objects of many different types as parts. This poses no real difficulty in defining generic operations. We need only install appropriate generic operations for performing the necessary manipulations of the parts of the compound types. In fact, we saw that polynomials form a kind of “recursive data abstraction,” in that parts of a polynomial may themselves be polynomials. Our generic operations and our data-directed programming style can handle this complication without much trouble.

私達の多項式システムはある型 (polynomials) のオブジェクトがどのようにして事実上多くの異なる型のオブジェクトをその部分として持つ複雑なオブジェクトになり得るのかについて説明しました。これはジェネリックな命令を定義する場合の実際の困難さは何も引き起していません。複合型の部分の必要な操作を実行するために適切なジェネリック命令をインストールすることのみが必要です。実際に多項式がある種の“再帰的データ抽象化”を形成し、多項式のその部分においてそれ自身が多項式である場合があることを学びました。私達のジェネリック命令とデータ適従プログラミングスタイルはこの複雑さを大した問題無しに扱うことができます。

On the other hand, polynomial algebra is a system for which the data types cannot be naturally arranged in a tower. For instance, it is possible to have polynomials in  $x$  whose coefficients are polynomials in  $y$ . It is also possible to have polynomials in  $y$  whose coefficients are polynomials in  $x$ . Neither of these types is “above” the other in any natural way, yet it is often necessary to add together elements from each set. There are several ways to do this. One possibility is to convert one polynomial to the type of the other by expanding and rearranging terms so that both polynomials have the same principal variable. One can impose a towerlike structure on this by ordering the variables and thus always converting any polynomial to a “canonical form” with the highest-priority variable dominant and the lower-priority variables buried in the coefficients. This strategy works fairly well, except that the conversion may expand a polynomial unnecessarily, making it hard to read and perhaps less efficient to work with. The tower strategy is certainly not natural for this domain or for any domain where the user can invent new types dynamically using old types in various combining forms, such as trigonometric functions, power series, and integrals.

一方で多項式代数はデータ型が自然にタワーに配置できないシステムです。

例えば係数が  $y$  の多項式である  $x$  の多項式を持つことができます。また係数が  $x$  の多項式である  $y$  の多項式を持つことも可能です。これらの型のどちらももう一方の“上”には自然には成り得ません。その上各集合から両者の要素を足す必要は良くあります。これを行う方法はいくつか存在します。1つの可能性としてはある多項式をもう一方の多項式の型に項の展開と再配置を行うことで両者の多項式が同じ主な変数を持つように変換する方法が考えられます。この上に変数で順序付けるタワーの様な構造を強制することで、常に任意の多項式を最優先の変数が主で低優先度の変数が係数に埋め込まれた“基底形式”に変換することができます。この戦略はとても良く行きます。ただし変換が多項式を不必要に展開するかもしれないため、読み難くそして恐らく非効率にしてしまいます。タワーの戦略は全くこの領域では全く自然ではありません。またはユーザが新しい型を古い型を用いて種々の接続形式にて動的に創作する領域、例えば三角関数、累級数、積分等の任意の領域には自然ではないでしょう。

It should not be surprising that controlling coercion is a serious problem in the design of large-scale algebraic-manipulation systems. Much of the complexity of such systems is concerned with relationships among diverse types. Indeed, it is fair to say that we do not yet completely understand coercion. In fact, we do not yet completely understand the concept of a data type. Nevertheless, what we know provides us with powerful structuring and modularity principles to support the design of large systems.

強制をコントロールすることが巨大スケールの代数操作システムの設計において深刻な問題であることは驚くべきことではありません。そのようなシステムの多くの複雑性は様々な型の間の関係性に携わっています。私達はまだ完全には強制を理解していないと言うことは本当に公正でしょう。実際に私達はまだデータ型の概念を完全には理解していません。それでもなお、私達が知っていることは強力な構造化とモジュラー方式の原則を伴ない巨大システムの設計の支援を与えてくれます。

**Exercise 2.92:** By imposing an ordering on variables, extend the polynomial package so that addition and multiplication of polynomials works for polynomials in different variables. (This is not easy!)

変数の順序付けを強要することで多項式パッケージを拡張し多項式の加算と乗算が異なる変数の多項式に対しても働くようにせよ。  
(これは簡単ではない!)

## Extended exercise: Rational functions

We can extend our generic arithmetic system to include *rational functions*. These are “fractions” whose numerator and denominator are polynomials, such as

私達のジェネリック数値演算システムを拡張し *rational functions*(分数関数) を含むようにすることができます。分子と分母が多項式である以下の様な“分数”が存在します。

$$\frac{x+1}{x^3-1}.$$

The system should be able to add, subtract, multiply, and divide rational functions, and to perform such computations as

システムは分数関数の加算、減算、乗算、除算をできなければなりません。そして以下の様な計算を行うために、

$$\frac{x+1}{x^3-1} + \frac{x}{x^2-1} = \frac{x^3+2x^2+3x+1}{x^4+x^3-x-1}.$$

(Here the sum has been simplified by removing common factors. Ordinary “cross multiplication” would have produced a fourth-degree polynomial over a fifth-degree polynomial.)

(ここでは加算は共通因数を取り除くことで簡約されています。通常の“たすき掛け”なら 5 次多項式分の 4 次多項式の分数を生成しているでしょう。)

If we modify our rational-arithmetic package so that it uses generic operations, then it will do what we want, except for the problem of reducing fractions to lowest terms.

私達の分数演算パッケージを変更することでジェネリック命令を用いるようにすると分数を最小の項に簡約する問題を除いて望むことができます。

**Exercise 2.93:** Modify the rational-arithmetic package to use generic operations, but change `make-rat` so that it does not attempt to reduce fractions to lowest terms. Test your system by calling `make-rational` on two polynomials to produce a rational function

分数演算パッケージを変更しジェネリック命令を使用するようにせよ。ただし `make-rat` を変更し分数を最小の項に簡約することは試行しないようにせよ。あなたのシステムを `make-rational` を 2 つの多項式上にて呼び出し分数関数を生成することでテストせよ。

```
(define p1 (make-polynomial 'x '((2 1)(0 1))))
```

```
(define p2 (make-polynomial 'x '((3 1)(0 1))))
(define rf (make-rational p2 p1))
```

Now add `rf` to itself, using `add`. You will observe that this addition procedure does not reduce fractions to lowest terms.

ここで `rf` を自身に `add` を用いて足せ。この加算手続が分数を最小項に簡約しないことを確認するだろう。

We can reduce polynomial fractions to lowest terms using the same idea we used with integers: modifying `make-rat` to divide both the numerator and the denominator by their greatest common divisor. The notion of “greatest common divisor” makes sense for polynomials. In fact, we can compute the GCD of two polynomials using essentially the same Euclid’s Algorithm that works for integers.<sup>114</sup> The integer version is

多項式の分数を整数で用いたのと同じ考え方を用いて最小の項に簡約することができます。`make-rat` を変更し分子と分母の両方を最大公約数で割ります。“Greatest Common Denominator” (GCD: 最大公約数) の概念は多項式に対しても意味を成します。実際に 2 つの多項式の GCD を整数に対して働く、基本的に同じユークリッドのアルゴリズムを用いて求めることができます。<sup>115</sup> 整数版は以下のとおりです。

---

<sup>114</sup>The fact that Euclid’s Algorithm works for polynomials is formalized in algebra by saying that polynomials form a kind of algebraic domain called a *Euclidean ring*. A Euclidean ring is a domain that admits addition, subtraction, and commutative multiplication, together with a way of assigning to each element  $x$  of the ring a positive integer “measure”  $m(x)$  with the properties that  $m(xy) \geq m(x)$  for any nonzero  $x$  and  $y$  and that, given any  $x$  and  $y$ , there exists a  $q$  such that  $y = qx + r$  and either  $r = 0$  or  $m(r) < m(x)$ . From an abstract point of view, this is what is needed to prove that Euclid’s Algorithm works. For the domain of integers, the measure  $m$  of an integer is the absolute value of the integer itself. For the domain of polynomials, the measure of a polynomial is its degree.

<sup>115</sup>ユークリッドのアルゴリズムが多項式に対して働くという事実は代数学において多項式が *Euclidean ring* (ユークリッド環) と呼ばれるある種の代数の定義域を形成することにより形式化されます。ユークリッド環とは加算、減算、そして可換な乗算を許す定義域であり、環の各元  $x$  に対する正の整数の“大きさ” $m(x)$  の割り当て方法とそれに対する性質として任意の非ゼロな  $x$  と  $y$  に対し  $m(xy) \geq m(x)$  であると共に、与えられた任意の  $x$  と  $y$  に対し  $y = qx + r$  となる  $q$  が存在し、 $r = 0$  または  $m(r) < m(x)$  であることが言えます。抽象化的視点からこれがユークリッドのアルゴリズムがうまく行くに必要な条件です。整数の定義域に対して、整数の大きさ  $m$  はその整数の絶対値です。多項式の定義域においては多項式の大きさはその次数です。

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

Using this, we could make the obvious modification to define a GCD operation that works on term lists:

これを用いて、項リスト上で働く GCD 命令を定義するための明かな変更を行なうことができます。

```
(define (gcd-terms a b)
  (if (empty-termlist? b)
      a
      (gcd-terms b (remainder-terms a b))))
```

where `remainder-terms` picks out the remainder component of the list returned by the term-list division operation `div-terms` that was implemented in [Exercise 2.91](#).

ここで `remainder-terms` は [Exercise 2.91](#) で実装された項リストの除算命令 `div-terms` により返されるリストの剰余部を取り出します。

**Exercise 2.94:** Using `div-terms`, implement the procedure `remainder-terms` and use this to define `gcd-terms` as above. Now write a procedure `gcd-poly` that computes the polynomial GCD of two polys. (The procedure should signal an error if the two polys are not in the same variable.) Install in the system a generic operation `greatest-common-divisor` that reduces to `gcd-poly` for polynomials and to ordinary gcd for ordinary numbers. As a test, try

`div-terms` を用いて手続 `remainder-terms` を実装し、それを用いて `gcd-terms` を上記のように定義せよ。次に 2 つの多項式の多項式 GCD を求める手続 `gcd-poly` を書け。(この手続は 2 つの多項式が同じ変数でなければエラーを発しなければならない)。多項式に対しては `gcd-poly` を簡約し、通常の数値に対しては通常の `gcd` に簡約するジェネリック命令 `greatest-common-divisor` をシステムにインストールせよ。テストとして以下を試せ。

```
(define p1 (make-polynomial
              'x '((4 1) (3 -1) (2 -2) (1 2))))
(define p2 (make-polynomial 'x '((3 1) (1 -1))))
```

(`greatest-common-divisor p1 p2`)

and check your result by hand.

次にその結果を手でチェックせよ。

**Exercise 2.95:** Define  $P_1$ ,  $P_2$ , and  $P_3$  to be the polynomials  
以下の多項式  $P_1$ ,  $P_2$ ,  $P_3$  を定義せよ。

$$\begin{aligned}P_1 &: x^2 - 2x + 1, \\P_2 &: 11x^2 + 7, \\P_3 &: 13x + 5.\end{aligned}$$

Now define  $Q_1$  to be the product of  $P_1$  and  $P_2$  and  $Q_2$  to be the product of  $P_1$  and  $P_3$ , and use `greatest-common-divisor` (Exercise 2.94) to compute the GCD of  $Q_1$  and  $Q_2$ . Note that the answer is not the same as  $P_1$ . This example introduces noninteger operations into the computation, causing difficulties with the GCD algorithm.<sup>116</sup> To understand what is happening, try tracing `gcd-terms` while computing the GCD or try performing the division by hand.

次に  $P_1$  と  $P_2$  の積  $Q_1$ 、 $P_1$  と  $P_3$  の積  $Q_2$  を定義し、`greatest-common-divisor`(Exercise 2.94) を用いて  $Q_1$  と  $Q_2$  の GCD を求めよ。答が  $P_1$  と同じにならないことに注意せよ。これが非整数命令の演算が GCD に伴なう困難さを生じさせることの例を示している。<sup>117</sup> 何が起きたのか正しく理解するため、GCD を求める間 `gcd-terms` をトレースするかこの除算を手で試行してみよ。

We can solve the problem exhibited in Exercise 2.95 if we use the following modification of the GCD algorithm (which really works only in the case of polynomials with integer coefficients). Before performing any polynomial division in the GCD computation, we multiply the dividend by an integer constant factor, chosen to guarantee that no fractions will arise during the division process.

---

<sup>116</sup>In an implementation like MIT Scheme, this produces a polynomial that is indeed a divisor of  $Q_1$  and  $Q_2$ , but with rational coefficients. In many other Scheme systems, in which division of integers can produce limited-precision decimal numbers, we may fail to get a valid divisor.

<sup>117</sup>MIT Scheme の様な実装ではこの問題は  $Q_1$  と  $Q_2$  の実際の約数を分数係数を伴なって生成します。多くの Scheme システムでは整数の除算が精度に限界のある小数を生成するため、正しい約数を得るのに失敗します。

Our answer will thus differ from the actual GCD by an integer constant factor, but this does not matter in the case of reducing rational functions to lowest terms; the GCD will be used to divide both the numerator and denominator, so the integer constant factor will cancel out.

**Exercise 2.95**で示された問題を以下に示す(整数係数の多項式の場合のみ実際には動作する) GCD アルゴリズムの変更を用いて解決することができます。GCD の演算中の一切の多項式の除算の前に、被除数を一切の分数が除算処理の間に現れないよう保証するために選ばれた整数定数因数を掛けます。答は従って実際の GCD より整数定数因数の分異なります。しかしこれは分数関数を最小の項に簡約する場合には問題になりません。GCD は分子と分母の両方を割るために利用されるため、整数定数因数は相殺されます。

More precisely, if  $P$  and  $Q$  are polynomials, let  $O_1$  be the order of  $P$  (i.e., the order of the largest term of  $P$ ) and let  $O_2$  be the order of  $Q$ . Let  $c$  be the leading coefficient of  $Q$ . Then it can be shown that, if we multiply  $P$  by the *integerizing factor*  $c^{1+O_1-O_2}$ , the resulting polynomial can be divided by  $Q$  by using the `div-terms` algorithm without introducing any fractions. The operation of multiplying the dividend by this constant and then dividing is sometimes called the *pseudodivision* of  $P$  by  $Q$ . The remainder of the division is called the *pseudoremainder*.

より正確に述べれば、もし  $P$  と  $Q$  が多項式である場合、 $O_1$  を  $P$  の次数とし(つまり  $P$  の最大項の次数とし)、 $O_2$  を  $Q$  の次数とします。 $c$  を  $Q$  の第一の係数とします。すると  $P$  を *integerizing factor*(整数化因数) $c^{1+O_1-O_2}$  で掛けると、結果の多項式は `div-terms` アルゴリズムを用いて一切の分数を生じずに  $Q$  で割ることができます。被除数をこの定数で乗算した後に割る命令は時々  $P$  の  $Q$  による *pseudodivision*(擬除算) と呼ばれます。除算の剰余は *pseudoremainder*(擬剰余) と呼ばれます。

### Exercise 2.96:

- Implement the procedure `pseudoremainder-terms`, which is just like `remainder-terms` except that it multiplies the dividend by the integerizing factor described above before calling `div-terms`. Modify `gcd-terms` to use `pseudoremainder-terms`, and verify that `greatest-common-divisor` now produces an answer with integer coefficients on the example in **Exercise 2.95**.

手続 `pseudoremainder-terms` を実装せよ。これは `remainder-`

`terms` と同様であるが `div-terms` を呼ぶ前に被除数を上で説明した整数化因数で掛ける。`gcd-terms` を変更し `pseudoremainder-terms` を用いるようにし、`greatest-common-divisor` が整数係数の答を [Exercise 2.95](#) の例にて生ずることを確認せよ。

- b The GCD now has integer coefficients, but they are larger than those of  $P_1$ . Modify `gcd-terms` so that it removes common factors from the coefficients of the answer by dividing all the coefficients by their (integer) greatest common divisor.

GCD はこれで整数係数を得る。しかしそれらは  $P_1$  の物よりも大きい。`gcd-terms` を変更し解の係数から全ての係数をそれらの(整数)の最大公約数により割ることで共通因数を取り除くようにせよ。

Thus, here is how to reduce a rational function to lowest terms:

従って、以下に分数関数をどのようにして規約分数に簡約するかを説明します。

- Compute the GCD of the numerator and denominator, using the version of `gcd-terms` from [Exercise 2.96](#).

[Exercise 2.96](#) の `gcd-terms` の版を用いて、分子と分母の GCD を求める

- When you obtain the GCD, multiply both numerator and denominator by the same integerizing factor before dividing through by the GCD, so that division by the GCD will not introduce any noninteger coefficients. As the factor you can use the leading coefficient of the GCD raised to the power  $1 + O_1 - O_2$ , where  $O_2$  is the order of the GCD and  $O_1$  is the maximum of the orders of the numerator and denominator. This will ensure that dividing the numerator and denominator by the GCD will not introduce any fractions.

GCD を得たら分子と分母の両方に同じ整数化因数を GCD で割る前に掛けることで GCD による除算が非整数な係数を生じないようにする。因数として GCD の最初の係数を  $1 + O_1 - O_2$  乗した物を用いることができ、この時  $O_2$  は GCD の次数であり、 $O_1$  は分子と分母の最大次数である。こうすることで分子と分母を GCD で割っても分数を生じない。

- The result of this operation will be a numerator and denominator with integer coefficients. The coefficients will normally be very large because of all of the integerizing factors, so the last step is to remove the redundant

factors by computing the (integer) greatest common divisor of all the coefficients of the numerator and the denominator and dividing through by this factor.

この操作の結果は分子と分母が整数係数になる。係数は通常とても巨大になる。理由の全ては整数化因数のせいだ。そのため最終ステップは分子と分母の全ての係数の(整数の)最大公約数を求めてこの約数で割ることで冗長な因数を取り除くことである。

### Exercise 2.97:

- a Implement this algorithm as a procedure `reduce-terms` that takes two term lists `n` and `d` as arguments and returns a list `nn`, `dd`, which are `n` and `d` reduced to lowest terms via the algorithm given above. Also write a procedure `reduce-poly`, analogous to `add-poly`, that checks to see if the two polys have the same variable. If so, `reduce-poly` strips off the variable and passes the problem to `reduce-terms`, then reattaches the variable to the two term lists supplied by `reduce-terms`.

このアルゴリズムを、2つの項リスト `n` と `d` を引数として取り上で説明されたアルゴリズムにて `n` と `d` を最小の項に簡約したリスト `nn` と `dd` を返す手続 `reduce-terms` として実装せよ。また `add-poly` と同様に2つの多項式が同じ変数を持つかチェックする手続 `reduce-poly` も書け。もしそうである場合 `reduce-poly` は変数を取り去り問題を `reduce-terms` に渡す。そして `reduce-terms` により与えられた2つの項リストに再び変数を取り付ける。

- b Define a procedure analogous to `reduce-terms` that does what the original `make-rat` did for integers:

元の `make-rat` が整数に対して行ったことを行う `reduce-terms` と同様の手続を定義せよ。

```
(define (reduce-integers n d)
  (let ((g (gcd n d)))
    (list (/ n g) (/ d g))))
```

and define `reduce` as a generic operation that calls `apply-generic` to dispatch to either `reduce-poly` (for polynomial

arguments) or `reduce-integers` (for `scheme-number` arguments). You can now easily make the rational-arithmetic package reduce fractions to lowest terms by having `make-rat` call `reduce` before combining the given numerator and denominator to form a rational number. The system now handles rational expressions in either integers or polynomials. To test your program, try the example at the beginning of this extended exercise:

次に `reduce` をジェネリック命令として定義する。これは `apply-generic` を呼び、(`polynomial` 型引数に対しては)`reduce-poly` を呼び出し、(`scheme-number` 型引数に対しては)`reduce-integers` を呼び出す。これで `make-rat` に与えられた分子と分母を接続して分数を形成する前に `reduce` を呼ばせることで、簡単に分数演算パッケージに分数を最小の項に約分させることができる。

```
(define p1 (make-polynomial 'x '((1 1) (0 1))))
(define p2 (make-polynomial 'x '((3 1) (0 -1))))
(define p3 (make-polynomial 'x '((1 1))))
(define p4 (make-polynomial 'x '((2 1) (0 -1))))
(define rf1 (make-rational p1 p2))
(define rf2 (make-rational p3 p4))
(add rf1 rf2)
```

See if you get the correct answer, correctly reduced to lowest terms.

正しい答を得るかどうか、正しく最小の項に簡約されるかどうか確認せよ。

The GCD computation is at the heart of any system that does operations on rational functions. The algorithm used above, although mathematically straightforward, is extremely slow. The slowness is due partly to the large number of division operations and partly to the enormous size of the intermediate coefficients generated by the pseudodivisions. One of the active areas in the development of algebraic-manipulation systems is the design of better algorithms for computing polynomial GCDs.<sup>118</sup>

---

<sup>118</sup>One extremely efficient and elegant method for computing polynomial GCDs was

GCD の計算は分数関数の操作を行うどんなシステムにおいても心臓部に存在します。上で用いられたアルゴリズムは数学的には簡単ですが非常に遅いです。。遅さの原因の一部は除算命令の大きな値であり、他には擬除算により生じる非常に大きな中間時の係数のためとなります。代数操作システムの活発な開発領域の 1 つは多項式の GCD を求めるより良いアルゴリズムの設計です。<sup>119</sup>

---

discovered by Richard Zippel (1979). The method is a probabilistic algorithm, as is the fast test for primality that we discussed in Chapter 1. Zippel's book (Zippel 1993) describes this method, together with other ways to compute polynomial GCDS.

<sup>119</sup>多項式の GCD を求めるための 1 つの著しく効率が良く洗練された手法は Richard Zippel (1979) により発見されました。この手法は Chapter 1 にて議論した素数性の高速なテストと同様の乱選アルゴリズムです。Zippel の本 (Zippel 1993) はこの手法を多項式の GCD を求める他の方法と共に解説しています。

# 3

## Modularity, Objects, and State

Μεταβάλλον ἀναπαύεται

(Even while it changes, it stands still.)

(例え変化している間も、それは静止していた)

—Heraclitus

Plus ça change, plus c'est la même chose.

(the more things change, the more they stay the same.)

(より多くが変化する程、より同じであり続ける)

—Alphonse Karr

The preceding chapters introduced the basic elements from which programs are made. We saw how primitive procedures and primitive data are combined to construct compound entities, and we learned that abstraction is vital in helping us to cope with the complexity of large systems. But these tools are not sufficient for designing programs. Effective program synthesis also requires organizational principles that can guide us in formulating the overall design of a program. In particular, we need strategies to help us structure large systems so that they will be *modular*, that is, so that they can be divided “naturally” into coherent parts that can be separately developed and maintained.

ここまでこの章はプログラムが作成される基礎的な要素を紹介しました。どのようにしてプリミティブな手続とプリミティブなデータが接続され複合要素

を構築するかについて学び、また抽象化が巨大システムの複雑さに立ち向かうことを手助けする核心であることを学習しました。しかしこれらのツールはプログラムを設計するのに十分ではありません。効果的なプログラム統合はプログラム設計全体の形式化をガイドすることが可能な組織的原則を必要とします。具体的には巨大システムの構造化を手助けする戦略が必要で、それによりそれらが *modular*(モジュラ) 化されるよう、つまり“自然に” 分離して開発と保守が可能な論理的部品に分割されるようにします。

One powerful design strategy, which is particularly appropriate to the construction of programs for modeling physical systems, is to base the structure of our programs on the structure of the system being modeled. For each object in the system, we construct a corresponding computational object. For each system action, we define a symbolic operation in our computational model. Our hope in using this strategy is that extending the model to accommodate new objects or new actions will require no strategic changes to the program, only the addition of the new symbolic analogs of those objects or actions. If we have been successful in our system organization, then to add a new feature or debug an old one we will have to work on only a localized part of the system.

物理システムをモデル化したプログラムの構築に特に適切な 1 つの強力な設計戦略はプログラムの構造をモデル化されるシステムの構造を元にすることです。システムの各オブジェクトに対して対応する演算オブジェクトを構築します。各システムのアクションに対しては演算モデル内の記号操作を定義します。この戦略を用いる見込は新しいオブジェクトやアクションを供給するためにモデルを拡張することはプログラムに対する戦略上の変更を必要としないことです。それらのオブジェクト、またはアクションの新しい記号上の類似物の追加のみ変更が必要です。システムの組織化にて成功しているのなら、新しい機能の追加や古い物のデバッグにおいてはシステムの特定の部分上のみで働く必要があります。

To a large extent, then, the way we organize a large program is dictated by our perception of the system to be modeled. In this chapter we will investigate two prominent organizational strategies arising from two rather different “world views” of the structure of systems. The first organizational strategy concentrates on *objects*, viewing a large system as a collection of distinct objects whose behaviors may change over time. An alternative organizational strategy concentrates on the *streams* of information that flow in the system, much as an electrical engineer views a signal-processing system.

すると大体の場合、巨大プログラムを体系化する方法はモデル化されるシ

システムの私達の認知により指示されます。この章では2つの大きく異なるシステム構造の“世界観”から浮かび上がる2つの顕著な体系化戦略について調査します。最初の体系化戦略は *objects*(オブジェクト)に集中し、巨大システムをその振舞が時間と共に変化する区別可能なオブジェクトの集合だと見ます。代替となる体系化戦略はシステム内を流れる情報の *streams*(ストリーム)に集中します。これは電子技術者の信号処理システムの視点と同じです。

Both the object-based approach and the stream-processing approach raise significant linguistic issues in programming. With objects, we must be concerned with how a computational object can change and yet maintain its identity. This will force us to abandon our old substitution model of computation ([Section 1.1.5](#)) in favor of a more mechanistic but less theoretically tractable *environment model* of computation. The difficulties of dealing with objects, change, and identity are a fundamental consequence of the need to grapple with time in our computational models. These difficulties become even greater when we allow the possibility of concurrent execution of programs. The stream approach can be most fully exploited when we decouple simulated time in our model from the order of the events that take place in the computer during evaluation. We will accomplish this using a technique known as *delayed evaluation*.

オブジェクトベースとストリーム処理の両方のアプローチは共にプログラミングにおける重大な言語上の問題を浮かび上がらせます。オブジェクトでは演算オブジェクトがどのように変化可能で、それでもその同一性を維持できるかについて関心を持たなければなりません。このことがより機械的な、しかし論理的に扱い難い演算の *environment model*(環境モデル)のために、私達の古い演算の置換モデル ([Section 1.1.5](#))を諦めさせることになります。オブジェクト、変化、同一性の取扱の難しさは私達の計算モデル内で時間に取り組むための必要性の基本的な結論です。これらの問題はプログラムの並行実行を許可する場合にさらに大きくなります。ストリームの取り組みは私達のモデル内でシミュレートされた時間を計算機の中で評価の間に発生したイベントの順から分断した時に最も全体に利用可能です。*delayed evaluation*(遅延評価)として知られるテクニックを用いてこれを達成します。

## 3.1 Assignment and Local State

We ordinarily view the world as populated by independent objects, each of which has a state that changes over time. An object is said to “have state”

if its behavior is influenced by its history. A bank account, for example, has state in that the answer to the question “Can I withdraw \$100?” depends upon the history of deposit and withdrawal transactions. We can characterize an object’s state by one or more *state variables*, which among them maintain enough information about history to determine the object’s current behavior. In a simple banking system, we could characterize the state of an account by a current balance rather than by remembering the entire history of account transactions.

私達は通常世界を独立したオブジェクトが占める物として見なします。各オブジェクトは時間に伴ない変化する状態を持ちます。オブジェクトはその過去にその振舞が影響される時、“状態を持つ”と呼びます。例えば銀行講座は預金と引き出しの取引の記録に依存する“私は \$100 引き出せるか?”という質問の答に状態を持ちます。オブジェクトの状態を 1 つ以上の *state variables*(状態変数)と見做すことができ、それらの間にオブジェクトの現在の振舞を決定するための歴史についての十分な情報を保存します。簡単な銀行システムでは口座の状態を口座の取引履歴全体を記憶するのではなく、現在の差引残高と見做すことができるでしょう。

In a system composed of many objects, the objects are rarely completely independent. Each may influence the states of others through interactions, which serve to couple the state variables of one object to those of other objects. Indeed, the view that a system is composed of separate objects is most useful when the state variables of the system can be grouped into closely coupled subsystems that are only loosely coupled to other subsystems.

多くのオブジェクトから成るシステムではオブジェクトが完全に独立していることは稀です。あるオブジェクトの状態変数を他のオブジェクトのそれに連結する相互作用を通して各オブジェクトが他の状態に影響を与えることがあるでしょう。実際に、システムが分離したオブジェクトから成るという見方は、システムの状態変数が密結合されたサブシステムが、他のサブシステムとは疎結合であるというグループに分けられる時最も便利です。

This view of a system can be a powerful framework for organizing computational models of the system. For such a model to be modular, it should be decomposed into computational objects that model the actual objects in the system. Each computational object must have its own *local state variables* describing the actual object’s state. Since the states of objects in the system being modeled change over time, the state variables of the corresponding computational objects must also change. If we choose to model the flow of time in

the system by the elapsed time in the computer, then we must have a way to construct computational objects whose behaviors change as our programs run. In particular, if we wish to model state variables by ordinary symbolic names in the programming language, then the language must provide an *assignment operator* to enable us to change the value associated with a name.

このシステムの見方はシステムの演算モデルの体系化に対する強力なフレームワークに成ります。そのようなモデルをモジュール化するためにはシステム内の実際のオブジェクトをモデル化する計算オブジェクトに分離せねばなりません。各計算オブジェクトは実際のオブジェクトの状態を説明するそれ自身の *local state variables*(ローカル状態変数)を持たねばなりません。モデル化されるシステム内のオブジェクトの状態は経時変化するため、計算オブジェクトに相対する状態変数も変化しなければなりません。もし私達がシステム内の時の流れを計算機内で経過する時でモデル化することを選択するのならば、振舞がプログラムが実行するにつれ変化する計算オブジェクトを構築する手段を持たねばなりません。具体的には、もし状態変数をプログラミング言語内の通常の記号名にてモデル化を行いたいのならば、その言語は名前に関連する値を変化することができる *assignment operator*(代入演算子)を提供せねばなりません。

### 3.1.1 Local State Variables

To illustrate what we mean by having a computational object with time-varying state, let us model the situation of withdrawing money from a bank account. We will do this using a procedure `withdraw`, which takes as argument an `amount` to be withdrawn. If there is enough money in the account to accommodate the withdrawal, then `withdraw` should return the balance remaining after the withdrawal. Otherwise, `withdraw` should return the message *Insufficient funds*. For example, if we begin with \$100 in the account, we should obtain the following sequence of responses using `withdraw`:

時間的に変化する状態を伴なう計算オブジェクトを持つことにより何を意味するのかを説明するために、銀行口座からお金を引き出す状況をモデル化してみましょう。これを引数として引き出される `amount`(金額)を取る手続 `withdraw`を用いて行います。もし口座の中に引き出しを受け入れるのに十分なお金があるのならば、`withdraw`は引き出しの後に残る差引残高を返さねばなりません。そうでなければ、`withdraw`は *Insufficient funds*(資金不足)というメッセージを返します。例えば口座を \$100 で始めた場合、`withdraw`を用いて

以下の一連の応答を受け取るはずです。

```
(withdraw 25)
75
(withdraw 25)
50
(withdraw 60)
"Insufficient funds"
(withdraw 15)
35
```

Observe that the expression `(withdraw 25)`, evaluated twice, yields different values. This is a new kind of behavior for a procedure. Until now, all our procedures could be viewed as specifications for computing mathematical functions. A call to a procedure computed the value of the function applied to the given arguments, and two calls to the same procedure with the same arguments always produced the same result.<sup>1</sup>

式 `(withdraw 25)` が 2 度評価され異なる値を返していることに注目して下さい。これは手続にとって新しい種類の振舞です。今まででは全ての手続は数学上の関数を計算する仕様だと見做すことができました。手続の呼出は与えられた引数に適用された関数の値を計算しました。そして同じ手続に同じ引数を与えば場合の 2 度の呼出は常に同じ結果を生じました。<sup>2</sup>

To implement `withdraw`, we can use a variable `balance` to indicate the balance of money in the account and define `withdraw` as a procedure that accesses `balance`. The `withdraw` procedure checks to see if `balance` is at least as large as the requested `amount`. If so, `withdraw` decrements `balance` by `amount` and returns the new value of `balance`. Otherwise, `withdraw` returns the *Insufficient funds* message. Here are the definitions of `balance` and `withdraw`:

---

<sup>1</sup> Actually, this is not quite true. One exception was the random-number generator in Section 1.2.6. Another exception involved the operation/type tables we introduced in Section 2.4.3, where the values of two calls to `get` with the same arguments depended on intervening calls to `put`. On the other hand, until we introduce assignment, we have no way to create such procedures ourselves.

<sup>2</sup> 実際にはこれは全く正しい訳ではありません。例の 1 つはSection 1.2.6 の乱数生成です。別の例はSection 2.4.3で紹介した命令-型テーブルに従って生じます。同じ引数を伴なう `get` の二度の呼出の値は間にに入る `put` に依存します。一方で、代入を紹介するまではそのような手続を自分達で作る方法は無かった訳です。

`withdraw` を実装するために、口座の差引残高を示す変数 `balance` を用い、`balance` にアクセスする手続 `withdraw` を定義します。`withdraw` 手続は `balance` が少なくとも要求された `amount` と同じ大きさであるかをチェックします。もしそうであれば `withdraw` は `balance` を `amount` 分減らし、新しい `balance` の値を返します。そうでなければ `withdraw` は残高不足のメッセージを返します。以下に `balance` と `withdraw` の定義を示します。

```
(define balance 100)
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))
```

Decrementing `balance` is accomplished by the expression  
`balance` を減らすのは次の式により行われます。

```
(set! balance (- balance amount))
```

This uses the `set!` special form, whose syntax is  
これは `set!` という特殊形式を用いています。その文法は次のとおりです。

```
(set! <name> <new-value>)
```

Here `<name>` is a symbol and `<new-value>` is any expression. `Set!` changes `<name>` so that its value is the result obtained by evaluating `<new-value>`. In the case at hand, we are changing `balance` so that its new value will be the result of subtracting `amount` from the previous value of `balance`.

ここでは `<name>` はシンボルであり、`<new-value>` は任意の式です。`set!` は `<name>` を変更し、その値が `<new-value>` を評価して得られた結果になります。この場合では `balance` を変更することでその新しい値が `balance` の以前の値から `amount` を引いた結果になります。<sup>3</sup>

---

<sup>3</sup>The value of a `set!` expression is implementation-dependent. `Set!` should be used only for its effect, not for its value.

`set!` 式の値は実装依存です。`set!` はその効果のためのみに用いられ、その値のために用いられてはなりません。

The name `set!` reflects a naming convention used in Scheme: Operations that change the values of variables (or that change data structures, as we will see in Section 3.3) are given names that end with an exclamation point. This is similar to the convention

`Withdraw` also uses the `begin` special form to cause two expressions to be evaluated in the case where the `if` test is true: first decrementing `balance` and then returning the value of `balance`. In general, evaluating the expression

`withdraw` はまた `begin` という特殊形式も使用しており、これは 2 つの式を `if` のテストが真の場合に評価されるようにします。最初に `balance` を減らし、次に `balance` の値を返します。一般的に以下の式を評価すると

`(begin <exp1> <exp2> ... <expk>)`

causes the expressions  $\langle exp_1 \rangle$  through  $\langle exp_k \rangle$  to be evaluated in sequence and the value of the final expression  $\langle exp_k \rangle$  to be returned as the value of the entire `begin` form.<sup>4</sup>

$\langle exp_1 \rangle$  から  $\langle exp_k \rangle$  までの式は続けて評価され最後の式  $\langle exp_k \rangle$  が `begin` の形式全体の値として返ります。<sup>5</sup>

Although `withdraw` works as desired, the variable `balance` presents a problem. As specified above, `balance` is a name defined in the global environment and is freely accessible to be examined or modified by any procedure. It would be much better if we could somehow make `balance` internal to `withdraw`, so that `withdraw` would be the only procedure that could access `balance` directly and any other procedure could access `balance` only indirectly (through calls to `withdraw`). This would more accurately model the notion that `balance` is a local state variable used by `withdraw` to keep track of the state of the account.

`withdraw` は望んだ通りに働きますが、変数 `balance` が問題を表します。上で指定されたように、`balance` はグローバル環境にて定義された名前であり自由に検査や変更のために任意の手続からアクセスすることができます。どうにかして `balance` を `withdraw` の内在にすることで `withdraw` のみが `balance` に

---

of designating predicates by names that end with a question mark.

その名前 `set!` は Scheme で用いられる名前付けの慣習が反映されています。変数の値を変更する命令（またはSection 3.3で学ぶデータ構造を変える物）は感嘆符（ピックリマーク）で終わる名前を与えられます。これは述語をクエスチョンマークで終わる名前で指定するのと同様です。

<sup>4</sup>We have already used `begin` implicitly in our programs, because in Scheme the body of a procedure can be a sequence of expressions. Also, the `(consequent)` part of each clause in a `cond` expression can be a sequence of expressions rather than a single expression.

<sup>5</sup>私達は既に `begin` を暗黙的にプログラムの中で使用しています。Scheme では手続のボディは連続する式となるからです。また `cond` 式の各節の `(consequent)` の部分は單一の式でなく一連の式にすることができます。

直接アクセスでき、他の手続のどれもが `balance` には間接的に (`withdraw` の呼出を通して) アクセスするようにできればともて良くなるでしょう。こうすることが口座の状態を追跡するため `balance` が `withdraw` により利用される局所状態変数であるという概念をより正確にモデル化します。

We can make `balance` internal to `withdraw` by rewriting the definition as follows:

定義を以下のように書き直すことで `balance` を `withdraw` に内在させることができます。

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))))
```

What we have done here is use `let` to establish an environment with a local variable `balance`, bound to the initial value 100. Within this local environment, we use `lambda` to create a procedure that takes `amount` as an argument and behaves like our previous `withdraw` procedure. This procedure—returned as the result of evaluating the `let` expression—is `new-withdraw`, which behaves in precisely the same way as `withdraw` but whose variable `balance` is not accessible by any other procedure.<sup>6</sup>

ここで起こなったのは `let` を用いて初期値 100 に束縛されたローカル変数 `balance` を持つ環境を設置しました。この局所環境の中では `lambda` を用いて `amount` を引数に取り以前の `withdraw` 手続と同様に振る舞う手続を作成しています。この手続—`let` 式の評価の結果として返される物—は `new-withdraw` であり正確に `withdraw` と同じ振舞をしますが、その変数 `balance` は他のどの手続からもアクセスできません。<sup>7</sup>

<sup>6</sup>In programming-language jargon, the variable `balance` is said to be *encapsulated* within the `new-withdraw` procedure. Encapsulation reflects the general system-design principle known as the *hiding principle*: One can make a system more modular and robust by protecting parts of the system from each other; that is, by providing information access only to those parts of the system that have a “need to know.”

<sup>7</sup>プログラミング言語の専門語において変数 `balance` は手続 `new-withdraw` にカプセル化されたと言います。カプセル化は *hiding principle*(隠蔽原則)として知られる一般的なシステム設計の原則を反映しています。隠蔽原則とはシステムの部分をお互いから守

Combining `set!` with local variables is the general programming technique we will use for constructing computational objects with local state. Unfortunately, using this technique raises a serious problem: When we first introduced procedures, we also introduced the substitution model of evaluation ([Section 1.1.5](#)) to provide an interpretation of what procedure application means. We said that applying a procedure should be interpreted as evaluating the body of the procedure with the formal parameters replaced by their values. The trouble is that, as soon as we introduce assignment into our language, substitution is no longer an adequate model of procedure application. (We will see why this is so in [Section 3.1.3](#).) As a consequence, we technically have at this point no way to understand why the `new-withdraw` procedure behaves as claimed above. In order to really understand a procedure such as `new-withdraw`, we will need to develop a new model of procedure application. In [Section 3.2](#) we will introduce such a model, together with an explanation of `set!` and local variables. First, however, we examine some variations on the theme established by `new-withdraw`.

`set!`をローカル変数と組み合わせることはローカルな状態を持つ計算オブジェクトを構築するのに用いる一般的なプログラミングテクニックです。残念なことに、このテクニックを用いることは深刻な問題をもたらします。私達が最初に手続を紹介した時、評価の置換モデル ([Section 1.1.5](#)) も手続の適用が何を意味するのかの解釈を説明するために提供しました。手続の適用は手続のボディを、形式パラメータをそれらの値で置換して評価することだと解釈されるべきだと述べました。問題は言語に代入を紹介すると直ぐに置換は最早手続の適用モデルとして適切ではなくなります(なぜそうなのかについては[Section 3.1.3](#)で学びます)。結果として技術的に今の時点ではなぜ `new-withdraw` 手続が上で主張された通りに振る舞うのか理解する手立てがありません。本当に `new-withdraw` の様な手続を理解するためには、手続適用の新しいモデルの開発が必要とします。[Section 3.2](#)においてそのようなモデルを `set!` とローカル変数の説明と共に紹介します。しかし最初に `new-withdraw` により設定される主題上のいくつかの変化について調査することにします。

The following procedure, `make-withdraw`, creates “withdrawal processors.” The formal parameter `balance` in `make-withdraw` specifies the initial amount

---

ることでよりモジュール化の推進と頑強なシステムを作成することができるということです。それはつまり情報へのアクセスを“知ることを必要とする”システムの部分ににも与えることによります。

of money in the account.<sup>8</sup>

以下の手続 `make-withdraw` は“引き出し処理”を作成します。`make-withdraw` の形式パラメータ `balance` は口座の初期残高を指定します。<sup>9</sup>

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

`Make-withdraw` can be used as follows to create two objects `W1` and `W2`:

`make-withdraw` は以下のように 2 つのオブジェクト `W1` と `W2` を作るのに使用できます。

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))

(W1 50)
50
(W2 70)
30
(W2 40)
"Insufficient funds"
(W1 40)
10
```

Observe that `W1` and `W2` are completely independent objects, each with its own local state variable `balance`. Withdrawals from one do not affect the other.

`W1` と `W2` が完全独立したオブジェクトであり、各々がそれ自身のローカル状態変数 `balance` を持っていることを観察して下さい。ある口座からの引き出

---

<sup>8</sup>In contrast with `new-withdraw` above, we do not have to use `let` to make `balance` a local variable, since formal parameters are already local. This will be clearer after the discussion of the environment model of evaluation in Section 3.2. (See also Exercise 3.10.)

<sup>9</sup>上の `new-withdraw` とは逆に、`balance` をローカル変数にするために `let` を使用する必要がありません。形式パラメータは既にローカル変数であるためです。Section 3.2 の環境の評価モデルの議論の後にこのことはより明白になります。(Exercise 3.10も参照して下さい)

しは別の口座には影響しません。

We can also create objects that handle deposits as well as withdrawals, and thus we can represent simple bank accounts. Here is a procedure that returns a “bank-account object” with a specified initial balance:

引き出し同様に預け入れを扱うオブジェクトを作ることもできます。従つて簡単な銀行口座を表現可能です。以下が指定した初期残高を持つ“銀行口座オブジェクト”を返す手続です。

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT"
                       m)))))

dispatch)
```

Each call to `make-account` sets up an environment with a local state variable `balance`. Within this environment, `make-account` defines procedures `deposit` and `withdraw` that access `balance` and an additional procedure `dispatch` that takes a “message” as input and returns one of the two local procedures. The `dispatch` procedure itself is returned as the value that represents the bank-account object. This is precisely the *message-passing* style of programming that we saw in [Section 2.4.3](#), although here we are using it in conjunction with the ability to modify local variables.

各 `make-account` の呼出はローカル状態変数 `balance` を持つ環境を構築します。この環境の中で `make-account` は `balance` にアクセスする手続 `deposit` と `withdraw` を定義します。また追加の手続 `dispatch` は“メッセージ”を入力として取り 2 つのローカル手続の内 1 つを返します。`dispatch` 手続それ自身が銀行口座オブジェクトを表現する値として返されます。これはまさに [Section](#)

2.4.3で学んだ*message-passing*(メッセージパッシング) プログラミングスタイルです。ただしここではそれをローカル変数を変更する能力と合わせて用いています。

Make-account can be used as follows:  
make-account は以下のように使用できます。

```
(define acc (make-account 100))
((acc 'withdraw) 50)
50
((acc 'withdraw) 60)
"Insufficient funds"
((acc 'deposit) 40)
90
((acc 'withdraw) 60)
30
```

Each call to `acc` returns the locally defined `deposit` or `withdraw` procedure, which is then applied to the specified `amount`. As was the case with `make-withdraw`, another call to `make-account`

各 `acc` の呼出は局所的に定義された `deposit` か `withdraw` 手続を返し、指定された `amount` に適用されます。`make-withdraw` を用いる場合でしたので、別の呼出し、

```
(define acc2 (make-account 100))
```

will produce a completely separate account object, which maintains its own local `balance`.

は完全に分離された口座オブジェクトを生成し、それ自身のローカルな `balance`を持ちます。

**Exercise 3.1:** An *accumulator* is a procedure that is called repeatedly with a single numeric argument and accumulates its arguments into a sum. Each time it is called, it returns the currently accumulated sum. Write a procedure `make-accumulator` that generates accumulators, each maintaining an independent sum. The input to `make-accumulator` should specify the initial value of the sum; for example

*accumulator* は 1 つの数値引数を持ち繰り返し呼ばれる手続で、引数を合計に蓄積する。呼び出される度に現在の累積和を返す。アキ

ユムレータ(累算器)を返す手続 `make-accumulator` を書け。アキュムレータはそれが独立した合計を持つ。`make-accumulator`への入力は累計の初期値を指定する。例えば、

```
(define A (make-accumulator 5))
(A 10)
15
(A 10)
25
```

**Exercise 3.2:** In software-testing applications, it is useful to be able to count the number of times a given procedure is called during the course of a computation. Write a procedure `make-monitored` that takes as input a procedure, `f`, that itself takes one input. The result returned by `make-monitored` is a third procedure, say `mf`, that keeps track of the number of times it has been called by maintaining an internal counter. If the input to `mf` is the special symbol `how-many-calls?`, then `mf` returns the value of the counter. If the input is the special symbol `reset-count`, then `mf` resets the counter to zero. For any other input, `mf` returns the result of calling `f` on that input and increments the counter. For instance, we could make a monitored version of the `sqrt` procedure:

ソフトウェアテストアプリケーションでは演算処理の間に与えられた手続が何度呼ばれたかを数えられると便利である。1引数手続 `f` を入力として取る手続 `make-monitored` を書け。`make-monitored` の返す結果は第三の手続(`mf` としよう)は内部カウンタを保持することで何回呼出されたかを追跡する。もし `mf` への入力が特別なシンボル `how-many-calls?` であるなら、`mf` はカウンタの値を返す。入力が特別なシンボル `reset-count` であるなら `mf` はカウンタをゼロにリセットする。任意の他の入力に対しては `mf` はその入力上の `f` 呼出の結果を返しカウンタを1増やす。例えば監視版の `sqrt` 手続を作ることができるだろう。

```
(define s (make-monitored sqrt))
(s 100)
10
(s 'how-many-calls?)
1
```

**Exercise 3.3:** Modify the `make-account` procedure so that it creates password-protected accounts. That is, `make-account` should take a symbol as an additional argument, as in

`make-account` 手続を変更しパスワードで守られた口座を作成するようにせよ。即ち `make-account` はシンボルを追加引数として以下のように取得する。

```
(define acc (make-account 100 'secret-password))
```

The resulting account object should process a request only if it is accompanied by the password with which the account was created, and should otherwise return a complaint:

結果の口座オブジェクトはリクエストをアカウント作成時のパスワードが付随する場合のみ処理を行いその他の場合には間違いだと返す。

```
((acc 'secret-password 'withdraw) 40)
60
((acc 'some-other-password 'deposit) 50)
"incorrect password"
```

**Exercise 3.4:** Modify the `make-account` procedure of Exercise 3.3 by adding another local state variable so that, if an account is accessed more than seven consecutive times with an incorrect password, it invokes the procedure `call-the-cops`.

Exercise 3.3の `make-account` 手続に別のローカル状態変数を追加することで変更し、口座が 7 回連続間違ったパスワードでアクセスされた場合に手続 `call-the-cops` (警察を呼ぶ) を実行するようにせよ。

### 3.1.2 The Benefits of Introducing Assignment

As we shall see, introducing assignment into our programming language leads us into a thicket of difficult conceptual issues. Nevertheless, viewing systems as collections of objects with local state is a powerful technique for maintaining a modular design. As a simple example, consider the design of a procedure `rand` that, whenever it is called, returns an integer chosen at random.

私達が学ぶに従い、代入を私達のプログラミング言語に導入したことは難しい概念上の問題の藪の中へと導きます。それでもなおシステムをローカルな状態を持つオブジェクトの集合として見ることはモジュラな設計を維持する為の強力なテクニックです。簡単な例として、呼ばれる度に無作為な（ランダムな）整数を返す手続 `rand` の設計について考えてみて下さい。

It is not at all clear what is meant by “chosen at random.” What we presumably want is for successive calls to `rand` to produce a sequence of numbers that has statistical properties of uniform distribution. We will not discuss methods for generating suitable sequences here. Rather, let us assume that we have a procedure `rand-update` that has the property that if we start with a given number  $x_1$  and form

“ランダムに選択”が何を意味するのかは全くわかりません。恐らく私達が欲しい物は `rand` への連続した呼出が統計上の性質として均一な分散を持つ一連の数値を生じて欲しいのでしょう。ここでは適切な数列を生成する手法については議論しません。そうでなく、数値  $x_1$  を与えて開始した場合に以下の数列を生成する性質を持つ手続 `rand-update` を既に持っていると想定しましょう。

```
x2 = (rand-update x1)
x3 = (rand-update x2)
```

then the sequence of values  $x_1, x_2, x_3, \dots$  will have the desired statistical properties.<sup>10</sup>

すると数列  $x_1, x_2, x_3, \dots$  は望まれた統計的性質特性を持つでしょう。<sup>11</sup>

---

<sup>10</sup>One common way to implement `rand-update` is to use the rule that  $x$  is updated to  $ax + b$  modulo  $m$ , where  $a$ ,  $b$ , and  $m$  are appropriately chosen integers. Chapter 3 of Knuth 1981 includes an extensive discussion of techniques for generating sequences of random numbers and establishing their statistical properties. Notice that the `rand-update` procedure computes a mathematical function: Given the same input twice, it produces the same output. Therefore, the number sequence produced by `rand-update` certainly is not “random,” if by “random” we insist that each number in the sequence is unrelated to the preceding number. The relation between “real randomness” and so-called *pseudo-random* sequences, which are produced by well-determined computations and yet have suitable statistical properties, is a complex question involving difficult issues in mathematics and philosophy. Kolmogorov, Solomonoff, and Chaitin have made great progress in clarifying these issues; a discussion can be found in Chaitin 1975.

<sup>11</sup>`rand-update` を実装する 1 つの一般的な方法は  $x$  は  $ax + b$  modulo  $m$  に更新されるとする、この時  $a, b, m$  は適切に選択された整数であるというルールを用いることです。Knuth 1981 の 3 章は広範囲に及ぶ乱数列を生成するためのテクニックの議論を含んでお

We can implement `rand` as a procedure with a local state variable `x` that is initialized to some fixed value `random-init`. Each call to `rand` computes `rand-update` of the current value of `x`, returns this as the random number, and also stores this as the new value of `x`.

`rand` をある固定値 `random-init` で初期化されるローカル状態変数 `x` を持つ手続として実装できます。`rand` への各呼出は現在の `x` の値の `rand-update` を演算し、これを乱数として返し、また同時にこの値を `x` の新しい値として格納します。

```
(define rand ((x random-init))
  (lambda ()
    (set! x (rand-update x))
    x)))
```

Of course, we could generate the same sequence of random numbers without using assignment by simply calling `rand-update` directly. However, this would mean that any part of our program that used random numbers would have to explicitly remember the current value of `x` to be passed as an argument to `rand-update`. To realize what an annoyance this would be, consider using random numbers to implement a technique called *Monte Carlo simulation*.

もちろん、代入を用いずに単純に `rand-update` を直接呼ぶことで同じ乱数列を生成することも可能でしたでしょう。しかし、これは私達のプログラムの乱数を用いる任意の部分が明示的に `x` の現在の値を `rand-update` の引数として渡すために記憶せねばならないことを意味することになります。これがどれだけ不快であるかを気付くために、乱数を *Monte Carlo simulation* (モンテカルロシミュレーション) と呼ばれるテクニックを実装するために乱数を用いる場合について考えてみましょう。

The Monte Carlo method consists of choosing sample experiments at random from a large set and then making deductions on the basis of the probabilities estimated from tabulating the results of those experiments. For example,

---

り、またそれらの統計的性質を規定しています。`rand-update` 手続が数学上の関数を計算していることに注意して下さい。同じ入力を 2 回与えられれば同じ出力を生成します。従って `rand-update` により生成される数列は“ランダム”が数列のどの数値も以前の数値に関係が無いと主張するのであれば、明らかに“ランダム”ではありません。“真の無作為性(ランダムネス)”と上手く決定された計算で生成されるがそれでも適切な統計上特性を持つ *pseudo-random*(擬似乱数) 列の間の関係は数学と哲学の難しい問題を巻き込む複雑な質問です。Kolmogorov, Solomonoff, それに Chaitin はこれらの問題の解明において大きな進展を上げました。これに関する議論は Chaitin 1975 に見つかります。

we can approximate  $\pi$  using the fact that  $6/\pi^2$  is the probability that two integers chosen at random will have no factors in common; that is, that their greatest common divisor will be 1.<sup>12</sup> To obtain the approximation to  $\pi$ , we perform a large number of experiments. In each experiment we choose two integers at random and perform a test to see if their GCD is 1. The fraction of times that the test is passed gives us our estimate of  $6/\pi^2$ , and from this we obtain our approximation to  $\pi$ .

モンテカルロ法は巨大集合から無作為にサンプル試行を選択することと、その次にそれらの試行上の結果の集計から推測された確率を基準にして演繹を行うことから成り立ちます。例えば  $\pi$  を  $6/\pi^2$  は 2 つの無作為に選択された整数に公約数が無い場合の確率であるという事実を用いて近似値を求められます。言い換えると、2 つの整数の最大公約数が 1 になる場合ということです。<sup>13</sup>  $\pi$  の近似値を求めるためには数多くの試行を行います。各試行において 2 つの整数を無作為に選択し、それらの GCD が 1 であるかをテストします。テストをパスした回数の割合は  $6/\pi^2$  の近似値を与えてくれます。この値から  $\pi$  の近似値を得ます。

The heart of our program is a procedure `monte-carlo`, which takes as arguments the number of times to try an experiment, together with the experiment, represented as a no-argument procedure that will return either true or false each time it is run. `Monte-carlo` runs the experiment for the designated number of trials and returns a number telling the fraction of the trials in which the experiment was found to be true.

プログラムの心臓部は手続 `monte-carlo` です。これは試行回数と引数が無く実行される度に真偽値を返す手続として表される試行を引数として取ります。`monte-carlo` は試行を指定された回数実行し、試行が真と判定された割合を表す数値を返します。

```
(define (estimate-pi trials)
  (sqrt (/ 6 (monte-carlo trials cesaro-test))))
(define (cesaro-test)
  (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment)
```

---

<sup>12</sup>This theorem is due to E. Cesàro. See section 4.5.2 of Knuth 1981 for a discussion and a proof.

<sup>13</sup>この定理は E. Cesàro によるものです。その議論と証明については Knuth 1981 の節 4.5.2 を参照して下さい。

```

(define (iter trials-remaining trials-passed)
  (cond ((= trials-remaining 0)
         (/ trials-passed trials))
        ((experiment)
         (iter (- trials-remaining 1)
               (+ trials-passed 1)))
        (else
         (iter (- trials-remaining 1)
               trials-passed))))
  (iter trials 0))

```

Now let us try the same computation using `rand-update` directly rather than `rand`, the way we would be forced to proceed if we did not use assignment to model local state:

さて同じ計算を `rand` の代わりに `rand-update` を用いてやってみましょう。局所状態をモデル化するために代入を用いない場合に続行を強制される手法です。

```

(define (estimate-pi trials)
  (sqrt (/ 6 (random-gcd-test trials random-init)))))
(define (random-gcd-test trials initial-x)
  (define (iter trials-remaining trials-passed x)
    (let ((x1 (rand-update x)))
      (let ((x2 (rand-update x1)))
        (cond ((= trials-remaining 0)
               (/ trials-passed trials))
              ((= (gcd x1 x2) 1)
               (iter (- trials-remaining 1)
                     (+ trials-passed 1)
                     x2)))
        (else
         (iter (- trials-remaining 1)
               trials-passed
               x2))))))
  (iter trials 0 initial-x))

```

While the program is still simple, it betrays some painful breaches of modularity. In our first version of the program, using `rand`, we can express the

Monte Carlo method directly as a general `monte-carlo` procedure that takes as an argument an arbitrary `experiment` procedure. In our second version of the program, with no local state for the random-number generator, `random-gcd-test` must explicitly manipulate the random numbers `x1` and `x2` and recycle `x2` through the iterative loop as the new input to `rand-update`. This explicit handling of the random numbers intertwines the structure of accumulating test results with the fact that our particular experiment uses two random numbers, whereas other Monte Carlo experiments might use one random number or three. Even the top-level procedure `estimate-pi` has to be concerned with supplying an initial random number. The fact that the random-number generator's insides are leaking out into other parts of the program makes it difficult for us to isolate the Monte Carlo idea so that it can be applied to other tasks. In the first version of the program, assignment encapsulates the state of the random-number generator within the `rand` procedure, so that the details of random-number generation remain independent of the rest of the program.

プログラムは今も単純ではありますが、いくつかモジュール方式に対する苦痛を伴なう侵害行為があります。`rand` を用いる最初の版ではモンテカルロ法を直接、引数として任意の `experiment` 手続を取り全体的な `monte-carlo` 手続にて表すことができました。乱数生成に対する状態変数の無い 2 つ目の版では `random-gcd-test` が明示的に乱数 `x1` と `x2` を管理し、`x2` を繰り返しのループを通して `rand-update` に対する新しい入力としてリサイクルしなければなりません。この明示的な乱数の取扱はテスト結果の蓄積構造と私達の試行が 2 つの乱数を利用するという事実と一緒に密に結合します。例え他のモンテカルロの試行が 1 つや 3 つの乱数を使うにしてもです。トップレベルの手続 `estimate-pi` ですら乱数の初期値を提供することに関心を持たねばなりません。乱数生成器の内部がプログラムの他の部分に漏れ出すことはモンテカルロの考えを分離し他のタスクに適用することを難しくします。プログラムの最初の版では代入が乱数生成器の状態を `rand` 手続の中にカプセル化しているため乱数生成器の詳細はプログラムの他の部分からの独立を維持しています。

The general phenomenon illustrated by the Monte Carlo example is this: From the point of view of one part of a complex process, the other parts appear to change with time. They have hidden time-varying local state. If we wish to write computer programs whose structure reflects this decomposition, we make computational objects (such as bank accounts and random-number generators) whose behavior changes with time. We model state with local state variables, and we model the changes of state with assignments to those variables.

モンテカルロの例にて説明された一般的な事象は以下のとおりです。複雑なプロセスの一部の視点からは他の部分は時間に従い変化するように見えます。それらは時間と共に変化するローカルな状態を隠しています。もしこの分解を反映する構造を持つ計算機プログラムを書きたいのならば、振舞が時間と共に変換する(銀行口座と乱数生成器の様な)計算オブジェクトを作成します。私達は状態をローカル状態変数を用いてモデル化し、状態の変化をそれらの変数への代入にてモデル化します。

It is tempting to conclude this discussion by saying that, by introducing assignment and the technique of hiding state in local variables, we are able to structure systems in a more modular fashion than if all state had to be manipulated explicitly, by passing additional parameters. Unfortunately, as we shall see, the story is not so simple.

この議論を次のように述べることで結論付けることは魅力的です。曰く、代入と状態を局所変数に隠す技術を紹介することで、追加のパラメータを渡すことで全ての状態が明示的に操作されなければならない場合よりも、よりモジュール化を行う方法でシステムの構造化を行えます、と。残念ながらこれから学ぶように、このお話をそんなに簡単ではありません。

**Exercise 3.5:** *Monte Carlo integration* is a method of estimating definite integrals by means of Monte Carlo simulation. Consider computing the area of a region of space described by a predicate  $P(x, y)$  that is true for points  $(x, y)$  in the region and false for points not in the region. For example, the region contained within a circle of radius 3 centered at  $(5, 7)$  is described by the predicate that tests whether  $(x - 5)^2 + (y - 7)^2 \leq 3^2$ . To estimate the area of the region described by such a predicate, begin by choosing a rectangle that contains the region. For example, a rectangle with diagonally opposite corners at  $(2, 4)$  and  $(8, 10)$  contains the circle above. The desired integral is the area of that portion of the rectangle that lies in the region. We can estimate the integral by picking, at random, points  $(x, y)$  that lie in the rectangle, and testing  $P(x, y)$  for each point to determine whether the point lies in the region. If we try this with many points, then the fraction of points that fall in the region should give an estimate of the proportion of the rectangle that lies in the region. Hence, multiplying this fraction by the area of the entire rectangle should produce an estimate of the integral.

*Monte Carlo integration* モンテカルロ積分はモンテカルロ・シミュレーションを用いて定積分を推測する手法だ。述語  $P(x, y)$  で記述される空間の領域の面積を計算する場合について考えてみる。述語  $P(x, y)$  は点  $(x, y)$  が領域の中であれば真であり、そうでなければ偽である。例えば中心  $(5, 7)$ 、半径 3 の円に含まれる領域は  $(x - 5)^2 + (y - 7)^2 \leq 3^2$  であるかテストする述語にて記述される。そのよう述語で記述された領域の面積を推測するためにその領域を含む長方形を選択することから始める。例として対角線上の角を  $(2, 4)$  と  $(8, 10)$  に持つ長方形は先程の円を含む。期待される積分はその領域が位置する長方形の一部の面積だ。長方形の中の点  $(x, y)$  を不作為に選択し、各点に対し  $P(x, y)$  をテストしその点が元の領域の中であるかどうかを決定することで積分を推定することができる。もしこの試行を数多くの点で行えば領域の中に落ちる点の割合は長方形の内のその領域の割合の推定値を与えるはずだ。従ってこの割合に長方形全体の面積を掛けることで積分の推定値を生成可能である。

Implement Monte Carlo integration as a procedure `estimate-integral` that takes as arguments a predicate `P`, upper and lower bounds `x1`, `x2`, `y1`, and `y2` for the rectangle, and the number of trials to perform in order to produce the estimate. Your procedure should use the same `monte-carlo` procedure that was used above to estimate  $\pi$ . Use your `estimate-integral` to produce an estimate of  $\pi$  by measuring the area of a unit circle.

モンテカルロ積分を手続 `estimate-integral` として実装せよ。これは引数として述語 `P`、長方形の上下界として `x1`, `x2`, `y1`, `y2`、そして推定値を生成するため実行する試行回数を取る。手続は上で  $\pi$  を推測するために使用した `monte-carlo` 手続を同じく使用せねばならない。`estimate-integral` を用いて  $\pi$  の推測値を単位円の面積を測ることで求めよ。

You will find it useful to have a procedure that returns a number chosen at random from a given range. The following `random-in-range` procedure implements this in terms of the `random` procedure used in [Section 1.2.6](#), which returns a nonnegative number less than its input.<sup>14</sup>

---

<sup>14</sup>MIT Scheme provides such a procedure. If `random` is given an exact integer (as in

与えられた値域から不作為に選択された数値を返す手続を持つことが便利であると発見するかもしれない。以下の `random-in-range` 手続はこれを [Section 1.2.6](#) で使用した `random` 手続を用いて実装する。これは入力より小さな非負数を返す。<sup>15</sup>

```
(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (random range))))
```

**Exercise 3.6:** It is useful to be able to reset a random-number generator to produce a sequence starting from a given value. Design a new `rand` procedure that is called with an argument that is either the symbol `generate` or the symbol `reset` and behaves as follows: `(rand 'generate)` produces a new random number; `((rand 'reset) <new-value>)` resets the internal state variable to the designated `<new-value>`. Thus, by resetting the state, one can generate repeatable sequences. These are very handy to have when testing and debugging programs that use random numbers.

乱数生成器を与えた値から始まる列を生成するためリセットすることができれば便利である。シンボル `generate` またはシンボル `reset` のどちらかを引数として呼び出す新しい `rand` 手続を設計せよ。これは次のように振る舞う。`(rand 'generate)` は新しい乱数を生成する。`((rand 'reset) <new-value>)` は内部の状態変数を指定された `<new-value>` でリセットする。従って状態をリセットすることで繰り返し可能な列の生成が行える。これは乱数を用いるプログラムのテストやデバッグにおいてとても役に立つ。

### 3.1.3 The Costs of Introducing Assignment

As we have seen, the `set!` operation enables us to model objects that have local state. However, this advantage comes at a price. Our programming lan-

---

[Section 1.2.6](#)) it returns an exact integer, but if it is given a decimal value (as in this exercise) it returns a decimal value.

<sup>15</sup>MIT Scheme はそのような手続を提供します。もし `random` が ([Section 1.2.6](#) での様に) 整数を渡されれば、整数を返します。しかし (この課題のように) 小数を渡された場合には小数を返します。

guage can no longer be interpreted in terms of the substitution model of procedure application that we introduced in [Section 1.1.5](#). Moreover, no simple model with “nice” mathematical properties can be an adequate framework for dealing with objects and assignment in programming languages.

ここまで見てきたとおり、`set!` 命令はローカルな状態を持つオブジェクトのモデル化を可能にします。しかしこの利点は犠牲を伴ないです。私達のプログラミング言語は[Section 1.1.5](#)で紹介した手続適用の置換モデルを用いて説明することができません。加えて、プログラミング言語の間にオブジェクトと代入を取り扱うための適切なフレームワークとなる“良い”数学上の特性を伴なう簡単なモデルが存在しません。

So long as we do not use assignments, two evaluations of the same procedure with the same arguments will produce the same result, so that procedures can be viewed as computing mathematical functions. Programming without any use of assignments, as we did throughout the first two chapters of this book, is accordingly known as *functional programming*.

代入を使わない限り、同じ引数を伴なう同じ手続の二度の評価は同じ結果を生じ、手続は数学上の関数の計算と見ることができます。私達がこの本の最初の二章を通じて行ってきたような代入を使用しないプログラミングは、それ故に*functional programming*(関数型プログラミング)として知られています。

To understand how assignment complicates matters, consider a simplified version of the `make-withdraw` procedure of [Section 3.1.1](#) that does not bother to check for an insufficient amount:

代入が問題をどのように困難にするかを理解するために、[Section 3.1.1](#)の `make-withdraw` 手続を残額が十分であるかのチェックを行わない様に単純化した版について考えます。

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
(define W (make-simplified-withdraw 25))
(W 20)
5
(W 10)
-5
```

Compare this procedure with the following `make-decrementer` procedure, which

does not use `set!`:

この手続と以下の `set!` を使用しない `make-decremener` 手続とを比べてみて下さい。

```
(define (make-decremener balance)
  (lambda (amount)
    (- balance amount)))
```

`Make-decremener` returns a procedure that subtracts its input from a designated amount `balance`, but there is no accumulated effect over successive calls, as with `make-simplified-withdraw`:

`make-decremener` は指定された残高 `balance` からその入力を引きます。しかし連続した呼び出しにおいて `make-simplified-withdraw` のような累積効果はありません。

```
(define D (make-decremener 25))
(D 20)
5
(D 10)
15
```

We can use the substitution model to explain how `make-decremener` works. For instance, let us analyze the evaluation of the expression

`make-decremener` がどのように働くかの説明には置換モデルを使用できます。例えば以下の式の評価を解析してみましょう。

```
((make-decremener 25) 20)
```

We first simplify the operator of the combination by substituting 25 for `balance` in the body of `make-decremener`. This reduces the expression to

最初に結合のオペレータを `make-decremener` のボディの `balance` を 25 と置き換えることにより簡約します。式は以下のようになります。

```
((lambda (amount) (- 25 amount)) 20)
```

Now we apply the operator by substituting 20 for `amount` in the body of the `lambda` expression:

`lambda` 式のボディにある `amount` を 20 と置き換えることでオペレータを適用します。

```
(- 25 20)
```

The final answer is 5.

最終的な答は 5 です。

Observe, however, what happens if we attempt a similar substitution analysis with `make-simplified-withdraw`:

しかもしも `make-simplified-withdraw` に対しても同様の置換分析を試みれば何が起こるか観察してみて下さい。

```
((make-simplified-withdraw 25) 20)
```

We first simplify the operator by substituting 25 for `balance` in the body of `make-simplified-withdraw`. This reduces the expression to<sup>16</sup>

最初に `make-simplified-withdraw` のボディにある `balance` を 25 に置き換えることでオペレータを簡約します。これにより式は以下のようになります。

<sup>17</sup>

```
((lambda (amount) (set! balance (- 25 amount)) 25) 20)
```

Now we apply the operator by substituting 20 for `amount` in the body of the `lambda` expression:

ここで `lambda` 式のボディの中の `amount` を 20 に置き換えてオペレータを適用します。

```
(set! balance (- 25 20)) 25
```

If we adhered to the substitution model, we would have to say that the meaning of the procedure application is to first set `balance` to 5 and then return 25 as the value of the expression. This gets the wrong answer. In order to get the correct answer, we would have to somehow distinguish the first occurrence of `balance` (before the effect of the `set!`) from the second occurrence of `balance` (after the effect of the `set!`), and the substitution model cannot do this.

もし置換モデルに執着するのであれば、手続の適用の意味は最初に `balance` を 5 に設定し、次に式の値として 25 を返すと言わざるを得ません。これは間違った答を得ます。正しい答を得るためにには、どうにかして最初の位置の

---

<sup>16</sup>We don't substitute for the occurrence of `balance` in the `set!` expression because the `(name)` in a `set!` is not evaluated. If we did substitute for it, we would get `(set! 25 (- 25 amount))`, which makes no sense.

<sup>17</sup>`set!` 式に存在する `balance` は置き換えません。なぜなら `set!` 内の `(name)` は評価されないからです。もしこれを置き換えれば `(set! 25 (- 25 amount))` を得ることになりますが、これは意味がありません。

`balance(set! の効果以前)` を 2 つ目の `balance(set! の効果の後)` から区別せねばなりません。そして置換モデルはこれを行うことができません。

The trouble here is that substitution is based ultimately on the notion that the symbols in our language are essentially names for values. But as soon as we introduce `set!` and the idea that the value of a variable can change, a variable can no longer be simply a name. Now a variable somehow refers to a place where a value can be stored, and the value stored at this place can change. In Section 3.2 we will see how environments play this role of “place” in our computational model.

ここでの問題は置換は詰まるところ、私達の言語のシンボルが本質的に値の名前であるという概念に基いています。しかし `set!` と変数の値が変更できるという考えを紹介してから直ぐに、変数は最早単純な名前ではありません。今では変数はどうにかして値が格納できる場所を参照し、その場所に格納された値は変更することが可能です。Section 3.2にて、環境がどのようにしてこの“場所”的役割を演じるのかについて学びます。

## Sameness and change

The issue surfacing here is more profound than the mere breakdown of a particular model of computation. As soon as we introduce change into our computational models, many notions that were previously straightforward become problematical. Consider the concept of two things being “the same.”

ここで表出した問題は特定の演算モデルが単に崩壊したよりもずっと深淵です。私達の計算モデルに変更を紹介して直ぐに、以前は簡単であった多くの概念が難問と化します。2つの物が“同じ”であるという観念について考えてみましょう。

Suppose we call `make-decrementer` twice with the same argument to create two procedures:

`make-decrementer` を同じ引数を与えて二度呼び二つの手続を作成したとします。

```
(define D1 (make-decrementer 25))  
(define D2 (make-decrementer 25))
```

Are `D1` and `D2` the same? An acceptable answer is yes, because `D1` and `D2` have the same computational behavior—each is a procedure that subtracts its input from 25. In fact, `D1` could be substituted for `D2` in any computation without changing the result.

$D_1$  と  $D_2$  は同じでしょうか? 無難な答は YES です。 $D_1$  と  $D_2$  は同じ計算上の振舞を持ち、それぞれが入力から 25 を引く手続です。実際に  $D_1$  は任意の計算において結果を変えることなく  $D_2$  の代替にできます。

Contrast this with making two calls to `make-simplified-withdraw`:

これと `make-simplified-withdraw` の二度の呼出とを対比します。

```
(define W1 (make-simplified-withdraw 25))  
(define W2 (make-simplified-withdraw 25))
```

Are  $W_1$  and  $W_2$  the same? Surely not, because calls to  $W_1$  and  $W_2$  have distinct effects, as shown by the following sequence of interactions:

$W_1$  と  $W_2$  は同じでしょうか?もちろん違います。 $W_1$  と  $W_2$  の呼出は区別可能な効果を持ちます。以下の応答列によりそれが示されます。

```
(W1 20)  
5  
(W1 20)  
-15  
(W2 20)  
5
```

Even though  $W_1$  and  $W_2$  are “equal” in the sense that they are both created by evaluating the same expression, (`make-simplified-withdraw 25`), it is not true that  $W_1$  could be substituted for  $W_2$  in any expression without changing the result of evaluating the expression.

例え  $W_1$  と  $W_2$  が同じ式 (`make-simplified-withdraw 25`) を評価することで作成されたという点で “同じ” であっても、 $W_1$  が式の評価の結果を変えずに任意の式で  $W_2$  の代替になるかというのは正しくありません。

A language that supports the concept that “equals can be substituted for equals” in an expression without changing the value of the expression is said to be *referentially transparent*. Referential transparency is violated when we include `set!` in our computer language. This makes it tricky to determine when we can simplify expressions by substituting equivalent expressions. Consequently, reasoning about programs that use assignment becomes drastically more difficult.

式において式の値を変化せずに “等しい物は等しい物で置き換えられる” という観念を支持する言語は *referentially transparent* (参照透明) と呼ばれます。参照透明は私達の計算機言語に `set!` を含めた時侵害されました。これがいつ

式を等価な式で置き換えることで簡約できるかを決定することを扱いにくくします。結果的に、代入を用いるプログラムについての推測は大幅により難しくなります。

Once we forgo referential transparency, the notion of what it means for computational objects to be “the same” becomes difficult to capture in a formal way. Indeed, the meaning of “same” in the real world that our programs model is hardly clear in itself. In general, we can determine that two apparently identical objects are indeed “the same one” only by modifying one object and then observing whether the other object has changed in the same way. But how can we tell if an object has “changed” other than by observing the “same” object twice and seeing whether some property of the object differs from one observation to the next? Thus, we cannot determine “change” without some *a priori* notion of “sameness,” and we cannot determine sameness without observing the effects of change.

参照透明を無しで済ませば、計算オブジェクトが“同じ”であることを意味する概念が形式的に捉えることが難しくなります。本当に実際の世界での“等価”的な意味は私達のプログラムモデルはそれ自身において全く明確になりません。一般的に2つの恐らく同じオブジェクトが本当に“同じ物”であるかは一方のオブジェクトを変更した場合にもう一方のオブジェクトが同様に変化したかを観察するしか手立てがありません。しかしオブジェクトが“変更された”ことを“同じ”オブジェクトを2回観察し、オブジェクトのある属性が1回目の観察から次に対して異なるかどうかを見る以外にどうやって判断できるのでしょうか。従って“同一性”的な定義の候補としての *a priori*(先驗的な) 概念無しに“変化”を判断することができません。そして変化の結果を観察せずに同一性を判断することはできないのです。

As an example of how this issue arises in programming, consider the situation where Peter and Paul have a bank account with \$100 in it. There is a substantial difference between modeling this as

この問題がプログラミングにおいてどのように発生するかの例として、PeterとPaulが\$100入っている口座を持っている状態について考えましょう。これをモデル化するに当たって以下の定義と

```
(define peter-acc (make-account 100))  
(define paul-acc (make-account 100))
```

and modeling it as

以下の定義では大きな違いがあります。

```
(define peter-acc (make-account 100))
(define paul-acc peter-acc)
```

In the first situation, the two bank accounts are distinct. Transactions made by Peter will not affect Paul's account, and vice versa. In the second situation, however, we have defined `paul-acc` to be *the same thing* as `peter-acc`. In effect, Peter and Paul now have a joint bank account, and if Peter makes a withdrawal from `peter-acc` Paul will observe less money in `paul-acc`. These two similar but distinct situations can cause confusion in building computational models. With the shared account, in particular, it can be especially confusing that there is one object (the bank account) that has two different names (`peter-acc` and `paul-acc`); if we are searching for all the places in our program where `paul-acc` can be changed, we must remember to look also at things that change `peter-acc`.<sup>18</sup>

最初の状況では、2つの銀行口座は区別できます。Peterにより行われた取引はPaulの口座には影響を与えません。逆も同じです。2つ目の状況ではしかし、`paul-acc`が`peter-acc`と同じ物になるよう定義しました。実際にPeterとPaulは今では連結銀行口座を持っておりPeterが`peter-acc`から引き出しを行えばPaulは`paul-acc`の残額が減ったことを観察するでしょう。これらの2つの似ているが区別できる状況は計算モデルの構築において混乱の元となり得ます。具体的には、共有口座のために1つのオブジェクト(銀行口座)が2つの異なる名前(`peter-acc`と`paul-acc`)を持つことは特に混乱します。プログラムの中で`paul-acc`を変更することができる箇所を全て探す場合、`peter-acc`を変更する箇所もまた探さねばならないことを覚えておかなければなりません。<sup>19</sup>

---

<sup>18</sup>The phenomenon of a single computational object being accessed by more than one name is known as *aliasing*. The joint bank account situation illustrates a very simple example of an alias. In Section 3.3 we will see much more complex examples, such as “distinct” compound data structures that share parts. Bugs can occur in our programs if we forget that a change to an object may also, as a “side effect,” change a “different” object because the two “different” objects are actually a single object appearing under different aliases. These so-called *side-effect bugs* are so difficult to locate and to analyze that some people have proposed that programming languages be designed in such a way as to not allow side effects or aliasing (Lampson et al. 1981; Morris et al. 1980).

<sup>19</sup>単一の計算オブジェクトが複数の名前によりアクセスされる事象は*aliasing*(エイリアシング)として知られています。連結銀行口座の状況はエイリアスのとても簡単な例を説明します。Section 3.3では“識別可能”な複合データ構造が一部を共有するような

With reference to the above remarks on “sameness” and “change,” observe that if Peter and Paul could only examine their bank balances, and could not perform operations that changed the balance, then the issue of whether the two accounts are distinct would be moot. In general, so long as we never modify data objects, we can regard a compound data object to be precisely the totality of its pieces. For example, a rational number is determined by giving its numerator and its denominator. But this view is no longer valid in the presence of change, where a compound data object has an “identity” that is something different from the pieces of which it is composed. A bank account is still “the same” bank account even if we change the balance by making a withdrawal; conversely, we could have two different bank accounts with the same state information. This complication is a consequence, not of our programming language, but of our perception of a bank account as an object. We do not, for example, ordinarily regard a rational number as a changeable object with identity, such that we could change the numerator and still have “the same” rational number.

上記の“同一性”と“変更”上の見解への参照と共に、もしPeterとPaulは差引残高を調べられるだけで差引残高を変更する命令を実行することができない場合、2つの口座が区別できるかどうかという問題が無意味になるのかについて注意して下さい。一般的に、データオブジェクトを変更しない限り、複合データオブジェクトをまさにそれの部分の全体であると見做すことができます。例えば、分数はその分子と分母により決定されます。しかしこの見方は変更が存在する時には、複合データオブジェクトがそれが組み立てられている部品とは異質の“アイデンティティ(自己同一性)”を持つ場合には有効ではありません。銀行口座は例え引き出しを行うことで残高を変更しても依然として“同じ”銀行口座です。反対に、同じ状態情報を持つ2つの異なる銀行口座を持つこともできるでしょう。この複雑さは私達のプログラミング言語による物ではなく、私達のオブジェクトとしての銀行口座の認知によるものです。例えば私達は通常分数を同一性を保ちながら変更可能なオブジェクトだとは見做しません。分子を変更したら“同じ”分数をだとは思いません。

---

さらに複雑な例について学びます。バグはプログラムの中でオブジェクトに対する変更が“副作用”として“異なる”オブジェクトに対しても変更を行い得る場合を忘れている時に発生します。2つの“異なる”オブジェクトが実際には異なるエイリアスの下に現れる单一のオブジェクトであるためです。これらは *side-effect bugs*(副作用バグ)と呼ばれる物で位置の特定や分析がとても難しいため一部の人々はプログラミング言語は副作用やエイリアスを許可しないよう設計されるべきだと提案しています。(Lampson et al. 1981; Morris et al. 1980)

## Pitfalls of imperative programming

In contrast to functional programming, programming that makes extensive use of assignment is known as *imperative programming*. In addition to raising complications about computational models, programs written in imperative style are susceptible to bugs that cannot occur in functional programs. For example, recall the iterative factorial program from [Section 1.2.1](#):

関数型プログラミングとは反対に、代入を広範囲に用いるプログラミングは *imperative programming*(命令型プログラミング)として知られています。計算モデルに関する複雑さを上げるのに加えて、命令型スタイルで書かれたプログラムは関数型プログラムでは起こり得ないバグを起こしやすくなります。例えば[Section 1.2.1](#)の反復指數プログラムを思い出して下さい。

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product) (+ counter 1))))
  (iter 1 1))
```

Instead of passing arguments in the internal iterative loop, we could adopt a more imperative style by using explicit assignment to update the values of the variables `product` and `counter`:

内部の反復ループ内で引数を渡す代わりに変数 `product` と `counter` の値の明示的な代入を用いることでより命令型のスタイルを受け入れることができます

```
(define (factorial n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! product (* counter product))
                 (set! counter (+ counter 1))
                 (iter)))))
    (iter)))
```

This does not change the results produced by the program, but it does introduce a subtle trap. How do we decide the order of the assignments? As it happens, the program is correct as written. But writing the assignments in the opposite order

これはプログラムにより生成される結果に違いがありません。しかし微妙な罠を招いています。私達は代入の順序をどのように決めたのでしょうか。たまたま上のプログラムは正しく書かれています。しかし代入を逆順に書くことは

```
(set! counter (+ counter 1))  
(set! product (* counter product))
```

would have produced a different, incorrect result. In general, programming with assignment forces us to carefully consider the relative orders of the assignments to make sure that each statement is using the correct version of the variables that have been changed. This issue simply does not arise in functional programs.<sup>20</sup>

異なる間違った結果を生じることでしょう。一般的に代入を伴なうプログラミングは各命令が変更された変数の正しい版を用いることを確認するため、私達に注意深く代入の相対順序を考えることを強制します。この問題は単純に関数型プログラミングでは起こりません。<sup>21</sup>

The complexity of imperative programs becomes even worse if we consider applications in which several processes execute concurrently. We will return

---

<sup>20</sup>In view of this, it is ironic that introductory programming is most often taught in a highly imperative style. This may be a vestige of a belief, common throughout the 1960s and 1970s, that programs that call procedures must inherently be less efficient than programs that perform assignments. (Steele 1977 debunks this argument.) Alternatively it may reflect a view that step-by-step assignment is easier for beginners to visualize than procedure call. Whatever the reason, it often saddles beginning programmers with “should I set this variable before or after that one” concerns that can complicate programming and obscure the important ideas.

<sup>21</sup>この視点ではプログラミング入門が高度に命令型スタイルを用いながら最も頻繁に教えられていることは皮肉な事です。これは1960年代から1970年代までの間中、手続を呼ぶプログラムは本質的に代入を実行するプログラムよりも非効率であるに違いないという共通の信念の名残でしょう。(Steele 1977がこの論争が誤りであることを示しました)。あるいは行毎の代入を思い浮かべることが初心者にとって手続呼出よりも簡単であるという見方もあるでしょう。どのような理由しろ、このことは初級プログラマに対し“私はこの変数をあれより前か後に設定すべきか?”といったプログラミングを複雑にし、重要な考慮点を不明瞭にする心配事をしばしば負わせることになります。

to this in [Section 3.4](#). First, however, we will address the issue of providing a computational model for expressions that involve assignment, and explore the uses of objects with local state in designing simulations.

命令型プログラムの複雑さは複数のプロセスが並行に実行されるアプリケーションを考える場合により悪くなります。この点については[Section 3.4](#)にて戻ります。しかし最初に代入を含む表現のための計算モデルを提供する場合の問題を提示します。そしてシミュレーションの設計においてローカルな状態を持つオブジェクトの使用を検討します。

**Exercise 3.7:** Consider the bank account objects created by `make-account`, with the password modification described in [Exercise 3.3](#). Suppose that our banking system requires the ability to make joint accounts. Define a procedure `make-joint` that accomplishes this. `Make-joint` should take three arguments. The first is a password-protected account. The second argument must match the password with which the account was defined in order for the `make-joint` operation to proceed. The third argument is a new password. `Make-joint` is to create an additional access to the original account using the new password. For example, if `peter-acc` is a bank account with password `open-sesame`, then

[Exercise 3.3](#)で記述したパスワード変更を用いる `make-account` により作成された銀行口座オブジェクトについて考える。私達の銀行システムが連結口座の開設能力を必要とすると仮定しよう。これを達成する手続 `make-joint` を定義せよ。`make-joint` は 3 つの引数を取らねばならない。第一はパスワードで守られた口座である。第二引数はパスワードで `make-joint` 命令が成功するためには口座が開設された時点のパスワードに合致しなければならない。第三引数は新しいパスワードである。`make-joint` は元の口座に対して新しいパスワードを用いる追加のアクセスを作成する。例えば `peter-acc` がパスワード `open-sesame` を用いる銀行口座であれば、

```
(define paul-acc  
  (make-joint peter-acc 'open-sesame 'rosebud))
```

will allow one to make transactions on `peter-acc` using the name `paul-acc` and the password `rosebud`. You may wish to modify your solution to [Exercise 3.3](#) to accommodate this new feature.

上記は `peter-acc` に対し名前 `paul-acc` とパスワード `rosebud` を用いて取引することを可能にする。この新しい機能に対応するためあなたの Exercise 3.3 への解答を変更したいと思うだろう。

**Exercise 3.8:** When we defined the evaluation model in Section 1.1.3, we said that the first step in evaluating an expression is to evaluate its subexpressions. But we never specified the order in which the subexpressions should be evaluated (e.g., left to right or right to left). When we introduce assignment, the order in which the arguments to a procedure are evaluated can make a difference to the result. Define a simple procedure `f` such that evaluating

Section 1.1.3 にて評価モデルを定義した時、式の評価の最初のステップはその部分式を評価することだと述べました。しかし部分式を評価する順についてには指定しませんでした。(例えば左から右や右から左です)。代入を導入する時、手続に対する引数が評価される順は結果に違いを起こせます。以下の式を評価した時に、

```
(+ (f 0) (f 1))
```

will return 0 if the arguments to `+` are evaluated from left to right but will return 1 if the arguments are evaluated from right to left. `+` の引数が左から右へ評価された場合に 0 を返し、右から左へ評価された場合に 1 を返すようにする簡単な手続 `f` を定義せよ。

## 3.2 The Environment Model of Evaluation

When we introduced compound procedures in Chapter 1, we used the substitution model of evaluation (Section 1.1.5) to define what is meant by applying a procedure to arguments:

複合手続を Chapter 1 で紹介した時、手続を引数に適用することが何を意味するか定義するため評価の置換モデルを使用しました (Section 1.1.5)。

- To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

複合手続を引数に適用するため、手続のボディを各形式パラメータを相対する引数で置き換えて評価する。

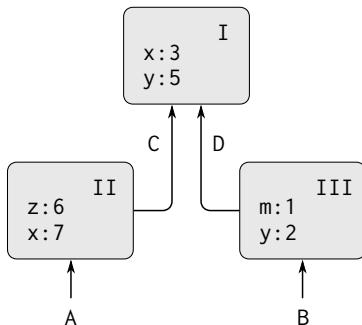
Once we admit assignment into our programming language, such a definition is no longer adequate. In particular, [Section 3.1.3](#) argued that, in the presence of assignment, a variable can no longer be considered to be merely a name for a value. Rather, a variable must somehow designate a “place” in which values can be stored. In our new model of evaluation, these places will be maintained in structures called *environments*.

一旦代入を私達のプログラミング言語で認めれば、そのような定義は最早適切ではありません。具体的には[Section 3.1.3](#)で議論しましたが、代入の出現により、変数は最早単に値に対する名前であると考えることができません。そうでなく、変数はどうにかして値が格納できる“場所”を指定することになります。私達の新しい評価モデルではこれらの場所は*environments(環境)*と呼ばれる構造に保存されます。

An environment is a sequence of *frames*. Each frame is a table (possibly empty) of *bindings*, which associate variable names with their corresponding values. (A single frame may contain at most one binding for any variable.) Each frame also has a pointer to its *enclosing environment*, unless, for the purposes of discussion, the frame is considered to be *global*. The *value of a variable* with respect to an environment is the value given by the binding of the variable in the first frame in the environment that contains a binding for that variable. If no frame in the sequence specifies a binding for the variable, then the variable is said to be *unbound in the environment*.

環境は*frames(フレーム)*の列です。各フレームは*bindings(束縛)*の(空の可能性のある)テーブルで、変数名とそれらが相対する値とを結び付けます。(单一のフレームは任意の変数に対してたかだか1つの束縛を保持します)。各フレームはまた議論の目的のためフレームが*global(グローバル、大域的)*だと認識されない限り、*enclosing environment(外部環境)*へのポインタを持ちます。環境に対して*value of a variable(変数の値)*はその変数に対する束縛を持つ環境内の最初のフレーム内の変数の束縛により与えられる値です。もし列内の全てのフレームがその変数に対する束縛を指定しない場合、その変数はその環境に*unbound(束縛されない)*と呼びます。

[Figure 3.1](#) shows a simple environment structure consisting of three frames, labeled I, II, and III. In the diagram, A, B, C, and D are pointers to environments. C and D point to the same environment. The variables *z* and *x* are bound in frame II, while *y* and *x* are bound in frame I. The value of *x* in environment D is 3. The value of *x* with respect to environment B is also 3. This is determined as follows: We examine the first frame in the sequence (frame III)



**Figure 3.1:** A simple environment structure.

and do not find a binding for  $x$ , so we proceed to the enclosing environment D and find the binding in frame I. On the other hand, the value of  $x$  in environment A is 7, because the first frame in the sequence (frame II) contains a binding of  $x$  to 7. With respect to environment A, the binding of  $x$  to 7 in frame II is said to *shadow* the binding of  $x$  to 3 in frame I.

Figure 3.1はI, II, IIIとラベリングした3つのフレームから成る簡単な環境構造を示しています。図の中でA, B, C, Dは環境へのポインタです。CとDは同じ環境を差しています。変数 $z$ と $x$ はフレームIIに束縛され、一方 $y$ と $x$ はフレームIに束縛されます。環境Dの $x$ の値は3です。環境Bに対する $x$ の値もまた3です。これは次のように決定されます。列の最初のフレーム(フレームIII)を調べますが $x$ に対する束縛を見つけられません。そのため外部環境Dで続けてフレームIの中に束縛を見つけます。一方で環境Aでの $x$ の値は7です。列の最初のフレーム(フレームII)が $x$ から7への束縛を含んでいるからです。環境Aに対して、フレームII内の $x$ から7への束縛はフレームIの $x$ から3への束縛を*shadow*(隠蔽する)と言われます。

The environment is crucial to the evaluation process, because it determines the context in which an expression should be evaluated. Indeed, one could say that expressions in a programming language do not, in themselves, have any meaning. Rather, an expression acquires a meaning only with respect to some environment in which it is evaluated. Even the interpretation of an expression as straightforward as  $(+ 1 1)$  depends on an understanding that one is operating in a context in which  $+$  is the symbol for addition. Thus, in our model of

evaluation we will always speak of evaluating an expression with respect to some environment. To describe interactions with the interpreter, we will suppose that there is a global environment, consisting of a single frame (with no enclosing environment) that includes values for the symbols associated with the primitive procedures. For example, the idea that `+` is the symbol for addition is captured by saying that the symbol `+` is bound in the global environment to the primitive addition procedure.

環境は評価プロセスに対し不可欠な存在です。式が評価されるべきコンテキスト(文脈)を決定するためです。実際にプログラミング言語の式、それ自身は意味を持たないと見えるでしょう。そうでなく、式はそれが評価されるある環境に対してのみ意味を獲得します。`(+ 1 1)`のような簡単な式の逐次実行でさえ、`+`が加算のためのシンボルであるというコンテキストのなかで操作しているという合意に依存しています。従って私達の評価モデルにおいて私達は常にある環境に対して式を評価すると述べます。インタプリタとの相互作用を説明するために、単一のフレームから成り立ち(外部環境を持たず)、プリミティブな手続に関連するシステムの値を持つグローバル環境が存在すると仮定します。例えば`+`が加算に対するシンボルであるという考えは、シンボル`+`がグローバル環境においてプリミティブな加算手続に対し束縛されていると言うことで捉えられます。

### 3.2.1 The Rules for Evaluation

The overall specification of how the interpreter evaluates a combination remains the same as when we first introduced it in [Section 1.1.3](#):

インタプリタが組み合わせをどのように評価するかの全体的な仕様は最初に[Section 1.1.3](#)にて紹介した時と同じに残っています。

- To evaluate a combination: 組み合わせを評価するために
  1. Evaluate the subexpressions of the combination.<sup>22</sup>

---

<sup>22</sup>Assignment introduces a subtlety into step 1 of the evaluation rule. As shown in [Exercise 3.8](#), the presence of assignment allows us to write expressions that will produce different values depending on the order in which the subexpressions in a combination are evaluated. Thus, to be precise, we should specify an evaluation order in step 1 (e.g., left to right or right to left). However, this order should always be considered to be an implementation detail, and one should never write programs that depend on some particular order. For instance, a sophisticated compiler might optimize a program by varying the order in which subexpressions are evaluated.

組み合わせの部分式を評価する。<sup>23</sup>

2. Apply the value of the operator subexpression to the values of the operand subexpressions.

オペレータ部分式の値をオペランド部分式の値に適用する。

The environment model of evaluation replaces the substitution model in specifying what it means to apply a compound procedure to arguments.

評価の環境モデルは置換モデルを複合手続を引数に適用することの意味を指定することで置き換えます。

In the environment model of evaluation, a procedure is always a pair consisting of some code and a pointer to an environment. Procedures are created in one way only: by evaluating a  $\lambda$ -expression. This produces a procedure whose code is obtained from the text of the  $\lambda$ -expression and whose environment is the environment in which the  $\lambda$ -expression was evaluated to produce the procedure. For example, consider the procedure definition

評価の環境モデルでは手続は常にあるコードと環境へのポインタのペアから成り立ちます。手續はただ 1 つの方法で作成されます。それは  $\lambda$  式を評価することです。これによりコードが  $\lambda$  式のテキストから得られる手續が生成され、その環境は  $\lambda$  式が手續を生成するために評価された環境になります。例えば以下の手続定義について考えてみましょう。

```
(define (square x)
  (* x x))
```

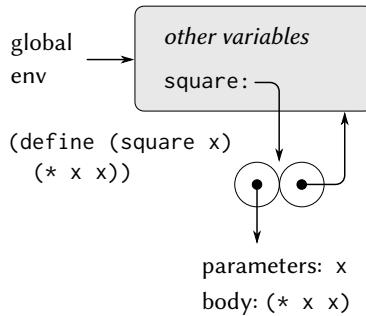
evaluated in the global environment. The procedure definition syntax is just syntactic sugar for an underlying implicit  $\lambda$ -expression. It would have been equivalent to have used

この式はグローバル環境で評価されました。この手続定義の文法は根底にある暗黙的な  $\lambda$  式のための構文糖です。これは次を行った場合と等価です。

```
(define square
  (lambda (x) (* x x)))
```

---

<sup>23</sup>代入は評価ルールのステップ 1 に微妙さを取り込みます。Exercise 3.8 に示されるように代入の存在は組み合わせの部分式がどの順で評価されるかに依存して異なる値を生じます。従って正確に述べればステップ 1 における評価順を指定せねばなりません。(例えば左から右や右から左等)。しかしこの順は常に実装上の詳細と考えられねばなりません。例えば洗練されたコンパイラはどの部分式が評価されるかの順を最適化のために変えるかもしれません。



**Figure 3.2:** Environment structure produced by evaluating `(define (square x) (* x x))` in the global environment.

which evaluates `(lambda (x) (* x x))` and binds **square** to the resulting value, all in the global environment.

これは `(lambda (x) (* x x))` を評価し、全てグローバル環境において **square** をその結果に束縛します。

Figure 3.2 shows the result of evaluating this `define` expression. The procedure object is a pair whose code specifies that the procedure has one formal parameter, namely **x**, and a procedure body `(* x x)`. The environment part of the procedure is a pointer to the global environment, since that is the environment in which the  $\lambda$ -expression was evaluated to produce the procedure. A new binding, which associates the procedure object with the symbol **square**, has been added to the global frame. In general, `define` creates definitions by adding bindings to frames.

Figure 3.2はこの `define` 式の評価結果を示します。手続オブジェクトは手続が 1 つの形式パラメータ **x** を持ち手続のボディが `(* x x)` ことをコードが指定するペアです。手続の環境部分はグローバル環境へのポインタです。それが入式が手続を生じるため評価される環境なためです。シンボル **square** と手続オブジェクトを関連付ける新しい束縛はグローバルなフレームに追加されます。一般的に `define` はフレームに束縛を追加することで定義を作成します。

Now that we have seen how procedures are created, we can describe how procedures are applied. The environment model specifies: To apply a procedure

to arguments, create a new environment containing a frame that binds the parameters to the values of the arguments. The enclosing environment of this frame is the environment specified by the procedure. Now, within this new environment, evaluate the procedure body.

これで手続がどのように作成されるのか学んだので手続がどのように適用されるのかを説明することができます。環境モデルは以下のことを指定します。手続を引数に適用するために、パラメタを引数の値に束縛するフレームを含む新しい環境を作成します。このフレームの外部環境は手続により指定された環境です。さて、この新しい環境で手続のボディを評価します。

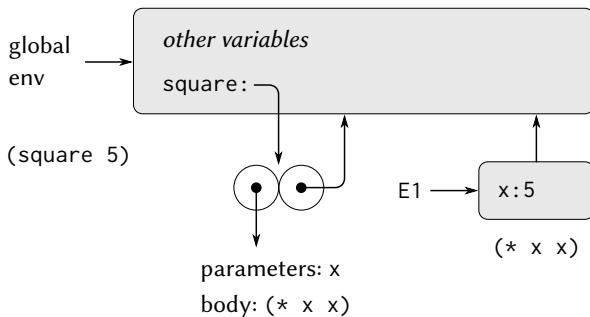
To show how this rule is followed, Figure 3.3 illustrates the environment structure created by evaluating the expression (`square 5`) in the global environment, where `square` is the procedure generated in Figure 3.2. Applying the procedure results in the creation of a new environment, labeled E1 in the figure, that begins with a frame in which `x`, the formal parameter for the procedure, is bound to the argument 5. The pointer leading upward from this frame shows that the frame's enclosing environment is the global environment. The global environment is chosen here, because this is the environment that is indicated as part of the `square` procedure object. Within E1, we evaluate the body of the procedure,  $(\ast \ x \ x)$ . Since the value of `x` in E1 is 5, the result is  $(\ast \ 5 \ 5)$ , or 25.

このルールがどのように従われるかについて示すため、Figure 3.3は式(`square 5`)をグローバル環境にて`square`がFigure 3.2にて生成された手続ある場合に評価することで作成された環境構造を図示しています。この手続の適用は図でE1と示される新しい環境の作成に帰着し、手続の形式パラメタ`x`が引数5に束縛されているフレームで始まっています。このフレームから情報へ向かうポインタはフレームの外部環境がグローバル環境であることを示します。`square`手続オブジェクトの一部として示される環境であるためここでグローバル環境が選択されます。E1の中では手続のボディ $(\ast \ x \ x)$ を評価します。E1中の`x`の値は5であるため結果は $(\ast \ 5 \ 5)$ 、つまり25です。

The environment model of procedure application can be summarized by two rules:

手続適用の環境モデルは2つのルールでまとめられます。

- A procedure object is applied to a set of arguments by constructing a frame, binding the formal parameters of the procedure to the arguments of the call, and then evaluating the body of the procedure in the context of the new environment constructed. The new frame has as its enclosing



**Figure 3.3:** Environment created by evaluating `(square 5)` in the global environment.

environment the environment part of the procedure object being applied.  
手続オブジェクトはフレームを構築、手続の形式パラメタを呼出の引数へ束縛し、新しく構築された環境のコンテキストにて手続のボディを評価することで引数の集合に手続を適用することができる。

- A procedure is created by evaluating a  $\lambda$ -expression relative to a given environment. The resulting procedure object is a pair consisting of the text of the  $\lambda$ -expression and a pointer to the environment in which the procedure was created.  
手続は与えられた環境に関する  $\lambda$  式を評価することで作成される。結果としての手続オブジェクトは  $\lambda$  式のテキストと手続が作成された環境へのポインタから成るペアである。

We also specify that defining a symbol using `define` creates a binding in the current environment frame and assigns to the symbol the indicated value.<sup>24</sup> Finally, we specify the behavior of `set!`, the operation that forced us to introduce the environment model in the first place. Evaluating the expression

---

<sup>24</sup>If there is already a binding for the variable in the current frame, then the binding is changed. This is convenient because it allows redefinition of symbols; however, it also means that `define` can be used to change values, and this brings up the issues of assignment without explicitly using `set!`. Because of this, some people prefer redefinitions of existing symbols to signal errors or warnings.

`(set! <variable> <value>)` in some environment locates the binding of the variable in the environment and changes that binding to indicate the new value. That is, one finds the first frame in the environment that contains a binding for the variable and modifies that frame. If the variable is unbound in the environment, then `set!` signals an error.

`define` を用いてのシンボルの定義は現在の環境フレームに束縛を作成し、そのシンボルに指示された値を束縛することもまた指摘します。<sup>25</sup> 最後に、`set!` の振舞を指定します。私達にそもそも環境モデルの導入を強いた命令です。ある環境で式 `(set! <variable> <value>)` を評価することはその環境に束縛を位置付け、その束縛を新しい値を示すよう変更します。つまり `set!` は環境でその変数の束縛を持つ最初のフレームを探しそのフレームを変更します。もし変数がその環境では束縛されていないのであれば `set!` はエラーを発します。

These evaluation rules, though considerably more complex than the substitution model, are still reasonably straightforward. Moreover, the evaluation model, though abstract, provides a correct description of how the interpreter evaluates expressions. In Chapter 4 we shall see how this model can serve as a blueprint for implementing a working interpreter. The following sections elaborate the details of the model by analyzing some illustrative programs.

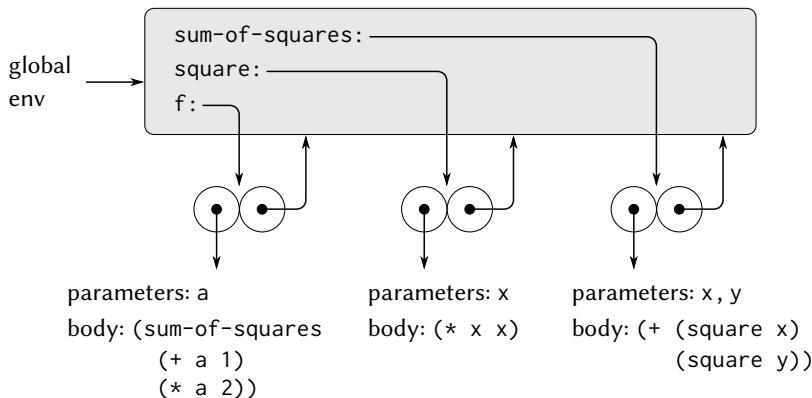
これらの評価ルールは置換モデルより大幅により複雑ですが、依然として適度に容易です。さらに環境モデルは抽象的ですがインタプリタが式をどのように評価するかの正し説明を与えます。Chapter 4ではこのモデルがどのようにうまく働くインタプリタの実装のための設計図としての役を果たすのかについて学ぶことになります。残りの節ではいくつかの実例となるプログラムを分析することによりこのモデルについての詳細を述べます。

### 3.2.2 Applying Simple Procedures

When we introduced the substitution model in Section 1.1.5 we showed how the combination `(f 5)` evaluates to 136, given the following procedure definitions:

---

<sup>25</sup>もし既にその変数への束縛が現在のフレームに存在する場合、束縛は変更されます。これはシンボルの再定義を可能にするため便利です。しかし `define` が値の変更に使用できること、そしてこれが明示的に `set!` を使用せずとも代入の問題を持ち出すことを意味します。このため既存のシンボルの再定義に対しエラーや警告を発することを好む人達もいます。



**Figure 3.4:** Procedure objects in the global frame.

Section 1.1.5にて置換モデルを紹介した時、以下の手続定義を与えられた場合に合成( $f \circ g$ )がどのように $136$ として評価されるかについて説明しました。

```
(define (square x)
  (* x x))
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

We can analyze the same example using the environment model. Figure 3.4 shows the three procedure objects created by evaluating the definitions of `f`, `square`, and `sum-of-squares` in the global environment. Each procedure object consists of some code, together with a pointer to the global environment.

同じ式を環境モデルを用いて分析できます。Figure 3.4は3つの手続オブジェクトが `f`, `square`, and `sum-of-squares` の定義を評価することでグローバル環境に作成されたことを示します。各手続オブジェクトはいくつかのコードとグローバル環境へのポインタから成り立ちます。

In Figure 3.5 we see the environment structure created by evaluating the expression (f 5). The call to `f` creates a new environment `E1` beginning with a frame in which `a`, the formal parameter of `f`, is bound to the argument `5`. In

E1, we evaluate the body of f:

Figure 3.5は式 (f 5) を評価することで作成された環境構造です。f の呼出により f の形式パラメタ a が引数 5 に束縛されるフレームで始まる新しい環境 E1 が作成されます。E1 の中で f のボディを評価します。

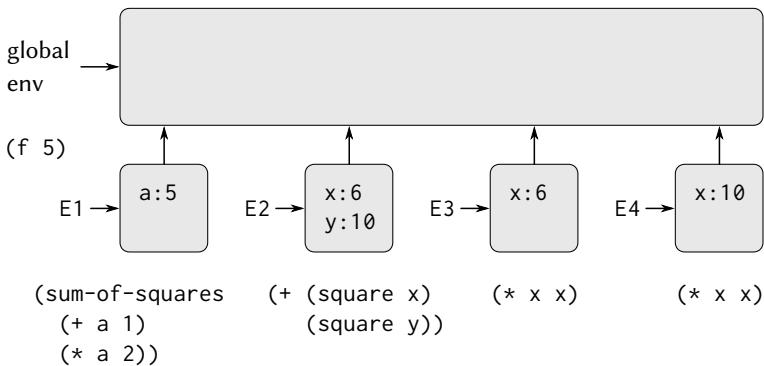
```
(sum-of-squares (+ a 1) (* a 2))
```

To evaluate this combination, we first evaluate the subexpressions. The first subexpression, **sum-of-squares**, has a value that is a procedure object. (Notice how this value is found: We first look in the first frame of E1, which contains no binding for **sum-of-squares**. Then we proceed to the enclosing environment, i.e. the global environment, and find the binding shown in Figure 3.4.) The other two subexpressions are evaluated by applying the primitive operations + and \* to evaluate the two combinations (+ a 1) and (\* a 2) to obtain 6 and 10, respectively.

この合成式を評価するために最初に部分式を評価します。最初の部分式 **sum-of-squares** は手続オブジェクトである値を持っています。(この値がどのように見つけられるかに注意して下さい。最初に E1 の第一フレームを調べますが **sum-of-squares** の束縛はありません。次に外部環境に進みます。つまりグローバル環境です。そこでFigure 3.4に示すように束縛を見つけます)。他の 2 つの部分式はプリミティブな命令 + と \* を、2 つの合成式 (+ a 1) と (\* a 2) を評価しそれぞれ 6 と 10 を得るために適用することで評価されます。

Now we apply the procedure object **sum-of-squares** to the arguments 6 and 10. This results in a new environment E2 in which the formal parameters x and y are bound to the arguments. Within E2 we evaluate the combination (+ (**square** x) (**square** y)). This leads us to evaluate (**square** x), where **square** is found in the global frame and x is 6. Once again, we set up a new environment, E3, in which x is bound to 6, and within this we evaluate the body of **square**, which is (\* x x). Also as part of applying **sum-of-squares**, we must evaluate the subexpression (**square** y), where y is 10. This second call to **square** creates another environment, E4, in which x, the formal parameter of **square**, is bound to 10. And within E4 we must evaluate (\* x x).

これで手続オブジェクト **sum-of-squares** を引数 6 と 10 に適用します。結果は形式パラメタ x と y が引数に束縛される新しい環境 E2 へ帰着します。E2 内では合成 (+ (**square** x) (**square** y)) を評価します。これが (**square** x) の評価へと移り、**square** はグローバルフレームで見つかり、x は 6 です。もう一度、新しい環境 E3 を立ち上げ、x は 6 に束縛され E3 の中で **square** のボデ



**Figure 3.5:** Environments created by evaluating `(f 5)` using the procedures in Figure 3.4.

イ `(* x x)` が評価されます。また `sum-of-squares` の適用の一部として部分式 `(square y)` も評価さねばならずそこでは `y` は 10 です。この 2 つ目の `square` の呼出がまた別の環境 `E4` を作成し、そこでは `square` の形式パラメタ `x` は 10 に束縛されます。そして `E4` の中では `(* x x)` を評価せねばなりません。

The important point to observe is that each call to `square` creates a new environment containing a binding for `x`. We can see here how the different frames serve to keep separate the different local variables all named `x`. Notice that each frame created by `square` points to the global environment, since this is the environment indicated by the `square` procedure object.

確認すべき重要な点は `square` の各呼出が `x` の束縛を持つ新しい環境を構築することです。ここで私達は異なるフレームがどのようにして全て `x` と名付けられた異なるローカル変数の独立を保つかについて見ることができます。`square` により作られた各フレームがグローバル環境を差していることに注意して下さい。これは `square` 手続オブジェクトが指す環境であるためです。

After the subexpressions are evaluated, the results are returned. The values generated by the two calls to `square` are added by `sum-of-squares`, and this result is returned by `f`. Since our focus here is on the environment structures, we will not dwell on how these returned values are passed from call to call; however, this is also an important aspect of the evaluation process, and we will

return to it in detail in Chapter 5.

部分式が評価された後に結果が返されます。2つの `square` の呼出により作成された値は `sum-of-squares` により加算され、この結果が `f` により返されます。ここでの私達の焦点は環境構造上にありますのでこれらの返された値が呼出から呼出へどのように渡されるかについては長々と説明は致しません。しかし、これはまた評価処理の重要な側面であり、Chapter 5にてこれの詳細に戻ります。

**Exercise 3.9:** In Section 1.2.1 we used the substitution model to analyze two procedures for computing factorials, a recursive version

Section 1.2.1にて指数演算のための2つの手続を解析するために置換モデルを使用した。以下が再帰版であり、

```
(define (factorial n)
  (if (= n 1) 1 (* n (factorial (- n 1)))))
```

and an iterative version 以下は反復版である。

```
(define (factorial n) (fact-iter 1 1 n))
(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

Show the environment structures created by evaluating  
(`factorial 6`) using each version of the `factorial` procedure.<sup>26</sup>

各版の `factorial` 手続を用いて (`factorial 6`) を評価した場合に作成される環境構造を示せ。<sup>27</sup>

<sup>26</sup>The environment model will not clarify our claim in Section 1.2.1 that the interpreter can execute a procedure such as `fact-iter` in a constant amount of space using tail recursion. We will discuss tail recursion when we deal with the control structure of the interpreter in Section 5.4.

<sup>27</sup>環境モデルはインタプリタは `fact-iter` のような手続を末尾再帰を用いることで一定量の記憶域にて実行できるという Section 1.2.1での私達の主張を明確にはしません。末尾再帰についてはSection 5.4にてインタプリタのコントロール構造を取り扱う時に議論します。

### 3.2.3 Frames as the Repository of Local State

We can turn to the environment model to see how procedures and assignment can be used to represent objects with local state. As an example, consider the “withdrawal processor” from [Section 3.1.1](#) created by calling the procedure

手続と代入がどのようにしてローカルな状態を持つオブジェクトを表現するために利用できるかを知るために環境モデルに助けを求めることができます。

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

Let us describe the evaluation of

次の定義の評価について説明してみましょう。

```
(define W1 (make-withdraw 100))
```

followed by

以下のように用いたとします。

```
(W1 50)
```

```
50
```

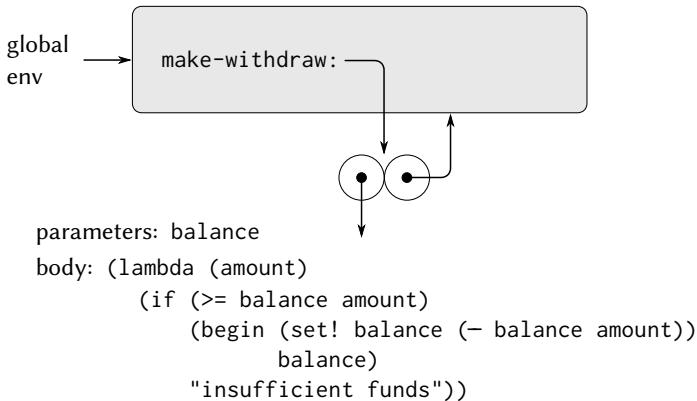
[Figure 3.6](#) shows the result of defining the `make-withdraw` procedure in the global environment. This produces a procedure object that contains a pointer to the global environment. So far, this is no different from the examples we have already seen, except that the body of the procedure is itself a  $\lambda$ -expression.

[Figure 3.6](#)はグローバル環境における `make-withdraw` 手続の定義の結果を示します。グローバル環境へのポインタを持つ手続オブジェクトを作成します。今の所、これは今までに見た例から異なる点はありません。ただし手続のボディそれ自身が  $\lambda$  式であることが異なります。

The interesting part of the computation happens when we apply the procedure `make-withdraw` to an argument:

演算の面白い部分は手続 `make-withdraw` を引数に適用した時に起こります。

```
(define W1 (make-withdraw 100))
```

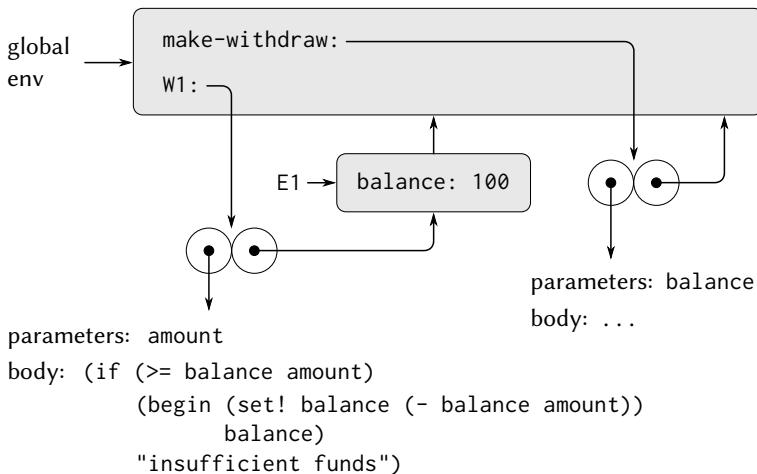


**Figure 3.6:** Result of defining `make-withdraw` in the global environment.

We begin, as usual, by setting up an environment  $E_1$  in which the formal parameter `balance` is bound to the argument 100. Within this environment, we evaluate the body of `make-withdraw`, namely the  $\lambda$ -expression. This constructs a new procedure object, whose code is as specified by the `lambda` and whose environment is  $E_1$ , the environment in which the `lambda` was evaluated to produce the procedure. The resulting procedure object is the value returned by the call to `make-withdraw`. This is bound to  $W_1$  in the global environment, since the `define` itself is being evaluated in the global environment. [Figure 3.7](#) shows the resulting environment structure.

通常通りに形式パラメタ `balance` が引数 100 に束縛される環境  $E_1$  を設定することから始まります。この環境の中で `make-withdraw` のボディ、即ち入式を評価します。これがコードは `lambda` で指定され、環境が  $E_1$  である新しい手続オブジェクトが構築されます。その  $E_1$  の中で `lambda` が手続を生成するため評価されています。結果の手続オブジェクトは `make-withdraw` を呼び出して返された値です。これはグローバル環境にて  $W_1$  に束縛されます。`define` 自身がグローバル環境にて評価されたためです。[Figure 3.7](#)は結果の環境構造を示します。

Now we can analyze what happens when  $W_1$  is applied to an argument:



**Figure 3.7:** Result of evaluating `(define W1 (make-withdraw 100))`.

これで `W1` が引数に適用された時に何が起こるかを解析できます。

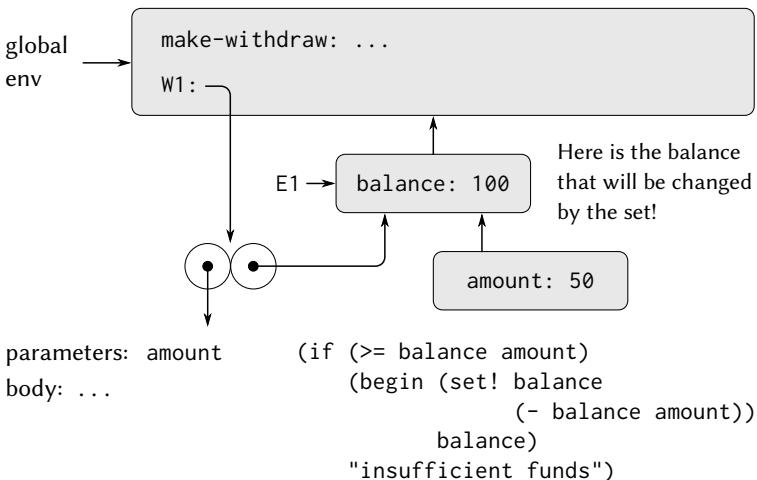
`(W1 50)`

`50`

We begin by constructing a frame in which `amount`, the formal parameter of `W1`, is bound to the argument `50`. The crucial point to observe is that this frame has as its enclosing environment not the global environment, but rather the environment `E1`, because this is the environment that is specified by the `W1` procedure object. Within this new environment, we evaluate the body of the procedure:

`W1` の形式パラメタ `amount` が引数 `50` に束縛されるフレームを構築することから始めます。観察すべき重大な点はこのフレームがその外部環境としてグローバル環境ではなく環境 `E1` を持っている点です。これが `W1` 手続オブジェクトにより指示される環境だからです。この新しい環境の中で手続のボディを評価します。

```
(if (>= balance amount)
  (begin (set! balance (- balance amount)))
```



**Figure 3.8:** Environments created by applying the procedure object W1.

```

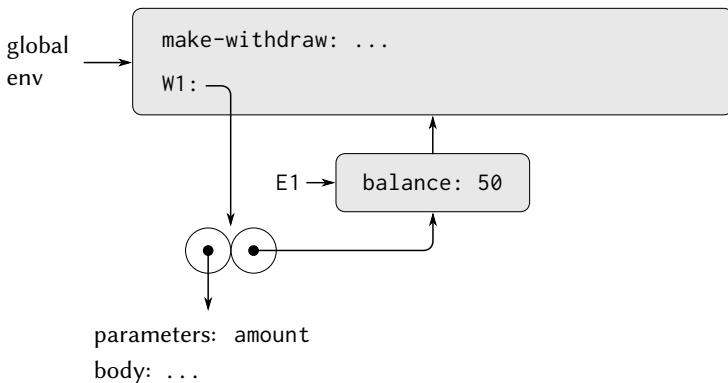
  balance)
"Insufficient funds")

```

The resulting environment structure is shown in Figure 3.8. The expression being evaluated references both `amount` and `balance`. `Amount` will be found in the first frame in the environment, while `balance` will be found by following the enclosing-environment pointer to E1.

結果の環境構造はFigure 3.8に示されます。評価された式は `amount` と `balance` の両方を参照します。`amount` は環境の最初のフレームに見つかりますが、`balance` は外部環境ポインタに従って E1 にて見つかります。

When the `set!` is executed, the binding of `balance` in E1 is changed. At the completion of the call to W1, `balance` is 50, and the frame that contains `balance` is still pointed to by the procedure object W1. The frame that binds `amount` (in which we executed the code that changed `balance`) is no longer relevant, since the procedure call that constructed it has terminated, and there are no pointers to that frame from other parts of the environment. The next time W1 is called, this will build a new frame that binds `amount` and whose enclosing environment



**Figure 3.9:** Environments after the call to W1.

is E1. We see that E1 serves as the “place” that holds the local state variable for the procedure object W1. Figure 3.9 shows the situation after the call to W1.

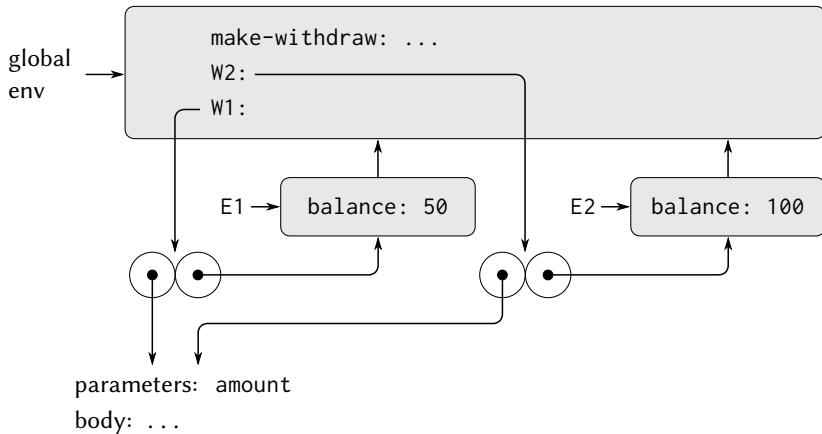
`set!` が実行された時、E1 中の `balance` の束縛は変更されます。W1 の呼出が終了する時 `balance` は 50 で、`balance` を含むフレームは依然手続オブジェクト W1 から指されています。`amount` を束縛する（その中で `balance` を変更するコードを実行した）フレームは最早関係が存在しません。それを構築した手続呼出は停止したためです。そしてその環境の他の部分からそのフレームを指すポインタは存在しません。次回 W1 が呼ばれた時、`amount` を束縛する新しいフレームが構築されその外部環境は E1 になります。私達は E1 が手続オブジェクト W1 のためのローカル状態を持つ“場所”的役割を果たすのを見ました。Figure 3.9 は W1 を呼び出した後の状況を示します。

Observe what happens when we create a second “withdraw” object by making another call to `make-withdraw`:

二つ目の “withdraw” オブジェクトを別の `make-withdraw` 呼出を行うことで作成した時に何が起こるかについて観察して下さい。

```
(define W2 (make-withdraw 100))
```

This produces the environment structure of Figure 3.10, which shows that W2 is a procedure object, that is, a pair with some code and an environment. The environment E2 for W2 was created by the call to `make-withdraw`. It contains



**Figure 3.10:** Using `(define W2 (make-withdraw 100))` to create a second object.

a frame with its own local binding for `balance`. On the other hand, `W1` and `W2` have the same code: the code specified by the  $\lambda$ -expression in the body of `make-withdraw`.<sup>28</sup> We see here why `W1` and `W2` behave as independent objects. Calls to `W1` reference the state variable `balance` stored in `E1`, whereas calls to `W2` reference the `balance` stored in `E2`. Thus, changes to the local state of one object do not affect the other object.

これによりFigure 3.10の環境構造が生成され `W2` が手続オブジェクトであり、ある程度のコードと環境によるペアであることを示しています。`W2` のための環境 `E2` は `make-withdraw` の呼出により作成されます。それ専用の `balance` のためのローカルな状態を持つフレームを含みます。一方で `W1` と `W2` は同じコードを持ちます。`make-withdraw` のボディ内の  $\lambda$  式によりコードは指定されています。<sup>29</sup> なぜ `W1` と `W2` が独立したオブジェクトとして振る舞うのかをここで

<sup>28</sup>Whether `W1` and `W2` share the same physical code stored in the computer, or whether they each keep a copy of the code, is a detail of the implementation. For the interpreter we implement in Chapter 4, the code is in fact shared.

<sup>29</sup>`W1` と `W2` が計算機内の同じ物理コードを共有しているかどうか、またはそれぞれがコードのコピーを持っているのかは実装上の詳細です。Chapter 4で実装するインタプリタ

見ました。W1 の呼出は E1 に格納された状態変数 `balance` を参照し、一方 W2 の呼出は E2 に格納された `balance` を参照します。従って一方のオブジェクトのローカル状態への変更は他方のオブジェクトに影響を与えません。

**Exercise 3.10:** In the `make-withdraw` procedure, the local variable `balance` is created as a parameter of `make-withdraw`. We could also create the local state variable explicitly, using `let`, as follows: `make-withdraw` 手続ではローカル変数 `balance` は `make-withdraw` のパラメタとして作成される。ローカル状態変数を明示的に `let` を使って以下の様に作成することもできる。

```
(define (make-withdraw initial-amount)
  (let ((balance initial-amount))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))))
```

Recall from Section 1.3.2 that `let` is simply syntactic sugar for a procedure call:

Section 1.3.2で `let` は手続呼出のための単純な構文糖であったことを思い出そう。

```
(let ((<var> <exp>)) <body>)
```

is interpreted as an alternate syntax for

上記は代替的な文法として以下に翻訳される。

```
((lambda (<var>) <body>) <exp>)
```

Use the environment model to analyze this alternate version of `make-withdraw`, drawing figures like the ones above to illustrate the interactions

環境モデルを用いてこの `make-withdraw` の代替版を解析し、先に記述したような図を描き相互作用を説明せよ。

---

ではコードは実際に共有されます。

```
(define W1 (make-withdraw 100))
(W1 50)
(define W2 (make-withdraw 100))
```

Show that the two versions of `make-withdraw` create objects with the same behavior. How do the environment structures differ for the two versions?

`make-withdraw` の 2 つの版が同じ振舞を持つオブジェクトを作成することを示せ。環境構造は 2 つの版でどのように違うか？

### 3.2.4 Internal Definitions

Section 1.1.8 introduced the idea that procedures can have internal definitions, thus leading to a block structure as in the following procedure to compute square roots:

Section 1.1.8では手続が内部定義を持つことができ、結果としてブロック構造へと導くことを説明しました。以下の平方根を求める手続がその例です。

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

Now we can use the environment model to see why these internal definitions behave as desired. Figure 3.11 shows the point in the evaluation of the expression `(sqrt 2)` where the internal procedure `good-enough?` has been called for the first time with `guess` equal to 1.

さて環境モデルを用いてなぜこれらの内部定義が希望通りに振る舞うのかを調べることができます。Figure 3.11は内部手続 `good-enough?` が `guess` が 1 に等しい場合に最初に呼ばれた状態で式 `(sqrt 2)` を評価した時点を示しています。

Observe the structure of the environment. `Sqrt` is a symbol in the global environment that is bound to a procedure object whose associated environment is the global environment. When `sqrt` was called, a new environment E1 was formed, subordinate to the global environment, in which the parameter `x` is bound to 2. The body of `sqrt` was then evaluated in E1. Since the first expression in the body of `sqrt` is

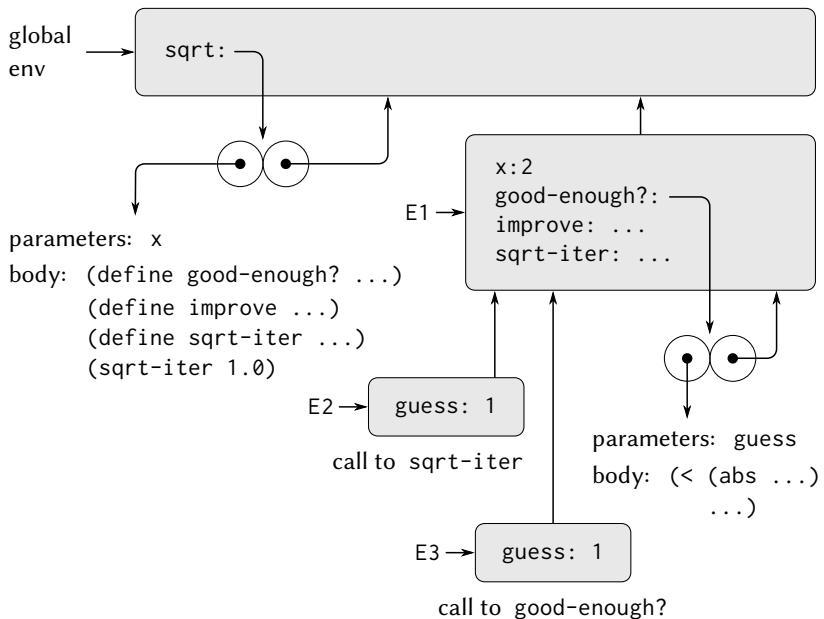
環境構造を観察して下さい。`sqrt` はグローバル環境におけるシンボルであり手続オブジェクトに束縛され、その関連する環境はグローバル環境です。`sqrt` が呼ばれた時、新しい環境 E1 が形成されグローバル環境の下位に置かれ、その中ではパラメタ `x` が 2 に束縛されます。`sqrt` のボディが次に E1 の中に評価されます。`sqrt` のボディの最初の式は以下であり、

```
(define (good-enough? guess)
  (< (abs (- (square guess) x)) 0.001))
```

evaluating this expression defined the procedure `good-enough?` in the environment E1. To be more precise, the symbol `good-enough?` was added to the first frame of E1, bound to a procedure object whose associated environment is E1. Similarly, `improve` and `sqrt-iter` were defined as procedures in E1. For conciseness, Figure 3.11 shows only the procedure object for `good-enough?`.

この式を評価すると手続 `good-enough?` が環境 E1 の中に定義されます。具体的には、シンボル `good-enough?` が E1 の最初のフレームに追加され環境 E1 を指す手続オブジェクトに束縛されます。同様に `improve` と `sqrt-iter` が E1 の中に手続として定義されます。簡潔さのために、Figure 3.11 は `good-enough?` に対する手続オブジェクトのみを示しています。

After the local procedures were defined, the expression (`sqrt-iter` 1.0) was evaluated, still in environment E1. So the procedure object bound to `sqrt-iter` in E1 was called with 1 as an argument. This created an environment E2 in which `guess`, the parameter of `sqrt-iter`, is bound to 1. `Sqrt-iter` in turn called `good-enough?` with the value of `guess` (from E2) as the argument for `good-enough?`. This set up another environment, E3, in which `guess` (the parameter of `good-enough?`) is bound to 1. Although `sqrt-iter` and `good-enough?` both have a parameter named `guess`, these are two distinct local variables located in different frames. Also, E2 and E3 both have E1 as their enclosing environment, because the `sqrt-iter` and `good-enough?` procedures both have E1 as their environment part. One consequence of this is that the symbol `x` that appears in the body of `good-enough?` will reference the binding



**Figure 3.11:** `Sqrt` procedure with internal definitions.

of `x` that appears in `E1`, namely the value of `x` with which the original `sqrt` procedure was called.

ローカル手続が定義された後に、式 `(sqrt-iter 1.0)` がまた環境 `E1` の中で評価されます。そのため `E1` の中で `sqrt-iter` に束縛された手続オブジェクトが引数 1 にて呼ばれます。これが環境 `E2` を作成し `sqrt-iter` のパラメタである `guess` が 1 に束縛される。`sqrt-iter` は次に `good-enough?` を (`E2` の) `guess` の値を引数として呼びます。これが別の環境 `E3` を構築し (`good-enough?` の引数である) `guess` が 1 に束縛されます。`sqrt-iter` と `good-enough?` の両方が `guess` という名前のパラメタを持ちますが、2 つの区別可能なローカル変数が異なるフレームの中に存在します。また `E2` と `E3` の両方が `E1` を外部環境として持っています。手続 `sqrt-iter` と `good-enough?` の両方が `E1` をそれらの環境部分として持つためです。この結果の 1 つとして `good-enough?` のボディ内の

シンボル `x` は `E1` 内に存在する `x` の束縛を参照します。即ち元の `sqrt` 手続が呼ばれた時の `x` の値です。

The environment model thus explains the two key properties that make local procedure definitions a useful technique for modularizing programs:

環境モデルは従ってローカル手続定義をプログラムのモジュール化するための便利なテクニックとする 2 つの鍵となる性質を説明します。

- The names of the local procedures do not interfere with names external to the enclosing procedure, because the local procedure names will be bound in the frame that the procedure creates when it is run, rather than being bound in the global environment.

ローカル手続の名前は(直の)外部手続の外側の名前と衝突しない。ローカル手続の名前は手続が実行される時に作成したフレーム内にて束縛されるのであり、グローバル環境内で束縛される訳でありません。

- The local procedures can access the arguments of the enclosing procedure, simply by using parameter names as free variables. This is because the body of the local procedure is evaluated in an environment that is subordinate to the evaluation environment for the enclosing procedure.

ローカルな手続はそれを内包する外部手続の引数にアクセスすることができます。単純にパラメタの名前を自由変数として用いるだけです。これはローカル手続のボディは外部手続のための評価環境の下位に置かれる環境内で評価されるためです。

**Exercise 3.11:** In Section 3.2.3 we saw how the environment model described the behavior of procedures with local state. Now we have seen how internal definitions work. A typical message-passing procedure contains both of these aspects. Consider the bank account procedure of Section 3.1.1:

Section 3.2.3では環境モデルがどのようにローカルな状態を持つ手続の振舞を説明するかについて学んだ。ここまでで内部定義がどのように働くかについて理解した。典型的なメッセージパッシング手続はこれらの側面の両方を持っている。Section 3.1.1の銀行口座について考えよう。

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
```

```

(begin (set! balance (- balance amount))
       balance)
      "Insufficient funds"))
(define (deposit amount)
  (set! balance (+ balance amount))
  balance)
(define (dispatch m)
  (cond ((eq? m 'withdraw) withdraw)
        ((eq? m 'deposit) deposit)
        (else
         (error "Unknown request:
                  MAKE-ACCOUNT"
               m)))))

dispatch)

```

Show the environment structure generated by the sequence of interactions

以下の応答により生成される環境構造を示せ。

```

(define acc (make-account 50))
((acc 'deposit) 40)
90
((acc 'withdraw) 60)
30

```

Where is the local state for `acc` kept? Suppose we define another account

`acc` の局所状態はどこにあるか? 別の口座を定義したとする。

```
(define acc2 (make-account 100))
```

How are the local states for the two accounts kept distinct? Which parts of the environment structure are shared between `acc` and `acc2`?

2つの口座の局所状態はどのように区別されるか? 環境構造のどの部分が `acc` と `acc2` にて共有されるか?

### 3.3 Modeling with Mutable Data

Chapter 2 dealt with compound data as a means for constructing computational objects that have several parts, in order to model real-world objects that have several aspects. In that chapter we introduced the discipline of data abstraction, according to which data structures are specified in terms of constructors, which create data objects, and selectors, which access the parts of compound data objects. But we now know that there is another aspect of data that Chapter 2 did not address. The desire to model systems composed of objects that have changing state leads us to the need to modify compound data objects, as well as to construct and select from them. In order to model compound objects with changing state, we will design data abstractions to include, in addition to selectors and constructors, operations called *mutators*, which modify data objects. For instance, modeling a banking system requires us to change account balances. Thus, a data structure for representing bank accounts might admit an operation

Chapter 2では複合データを計算オブジェクトを構築する手段として扱いました。これは複数の側面を持つ実際の世界のオブジェクトをモデル化するためにいくつかの部品を持ちます。またChapter 2ではデータオブジェクトを作成するコンストラクタと、複合データオブジェクトの部品にアクセスするセレクタを用いてどのデータ構造が指定されるかに準ずるデータ抽象化の規律についても紹介しました。しかし今ではChapter 2が解決しなかったデータの別の側面があることを私達は知りました。状態が変化するオブジェクトにより成るシステムをモデル化したいという欲求複合データオブジェクトを構築することやそれらから選択することと同様に変更することの必要性へと導きます。変換する状態を持つ複合オブジェクトをモデル化するために、セレクタやコンストラクタに追加して、データオブジェクトを変更する*mutators*(ミュータータ、変化させる物)と呼ばれる命令を含むようにデータ抽象化を設計することにします。例えば、銀行システムのモデル化は口座の差引残高を変更する必要があります。従って銀行口座を表現するデータ構造は以下の命令を許可するでしょう。

```
(setbalance! <account> <newvalue>)
```

that changes the balance of the designated account to the designated new value. Data objects for which mutators are defined are known as *mutable data objects*.

これは指定した口座の差引残高を指定した新しい値に変更します。ミュータータが定義されたデータオブジェクトは*mutable data objects*(ミュータブル

データオブジェクト、変更可能なオブジェクト)として知られます。

Chapter 2 introduced pairs as a general-purpose “glue” for synthesizing compound data. We begin this section by defining basic mutators for pairs, so that pairs can serve as building blocks for constructing mutable data objects. These mutators greatly enhance the representational power of pairs, enabling us to build data structures other than the sequences and trees that we worked with in Section 2.2. We also present some examples of simulations in which complex systems are modeled as collections of objects with local state.

Chapter 2は複合データを合成するため汎用目的の“糊”としてのペアを紹介しました。この節はペアのための基本的なミュータータを定義することから始め、ペアが変更可能なデータオブジェクトを構築するための架設ブロックとして供給できるようにします。これらのミュータータはペアの表現力を大きく拡張し、Section 2.2で用いた列と木以外のデータ構造を構築することを可能にします。複雑なシステムが局所状態を持つオブジェクトの集合としてモデル化されるシミュレーションのいくつかの例も紹介します。

### 3.3.1 Mutable List Structure

The basic operations on pairs—`cons`, `car`, and `cdr`—can be used to construct list structure and to select parts from list structure, but they are incapable of modifying list structure. The same is true of the list operations we have used so far, such as `append` and `list`, since these can be defined in terms of `cons`, `car`, and `cdr`. To modify list structures we need new operations.

ペア上の基本的な命令—`cons`, `car`, `cdr`—はリスト構造の構築とリスト構造からの部品の選択に用いることができます。しかしそれらはリスト構造を変更する能力はありませんでした。同じことが今までに使用した `append` や `list` の様なリスト命令にも正しいと言えます。これらが `cons`, `car`, `cdr` を用いて定義できるためです。リスト構造を変更するため新しい命令が必要です。

The primitive mutators for pairs are `set-car!` and `set-cdr!`. `Set-car!` takes two arguments, the first of which must be a pair. It modifies this pair, replacing the `car` pointer by a pointer to the second argument of `set-car!`.<sup>30</sup>

ペアのプリミティブなミュータータは `set-car!` と `set-cdr!` です。`set-car!` は 2 つの引数を取り、第一引数はペアでなければなりません。このペアの

---

<sup>30</sup> `Set-car!` and `set-cdr!` return implementation-dependent values. Like `set!`, they should be used only for their effect.

`car` ポインタを `set-car!` の第二引数へのポインタで置き換えることでペアを変更します。<sup>31</sup>

As an example, suppose that `x` is bound to the list `((a b) c d)` and `y` to the list `(e f)` as illustrated in Figure 3.12. Evaluating the expression `(set-car! x y)` modifies the pair to which `x` is bound, replacing its `car` by the value of `y`. The result of the operation is shown in Figure 3.13. The structure `x` has been modified and would now be printed as `((e f) c d)`. The pairs representing the list `(a b)`, identified by the pointer that was replaced, are now detached from the original structure.<sup>32</sup>

例としてFigure 3.12に示すように `x` がリスト `((a b) c d)` に、`y` がリスト `(e f)` に束縛されているとします。式 `(set-car! x y)` の評価は `x` が束縛されているペアを変更し、その `car` を `y` の値で置き換えます。命令の結果はFigure 3.13に示されています。構造 `x` が変更され `((e f) c d)` となりました。リスト `(a b)` を表すペアは、置き換えられたポインタにより特定されていましが、元の構造から取り外されました。<sup>33</sup>

Compare Figure 3.13 with Figure 3.14, which illustrates the result of executing `(define z (cons y (cdr x)))` with `x` and `y` bound to the original lists of Figure 3.12. The variable `z` is now bound to a new pair created by the `cons` operation; the list to which `x` is bound is unchanged.

Figure 3.13とFigure 3.14を比べてください。これは `x` と `y` がFigure 3.12の元のリストに束縛されている時に `(define z (cons y (cdr x)))` を実行した結果を図示しています。変数 `z` はこれで `cons` 命令により作成された新しいペアに束縛されます。`x` が束縛されるリストは変更されません。

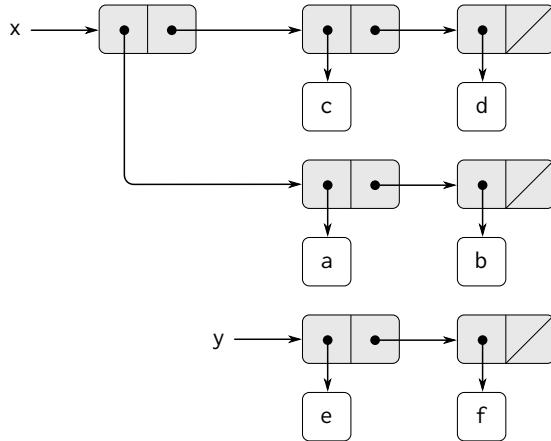
The `set-cdr!` operation is similar to `set-car!`. The only difference is that the `cdr` pointer of the pair, rather than the `car` pointer, is replaced. The effect of executing `(set-cdr! x y)` on the lists of Figure 3.12 is shown in Figure 3.15. Here the `cdr` pointer of `x` has been replaced by the pointer to `(e f)`. Also, the list `(c d)`, which used to be the `cdr` of `x`, is now detached from the structure.

---

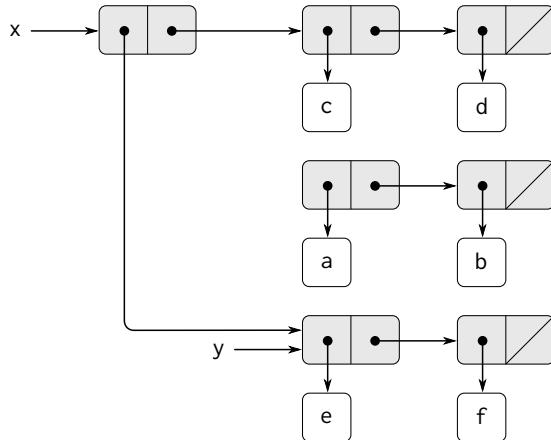
<sup>31</sup>`set-car!` と `set-cdr!` は実装依存な値を返します。`set!` と同様にそれらはそれらの効果のためだけに使用されるべきです。

<sup>32</sup>We see from this that mutation operations on lists can create “garbage” that is not part of any accessible structure. We will see in Section 5.3.2 that Lisp memory-management systems include a *garbage collector*, which identifies and recycles the memory space used by unneeded pairs.

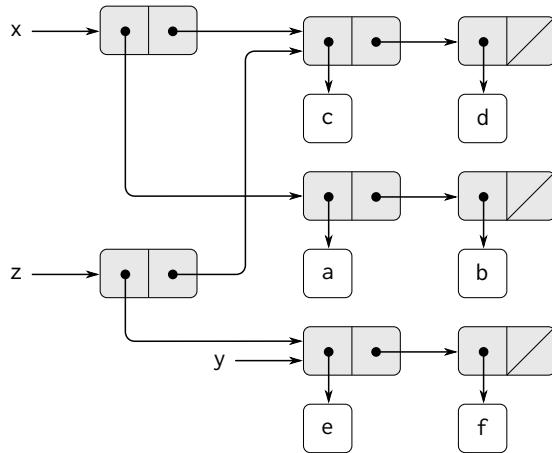
<sup>33</sup>この点からリストの変更命令はどのアクセス可能な構造の部分でもない“garbage”(ゴミ)を作り得ることがわかります。Section 5.3.2にて Lisp のメモリ管理システムが *garbage collector*(ガベージコレクタ、清掃局員)を持ち、それにより必要な無いペアにより使用されているメモリ空間を判断しリサイクルを行います。



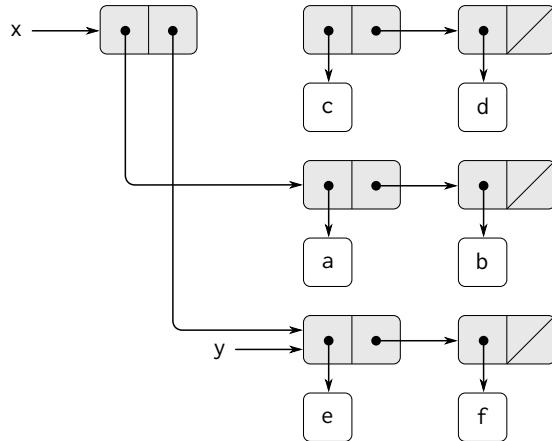
**Figure 3.12:** Lists  $x$ : ((a b) c d) and  $y$ : (e f).



**Figure 3.13:** Effect of  $(\text{set-car! } x y)$  on the lists in [Figure 3.12](#).



**Figure 3.14:** Effect of `(define z (cons y (cdr x)))` on the lists in [Figure 3.12](#).



**Figure 3.15:** Effect of `(set-cdr! x y)` on the lists in [Figure 3.12](#).

`set-cdr!` 命令は `set-car!` と同様です。違いは `car` ポインタでなく、`cdr` ポインタが置き換えられます。[Figure 3.12](#)のリスト上での `(set-cdr! x y)` の実行の結果は[Figure 3.15](#)に示されます。ここでは `x` の `cdr` ポインタは `(e f)` へのポインタにて置き換えられます。また `x` の `cdr` として用いられるリスト `(c d)` はこれで構造から取り外されます。

`Cons` builds new list structure by creating new pairs, while `set-car!` and `set-cdr!` modify existing pairs. Indeed, we could implement `cons` in terms of the two mutators, together with a procedure `get-new-pair`, which returns a new pair that is not part of any existing list structure. We obtain the new pair, set its `car` and `cdr` pointers to the designated objects, and return the new pair as the result of the `cons`.<sup>34</sup>

`cons` は新しいリスト構造を新しいペアを作成することで構築します。一方、`set-car!` と `set-cdr!` は既存のペアを変更します。実際に 2 つのミュータータと既存のリスト構造の一部ではない新しいペアを返す `get-new-pair` を一緒に用いて `cons` を実装することができます。新しいペアを得てからその `car` と `cdr` ポインタに指定されたオブジェクトを設定し、`cons` の結果として返します。<sup>35</sup>

```
(define (cons x y)
  (let ((new (get-new-pair)))
    (set-car! new x)
    (set-cdr! new y)
    new))
```

**Exercise 3.12:** The following procedure for appending lists was introduced in [Section 2.2.1](#):

リストを接続するための以下の手続は[Section 2.2.1](#)で紹介した。

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

---

<sup>34</sup>`Get-new-pair` is one of the operations that must be implemented as part of the memory management required by a Lisp implementation. We will discuss this in [Section 5.3.1](#).

<sup>35</sup>`get-new-pair` は Lisp 実装にて必要とされるメモリ管理の一部として実装されなければならない命令の 1 つです。これについては [Section 5.3.1](#)にて議論します。

`Append` forms a new list by successively `consing` the elements of `x` onto `y`. The procedure `append!` is similar to `append`, but it is a mutator rather than a constructor. It appends the lists by splicing them together, modifying the final pair of `x` so that its `cdr` is now `y`. (It is an error to call `append!` with an empty `x`.)

`append` は `y` に連続して `x` の要素を `cons` することで新しいリストを作る。手続 `append!` は `append` と同様だが、コンストラクタではなくミュータータである。これは `x` の最後のペアを変更しその `cdr` を `y` にし両者を繋ぎ合わせることで `append`(付け加え) する。`(append! x)` を空の `x` にて呼ぶのはエラーとなる。

```
(define (append! x y)
  (set-cdr! (last-pair x) y)
  x)
```

Here `last-pair` is a procedure that returns the last pair in its argument:

ここで `last-pair` はその引数の最後のペアを返す手続である。

```
(define (last-pair x)
  (if (null? (cdr x)) x (last-pair (cdr x))))
```

Consider the interaction

以下の応答について考えよ。

```
(define x (list 'a 'b))
(define y (list 'c 'd))
(define z (append x y))
z
(a b c d)
(cdr x)
<response>
(define w (append! x y))
w
(a b c d)
(cdr x)
<response>
```

What are the missing `<response>`s? Draw box-and-pointer diagrams to explain your answer.

欠けている `<response>` は何か？ 箱とポインタの図をあなたの答を説明するために描け。

**Exercise 3.13:** Consider the following `make-cycle` procedure, which uses the `last-pair` procedure defined in Exercise 3.12:

次の `make-cycle` 手続について考えよ。これはExercise 3.12で定義した `last-pair` 手続を用いる。

```
(define (make-cycle x)
  (set-cdr! (last-pair x) x)
  x)
```

Draw a box-and-pointer diagram that shows the structure `z` created by

以下の様に作成される `z` を表す箱とポインタの図を描け。

```
(define z (make-cycle (list 'a 'b 'c)))
```

What happens if we try to compute `(last-pair z)`?

`(last-pair z)` を演算すると何が置くるか？

**Exercise 3.14:** The following procedure is quite useful, although obscure:

以下の手続はとても便利であるが不明瞭である。

```
(define (mystery x)
  (define (loop x y)
    (if (null? x)
        y
        (let ((temp (cdr x)))
          (set-cdr! x y)
          (loop temp x))))
  (loop x '()))
```

`Loop` uses the “temporary” variable `temp` to hold the old value of the `cdr` of `x`, since the `set-cdr!` on the next line destroys the `cdr`. Explain what `mystery` does in general. Suppose `v` is defined

by `(define v (list 'a 'b 'c 'd))`. Draw the box-and-pointer diagram that represents the list to which `v` is bound. Suppose that we now evaluate `(define w (mystery v))`. Draw box-and-pointer diagrams that show the structures `v` and `w` after evaluating this expression. What would be printed as the values of `v` and `w`?

`loop` は “temporary”(一時的) な変数 `temp` を用いて `x` の `cdr` を保存する。次の行の `set-cdr!` が `cdr` を破壊するためである。`mystery` が通常何を行うのか説明せよ。`v` が `(define v (list 'a 'b 'c 'd))` で定義されているとする。`v` が束縛されるリストを表す箱とポインタの図を描け。次に `(define w (mystery v))` を評価したとする。この式を評価した後の `v` と `w` の構造を表す箱とポインタの図を描け。`v` と `w` の値として何が表示されるか？

## Sharing and identity

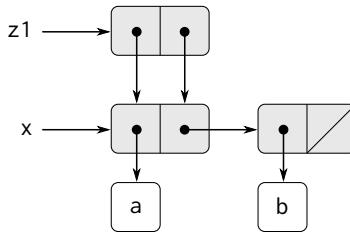
We mentioned in Section 3.1.3 the theoretical issues of “sameness” and “change” raised by the introduction of assignment. These issues arise in practice when individual pairs are *shared* among different data objects. For example, consider the structure formed by

Section 3.1.3で代入の導入に伴う“同一性”と“変更”という論理的な問題について記述しました。これらの問題は実際の所個々のペアが異なるデータオブジェクトの間で *shared*(共有) されている時に問題となります。例えば、以下の様に形成される構造について考えてみて下さい。

```
(define x (list 'a 'b))
(define z1 (cons x x))
```

As shown in Figure 3.16, `z1` is a pair whose `car` and `cdr` both point to the same pair `x`. This sharing of `x` by the `car` and `cdr` of `z1` is a consequence of the straightforward way in which `cons` is implemented. In general, using `cons` to construct lists will result in an interlinked structure of pairs in which many individual pairs are shared by many different structures.

Figure 3.16で示されるように、`z1` はその `car` と `cdr` の両者が同じペア `x` を指している。この `z1` の `car` と `cdr` による `x` の共有は `cons` が直接的な方法で実装されていることによる結果です。一般的に `cons` を用いてリストを構築することは多くの個別のペアが多く異なる構造において共有される、ペアの連結構造に帰着します。



**Figure 3.16:** The list  $z1$  formed by  $(\text{cons } x \ x)$ .

In contrast to Figure 3.16, Figure 3.17 shows the structure created by Figure 3.16とは対照的に、Figure 3.17は以下の式で作成された構造を示します。

```
(define z2 (cons (list 'a 'b) (list 'a 'b)))
```

In this structure, the pairs in the two  $(a \ b)$  lists are distinct, although the actual symbols are shared.<sup>36</sup>

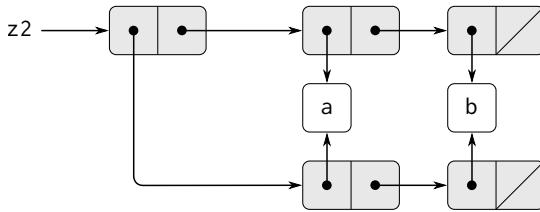
この構造においては、2つの  $(a \ b)$  リスト内のペアは実際のシンボルが共有されていても区別可能です。<sup>37</sup>

When thought of as a list,  $z1$  and  $z2$  both represent “the same” list,  $((a \ b) \ a \ b)$ . In general, sharing is completely undetectable if we operate on lists using only `cons`, `car`, and `cdr`. However, if we allow mutators on list structure, sharing becomes significant. As an example of the difference that sharing can make, consider the following procedure, which modifies the `car` of the structure to which it is applied:

---

<sup>36</sup>The two pairs are distinct because each call to `cons` returns a new pair. The symbols are shared; in Scheme there is a unique symbol with any given name. Since Scheme provides no way to mutate a symbol, this sharing is undetectable. Note also that the sharing is what enables us to compare symbols using `eq?`, which simply checks equality of pointers.

<sup>37</sup>2つのペアは各 `cons` 呼出が新しいペアを返すため区別可能です。シンボルは共有されています。Schemeではどの与えられた名前にも固有のシンボルが存在します。Schemeがシンボルを変更する手段を全く提供しないため、この共有は判別不可能です。共有が単純にポインタの等価性をチェックする `eq?` を用いてシンボルで比較することを可能にする物であることにも注意して下さい。



**Figure 3.17:** The list  $z_2$  formed by `(cons (list 'a 'b) (list 'a 'b))`.

リストとして考えた時、 $z_1$  と  $z_2$  の両方が“同じ”リスト  $((a\ b)\ a\ b)$  を表現します。一般的に共有はリスト上で用いる命令が `cons`, `car`, `cdr` だけならば完全に検出不可能です。しかしリスト構造上で変更を許可するのであれば、共有に気付くことができます。共有が作成できる違いの例として、適用された引数の構造の `car` を変更する以下の手続について考えてみましょう。

```
(define (set-to-wow! x) (set-car! (car x) 'wow) x)
```

Even though  $z_1$  and  $z_2$  are “the same” structure, applying `set-to-wow!` to them yields different results. With  $z_1$ , altering the `car` also changes the `cdr`, because in  $z_1$  the `car` and the `cdr` are the same pair. With  $z_2$ , the `car` and `cdr` are distinct, so `set-to-wow!` modifies only the `car`:

例え  $z_1$  と  $z_2$  が“同じ”構造だとしても、`set-to-wow!` をそれらに適用すると異なる結果を返します。 $z_1$  では `car` の変更は `cdr` も変更します。 $z_1$  では `car` と `cdr` が同じペアであるためです。 $z_2$  では `car` と `cdr` は区別可能なので `set-to-wow!` は `car` のみを変更します。

```
z1
((a b) a b)
(set-to-wow! z1)
((wow b) wow b)

z2
((a b) a b)
(set-to-wow! z2)
((wow b) a b)
```

One way to detect sharing in list structures is to use the predicate `eq?`, which we introduced in [Section 2.3.1](#) as a way to test whether two symbols are equal. More generally, `(eq? x y)` tests whether `x` and `y` are the same object (that is, whether `x` and `y` are equal as pointers). Thus, with `z1` and `z2` as defined in [Figure 3.16](#) and [Figure 3.17](#), `(eq? (car z1) (cdr z1))` is true and `(eq? (car z2) (cdr z2))` is false.

リスト構造中の共有を見つける一つの方法は[Section 2.3.1](#)で2つのシンボルが等しいかテストする方法として紹介した述語 `eq?` を用います。より一般的には `(eq? x y)` は `x` と `y` が同じオブジェクトであるかをテストします(これはつまり `x` と `y` はポインタとして等しいかです)。従って[Figure 3.16](#)と[Figure 3.17](#)で示すよう定義された `z1` と `z2`

As will be seen in the following sections, we can exploit sharing to greatly extend the repertoire of data structures that can be represented by pairs. On the other hand, sharing can also be dangerous, since modifications made to structures will also affect other structures that happen to share the modified parts. The mutation operations `set-car!` and `set-cdr!` should be used with care; unless we have a good understanding of how our data objects are shared, mutation can have unanticipated results.<sup>38</sup>

以降の節で示されるように、ペアで表現可能なデータ構造のレパートリを大きく拡張することが共有を用いてできます。一方で、共有はまた危険であり構造に対して行われる変更がたまたま部品を共有する他の構造に対しても影響を与えます。ミューターティアである `set-car!` と `set-cdr!` は注意深く利用せねばなりません。データオブジェクトがどのように共有されているかを良く理解しなければ変更は予期しない結果を引き起します。<sup>39</sup>

---

<sup>38</sup>The subtleties of dealing with sharing of mutable data objects reflect the underlying issues of “sameness” and “change” that were raised in [Section 3.1.3](#). We mentioned there that admitting change to our language requires that a compound object must have an “identity” that is something different from the pieces from which it is composed. In Lisp, we consider this “identity” to be the quality that is tested by `eq?`, i.e., by equality of pointers. Since in most Lisp implementations a pointer is essentially a memory address, we are “solving the problem” of defining the identity of objects by stipulating that a data object “itself” is the information stored in some particular set of memory locations in the computer. This suffices for simple Lisp programs, but is hardly a general way to resolve the issue of “sameness” in computational models.

<sup>39</sup>ミュータブルなデータオブジェクトの共有の取扱の微妙な部分は[Section 3.1.3](#)で取り上げられた“等価性”と“変更”的根底に横たわる問題を反映しています。そこでは私達の言語に変更を許すことは複合データがそれを構成する部分から何かが異なるという

**Exercise 3.15:** Draw box-and-pointer diagrams to explain the effect of `set-to-wow!` on the structures `z1` and `z2` above.

上記の構造 `z1` と `z2` 上での `set-to-wow!` の効果を説明する箱とポインタの図を描け。

**Exercise 3.16:** Ben Bitdiddle decides to write a procedure to count the number of pairs in any list structure. “It’s easy,” he reasons. “The number of pairs in any structure is the number in the `car` plus the number in the `cdr` plus one more to count the current pair.” So Ben writes the following procedure:

Ben Bitdiddle は任意のリスト構造内のペアの数を数える手続を書くことに決めた。“簡単だよね”と彼は思った。“任意の構造内のペアの数は `car` の中の数と `cdr` の中の数の和に現在のペアを数えるために 1 を足した物”。だから Ben は以下の手続を書いた。

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
          (count-pairs (cdr x))
          1)))
```

Show that this procedure is not correct. In particular, draw box-and-pointer diagrams representing list structures made up of exactly three pairs for which Ben’s procedure would return 3; return 4; return 7; never return at all.

この手続が正しくないことを示せ。具体的にはきっちり 3 つのペアにより作られ Ben の手続が 3, 4, 7 を返すだろう、また Ben の手続が絶対に終了しないリスト構造を表現する箱とポインタの図を描け。

---

“自己同一性”を持たねばならないことを述べました。Lisp ではこの“自己同一性”を `eq?` にてテストされる性質だと考えます。即ち、ポインタの等価性です。多くの Lisp 実装ではポインタが本質的にはメモリアドレスですので、オブジェクトの自己同一性を定義することの“問題の解決”はデータオブジェクト“それ自身”がいくつかの特定の計算機内のメモリ上の場所の集合に格納された情報であることを要求することにより解決します。これは単純な Lisp プログラムには十分ですが、計算モデルの“同一性”的問題を解決する一般的な方法ではありません。

**Exercise 3.17:** Devise a correct version of the `count-pairs` procedure of [Exercise 3.16](#) that returns the number of distinct pairs in any structure. (Hint: Traverse the structure, maintaining an auxiliary data structure that is used to keep track of which pairs have already been counted.)

[Exercise 3.16](#)の `count-pairs` 手続の正しい版を考案せよ。これは任意の構造の中の固有のペアの数を返す。(ヒント: 構造を横断しながらどのペアが既に数えられたかを追跡するために使用する補助的なデータ構造を保存する)。

**Exercise 3.18:** Write a procedure that examines a list and determines whether it contains a cycle, that is, whether a program that tried to find the end of the list by taking successive `cdrs` would go into an infinite loop. [Exercise 3.13](#) constructed such lists.

リストを検査しそれが循環を持つかどうか判断せよ。つまりリストの最後を見つけようとしたプログラムが連続して `cdr` を取ることで無限ループに入るかどうかを判定せよ。[Exercise 3.13](#)にてそのようなリストを構築した。

**Exercise 3.19:** Redo [Exercise 3.18](#) using an algorithm that takes only a constant amount of space. (This requires a very clever idea.) [Exercise 3.18](#)を一定の容量のメモリのみを用いるアルゴリズムを用いて再度行え。(これはとても巧妙なアイデアを必要とする)。

## Mutation is just assignment

When we introduced compound data, we observed in [Section 2.1.3](#) that pairs can be represented purely in terms of procedures:

複合データを紹介した時、[Section 2.1.3](#)にてペアが手続をのみを利用することで表現できることを観察しました。

```
(define (cons x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (error "Undefined operation: CONS" m))))
  dispatch)
(define (car z) (z 'car))
```

```
(define (cdr z) (z 'cdr))
```

The same observation is true for mutable data. We can implement mutable data objects as procedures using assignment and local state. For instance, we can extend the above pair implementation to handle `set-car!` and `set-cdr!` in a manner analogous to the way we implemented bank accounts using `make-account` in Section 3.1.1:

同じ観察結果がミュータブルなデータに対しても正しいと言えます。ミュータブル(可変)なデータオブジェクトを代入と局所状態を用いることで手続として実装可能です。例として上のペアの実装を拡張し、Section 3.1.1で`make-account`を用いて銀行口座を実装した方法とある程度類似して、`set-car!`と`set-cdr!`を扱うことができます。

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else
            (error "Undefined operation: CONS" m))))
  dispatch)
(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
(define (set-car! z new-value)
  ((z 'set-car!) new-value) z)
(define (set-/cdr! z new-value)
  ((z 'set-cdr!) new-value) z)
```

Assignment is all that is needed, theoretically, to account for the behavior of mutable data. As soon as we admit `set!` to our language, we raise all the issues, not only of assignment, but of mutable data in general.<sup>40</sup>

---

<sup>40</sup>On the other hand, from the viewpoint of implementation, assignment requires us to modify the environment, which is itself a mutable data structure. Thus, assignment and mutation are equipotent: Each can be implemented in terms of the other.

代入が可変データの振舞を説明するために論理上必要な物全てです。私達の言語に `set!` を認めるに直ぐに、代入の問題のみでなく、一般的な可変データの全ての問題を引き起しました。<sup>41</sup>

**Exercise 3.20:** Draw environment diagrams to illustrate the evaluation of the sequence of expressions

以下の連続した式の評価を説明する環境の図を描け。

```
(define x (cons 1 2))
(define z (cons x x))
(set-car! (cdr z) 17)
(car x)
17
```

using the procedural implementation of pairs given above. (Compare Exercise 3.11.)

上で与えられた手続型の実装を用いよ。(Exercise 3.11と比較せよ)。

### 3.3.2 Representing Queues

The mutators `set-car!` and `set-cdr!` enable us to use pairs to construct data structures that cannot be built with `cons`, `car`, and `cdr` alone. This section shows how to use pairs to represent a data structure called a queue. Section 3.3.3 will show how to represent data structures called tables.

ミューテータの `set-car!` と `set-cdr!` はペアを用いて `cons`, `car`, `cdr` のみでは不可能なデータ構造を構築可能です。この節ではキューと呼ばれるデータ構造を表現するためにどのようにペアを用いるかについて示します。Section 3.3.3ではテーブル(表)と呼ばれるデータ構造の表現方法について学びます。

A *queue* is a sequence in which items are inserted at one end (called the *rear* of the queue) and deleted from the other end (the *front*). Figure 3.18 shows an initially empty queue in which the items `a` and `b` are inserted. Then `a` is removed, `c` and `d` are inserted, and `b` is removed. Because items are always removed in the order in which they are inserted, a queue is sometimes called a *FIFO* (first in, first out) buffer.

---

<sup>41</sup>一方で、実装上の観点からは代入は環境を変更することを必要とし、環境はそれ自身が可変なデータ構造です。従って代入と変更は等位です。つまり一方は他方を用いることで実装可能です。

<u>Operation</u>	<u>Resulting Queue</u>
(define q (make-queue))	
(insert-queue! q 'a)	a
(insert-queue! q 'b)	a b
(delete-queue! q)	b
(insert-queue! q 'c)	b c
(insert-queue! q 'd)	b c d
(delete-queue! q)	c d

**Figure 3.18:** Queue operations.

*queue(キュー)* はアイテムが一方の端 (*rear*(リア、終端)) に挿入され、他方の端 (*front*(フロント、先端)) から削除される列です。Figure 3.18は初期化時に空のキューにアイテム *a* と *b* が挿入された状態を示しています。次に *a* が削除され、*c* と *d* が挿入され、*b* が削除されます。アイテムは常に挿入順に削除されるためキューは時々 *FIFO*(first in, first out)(先入れ先出し) バッファと呼ばれます。

In terms of data abstraction, we can regard a queue as defined by the following set of operations:

データ抽象化の観点ではキューを以下の操作の集合であると見做すことができます。

- a constructor: (*make-queue*) returns an empty queue (a queue containing no items).

コンストラクタ : (*make-queue*) は空のキュー (アイテムを全く持たないキュー) を返す

- two selectors:

2つのセレクタ :

(*empty-queue?* *<queue>*)

tests if the queue is empty.

キューが空であるかテストする

(*front-queue* *<queue>*)

returns the object at the front of the queue, signaling an error if the queue is empty; it does not modify the queue.

キューの先頭のオブジェクトを返す。もしキューが空ならエラーを発す。  
キューを変更しない。

- two mutators:

2つのミューテータ：

```
(insert-queue! <queue> <item>)
```

inserts the item at the rear of the queue and returns the modified queue as its value.

キューの最後尾にアイテムを挿入し、変更されたキューをその値として返す。

```
(delete-queue! <queue>)
```

removes the item at the front of the queue and returns the modified queue as its value, signaling an error if the queue is empty before the deletion.

キューの先頭のアイテムを削除し、その値として変更されたキューを返す。もしキューが削除前に空であればエラーを発す。

Because a queue is a sequence of items, we could certainly represent it as an ordinary list; the front of the queue would be the `car` of the list, inserting an item in the queue would amount to appending a new element at the end of the list, and deleting an item from the queue would just be taking the `cdr` of the list. However, this representation is inefficient, because in order to insert an item we must scan the list until we reach the end. Since the only method we have for scanning a list is by successive `cdr` operations, this scanning requires  $\Theta(n)$  steps for a list of  $n$  items. A simple modification to the list representation overcomes this disadvantage by allowing the queue operations to be implemented so that they require  $\Theta(1)$  steps; that is, so that the number of steps needed is independent of the length of the queue.

キューはアイテムの列であるため確かに順序有りリストであると表現できます。キューの先頭はリストの `car` であり、キューにアイテムを挿入するのは新しい要素をリストの最後に追加することで、キューからのアイテムの削除はただリストの `cdr` を得ることと言えるでしょう。しかしこの表現は非効率です。なぜならアイテムを挿入するためにはリストを終端まで走査しなければなりま

せん。リストの走査のための手段は `cdr` 命令を連続して用いるしかなく、この走査は  $n$  アイテムのリストに対し  $\Theta(n)$  ステップを必要とします。リスト表現に対する簡単な変更がこの欠点を克服し  $\Theta(1)$  ステップを必要とするキュー命令の実装を可能にします。これはつまり必要なステップ数がキューの長さから独立するということです。

The difficulty with the list representation arises from the need to scan to find the end of the list. The reason we need to scan is that, although the standard way of representing a list as a chain of pairs readily provides us with a pointer to the beginning of the list, it gives us no easily accessible pointer to the end. The modification that avoids the drawback is to represent the queue as a list, together with an additional pointer that indicates the final pair in the list. That way, when we go to insert an item, we can consult the rear pointer and so avoid scanning the list.

リスト表現による困難はリストの終端を見つけるための走査が必要である点から生じています。走査が必要な理由はリストをペアの鎖として表現する標準的な方法が、事前にリストの先頭へのポインタを提供するのに対し、終端を指す簡単にアクセス可能なポインタを提供しないためです。欠点を避けるための変更としてキューをリストとしながらリストの最終ペアを示す追加のポインタをも用いて表現します。この方法ではアイテムを挿入する場合に終端ポインタを調べることでリストの走査を避ることができます。

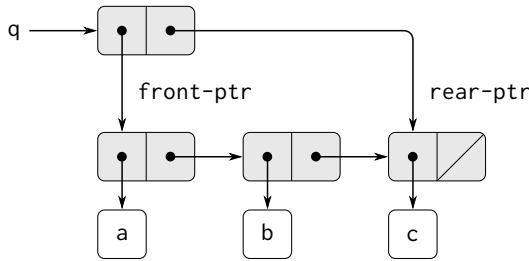
A queue is represented, then, as a pair of pointers, `front-ptr` and `rear-ptr`, which indicate, respectively, the first and last pairs in an ordinary list. Since we would like the queue to be an identifiable object, we can use `cons` to combine the two pointers. Thus, the queue itself will be the `cons` of the two pointers. [Figure 3.19](#) illustrates this representation.

するとキューはポインタのペア、`front-ptr` と `rear-ptr` として表現されます。それぞれが通常のリストの先頭と最後のペアを指します。キューを識別可能なオブジェクトにするために 2 つのポインタを接続するのに `cons` を用います。従ってキューそれ自身が 2 つのポインタの `cons` になります。[Figure 3.19](#) はこの表現を図示します。

To define the queue operations we use the following procedures, which enable us to select and to modify the front and rear pointers of a queue:

キューの命令を定義するために以下の手続を用います。これはキューの先端と終端のポインタの選択、変更を可能にします。

```
(define (front-ptr queue) (car queue))
(define (rear-ptr queue) (cdr queue))
```



**Figure 3.19:** Implementation of a queue as a list with front and rear pointers.

```
(define (set-front-ptr! queue item) (set-car! queue item))
(define (set-rear-ptr! queue item) (set-cdr! queue item))
```

Now we can implement the actual queue operations. We will consider a queue to be empty if its front pointer is the empty list:

これで実際のキューの命令を実装できます。もし先端のポインタが空リストならばキューは空であると考えることにします。

```
(define (empty-queue? queue) (null? (front-ptr queue)))
```

The `make-queue` constructor returns, as an initially empty queue, a pair whose `car` and `cdr` are both the empty list:

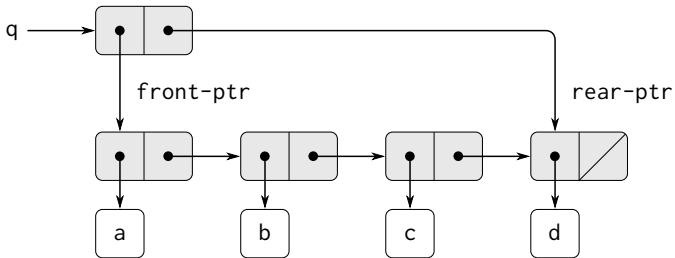
`make-queue` コンストラクタは初期値として空キューを意味する `car` と `cdr` の両方が空リストのペアを返します。

```
(define (make-queue) (cons '() '()))
```

To select the item at the front of the queue, we return the `car` of the pair indicated by the front pointer:

キューの頭のアイテムを選択するために先端ポインタが指すべきアの `car` を返します。

```
(define (front-queue queue)
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (front-ptr queue))))
```



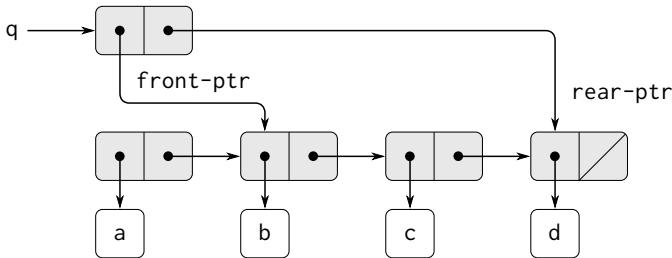
**Figure 3.20:** Result of using `(insert-queue! q 'd)` on the queue of Figure 3.19.

To insert an item in a queue, we follow the method whose result is indicated in Figure 3.20. We first create a new pair whose `car` is the item to be inserted and whose `cdr` is the empty list. If the queue was initially empty, we set the front and rear pointers of the queue to this new pair. Otherwise, we modify the final pair in the queue to point to the new pair, and also set the rear pointer to the new pair.

キューにアイテムを挿入するために、Figure 3.20が示す結果を成す手法に従います。最初に `car` が挿入するアイテムであり `cdr` が空リストである新しいペアを作成します。もしキューが空であるならキューの先端と終端のポインタにこの新しいペアを設定します。そうでなければキューの最終ペアを新しいペアを指すように変更し、また終端ポインタを新しいペアを指すようにします。

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
            (set-cdr! (rear-ptr queue) new-pair)
            (set-rear-ptr! queue new-pair)
            queue))))
```

To delete the item at the front of the queue, we merely modify the front pointer



**Figure 3.21:** Result of using `(delete-queue! q)` on the queue of Figure 3.20.

so that it now points at the second item in the queue, which can be found by following the `cdr` pointer of the first item (see Figure 3.21):<sup>42</sup>

キューの頭のアイテムを削除するために、ただ単に先端ポインタを変更しキューの二つ目のアイテムを指すようにします。これは最初のアイテムの `cdr` ポインタに従うだけで見つけられます。(Figure 3.21参照)<sup>43</sup>

```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else (set-front-ptr! queue (cdr (front-ptr queue)))
              queue))))
```

**Exercise 3.21:** Ben Bitdiddle decides to test the queue implementation described above. He types in the procedures to the Lisp interpreter and proceeds to try them out:

---

<sup>42</sup>If the first item is the final item in the queue, the front pointer will be the empty list after the deletion, which will mark the queue as empty; we needn't worry about updating the rear pointer, which will still point to the deleted item, because `empty-queue?` looks only at the front pointer.

<sup>43</sup>もし最初のアイテムがキューの最終アイテムでもある場合、先端ポインタは削除後に空リストになるでしょう。これはキューを空の状態にします。終端ポインタの更新を心配する必要はありません。これは依然として削除されたアイテムを指しますが、`empty-queue?` は先端ポインタしか見ません。

Ben Bitdiddle は上で説明されたキューの実装をテストすることに決めた。彼は Lisp インタプリタに対し手続を入力し、続いて以下のように試行を行った。

```
(define q1 (make-queue))

(insert-queue! q1 'a)
((a) a)

(insert-queue! q1 'b)
((a b) b)

(delete-queue! q1)
(() b)
```

“It’s all wrong!” he complains. “The interpreter’s response shows that the last item is inserted into the queue twice. And when I delete both items, the second **b** is still there, so the queue isn’t empty, even though it’s supposed to be.” Eva Lu Ator suggests that Ben has misunderstood what is happening. “It’s not that the items are going into the queue twice,” she explains. “It’s just that the standard Lisp printer doesn’t know how to make sense of the queue representation. If you want to see the queue printed correctly, you’ll have to define your own print procedure for queues.” Explain what Eva Lu is talking about. In particular, show why Ben’s examples produce the printed results that they do. Define a procedure `print-queue` that takes a queue as input and prints the sequence of items in the queue.

“間違っている！”と彼は文句を言った。“インタプリタの応答は最後のアイテムがキューに二回挿入されていることを示している。そして僕が両方のアイテムを消しても二つ目の **b** がまだそこにある。だからキューは空になるべきなのにそうならない”。Eva Lu Ator は Ben が何が起きたのか間違って理解していると示唆した。“アイテムはキューに二回入ってはいないわ”と彼女は説明した。“Lisp 標準の応答がキュー表現の意味をどのように理解するのか知ら

ないだけ。もしあなたがキューが正しく表示されるのを見たいなら自分でキューを表示する手続を定義する必要があるわ”。Eva Lu が話していることを説明せよ。具体的にはなぜ Ben の例がそのような表示の結果になるのか示せ。キューを入力に取りキュー内のアイテムの列を表示する手続 `print-queue` を定義せよ。

**Exercise 3.22:** Instead of representing a queue as a pair of pointers, we can build a queue as a procedure with local state. The local state will consist of pointers to the beginning and the end of an ordinary list. Thus, the `make-queue` procedure will have the form キューをポインタのペアとして表現する代わりに、キューを局所状態を持つ手続として構築することができる。局所状態は通常のリストの先端と終端へのポインタから成る。従って `make-queue` 手続は以下の形式となる。

```
(define (make-queue)
  (let ((front-ptr ... )
        (rear-ptr ... ))
    (definitions of internal procedures)
    (define (dispatch m) ...)
    dispatch))
```

Complete the definition of `make-queue` and provide implementations of the queue operations using this representation.

`make-queue` の定義を完成させ、この表現を用いたキューの命令を実装せよ。

**Exercise 3.23:** A *deque* (“double-ended queue”) is a sequence in which items can be inserted and deleted at either the front or the rear. Operations on deques are the constructor `make-deque`, the predicate `empty-deque?`, selectors `front-deque` and `rear-deque`, mutators `front-insert-deque!`, `rear-insert-deque!`, `front-delete-deque!`, and `rear-delete-deque!`. Show how to represent deques using pairs, and give implementations of the operations.<sup>44</sup> All operations should be accomplished in  $\Theta(1)$  steps.

---

<sup>44</sup>Be careful not to make the interpreter try to print a structure that contains cycles. (See Exercise 3.13.)

`deque`(“double-ended queue”、両頭キュー) はアイテムの挿入と消去が先端と終端の両方に対して行える列である。`deque` 上の命令はコンストラクタ `make-deque`、述語 `empty-deque?`、セレクタ `front-deque` と `rear-deque`、ミュータータ `front-insert-deque!`, `rear-insert-deque!`, `front-delete-deque!`, `rear-delete-deque!` である。ペアを用いてどのように `deque` を表現するか示せ。また命令の実装を提供せよ。<sup>45</sup> 全ての命令は  $\Theta(1)$  ステップで達成すること。

### 3.3.3 Representing Tables

When we studied various ways of representing sets in [Chapter 2](#), we mentioned in [Section 2.3.3](#) the task of maintaining a table of records indexed by identifying keys. In the implementation of data-directed programming in [Section 2.4.3](#), we made extensive use of two-dimensional tables, in which information is stored and retrieved using two keys. Here we see how to build tables as mutable list structures.

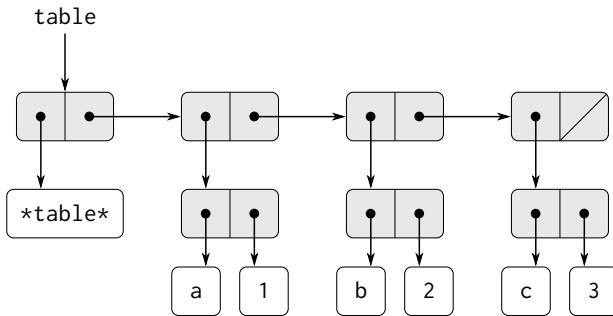
[Chapter 2](#)で種々の集合の表現について学んだ時、[Section 2.3.3](#)にてキーで同定する索引を持つレコードの表を保存する作業について述べました。[Section 2.4.3](#)でのデータ適従プログラミングの実装において二次元テーブルの広範な使用を行い、情報は 2 つのキーを用いて格納と取り出しされました。ここではどのように表をミュータブルなリスト構造として構築するかについて学びます。

We first consider a one-dimensional table, in which each value is stored under a single key. We implement the table as a list of records, each of which is implemented as a pair consisting of a key and the associated value. The records are glued together to form a list by pairs whose `cars` point to successive records. These gluing pairs are called the *backbone* of the table. In order to have a place that we can change when we add a new record to the table, we build the table as a *headed list*. A headed list has a special backbone pair at the beginning, which holds a dummy “record”—in this case the arbitrarily chosen symbol `*table*`. [Figure 3.22](#) shows the box-and-pointer diagram for the table

最初は一次元の表について考えます。各値が单一のキーの下に格納されます。テーブルをレコードのリストとして実装し、各レコードはキーと関連する値から成るペアとして実装します。レコードは `car` が次のレコードを指す

---

<sup>45</sup> インタプリタに循環を含む構造を表示させないように注意せよ。(Exercise 3.13 参照)。



**Figure 3.22:** A table represented as a headed list.

ペアによりリストを形成する様に連結されます。これらの連結されたペアは表の *backbone*(バックボーン、基幹)と呼ばれます。テーブルに新しいレコードを追加する時に変更可能な場所を得るために、テーブルを *headed list*(頭出しリスト)として構築します。頭出しリストは特別なバックボーンペアを最初に持ちます。これはダミーの“レコード”—今回の場合、自由裁量で選択したシンボル `*table*`—を持っていきます。Figure 3.22は以下のテーブルの箱とポインタの図を示しています。

```
a: 1
b: 2
c: 3
```

To extract information from a table we use the `lookup` procedure, which takes a key as argument and returns the associated value (or false if there is no value stored under that key). `Lookup` is defined in terms of the `assoc` operation, which expects a key and a list of records as arguments. Note that `assoc` never sees the dummy record. `Assoc` returns the record that has the given key as its `car`.<sup>46</sup> `Lookup` then checks to see that the resulting record returned by `assoc` is not false, and returns the value (the `cdr`) of the record.

テーブルから情報を抽出するには鍵を引数として取り相対する値(またはそのキーの下に値が格納されていない場合には `false`)を返す `lookup` 手続を用

<sup>46</sup>Because `assoc` uses `equal?`, it can recognize keys that are symbols, numbers, or list structure.

います。`lookup` はキーとレコードのリストを引数として期待する `assoc` 命令を用いて定義します。`assoc` がダミーレコードを絶対に参照しないことに注意して下さい。`assoc` は与えられたキーを `car` として持つレコードを返します。<sup>47</sup> すると `lookup` は `assoc` が返した結果のレコードが `false` でないかチェックし、そのレコードの値 (`cdr`) を返します。

```
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        false)))
(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))
```

To insert a value in a table under a specified key, we first use `assoc` to see if there is already a record in the table with this key. If not, we form a new record by `consing` the key with the value, and insert this at the head of the table's list of records, after the dummy record. If there already is a record with this key, we set the `cdr` of this record to the designated new value. The header of the table provides us with a fixed location to modify in order to insert the new record.<sup>48</sup>

値をテーブルに指定したキーの下に挿入するために、最初に `assoc` を用いて既にテーブルの中にこのキーを持つレコードが存在しないか確認します。もし無ければ鍵と値を `cons` することで新しいレコードを作成しこれをテーブルのレコードリストの先頭のダミーレコードの後ろに挿入します。もし既にこのキーのレコードが存在する場合にはそのレコードの `cdr` に新しい値を設定します。テーブルのヘッダは新しいレコードを挿入するために変更する固定位置を与えます。<sup>49</sup>

---

<sup>47</sup>`assoc` が `equal?` を用いるため、シンボル、数値、リスト構造であるキーを認識可能です。

<sup>48</sup>Thus, the first backbone pair is the object that represents the table “itself”; that is, a pointer to the table is a pointer to this pair. This same backbone pair always starts the table. If we did not arrange things in this way, `insert!` would have to return a new value for the start of the table when it added a new record.

<sup>49</sup>従って最初のバックボーンペアはテーブル“それ自身”を表現するオブジェクトです。テーブルを指すポインタはこのペアを指すポインタです。この同じバックボーンペアが

```
(define (insert! key value table)
  (let ((record (assoc key (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons key value)
                        (cdr table)))))

  'ok)
```

To construct a new table, we simply create a list containing the symbol `*table*`:

新しいテーブルを構築するためには単純にシンボル `*table*` を持つリストを作成します。

```
(define (make-table)
  (list '*table*))
```

## Two-dimensional tables

In a two-dimensional table, each value is indexed by two keys. We can construct such a table as a one-dimensional table in which each key identifies a subtable.

Figure 3.23 shows the box-and-pointer diagram for the table

二次元テーブルでは各値は 2 つのキーにより索引付けられます。そのようなテーブルを各キーが部分テーブルを特定する 1 次元テーブルとして構築することができます。Figure 3.23 は以下のテーブルを箱とポインタの図で示しています。

math:	+: 43	letters:	a: 97
	-: 45		b: 98
	*: 42		

which has two subtables. (The subtables don't need a special header symbol, since the key that identifies the subtable serves this purpose.)

これは 2 つ部分テーブルを持ちます。(部分テーブルは特別なヘッダシンボルを必要としません。部分テーブルを特定するキーがこの目的を果たすためです)。

---

常にテーブルを始めます。もしこのようにしなければ `insert!` は新しいレコードを追加した時にテーブルの新しい開始地点を返さなければならなくなるでしょう。

When we look up an item, we use the first key to identify the correct subtable. Then we use the second key to identify the record within the subtable.

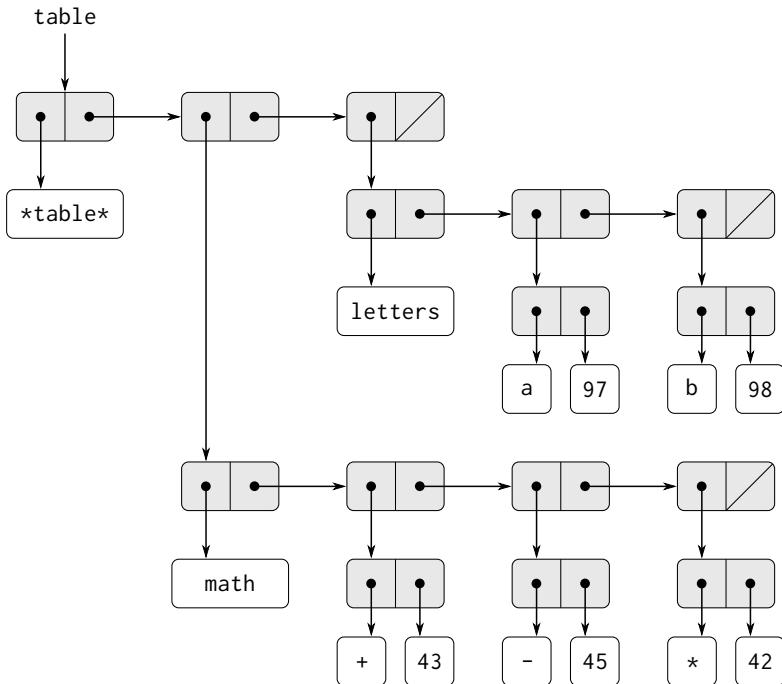
アイテムを探す時、最初のキーを用いて正しい部分テーブルを特定します。次に二つ目のキーを用いて部分テーブル中のレコードを特定します。

```
(define (lookup key-1 key-2 table)
  (let ((subtable
         (assoc key-1 (cdr table))))
    (if subtable
        (let ((record
               (assoc key-2 (cdr subtable))))
          (if record
              (cdr record)
              false))
        false)))
```

To insert a new item under a pair of keys, we use `assoc` to see if there is a subtable stored under the first key. If not, we build a new subtable containing the single record (`key-2, value`) and insert it into the table under the first key. If a subtable already exists for the first key, we insert the new record into this subtable, using the insertion method for one-dimensional tables described above:

キーのペアの下に新しいアイテムを挿入するには `assoc` を用いて最初のキーの下に部分テーブルが存在するかどうかを確認します。もし無ければ単一のレコード (`key-2, value`) を含む新しい部分テーブルを構築し、それを最初のキーの下にテーブルに挿入します。もし最初のキーに対する部分テーブルが既に存在する場合新しいレコードをこの部分テーブルに先に説明した一次元テーブルに対する挿入方法を用いて挿入します。

```
(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (set-cdr! record value)
              (set-cdr! subtable
                        (cons (cons key-2 value)
                              (cdr subtable))))))
        (cons (cons key-1 table)
              (cdr table))))
```



**Figure 3.23:** A two-dimensional table.

```
(set-cdr! table
  (cons (list key-1
    (cons key-2 value))
  (cdr table)))))

'ok)
```

### Creating local tables

The `lookup` and `insert!` operations defined above take the table as an argument. This enables us to use programs that access more than one table. An-

other way to deal with multiple tables is to have separate `lookup` and `insert!` procedures for each table. We can do this by representing a table procedurally, as an object that maintains an internal table as part of its local state. When sent an appropriate message, this “table object” supplies the procedure with which to operate on the internal table. Here is a generator for two-dimensional tables represented in this fashion:

上で定義された `lookup` と `insert!` 命令はテーブルを引数として取ります。これが複数のテーブルにアクセスするプログラムを許可します。複数のテーブルを扱う他の方法には各テーブルに対し分離された `lookup` と `insert!` 手続を持つ方法があります。これはテーブルを手続的に、その局所状態の一部に内部テーブルを持つオブジェクトとして表現することにより可能となります。適切なメッセージを送った時に、この“テーブルオブジェクト”は内部テーブルを操作する手続を提供します。以下にこの様式で表現された二次元テーブルのためのジェネレータ(生成器)を示します。

```
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key-1 key-2)
      (let ((subtable
             (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record
                  (assoc key-2 (cdr subtable))))
              (if record (cdr record) false))
            false)))
    (define (insert! key-1 key-2 value)
      (let ((subtable
             (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record
                  (assoc key-2 (cdr subtable))))
              (if record
                  (set-cdr! record value)
                  (set-cdr! subtable
                            (cons (cons key-2 value)
                                  (cdr subtable))))))
            (set-cdr! local-table
```

```

        (cons (list key-1 (cons key-2 value))
              (cdr local-table)))))

'ok)
(define dispatch m)
(cond ((eq? m 'lookup-proc) lookup)
      ((eq? m 'insert-proc!) insert!)
      (else (error "Unknown operation: TABLE" m))))
dispatch)

```

Using `make-table`, we could implement the `get` and `put` operations used in Section 2.4.3 for data-directed programming, as follows:

`make-table` を用いることでSection 2.4.3で用いたデータ適従プログラミングのための `get` と `put` を以下のように実装することができます。

```

(define operation-table (make-table))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))

```

`Get` takes as arguments two keys, and `put` takes as arguments two keys and a value. Both operations access the same local table, which is encapsulated within the object created by the call to `make-table`.

`get` は引数として 2 つのキーを取り、`put` は引数として 2 つのキーと値を取ります。両方の命令共に同じ局所テーブルをアクセスします。局所テーブルは `make-table` の呼出により作成されたオブジェクトの中にカプセル化されます。

**Exercise 3.24:** In the table implementations above, the keys are tested for equality using `equal?` (called by `assoc`). This is not always the appropriate test. For instance, we might have a table with numeric keys in which we don't need an exact match to the number we're looking up, but only a number within some tolerance of it. Design a table constructor `make-table` that takes as an argument a `same-key?` procedure that will be used to test “equality” of keys. `Make-table` should return a `dispatch` procedure that can be used to access appropriate `lookup` and `insert!` procedures for a local table.

上記のテーブル実装において、キーは `equal?` を用いて等価試験を行う。`(assoc` により呼び出される)。これは常に適切な試験ではない。例として数値キーを用いるテーブルを用いる場合に、検索時に

厳密に等しい必要が無く、ある許容範囲で数値を探したいかもしれない。キーの“等価性”を試験するのに用いられる `same-key?` 手続を引数として取るテーブルコンストラクタ `make-table` を定義せよ。`make-table` は内部テーブルに対して適切な手続 `lookup` と `insert!` にアクセスするのに使用可能な `dispatch` 手続を返さねばならない。

**Exercise 3.25:** Generalizing one- and two-dimensional tables, show how to implement a table in which values are stored under an arbitrary number of keys and different values may be stored under different numbers of keys. The `lookup` and `insert!` procedures should take as input a list of keys used to access the table.

1次元と二次元のテーブルを一般化せよ。任意の数のキーの下で値を格納し、異なる値を異なる数のキーの下格納できるテーブルをどのように実装するか示せ。`lookup` と `insert!` 手続は入力としてキーのリストを取りテーブルにアクセスする。

**Exercise 3.26:** To search a table as implemented above, one needs to scan through the list of records. This is basically the unordered list representation of [Section 2.3.3](#). For large tables, it may be more efficient to structure the table in a different manner. Describe a table implementation where the (key, value) records are organized using a binary tree, assuming that keys can be ordered in some way (e.g., numerically or alphabetically). (Compare [Exercise 2.66](#) of [Chapter 2](#).)

上で実装されたテーブルを検索するにはレコードのリストを走査しなければならない。これは基本的に [Section 2.3.3](#) の順序無しリスト表現である。大きなテーブルに対しては異なる様式でテーブルを構造化するほうが効率が良い。(キー, 値) のレコードが二分木を用いて体系化されるテーブルの実装を説明せよ。キーは何らかの方法にて順序付可能であると想定する。[\(Chapter 2 の Exercise 2.66 と比較せよ\)](#)。

**Exercise 3.27:** *Memoization* (also called *tabulation*) is a technique that enables a procedure to record, in a local table, values that have previously been computed. This technique can make a vast difference in the performance of a program. A memoized procedure maintains a table in which values of previous calls are stored

using as keys the arguments that produced the values. When the memoized procedure is asked to compute a value, it first checks the table to see if the value is already there and, if so, just returns that value. Otherwise, it computes the new value in the ordinary way and stores this in the table. As an example of memoization, recall from Section 1.2.2 the exponential process for computing Fibonacci numbers:

*memoization*(メモ化)(*tabulation*(表形式化)とも呼ばれる)とは手続の局所テーブルに事前に計算した値を記録することを可能するテクニックである。このテクニックはプログラムのパフォーマンスに大幅な違いを与えることができる。メモ化された手続は以前の呼出の値がその値を生成した引数をキーとして格納するテーブルを持つ。メモ化された手続が値を計算するよう命じられた時、最初に値が既にテーブルにないかチェックを行い、もし存在すれば単にその値を返す。そうでなければ新しい値を通常の方法で計算しテーブルに保存する。メモ化の例としてSection 1.2.2からフィボナッチ数を演算するための指数関数処理を思い出せ。

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2)))))))
```

The memoized version of the same procedure is  
同じ手続のメモ化版は以下である。

```
(define memo-fib
  (memoize
    (lambda (n)
      (cond ((= n 0) 0)
            ((= n 1) 1)
            (else (+ (memo-fib (- n 1))
                      (memo-fib (- n 2))))))))
```

where the memoizer is defined as

この時、*memoize* は以下の様に定義される。

```
(define (memoize f)
```

```

(let ((table (make-table)))
  (lambda (x)
    (let ((previously-computed-result
           (lookup x table)))
      (or previously-computed-result
          (let ((result (f x)))
            (insert! x result table)
            result))))))

```

Draw an environment diagram to analyze the computation of (`memo-fib 3`). Explain why `memo-fib` computes the  $n^{\text{th}}$  Fibonacci number in a number of steps proportional to  $n$ . Would the scheme still work if we had simply defined `memo-fib` to be (`memoize fib`)?  
`(memo-fib 3)` の演算を分析するための環境図を描け。なぜ `memo-fib` が  $n$  番目のフィボナッチ数を  $n$  に比例するステップ数で演算するのか説明せよ。単に `memo-fib` を (`memoize fib`) と定義した場合にも Scheme は正しく処理できるだろうか？

### 3.3.4 A Simulator for Digital Circuits

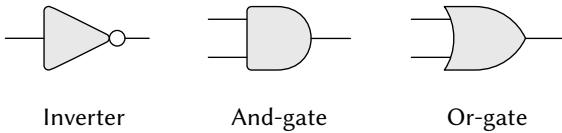
Designing complex digital systems, such as computers, is an important engineering activity. Digital systems are constructed by interconnecting simple elements. Although the behavior of these individual elements is simple, networks of them can have very complex behavior. Computer simulation of proposed circuit designs is an important tool used by digital systems engineers. In this section we design a system for performing digital logic simulations. This system typifies a kind of program called an *event-driven simulation*, in which actions (“events”) trigger further events that happen at a later time, which in turn trigger more events, and so on.

コンピュータのような複雑なデジタルシステムの設計は重要な工学の活動領域です。デジタルシステムは簡単な要素を相互接続することで構築されます。これらの個々の要素の振舞は単純ですが、それらのネットワークはとても複雑な振舞をします。提案された回路設計のコンピュータシミュレーションはデジタルシステムエンジニアにより使用される重要なツールです。この節ではデジタル論理シミュレーションを実行するためのシステムを設計します。このシステムは *event-driven simulation* (イベント駆動シミュレーション) と呼ばれる種

類の典型であり、その行動（“イベント”）は後に起こるさらなるイベントを引き起こし、順により多くのイベントを引き起します。

Our computational model of a circuit will be composed of objects that correspond to the elementary components from which the circuit is constructed. There are *wires*, which carry *digital signals*. A digital signal may at any moment have only one of two possible values, 0 and 1. There are also various types of digital *function boxes*, which connect wires carrying input signals to other output wires. Such boxes produce output signals computed from their input signals. The output signal is delayed by a time that depends on the type of the function box. For example, an *inverter* is a primitive function box that inverts its input. If the input signal to an inverter changes to 0, then one inverter-delay later the inverter will change its output signal to 1. If the input signal to an inverter changes to 1, then one inverter-delay later the inverter will change its output signal to 0. We draw an inverter symbolically as in Figure 3.24. An *and-gate*, also shown in Figure 3.24, is a primitive function box with two inputs and one output. It drives its output signal to a value that is the *logical and* of the inputs. That is, if both of its input signals become 1, then one and-gate-delay time later the and-gate will force its output signal to be 1; otherwise the output will be 0. An *or-gate* is a similar two-input primitive function box that drives its output signal to a value that is the *logical or* of the inputs. That is, the output will become 1 if at least one of the input signals is 1; otherwise the output will become 0.

私達の回路の計算モデルは回路を構築する基本となるコンポーネントに対応するオブジェクトにより成ります。*digital signals*(デジタル信号)を運ぶ*wires*(回路)が存在します。デジタル信号は任意の瞬間に可能な2つの値、0と1の内1つを取ります。また多様なタイプのデジタル *function boxes*(関数箱)が存在し、入力信号を運ぶ回路と別の出力回路を接続します。そのような箱は入力信号から計算された信号を出力します。出力信号は関数箱のタイプにより時間的に遅れを生じさせます。例えば *inverter*(逆変換器)は入力を反転するプリミティブな関数箱です。もし逆変換器への入力信号が0に変化したなら、ある逆変換器による遅延の後、逆変換器はその出力信号を1に変更します。もし逆変換器への入力信号が1に変化したならば、ある逆変換器による遅延の後、逆変換器は出力信号を0にします。逆変換器を記号としてFigure 3.24に示すように描きます。Figure 3.24に示される *and-gate*(AND ゲート)も2つの入力と1つの出力を持つプリミティブな関数箱です。入力の *logical and*(論理積)の値にその出力の値を駆動します。言い替えれば、もし入力信号の両方が1になれば



**Figure 3.24:** Primitive functions in the digital logic simulator.

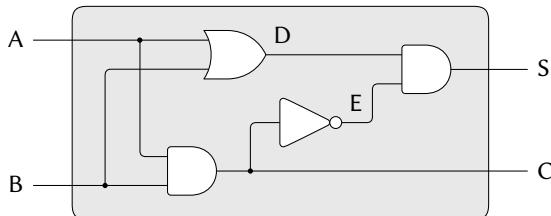
ある AND ゲートによる遅延の後に AND ゲートはその出力信号を 1 にします。そうでなければ出力は 0 です。or-gate(OR ゲート)も同様の 2 つの入力を持つプリミティブな関数箱でありその出力信号は入力に対する *logical or*(論理和)の値になります。言い替えれば出力はもし少くとも 1 つの入力信号が 1 であれば 1 になり、そうでなければ出力は 0 になります。

We can connect primitive functions together to construct more complex functions. To accomplish this we wire the outputs of some function boxes to the inputs of other function boxes. For example, the *half-adder* circuit shown in Figure 3.25 consists of an or-gate, two and-gates, and an inverter. It takes two input signals, A and B, and has two output signals, S and C. S will become 1 whenever precisely one of A and B is 1, and C will become 1 whenever A and B are both 1. We can see from the figure that, because of the delays involved, the outputs may be generated at different times. Many of the difficulties in the design of digital circuits arise from this fact.

プリミティブな関数を一緒に接続してより複雑な関数を構築できます。これを達成するためにある関数箱の出力から他の関数箱の入力へと回路を引きます。例えばFigure 3.25に示す *half-adder*(半加算器)はOR ゲート、2 つの AND ゲート、逆変換器から成り立ちます。これは 2 つの入力信号、A と B を取り 2 つの出力信号 S と C があります。S は正確に A と B の内 1 つが 1 であるならば 1 になり、C は A と B の両方が 1 の場合に 1 になります。遅延が生じるため出力が異なる時刻に生成されることが図から見てわかります。デジタル回路設計の困難の多くはこの事実から生じます。

We will now build a program for modeling the digital logic circuits we wish to study. The program will construct computational objects modeling the wires, which will “hold” the signals. Function boxes will be modeled by procedures that enforce the correct relationships among the signals.

今から私達が学習を望むデジタル論理回路をモデル化するためのプログラムを構築します。プログラムは回路をモデル化する計算モデルを構築します。



**Figure 3.25:** A half-adder circuit.

これは信号を“保持”します。関数箱は信号間の正しい関係を強制する手続によりモデル化されます。

One basic element of our simulation will be a procedure `make-wire`, which constructs wires. For example, we can construct six wires as follows:

私達のシミュレーションの基本的要素の1つは手続 `make-wire` であり回路を構築します。例として6つの回路を以下のように構築できます。

```
(define a (make-wire))
(define b (make-wire))
(define c (make-wire))
(define d (make-wire))
(define e (make-wire))
(define s (make-wire))
```

We attach a function box to a set of wires by calling a procedure that constructs that kind of box. The arguments to the constructor procedure are the wires to be attached to the box. For example, given that we can construct and-gates, or-gates, and inverters, we can wire together the half-adder shown in Figure 3.25:

ある関数箱を回路の集合に対してその種類の箱を構築する手続を呼ぶことにより取り付けることができます。コンストラクタ手続への引数は箱に取り付けられる回路です。例えばANDゲート、ORゲート、逆変換器を構築できる場合、Figure 3.25に示す半加算器を配線することができます。

```
(or-gate a b d)
ok
(and-gate a b c)
```

```
ok
(inverter c e)
ok
(and-gate d e s)
ok
```

Better yet, we can explicitly name this operation by defining a procedure `half-adder` that constructs this circuit, given the four external wires to be attached to the half-adder:

もっと良いことには、半加算器に取り付けられる 4 つの外部回路を与えた時、この回路を構築する手続 `half-adder` を定義することでこの操作に明示的に名前を付けることができます。

```
(define (half-adder a b s c)
  (let ((d (make-wire)) (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'ok))
```

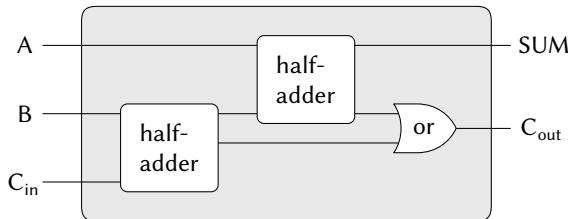
The advantage of making this definition is that we can use `half-adder` itself as a building block in creating more complex circuits. Figure 3.26, for example, shows a *full-adder* composed of two half-adders and an or-gate.<sup>50</sup> We can construct a full-adder as follows:

この定義を作ることの利点は `half-adder` それ自身をより複雑な回路を作成する時に建築用ブロックとして使用することができます。例えばFigure 3.26は 2 つの半加算器と 1 つの OR ゲートより組み立てられる*full-adder*(全加算器) を示しています。<sup>51</sup> 全加算器を以下のように構築できます。

---

<sup>50</sup>A full-adder is a basic circuit element used in adding two binary numbers. Here A and B are the bits at corresponding positions in the two numbers to be added, and  $C_{in}$  is the carry bit from the addition one place to the right. The circuit generates SUM, which is the sum bit in the corresponding position, and  $C_{out}$ , which is the carry bit to be propagated to the left.

<sup>51</sup>全加算器は 2 つの二進数の加算に用いられる基本的な回路要素です。ここで A と B は加算される 2 つの数の対応する位置のビットで、 $C_{in}$  は 1 つ右の加算からのキャリービット(桁上げビット)です。この回路は対応する位置の合計のビットである SUM と左に伝播されるキャリービットである  $C_{out}$  を算出します。



**Figure 3.26:** A full-adder circuit.

```
(define (full-adder a b c-in sum c-out)
  (let ((s (make-wire)) (c1 (make-wire)) (c2 (make-wire)))
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
    (or-gate c1 c2 c-out)
    'ok))
```

Having defined `full-adder` as a procedure, we can now use it as a building block for creating still more complex circuits. (For example, see [Exercise 3.30](#).)

手続として定義された `full-adder` を持つことでさらにより複雑な回路を作成するための建築ブロックとして利用することが可能です。(例えば [Exercise 3.30](#) を参照)。

In essence, our simulator provides us with the tools to construct a language of circuits. If we adopt the general perspective on languages with which we approached the study of Lisp in [Section 1.1](#), we can say that the primitive function boxes form the primitive elements of the language, that wiring boxes together provides a means of combination, and that specifying wiring patterns as procedures serves as a means of abstraction.

実質的に、私達のシミュレーターは回路の言語を構築するツールを提供します。もし [Section 1.1](#) における Lisp の学習への取り組みに用いた言語上の一般的な観点を受け入れれば、プリミティブな関数箱はプリミティブな言語の要素を形成し、箱の間に回路を引くことは組み合わせの手段を提供し、手続として回線を引くパターンを指定することは抽象化の手段としての役割を果たすということが言えます。

## Primitive function boxes

The primitive function boxes implement the “forces” by which a change in the signal on one wire influences the signals on other wires. To build function boxes, we use the following operations on wires:

プリミティブな関数箱はある回路上の信号の変化が他の配線上の信号に影響を与える“力”を実装します。関数箱を構築するため以下の回路上の命令を用います。

- `(get-signal <wire>)`

returns the current value of the signal on the wire.

回線上的信号の現在地を返す

- `(set-signal! <wire> <new value>)`

changes the value of the signal on the wire to the new value.

回路上の信号の値を新しい値に変更する

- `(add-action! <wire> <procedure of no arguments>)`

asserts that the designated procedure should be run whenever the signal on the wire changes value. Such procedures are the vehicles by which changes in the signal value on the wire are communicated to other wires.

指定された手続が回路上の信号が値を変化した場合常に実行される様に宣言する。そのような手続は、回路上の信号の値の変化が他の回路と通信を行うための伝達手段である。

In addition, we will make use of a procedure `after-delay` that takes a time delay and a procedure to be run and executes the given procedure after the given delay.

さらに手続 `after-delay` を使用し遅延時間と実行される手続を取得し、与えられた手続を遅延時間後に実行します。

Using these procedures, we can define the primitive digital logic functions. To connect an input to an output through an inverter, we use `add-action!` to associate with the input wire a procedure that will be run whenever the signal on the input wire changes value. The procedure computes the `logical-not` of the input signal, and then, after one `inverter-delay`, sets the output signal to be this new value:

これらの手続を用いてプリミティブなデジタル論理関数を定義できます。入力を逆変換器を通して出力に接続するために `add-action!` を用いて入力回路

と入力回路上の信号が値を変化する度に実行される手続を関連付けます。その手続は入力信号の `logical-not`(論理否定) を計算し、そして `inverter-delay` 後に出力信号にこの新しい値を設定します。

```
(define (inverter input output)
  (define (invert-input)
    (let ((new-value (logical-not (get-signal input))))
      (after-delay inverter-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! input invert-input) 'ok)
(define (logical-not s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else (error "Invalid signal" s))))
```

An and-gate is a little more complex. The action procedure must be run if either of the inputs to the gate changes. It computes the `logical-and` (using a procedure analogous to `logical-not`) of the values of the signals on the input wires and sets up a change to the new value to occur on the output wire after one `and-gate-delay`.

AND ゲートはより少しだけ複雑です。アクション手続はゲートへの入力のどちらかが変化した場合に実行されねばなりません。それが入力回路上の信号の値の `logical-and`(論理積) を (`logical-not` と類似の手続を用いて) 求め、出力回路上に起こる新しい値への変更を `and-gate-delay` 後に設定します。

```
(define (and-gate a1 a2 output)
  (define (and-action-procedure)
    (let ((new-value
           (logical-and (get-signal a1) (get-signal a2))))
      (after-delay
       and-gate-delay
       (lambda () (set-signal! output new-value)))))
  (add-action! a1 and-action-procedure)
  (add-action! a2 and-action-procedure)
  'ok))
```

**Exercise 3.28:** Define an or-gate as a primitive function box. Your `or-gate` constructor should be similar to `and-gate`.

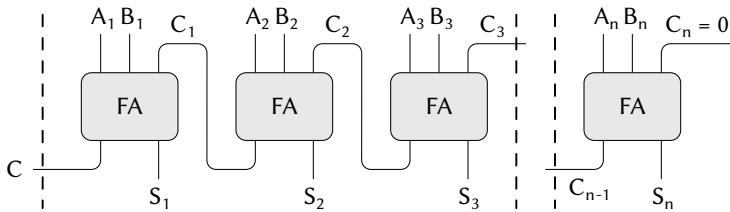
OR ゲートをプリミティブな関数箱として定義せよ。あなたの `or-gate` コンストラクタは `and-gate` と同様でなければならない。

**Exercise 3.29:** Another way to construct an or-gate is as a compound digital logic device, built from and-gates and inverters. Define a procedure `or-gate` that accomplishes this. What is the delay time of the or-gate in terms of `and-gate-delay` and `inverter-delay`?

OR ゲートを構築する別の方法は複合デジタル論理デバイスとして AND ゲートと逆変換器から構築するものである。これを達成する手続 `or-gate` を定義せよ。`and-gate-delay` と `inverter-delay` を用いた遅延時間はどのようになるか？

**Exercise 3.30:** Figure 3.27 shows a *ripple-carry adder* formed by stringing together  $n$  full-adders. This is the simplest form of parallel adder for adding two  $n$ -bit binary numbers. The inputs  $A_1, A_2, A_3, \dots, A_n$  and  $B_1, B_2, B_3, \dots, B_n$  are the two binary numbers to be added (each  $A_k$  and  $B_k$  is a 0 or a 1). The circuit generates  $S_1, S_2, S_3, \dots, S_n$ , the  $n$  bits of the sum, and  $C$ , the carry from the addition. Write a procedure `ripple-carry-adder` that generates this circuit. The procedure should take as arguments three lists of  $n$  wires each---the  $A_k$ , the  $B_k$ , and the  $S_k$ ---and also another wire  $C$ . The major drawback of the ripple-carry adder is the need to wait for the carry signals to propagate. What is the delay needed to obtain the complete output from an  $n$ -bit ripple-carry adder, expressed in terms of the delays for and-gates, or-gates, and inverters?

Figure 3.27 は  $n$  個の全加算器を繋げた *ripple-carry adder*(桁上げ伝播加算器) を示している。これは 2 つの  $n$  ビット二進数を足すための最も簡単な形式の並列加算器である。入力  $A_1, A_2, A_3, \dots, A_n$  と  $B_1, B_2, B_3, \dots, B_n$  は足すべき 2 つの二進数(各  $A_k$  と  $B_k$  は 0 か 1) である。回路は  $S_1, S_2, S_3, \dots, S_n$  の  $n$  ビットの和と、和算の桁上がりである  $C$  を生成する。この回路を生成する手続 `ripple-carry-adder` を書け。この手続は引数としてそれぞれ  $n$  個の配線を持つ 3 つのリスト— $A_k, B_k, S_k$ —と別の配線  $C$  を取る。桁上げ



**Figure 3.27:** A ripple-carry adder for  $n$ -bit numbers.

伝播加算器の主な欠点はキャリー信号の伝播を待つ必要があることである。 $n$  ビットの桁上げ伝播加算器における完全な出力を得るのに必要な遅延時間はいくらか? AND ゲート、OR ゲート、逆変換器の遅延時間から表現せよ。

## Representing wires

A wire in our simulation will be a computational object with two local state variables: a `signal-value` (initially taken to be 0) and a collection of `action-procedures` to be run when the signal changes value. We implement the wire, using message-passing style, as a collection of local procedures together with a `dispatch` procedure that selects the appropriate local operation, just as we did with the simple bank-account object in Section 3.1.1:

私達のシミュレーションにおけるワイヤ(wire, 配線、回路)は2つのローカルな状態変数を持つ計算オブジェクトになります。その2つは `signal-value`(信号値)(初期値は0)と信号が値を変えた時に実行される `action-procedures`(行動手続)の集合です。メッセージパッシングスタイルを用いてワイヤを局所手続の集合として適切な局所命令を選択する手続 `dispatch`と共に実装します。Section 3.1.1での簡単な銀行口座オブジェクトと同様に行います。

```
(define (make-wire)
  (let ((signal-value 0) (action-procedures '()))
    (define (set-my-signal! new-value)
      (if (not (= signal-value new-value))
          (begin (set! signal-value new-value)
                 (call-each action-procedures)))
          signal-value)))
```

```

'done))
(define (accept-action-procedure! proc)
  (set! action-procedures
    (cons proc action-procedures))
  (proc))
(define (dispatch m)
  (cond ((eq? m 'get-signal) signal-value)
        ((eq? m 'set-signal!) set-my-signal!)
        ((eq? m 'add-action!) accept-action-procedure!)
        (else (error "Unknown operation: WIRE" m))))
  dispatch))

```

The local procedure `set-my-signal!` tests whether the new signal value changes the signal on the wire. If so, it runs each of the action procedures, using the following procedure `call-each`, which calls each of the items in a list of no-argument procedures:

局所手続 `set-my-signal!` は新しい信号値が配線上の信号を変えるかチェックします。もしうそであれば全ての行動手続を以下の手続 `call-each` を用いて実行します。`call-each` は引数無し手続のリスト内の全てのアイテムを呼び出します。

```

(define (call-each procedures)
  (if (null? procedures)
      'done
      (begin ((car procedures))
             (call-each (cdr procedures))))))

```

The local procedure `accept-action-procedure!` adds the given procedure to the list of procedures to be run, and then runs the new procedure once. (See [Exercise 3.31](#).)

局所手続 `accept-action-procedure!` は与えられた手続を実行対象手続リストに追加します。次に新しい手続を一度実行します。([Exercise 3.31](#)参照)

With the local `dispatch` procedure set up as specified, we can provide the following procedures to access the local operations on wires:<sup>52</sup>

---

<sup>52</sup> These procedures are simply syntactic sugar that allow us to use ordinary procedural syntax to access the local procedures of objects. It is striking that we can interchange the role of “procedures” and “data” in such a simple way. For example, if

ローカルの `dispatch` 手続が指定通りに設定されていることから、以下の手続を与えて配線上の局所命令にアクセスすることができます。<sup>53</sup>

```
(define (get-signal wire) (wire 'get-signal))
(define (set-signal! wire new-value)
  ((wire 'set-signal!) new-value))
(define (add-action! wire action-procedure)
  ((wire 'add-action!) action-procedure))
```

Wires, which have time-varying signals and may be incrementally attached to devices, are typical of mutable objects. We have modeled them as procedures with local state variables that are modified by assignment. When a new wire is created, a new set of state variables is allocated (by the `let` expression in `make-wire`) and a new `dispatch` procedure is constructed and returned, capturing the environment with the new state variables.

時間的に変化する信号を持ち付加的に装置に取り付けられる配線はミュータブルなオブジェクトの特性を良く示しています。私達はそれを代入により変化するローカル状態変数を持つ手続としてモデル化しました。新しい配線が作成された時、新しい状態変数の信号は (`make-wire` 中の `let` 式により) 確保され、新しい `dispatch` 手續が構築され返され、新しい状態変数を持つ環境が確保されます。

The wires are shared among the various devices that have been connected to them. Thus, a change made by an interaction with one device will affect all the other devices attached to the wire. The wire communicates the change to its

---

we write `(wire 'get-signal)` we think of `wire` as a procedure that is called with the message `get-signal` as input. Alternatively, writing `(get-signal wire)` encourages us to think of `wire` as a data object that is the input to a procedure `get-signal`. The truth of the matter is that, in a language in which we can deal with procedures as objects, there is no fundamental difference between “procedures” and “data,” and we can choose our syntactic sugar to allow us to program in whatever style we choose.

53 これらの手続は単純にオブジェクトの局所手続にアクセスするために通常の手続的な文法を使用することを許可する構文糖に過ぎません。“手続”と“データ”的役割をそのような簡単な方法で交換できることは印象的です。例えばもし `(wire 'get-signal)` と書いた場合、私達は `wire` をメッセージ `get-signal` を入力として呼び出される手続だと考えるでしょう。その代わりに `(get-signal wire)` と書くことは私達に `wire` を手続 `get-signal` に対する入力としてのデータオブジェクトだと考えることを促します。この問題の真実は私達が手続をオブジェクトとして扱う言語には“手続”と“データ”的間に基本的な違いが存在せず、私達はどんなスタイルを選択してもプログラミングを可能にする構文糖を選択することができるということです。

neighbors by calling the action procedures provided to it when the connections were established.

配線は様々なデバイスの間で共有され、それらに対して接続されます。従ってあるデバイスとの応答により起こった変化はその配線に取り付けられた全ての他のデバイスに影響を与えます。配線は接続が開設された時に提供された行動手続を呼ぶことによりその近傍に対し変化を通知します。

## The agenda

The only thing needed to complete the simulator is `after-delay`. The idea here is that we maintain a data structure, called an *agenda*, that contains a schedule of things to do. The following operations are defined for agendas:

シミュレータを完成させるために必要な物は `after-delay` のみです。ここでアイデアは *agenda*(予定表) と呼ばれるデータ構造を保持し、それを行うべき予定を保存します。以下の命令は予定表のために定義されます。

- `(make-agenda)` returns a new empty agenda.  
`(make-agenda)` は新しい空の予定表を返す。
- `(empty-agenda? <agenda>)` is true if the specified agenda is empty.  
`(empty-agenda? <agenda>)` は指定した予定表が空であるなら真である。
- `(first-agenda-item <agenda>)` returns the first item on the agenda.  
`(first-agenda-item <agenda>)` は予定表の最初のアイテムを返す。
- `(remove-first-agenda-item! <agenda>)` modifies the agenda by removing the first item.  
`(remove-first-agenda-item! <agenda>)` は予定表から最初のアイテムを削除する。
- `(add-to-agenda! <time> <action> <agenda>)` modifies the agenda by adding the given action procedure to be run at the specified time.  
`(add-to-agenda! <time> <action> <agenda>)` は指定された時間後に実行される行動手続を追加する。
- `(current-time <agenda>)` returns the current simulation time.  
`(current-time <agenda>)` は現在のシミュレーション時間を返す。

The particular agenda that we use is denoted by `the-agenda`. The procedure `after-delay` adds new elements to `the-agenda`:

使用する予定表は `the-agenda` により指定されます。手続 `after-delay` は新しい要素を `the-agenda` に追加します。

```
(define (after-delay delay action)
  (add-to-agenda! (+ delay (current-time the-agenda))
                  action
                  the-agenda))
```

The simulation is driven by the procedure `propagate`, which operates on `the-agenda`, executing each procedure on the agenda in sequence. In general, as the simulation runs, new items will be added to the agenda, and `propagate` will continue the simulation as long as there are items on the agenda:

シミュレーションは手続 `propagate`(伝播) により駆動され、`the-agenda` 上で操作を行い、予定表上の各手続を順に実行します。一般的にシミュレータが実行されるにつれ、新しいアイテムが予定表に追加され、`propagate` はシミュレーションを予定表にアイテムが存在する間は続けます。

```
(define (propagate)
  (if (empty-agenda? the-agenda)
      'done
      (let ((first-item (first-agenda-item the-agenda)))
        (first-item)
        (remove-first-agenda-item! the-agenda)
        (propagate))))
```

## A sample simulation

The following procedure, which places a “probe” on a wire, shows the simulator in action. The probe tells the wire that, whenever its signal changes value, it should print the new signal value, together with the current time and a name that identifies the wire:

回路上に“probe”(プローブ、探針)を置く以下の手続は実行中のシミュレータを表示します。プローブは配線に対し信号値が変わる度に新しい信号値を現在に時刻と配線を識別する名前と一緒に表示せよと命じます。

```
(define (probe name wire)
  (add-action! wire
               (lambda ()
```

```

(newline)
(display name)
(display " ")
(display (current-time the-agenda))
(display " New-value = ")
(display (get-signal wire))))))

```

We begin by initializing the agenda and specifying delays for the primitive function boxes:

予定表の初期化とプリミティブな関数箱に対し遅延時間を指定することから始めます。

```

(define the-agenda (make-agenda))
(define inverter-delay 2)
(define and-gate-delay 3)
(define or-gate-delay 5)

```

Now we define four wires, placing probes on two of them:

ここで4つの配線を定義し、その内2つにプローブを仕込みます。

```

(define input-1 (make-wire))
(define input-2 (make-wire))
(define sum (make-wire))
(define carry (make-wire))

(probe 'sum sum)
sum 0 New-value = 0
(probe 'carry carry)
carry 0 New-value = 0

```

Next we connect the wires in a half-adder circuit (as in Figure 3.25), set the signal on `input-1` to 1, and run the simulation:

次に配線を (Figure 3.25の様に) 半加算器回路に接続し、`input-1` 上の信号を 1 に設定し、シミュレーションを実行します。

```

(half-adder input-1 input-2 sum carry)
ok

(set-signal! input-1 1)
done

```

```
(propagate)
sum 8 New-value = 1
done
```

The `sum` signal changes to 1 at time 8. We are now eight time units from the beginning of the simulation. At this point, we can set the signal on `input-2` to 1 and allow the values to propagate:

`sum` の進行は時刻 8において 1に変化しました。シミュレーションの開始から 8 単位時間が経過しました。この時点で `input-2` 上の信号を 1に設定し値の伝播を許可します。

```
(set-signal! input-2 1)
done
```

```
(propagate)
carry 11 New-value = 1
sum 16 New-value = 0
done
```

The `carry` changes to 1 at time 11 and the `sum` changes to 0 at time 16.  
`carry` は時刻 11 にて 1に変化し、`sum` は時刻 16 において 0に変化しました。

**Exercise 3.31:** The internal procedure `accept-action-procedure!` defined in `make-wire` specifies that when a new action procedure is added to a wire, the procedure is immediately run. Explain why this initialization is necessary. In particular, trace through the half-adder example in the paragraphs above and say how the system's response would differ if we had defined `accept-action-procedure!` as

`make-wire` 内で定義された内部手続 `accept-action-procedure!` は新しい行動手続が配線に追加された時に、その手続が即座に実行された。この初期化がなぜ必要であるのか説明せよ。具体的には、上の段落の半加算器の例をトレースし、システムの応答が、`accept-action-procedure!` が以下のように定義されていた場合にどのように異なるかについて述べよ。

```
(define (accept-action-procedure! proc)
  (set! action-procedures
    (cons proc action-procedures)))
```

## Implementing the agenda

Finally, we give details of the agenda data structure, which holds the procedures that are scheduled for future execution.

最後に将来に実行される予定の手続を保存する予定表データ構造の詳細について説明します。

The agenda is made up of *time segments*. Each time segment is a pair consisting of a number (the time) and a queue (see [Exercise 3.32](#)) that holds the procedures that are scheduled to be run during that time segment.

予定表は*time segments*(タイムセグメント、時間区分)により構成されています。各タイムセグメントは数値(時刻)と、そのタイムセグメントの間に実行されるよう予定された手続を持つキュー([Exercise 3.32](#)参照)から成るペアです。

```
(define (make-time-segment time queue)
  (cons time queue))
(define (segment-time s) (car s))
(define (segment-queue s) (cdr s))
```

We will operate on the time-segment queues using the queue operations described in [Section 3.3.2](#).

タイムセグメントのキューは[Section 3.3.2](#)で説明したキューの命令を用いて操作します。

The agenda itself is a one-dimensional table of time segments. It differs from the tables described in [Section 3.3.3](#) in that the segments will be sorted in order of increasing time. In addition, we store the *current time* (i.e., the time of the last action that was processed) at the head of the agenda. A newly constructed agenda has no time segments and has a current time of 0:<sup>54</sup>

予定表自身は1次元のタイムセグメントの表です。[Section 3.3.3](#)で説明された表との違いはセグメントが時間の増す順にソートされることです。加えて*current time*(現在時刻)(言い換えると最後に処理された行動の時刻)を予定表の頭に保存します。新しく構築された予定表はタイムセグメントを持っておらず現在時刻として0を持ちます。<sup>55</sup>

---

<sup>54</sup>The agenda is a headed list, like the tables in [Section 3.3.3](#), but since the list is headed by the time, we do not need an additional dummy header (such as the `*table*` symbol used with tables).

<sup>55</sup>予定表は[Section 3.3.3](#)のような頭出しリストですが、このリストは時刻による頭出しせど追加のダミーヘッダ(テーブルにて用いられた`*table*`シンボルのような物)を

```
(define (make-agenda) (list 0))
(define (current-time agenda) (car agenda))
(define (set-current-time! agenda time)
  (set-car! agenda time))
(define (segments agenda) (cdr agenda))
(define (set-segments! agenda segments)
  (set-cdr! agenda segments))
(define (first-segment agenda)
  (car (segments agenda)))
(define (rest-segments agenda)
  (cdr (segments agenda)))
```

An agenda is empty if it has no time segments:

予定表はタイムセグメントを持っていなければ空です。

```
(define (empty-agenda? agenda)
  (null? (segments agenda)))
```

To add an action to an agenda, we first check if the agenda is empty. If so, we create a time segment for the action and install this in the agenda. Otherwise, we scan the agenda, examining the time of each segment. If we find a segment for our appointed time, we add the action to the associated queue. If we reach a time later than the one to which we are appointed, we insert a new time segment into the agenda just before it. If we reach the end of the agenda, we must create a new time segment at the end.

予定表に行動(アクション)を追加するために、最初に予定表が空であるか確認します。もしそうならばアクションのためのタイムセグメントを作成し、それを予定表にインストールします。そうでなければ予定表を走査し、各セグメントの時刻を調べます。もし指定時刻が存在するならば対応するキューにアクションを追加します。もし指定時刻よりも後の時間に辿り着いたならば、新しいタイムセグメントを予定表のその時間の前に挿入します。もし予定表の最後まで辿り着いたならば新しいタイムセグメントを最後に作らねばなりません。

```
(define (add-to-agenda! time action agenda)
  (define (belongs-before? segments)
    (or (null? segments)
```

---

必要としません。

```

(< time (segment-time (car segments)))))

(define (make-new-time-segment time action)
  (let ((q (make-queue)))
    (insert-queue! q action)
    (make-time-segment time q)))

(define (add-to-segments! segments)
  (if (= (segment-time (car segments)) time)
      (insert-queue! (segment-queue (car segments))
                     action)
      (let ((rest (cdr segments)))
        (if (belongs-before? rest)
            (set-cdr!
             segments
             (cons (make-new-time-segment time action)
                   (cdr segments)))
            (add-to-segments! rest)))))

(let ((segments (segments agenda)))
  (if (belongs-before? segments)
      (set-segments!
       agenda
       (cons (make-new-time-segment time action)
             segments))
      (add-to-segments! segments))))

```

The procedure that removes the first item from the agenda deletes the item at the front of the queue in the first time segment. If this deletion makes the time segment empty, we remove it from the list of segments.<sup>56</sup>

予定表から最初のアイテムを削除する手続は最初のタイムセグメント中のキューの先頭のアイテムを削除します。もしこの削除がタイムセグメントを空にするのであれば、セグメントのリストからそれを削除します。<sup>57</sup>

---

<sup>56</sup>Observe that the `if` expression in this procedure has no `(alternative)` expression. Such a “one-armed `if statement`” is used to decide whether to do something, rather than to select between two expressions. An `if` expression returns an unspecified value if the predicate is false and there is no `(alternative)`.

<sup>57</sup>この手続の中の `if` 式が `(alternative)` 式を持っていないことに注意して下さい。このような“片腕の `if 文`”は 2 つの式の間から選択するのではなく何かをするかどうかを決定するのに使用されます。`if` 式は述語が偽になった場合に未定義の値を返し、

```
(define (remove-first-agenda-item! agenda)
  (let ((q (segment-queue (first-segment agenda))))
    (delete-queue! q)
    (if (empty-queue? q)
        (set-segments! agenda (rest-segments agenda)))))
```

The first agenda item is found at the head of the queue in the first time segment.  
Whenever we extract an item, we also update the current time:<sup>58</sup>

最初の予定表のアイテムは最初のタイムセグメントのキューの頭に見つかります。アイテムを抽出する度に現在時刻の更新も行います。<sup>59</sup>

```
(define (first-agenda-item agenda)
  (if (empty-agenda? agenda)
      (error "Agenda is empty: FIRST-AGENDA-ITEM")
      (let ((first-seg (first-segment agenda)))
        (set-current-time! agenda
                            (segment-time first-seg))
        (front-queue (segment-queue first-seg)))))
```

**Exercise 3.32:** The procedures to be run during each time segment of the agenda are kept in a queue. Thus, the procedures for each segment are called in the order in which they were added to the agenda (first in, first out). Explain why this order must be used. In particular, trace the behavior of an and-gate whose inputs change from 0, 1 to 1, 0 in the same segment and say how the behavior would differ if we stored a segment's procedures in an ordinary list, adding and removing procedures only at the front (last in, first out).

予定表の各タイムセグメントの間に実行される手続はキューに保存される。従って各セグメントの手続は予定表に追加された順に

---

*{alternative}* は有りません。

<sup>58</sup>In this way, the current time will always be the time of the action most recently processed. Storing this time at the head of the agenda ensures that it will still be available even if the associated time segment has been deleted.

<sup>59</sup>このようにして、現在時刻は常に最も最近に処理されたアクションの時刻になります。この時刻を予定表の頭に格納することで例え関連するタイムセグメントが削除されても依然として有効であることを確約します。

呼び出される (FIFO)。なぜこの順が使用されるべきか説明せよ。具体的には入力が 0,1 から 1,0 に同じセグメントにて変化した時の AND ゲートの振舞をトレースし、もしセグメントの手続を通常の順に格納し、手続の追加と削除を先頭でのみ行った場合 (LIFO) に振舞がどのように異なるかについて述べよ。

### 3.3.5 Propagation of Constraints

Computer programs are traditionally organized as one-directional computations, which perform operations on prespecified arguments to produce desired outputs. On the other hand, we often model systems in terms of relations among quantities. For example, a mathematical model of a mechanical structure might include the information that the deflection  $d$  of a metal rod is related to the force  $F$  on the rod, the length  $L$  of the rod, the cross-sectional area  $A$ , and the elastic modulus  $E$  via the equation

コンピュータプログラムは伝統的に一方向の演算として体系化されます。これは事前に指定した引数上で命令を実行し、望んだ出力を生成します。一方で私達は時折、量の間の関係を用いてシステムをモデル化します。例えば機械構造の数理的モデルは金属棒の偏差  $d$  が棒上の力  $F$ 、棒の長さ  $L$ 、断面積  $A$ 、弾性率  $E$  に方程式を通して関連するという情報を含むでしょう。

$$dAE = FL.$$

Such an equation is not one-directional. Given any four of the quantities, we can use it to compute the fifth. Yet translating the equation into a traditional computer language would force us to choose one of the quantities to be computed in terms of the other four. Thus, a procedure for computing the area  $A$  could not be used to compute the deflection  $d$ , even though the computations of  $A$  and  $d$  arise from the same equation.<sup>60</sup>

---

<sup>60</sup> Constraint propagation first appeared in the incredibly forward-looking SKETCHPAD system of Ivan Sutherland (1963). A beautiful constraint-propagation system based on the Smalltalk language was developed by Alan Borning (1977) at Xerox Palo Alto Research Center. Sussman, Stallman, and Steele applied constraint propagation to electrical circuit analysis (Sussman and Stallman 1975; Sussman and Steele 1980). TK!Solver (Konopasek and Jayaraman 1984) is an extensive modeling environment based on constraints.

そのような方程式は一方向ではありません。任意の4つの量を与えられることで、5つ目を計算することができます。けれども方程式を伝統的なコンピュータ言語へと翻訳することは1つの量を選択し他の4つを用いて求めることを私達は強制されます。従って断面積  $A$  を求める手続は偏差  $d$  を求めるることは、例え  $A$  と  $d$  の演算が同じ方程式から起こっても使用できません。<sup>61</sup>

In this section, we sketch the design of a language that enables us to work in terms of relations themselves. The primitive elements of the language are *primitive constraints*, which state that certain relations hold between quantities. For example, (`adder a b c`) specifies that the quantities  $a$ ,  $b$ , and  $c$  must be related by the equation  $a + b = c$ , (`multiplier x y z`) expresses the constraint  $xy = z$ , and (`constant 3.14 x`) says that the value of  $x$  must be 3.14.

この節では関係性自身を用いて働くことが可能な言語の設計を描きます。言語のプリミティブな要素は *primitive constraints*(プリミティブ制約)であり、幾らかの関係性が数量の間に保存されることを示します。例えは  $c$  は方程式  $a + b = c$  から参照されねばならず、(`multiplier x y z`) は制約  $xy = z$  を表し、(`constant 3.14 x`) は  $x$  の値が 3.14 に違いないと述べています。

Our language provides a means of combining primitive constraints in order to express more complex relations. We combine constraints by constructing *constraint networks*, in which constraints are joined by *connectors*. A connector is an object that “holds” a value that may participate in one or more constraints. For example, we know that the relationship between Fahrenheit and Celsius temperatures is

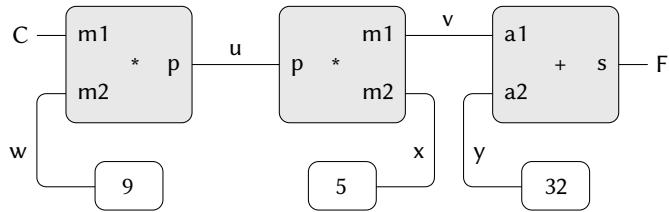
私達の言語はプリミティブ制約をより複雑な関係を表明するために接続する手段を提供します。制約を *constraint networks*(制約ネットワーク)を構築することで接続し、その中で制約は *connectors*(コネクタ)を用いて結合されます。コネクタは値を持つオブジェクトであり、1つ以上の制約に加わります。例えは華氏と摂氏の気温の間の関係が以下であることを知っています。

$$9C = 5(F - 32).$$

Such a constraint can be thought of as a network consisting of primitive adder, multiplier, and constant constraints (Figure 3.28). In the figure, we see on the

---

<sup>61</sup>制約伝播は最初に信じられない程先進的であった Ivan Sutherland (1963)による SKETCHPAD システムに現れました。Smalltalk をベースにした美しい制約伝播システムは Alan Borning (1977)により Xerox バロアルト研究センタにて開発されました。Sussman, Stallman, Steele の3人は制約伝播を電子回路分析に応用しました (Sussman and Stallman 1975; Sussman and Steele 1980)。TK!Solver(Konopasek and Jayaraman 1984) は制約をベースにした大規模モデリング環境です。



**Figure 3.28:** The relation  $9C = 5(F - 32)$  expressed as a constraint network.

left a multiplier box with three terminals, labeled  $m1$ ,  $m2$ , and  $p$ . These connect the multiplier to the rest of the network as follows: The  $m1$  terminal is linked to a connector  $C$ , which will hold the Celsius temperature. The  $m2$  terminal is linked to a connector  $w$ , which is also linked to a constant box that holds 9. The  $p$  terminal, which the multiplier box constrains to be the product of  $m1$  and  $m2$ , is linked to the  $p$  terminal of another multiplier box, whose  $m2$  is connected to a constant 5 and whose  $m1$  is connected to one of the terms in a sum.

そのような制約はプリミティブな加算器、乗算器、不变制約 (Figure 3.28) より成り立つネットワークとして考えることができます。図の中で左手に  $m1$ ,  $m2$ ,  $p$  の 3 つの端子を持つ乗算の箱を見ることができます。これらは乗算器を以下のネットワークの残りに接続します。 $m1$  端子は摂氏の気温を保持するコネクタ  $C$  にリンクされます。 $m2$  端子も 9 を持つ整数箱にリンクされます。乗算器の箱が  $m1$  と  $m2$  の積に制約を行う  $p$  端子は別の乗算器の箱の  $p$  端子に接続され、その箱の  $m2$  は整数 5 に、 $m1$  は合計の 1 つの端子に接続されます。

Computation by such a network proceeds as follows: When a connector is given a value (by the user or by a constraint box to which it is linked), it awakens all of its associated constraints (except for the constraint that just awakened it) to inform them that it has a value. Each awakened constraint box then polls its connectors to see if there is enough information to determine a value for a connector. If so, the box sets that connector, which then awakens all of its associated constraints, and so on. For instance, in conversion between Celsius and Fahrenheit,  $w$ ,  $x$ , and  $y$  are immediately set by the constant boxes to 9, 5, and 32, respectively. The connectors awaken the multipliers and the adder,

which determine that there is not enough information to proceed. If the user (or some other part of the network) sets  $C$  to a value (say 25), the leftmost multiplier will be awakened, and it will set  $u$  to  $25 \cdot 9 = 225$ . Then  $u$  awakens the second multiplier, which sets  $v$  to 45, and  $v$  awakens the adder, which sets  $f$  to 77.

このようなネットワークによる計算は以下の様に進行されます。コネクタに値が(ユーザ、またはリンクされた制約箱により)与えられた時、その関連する制約全てを(それを起こした制約を除いて)起こし、それらに値を得たことを伝えます。起きた制約箱は全て次にコネクタに対しこネクタの値を決定するのに十分な情報が存在するかを調査(poll)します。もしそうであれば、制約箱はコネクタに値を設定し、コネクタはすると関係する制約を全て起こします。これが繰り返されます。例として摂氏と華氏の間の換算では  $w, x, y$  は整数箱 9, 5, 32 それぞれにより直ぐに設定されます。コネクタは乗算器と加算器を起動し、それらは続行に必要な情報が十分ではないことを判断します。もしユーザ(またはネットワークの何らかの他の部分が)  $C$  に値(例えば 25)を設定すると最も左の乗算器が起動され、 $u$  に  $25 \cdot 9 = 225$  を設定します。すると  $u$  が 2 つ目の乗算器を起動し、それが  $v$  に 45 を設定します。そして  $v$  が加算器を起動し、加算器は  $f$  を 77 に設定します。

## Using the constraint system

To use the constraint system to carry out the temperature computation outlined above, we first create two connectors,  $C$  and  $F$ , by calling the constructor `make-connector`, and link  $C$  and  $F$  in an appropriate network:

制約システムを用いて上で説明された気温の計算を実行するには最初に 2 つのコネクタ、 $C$  と  $F$  をコンストラクタ `make-connector` を呼ぶことで作成し、 $C$  と  $F$  をあるべきネットワークにリンクします。

```
(define C (make-connector))
(define F (make-connector))
(celsius-fahrenheit-converter C F)
ok
```

The procedure that creates the network is defined as follows:

ネットワークを作成する手続は以下のように定義されます。

```
(define (celsius-fahrenheit-converter c f)
  (let ((u (make-connector)))
```

```

(v (make-connector))
(w (make-connector))
(x (make-connector))
(y (make-connector))
(multiplier c w u)
(multiplier v x u)
(adder v y f)
(constant 9 w)
(constant 5 x)
(constant 32 y)
'ok))

```

This procedure creates the internal connectors `u`, `v`, `w`, `x`, and `y`, and links them as shown in Figure 3.28 using the primitive constraint constructors `adder`, `multiplier`, and `constant`. Just as with the digital-circuit simulator of Section 3.3.4, expressing these combinations of primitive elements in terms of procedures automatically provides our language with a means of abstraction for compound objects.

この手続は内部コネクタ `u`, `v`, `w`, `x`, `y` を作成し、それらを Figure 3.28 に示されるようにプリミティブな制約コンストラクタ `adder`, `multiplier`, `constant` を用いてリンクします。

To watch the network in action, we can place probes on the connectors `C` and `F`, using a `probe` procedure similar to the one we used to monitor wires in Section 3.3.4. Placing a probe on a connector will cause a message to be printed whenever the connector is given a value:

実行中のネットワークを見るために、コネクタ `C` と `F` にプローブ(探針)をSection 3.3.4で配線の監視に用いた物と同様な `probe` 手続を用いて設置します。プローブのコネクタ上への設置はコネクタに値が与えられる度にメッセージが表示されるようにします。

```

(probe "Celsius temp" C)
(probe "Fahrenheit temp" F)

```

Next we set the value of `C` to 25. (The third argument to `set-value!` tells `C` that this directive comes from the `user`.)

次に `C` の値を 25 に設定します。`(set-value! ...)` への 3 つ目の引数は `C` にこの指示が `user` による物であることを伝えています)。

```
(set-value! C 25 'user)
Probe: Celsius temp = 25
Probe: Fahrenheit temp = 77
done
```

The probe on C awakens and reports the value. C also propagates its value through the network as described above. This sets F to 77, which is reported by the probe on F.

C 上のプローブが起動され値を報告します。C はまたその値を上で説明されたネットワークを通して伝播させます。これが F に 77 を設定し、F 上のプローブにより報告されます。

Now we can try to set F to a new value, say 212:

ここで F に新しい値、例えば 212 を設定してみましょう。

```
(set-value! F 212 'user)
Error! Contradiction (77 212)
```

The connector complains that it has sensed a contradiction: Its value is 77, and someone is trying to set it to 212. If we really want to reuse the network with new values, we can tell C to forget its old value:

コネクタが矛盾に気付いたと訴えています。その値は 77 の時、誰かが 212 を設定しようとしているのです。もし本当にネットワークを新しい値にて再利用したいのであれば C に古い値を忘れるように指示できます。

```
(forget-value! C 'user)
Probe: Celsius temp = ?
Probe: Fahrenheit temp = ?
done
```

C finds that the user, who set its value originally, is now retracting that value, so C agrees to lose its value, as shown by the probe, and informs the rest of the network of this fact. This information eventually propagates to F, which now finds that it has no reason for continuing to believe that its own value is 77. Thus, F also gives up its value, as shown by the probe.

C は元の値を設定した user が今撤回しているのに気付き、C はその値をなくすことにプローブが示すように同意し、ネットワークの残りにこの結果について伝えます。この情報が結果的に F に伝播し、F は今となってはそれ自身の値が 77 であると信じ続けるための理由が無いことに気付きます。従って F もまたその値を諦めプローブにより表示されます。

Now that  $F$  has no value, we are free to set it to 212:

これで  $F$  は値を持たず、私達は  $F$  に 212 を設定できます。

```
(set-value! F 212 'user)
Probe: Fahrenheit temp = 212
Probe: Celsius temp = 100
done
```

This new value, when propagated through the network, forces  $C$  to have a value of 100, and this is registered by the probe on  $C$ . Notice that the very same network is being used to compute  $C$  given  $F$  and to compute  $F$  given  $C$ . This nondirectionality of computation is the distinguishing feature of constraint-based systems.

この新しい値がネットワーク中に伝播された時、 $C$  に 100 の値を持つことを強制し、 $C$  上のプローブによりこのことが表されます。全く同じネットワークが  $F$  を与えて  $C$  を計算するのと、 $C$  を与えて  $F$  を計算することに用いられていることに注意して下さい。この方向性の無い演算が制約ベースシステムの特徴的な機能です。

## Implementing the constraint system

The constraint system is implemented via procedural objects with local state, in a manner very similar to the digital-circuit simulator of [Section 3.3.4](#). Although the primitive objects of the constraint system are somewhat more complex, the overall system is simpler, since there is no concern about agendas and logic delays.

制約システムは局所状態を持つ手続き型のオブジェクトにより、[Section 3.3.4](#) のデジタル回路シミュレータに良く似た作法で実装されます。制約システムのプリミティブなオブジェクトはいくらかより複雑ではあるものの、システム全体は予定表や論理遅延時間についての考慮が不要な分、よりシンプルです。

The basic operations on connectors are the following:

コネクタ上の基本的な命令は次のとおりです。

- (`(has-value? <connector>)`) tells whether the connector has a value.  
`(has-value? <connector>)` はコネクタが値を持つかどうか判断する
- (`(get-value <connector>)`) returns the connector's current value.  
`(get-value <connector>)` はコネクタの現在地を返す

- (`(set-value! <connector> <new-value> <informant>)`) indicates that the informant is requesting the connector to set its value to the new value.  
`(set-value! <connector> <new-value> <informant>)` は情報がコネクタに対しその値を新しい値に設定するよう要求することを示す
- (`(forget-value! <connector> <retractor>)`) tells the connector that the retractor is requesting it to forget its value.  
`(forget-value! <connector> <retractor>)` はコネクタに対し撤回を望む者が値を忘れるなどを要求していると伝える
- (`(connect <connector> <new-constraint>)`) tells the connector to participate in the new constraint.  
`(connect <connector> <new-constraint>)` はコネクタに対し新しい制約への参加を指示する

The connectors communicate with the constraints by means of the procedures `inform-about-value`, which tells the given constraint that the connector has a value, and `inform-about-no-value`, which tells the constraint that the connector has lost its value.

コネクタは与えられた制約にコネクタが値を持っていると伝える手続 `inform-about-value` と制約にコネクタが値を失ったと伝える手続 `inform-about-no-value` を用いて制約と通信を行います。

`Adder` constructs an adder constraint among summand connectors `a1` and `a2` and a `sum` connector. An adder is implemented as a procedure with local state (the procedure `me` below):

`adder` は加数コネクタ `a1` と `a2` と `sum` コネクタの間に加算器制約を構築するコンストラクタです。加算器は局所状態を持つ手続(下記の手続 `me`)として実装されます。

```
(define (adder a1 a2 sum)
  (define (process-new-value)
    (cond ((and (has-value? a1) (has-value? a2))
           (set-value! sum
                       (+ (get-value a1) (get-value a2))
                       me))
          ((and (has-value? a1) (has-value? sum))
           (set-value! a2
```

```

          (- (get-value sum) (get-value a1))
          me))
((and (has-value? a2) (has-value? sum))
 (set-value! a1
            (- (get-value sum) (get-value a2))
            me))))
(define (process-forget-value)
  (forget-value! sum me)
  (forget-value! a1 me)
  (forget-value! a2 me)
  (process-new-value))
(define (me request)
  (cond ((eq? request 'I-have-a-value)
         (process-new-value))
        ((eq? request 'I-lost-my-value)
         (process-forget-value))
        (else (error "Unknown request: ADDER" request))))
(connect a1 me)
(connect a2 me)
(connect sum me)
me)

```

Adder connects the new adder to the designated connectors and returns it as its value. The procedure `me`, which represents the adder, acts as a dispatch to the local procedures. The following “syntax interfaces” (see [Footnote 27](#) in [Section 3.3.4](#)) are used in conjunction with the dispatch:

`adder` は新しい加算器を指定されたコネクタに接続し自身をその値として返します。手続 `me` は加算器を表現し、ローカル手続を起動する者の役割を果たします。

```

(define (inform-about-value constraint)
  (constraint 'I-have-a-value))
(define (inform-about-no-value constraint)
  (constraint 'I-lost-my-value))

```

The adder’s local procedure `process-new-value` is called when the adder is informed that one of its connectors has a value. The adder first checks to see if both `a1` and `a2` have values. If so, it tells `sum` to set its value to the sum of the

two addends. The `informant` argument to `set-value!` is `me`, which is the adder object itself. If `a1` and `a2` do not both have values, then the adder checks to see if perhaps `a1` and `sum` have values. If so, it sets `a2` to the difference of these two. Finally, if `a2` and `sum` have values, this gives the adder enough information to set `a1`. If the adder is told that one of its connectors has lost a value, it requests that all of its connectors now lose their values. (Only those values that were set by this adder are actually lost.) Then it runs `process-new-value`. The reason for this last step is that one or more connectors may still have a value (that is, a connector may have had a value that was not originally set by the adder), and these values may need to be propagated back through the adder.

加算器のローカル手続 `process-new-value` はその加算器が繋るコネクタの内 1つが値を得た事を報された時に呼び出されます。加算器は最初に `a1` と `a2` の両方が値を持っているか確認します。もしそうならば `sum` に 2つの加数の和をその値として設定するように指示します。`set-value!` の `informant`(情報提供者)引数は加算器オブジェクト自身である `me` です。もし `a1` と `a2` の両方が値を持っていない場合、加算器はひょっとしたら `a1` と `sum` が値を持っていないか確認します。もしそうならば `a2` にその 2つの差を設定します。最後に `a2` と `sum` が値を持っているのならば加算器に `a1` を接待させるために十分な情報を持っていることになります。もし加算器がコネクタの 1つが値を失なったと報された場合、全てのコネクタに対しその値を捨てるよう指示します。(この加算器により設定された値のみが実際には失なわれます)。次に加算器は `process-new-value` を実行します。この理由は 1つ、またはそれ以上のコネクタが依然として値を持っている可能性があり(つまり、コネクタが元々その加算器により設定されたのではない値を持っている)、これらの値は加算器を通して伝播し返す必要があります。

A multiplier is very similar to an adder. It will set its `product` to 0 if either of the factors is 0, even if the other factor is not known.

乗算器は加算器にとても良く似ています。因数のどちらかが 0 なら例え他方の値がわからなくとも `product` を 0 にします。

```
(define (multiplier m1 m2 product)
  (define (process-new-value)
    (cond ((or (and (has-value? m1) (= (get-value m1) 0))
               (and (has-value? m2) (= (get-value m2) 0)))
           (set-value! product 0 me))
          ((and (has-value? m1) (has-value? m2))
           (set-value! product
```

```

(* (get-value m1) (get-value m2))
me))
((and (has-value? product) (has-value? m1))
(set-value! m2
(/ (get-value product)
(get-value m1))
me))
((and (has-value? product) (has-value? m2))
(set-value! m1
(/ (get-value product)
(get-value m2))
me))))
(define (process-forget-value)
(forget-value! product me)
(forget-value! m1 me)
(forget-value! m2 me)
(process-new-value))
(define (me request)
(cond ((eq? request 'I-have-a-value)
(process-new-value))
((eq? request 'I-lost-my-value)
(process-forget-value))
(else (error "Unknown request:
MULTIPLIER" request))))
(connect m1 me)
(connect m2 me)
(connect product me)
me)

```

A **constant** constructor simply sets the value of the designated connector. Any **I-have-a-value** or **I-lost-my-value** message sent to the constant box will produce an error.

**constant** コンストラクタは単純に指定されたコネクタの値を設定します。**I-have-a-value** と **I-lost-my-value** のどちらのメッセージが定数箱に送られてもエラーを発します。

```
(define (constant value connector)
```

```
(define (me request)
  (error "Unknown request: CONSTANT" request))
(connect connector me)
(set-value! connector value me)
me)
```

Finally, a probe prints a message about the setting or unsetting of the designated connector:

最後にプローブは指定されたコネクタの設定、設定解除のメッセージを表示します。

```
(define (probe name connector)
  (define (print-probe value)
    (newline) (display "Probe: ") (display name)
    (display " = ") (display value))
  (define (process-new-value)
    (print-probe (get-value connector)))
  (define (process-forget-value) (print-probe "?"))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
           (process-new-value))
          ((eq? request 'I-lost-my-value)
           (process-forget-value))
          (else (error "Unknown request: PROBE" request))))
  (connect connector me)
  me))
```

## Representing connectors

A connector is represented as a procedural object with local state variables **value**, the current value of the connector; **informant**, the object that set the connector's value; and **constraints**, a list of the constraints in which the connector participates.

コネクタは局所状態変数を持つ手続き型のオブジェクトとして表現され、**value** はコネクタの現在地、**informant** はコネクタの値を設定したオブジェクト、そして **constraints** はコネクタが参加する制約のリストです。

```

(define (make-connector)
  (let ((value false) (informant false) (constraints '()))
    (define (set-my-value newval setter)
      (cond ((not (has-value? me))
              (set! value newval)
              (set! informant setter)
              (for-each-except setter
                                inform-about-value
                                constraints))
            ((not (= value newval))
             (error "Contradiction" (list value newval)))
            (else 'ignored)))
    (define (forget-my-value retractor)
      (if (eq? retractor informant)
          (begin (set! informant false)
                 (for-each-except retractor
                                   inform-about-no-value
                                   constraints))
            'ignored))
    (define (connect new-constraint)
      (if (not (memq new-constraint constraints))
          (set! constraints
                (cons new-constraint constraints)))
          (if (has-value? me)
              (inform-about-value new-constraint)
              'done))
    (define (me request)
      (cond ((eq? request 'has-value?)
              (if informant true false))
            ((eq? request 'value) value)
            ((eq? request 'set-value!) set-my-value)
            ((eq? request 'forget) forget-my-value)
            ((eq? request 'connect) connect)
            (else (error "Unknown operation: CONNECTOR"
                         request))))
  me))

```

The connector's local procedure `set-my-value` is called when there is a request to set the connector's value. If the connector does not currently have a value, it will set its value and remember as `informant` the constraint that requested the value to be set.<sup>62</sup> Then the connector will notify all of its participating constraints except the constraint that requested the value to be set. This is accomplished using the following iterator, which applies a designated procedure to all items in a list except a given one:

コネクタの局所手続 `set-my-value` はコネクタの値を設定する要求が存在した時に呼ばれます。もしコネクタが現在値を持っていない場合、その値を設定し、値の設定を要求した制約を `informant` として記憶します。<sup>63</sup> 次にコネクタは参加している制約全てに対し値の設定を要求した制約を除いて通知します。これは以下の iterator(イテレータ、繰り返す者) を用いて達成されます。イテレータは指定された手続を与えられた 1 つを除いたリスト中の全てのアイテムに対して適用します。

```
(define (for-each-except exception procedure list)
  (define (loop items)
    (cond ((null? items) 'done)
          ((eq? (car items) exception) (loop (cdr items)))
          (else (procedure (car items))
                (loop (cdr items))))))
  (loop list))
```

If a connector is asked to forget its value, it runs the local procedure `forget-my-value`, which first checks to make sure that the request is coming from the same object that set the value originally. If so, the connector informs its associated constraints about the loss of the value.

もしコネクタがその値を忘れるよう指示されたなら、局所手続 `forget-my-value` を実行し、最初に要求が元々値を設定した同じオブジェクトからであるかを確認します。もしそうならばコネクタは関連する制約に値の喪失について伝えます。

The local procedure `connect` adds the designated new constraint to the list of constraints if it is not already in that list. Then, if the connector has a value,

---

<sup>62</sup>The `setter` might not be a constraint. In our temperature example, we used `user` as the `setter`.

<sup>63</sup>`setter` は制約ではないかもしれません。気温の例では `user` を `setter` として使用しました。

it informs the new constraint of this fact.

局所手続 `connect` は指定された新しい制約を制約リストに、既に存在しない場合には追加します。次にもしコネクタが値を持っているのならば、新しい制約にその事実を伝えます。

The connector's procedure `me` serves as a dispatch to the other internal procedures and also represents the connector as an object. The following procedures provide a syntax interface for the dispatch:

コネクタの手続 `me` は他の内部手続を実行する役割を果たし、またコネクタをオブジェクトとして表現します。以下の手続は起動のための文法上のインターフェイスを提供します。

```
(define (has-value? connector) (connector 'has-value?))
(define (get-value connector) (connector 'value))
(define (set-value! connector new-value informant)
  ((connector 'set-value!) new-value informant))
(define (forget-value! connector retractor)
  ((connector 'forget) retractor))
(define (connect connector new-constraint)
  ((connector 'connect) new-constraint))
```

**Exercise 3.33:** Using primitive multiplier, adder, and constant constraints, define a procedure `averager` that takes three connectors `a`, `b`, and `c` as inputs and establishes the constraint that the value of `c` is the average of the values of `a` and `b`.

プリミティブな乗算器、加算器、定数の制約を用いて、3つのコネクタ `a`, `b`, `c` を入力として取り、`c` の値が `a` と `b` の値の平均を見出す手続 `averager` を定義せよ。

**Exercise 3.34:** Louis Reasoner wants to build a squarer, a constraint device with two terminals such that the value of connector `b` on the second terminal will always be the square of the value `a` on the first terminal. He proposes the following simple device made from a multiplier:

Louis Reasoner は 2 つの端子を持ち、2 つ目の端子上のコネクタ `b` が常に 1 つ目の端子上の値 `a` の二乗である制約端末 `squarer` を構築したいと考えた。彼は以下の簡単な乗算から作られた端末を提案した。

```
(define (squarer a b) (multiplier a a b))
```

There is a serious flaw in this idea. Explain.

このアイデアには致命的な問題がある。説明せよ。

**Exercise 3.35:** Ben Bitdiddle tells Louis that one way to avoid the trouble in [Exercise 3.34](#) is to define a squarer as a new primitive constraint. Fill in the missing portions in Ben's outline for a procedure to implement such a constraint:

Ben Bitdiddle は Louis に [Exercise 3.34](#) の問題を避ける 1 つの方法として squarer を新しいプリミティブな制約として定義することを伝えた。Ben の新しい制約の輪郭の欠けている部分を埋めそのような文脈での実装を行え。

```
(define (squarer a b)
  (define (process-new-value)
    (if (has-value? b)
        (if (< (get-value b) 0)
            (error "square less than 0: SQUARER"
                  (get-value b))
            (alternative1))
        (alternative2)))
  (define (process-forget-value) (body1))
  (define (me request) (body2))
  (rest of definition)
  me)
```

**Exercise 3.36:** Suppose we evaluate the following sequence of expressions in the global environment:

以下のグローバル環境内の式の列を評価したとする。

```
(define a (make-connector))
(define b (make-connector))
(set-value! a 10 'user)
```

At some time during evaluation of the `set-value!`, the following expression from the connector's local procedure is evaluated:

`set-value!` の評価の間のある時点で、コネクタのローカル手続から以下の式が評価される。

```
(for-each-except
  setter inform-about-value constraints)
```

Draw an environment diagram showing the environment in which the above expression is evaluated.

上の式が評価される環境を示す環境の図を描け。

**Exercise 3.37:** The `celsius-fahrenheit-converter` procedure is cumbersome when compared with a more expression-oriented style of definition, such as

`celsius-fahrenheit-converter`(摂氏華氏変換器) 手続は以下の  
ような式指向なスタイルと比べた時に煩わしい。

```
(define (celsius-fahrenheit-converter x)
  (+ (* (/ (- x 32)) 5) 9))
(define C (make-connector))
(define F (celsius-fahrenheit-converter C)))
```

Here `c+`, `c*`, etc. are the “constraint” versions of the arithmetic operations. For example, `c+` takes two connectors as arguments and returns a connector that is related to these by an adder constraint:  
ここで `c+, c*` 等は数値演算命令の“制約”版である。例えば `c+` は  
2つのコネクタを引数として取り、これらに関係するコネクタを加  
算器制約にて返す。

```
(define (c+ x y)
  (let ((z (make-connector)))
    (adder x y z)
    z))
```

Define analogous procedures `c-`, `c*`, `c/`, and `cv` (constant value) that enable us to define compound constraints as in the converter example above.

同様の手続 `c-, c*, c/, cv`(定数) を定義し、複合制約を上記の変換  
器の例の様に定義できるようにせよ。<sup>64</sup>

---

<sup>64</sup>The expression-oriented format is convenient because it avoids the need to name

---

the intermediate expressions in a computation. Our original formulation of the constraint language is cumbersome in the same way that many languages are cumbersome when dealing with operations on compound data. For example, if we wanted to compute the product  $(a + b) \cdot (c + d)$ , where the variables represent vectors, we could work in “imperative style,” using procedures that set the values of designated vector arguments but do not themselves return vectors as values:

式指向形式は便利です。それは演算の中間間に名前を付ける必要性を回避できるためです。私達の元々の制約言語の形式は多くの言語が複合データを取り扱う場合と同様に面倒でした。例として、変数がベクトルを表現する場合に積  $(a + b) \cdot (c + d)$  を求めたい時、“命令型スタイル”で指定されたベクトルの値を設定するけれどもそれ自身はベクトルを値として返さない手続を用いて行うことは可能です。

```
(v-sum a b temp1)
(v-sum c d temp2)
(v-prod temp1 temp2 answer)
```

Alternatively, we could deal with expressions, using procedures that return vectors as values, and thus avoid explicitly mentioning `temp1` and `temp2`:

代替法として、ベクトルを値として返す手続を用いて式を用いて行うことも可能です。その場合、明示的に `temp1` と `temp2` を記述する必要を避けることができます。

```
(define answer (v-prod (v-sum a b) (v-sum c d)))
```

Since Lisp allows us to return compound objects as values of procedures, we can transform our imperative-style constraint language into an expression-oriented style as shown in this exercise. In languages that are impoverished in handling compound objects, such as Algol, Basic, and Pascal (unless one explicitly uses Pascal pointer variables), one is usually stuck with the imperative style when manipulating compound objects. Given the advantage of the expression-oriented format, one might ask if there is any reason to have implemented the system in imperative style, as we did in this section. One reason is that the non-expression-oriented constraint language provides a handle on constraint objects (e.g., the value of the `adder` procedure) as well as on connector objects. This is useful if we wish to extend the system with new operations that communicate with constraints directly rather than only indirectly via operations on connectors. Although it is easy to implement the expression-oriented style in terms of the imperative implementation, it is very difficult to do the converse.

Lisp は手続の値として複合オブジェクトを返すことができるため、命令型スタイル制約言語を式指向スタイルに課題で示されたように変形することができます。複合データの扱いが乏しい言語、例えば Algol, Basic, Pascal(明示的に Pascal のポインタ変数を用いる場合は除く) では通常複合オブジェクトを操作する場合に命令型スタイルに行き詰まります。式指向形式の利点を与えられるとある人はシステムを私達がこの節で行ったように命令型スタイルで実装することに何らかの意味があるのかと尋ねるかもしれません。1つの理由は非式指向の制約言語は制約オブジェクト上に、コネクタオブジェクト上

## 3.4 Concurrency: Time Is of the Essence

We've seen the power of computational objects with local state as tools for modeling. Yet, as Section 3.1.3 warned, this power extracts a price: the loss of referential transparency, giving rise to a thicket of questions about sameness and change, and the need to abandon the substitution model of evaluation in favor of the more intricate environment model.

私達はここまで局所状態をモデリングのためのツールとして持つ計算オブジェクトの力を学びました。それにもかかわらず、Section 3.1.3で警告したように、この力にはコストが伴います。参照等価性を失なうことは等価性と変更に関する問題のチケットを増加し、評価の置換モデルを断念し、より何回な環境モデルの支持を必要とします。

The central issue lurking beneath the complexity of state, sameness, and change is that by introducing assignment we are forced to admit *time* into our computational models. Before we introduced assignment, all our programs were timeless, in the sense that any expression that has a value always has the same value. In contrast, recall the example of modeling withdrawals from a bank account and returning the resulting balance, introduced at the beginning of Section 3.1.1:

状態、等価性、変更の複雑さの下に潜んでいる中心的課題は、代入を導入することにより私達は計算モデルの中に*time*(時間)の存在を認めることを強制されることです。代入の導入前は私達のプログラム全ては値を持つ任意の式が常に同じ値を持つという意味において恒久的でした。対照的に、Section 3.1.1で紹介した銀行口座からの引き出しと差引残高の返却のモデル化の例を思い出して下さい。

```
(withdraw 25)  
75  
(withdraw 25)  
50
```

Here successive evaluations of the same expression yield different values. This behavior arises from the fact that the execution of assignment statements (in

と同様にハンドルを提供します(例えば adder 手続の値)。これはもし我々がシステムをコネクタ上の命令を通して間接的に通信するだけでなく、制約と直接通信する新しい命令を用いてシステムを拡張したい場合にはとても便利です。式指向スタイルを命令型の実装を用いて実装するのは簡単ですが、逆はとても難しいのです。

this case, assignments to the variable `balance`) delineates *moments in time* when values change. The result of evaluating an expression depends not only on the expression itself, but also on whether the evaluation occurs before or after these moments. Building models in terms of computational objects with local state forces us to confront time as an essential concept in programming.

ここでは同じ式の一連の評価が異なる値を生じています。この振舞は代入文の実行（この場合では変数 `balance` への代入）が値が変化した *moments in time*（時間の瞬間）を描いています。式の評価の結果は式自身だけではなく、これらの瞬間の前か後に評価が行われたかにも依存します。局所状態を持つ計算モデルを用いたモデルの構築は私達にプログラミングにおける本質的な概念としての時間に直面することを強います。

We can go further in structuring computational models to match our perception of the physical world. Objects in the world do not change one at a time in sequence. Rather we perceive them as acting *concurrently*—all at once. So it is often natural to model systems as collections of computational processes that execute concurrently. Just as we can make our programs modular by organizing models in terms of objects with separate local state, it is often appropriate to divide computational models into parts that evolve separately and concurrently. Even if the programs are to be executed on a sequential computer, the practice of writing programs as if they were to be executed concurrently forces the programmer to avoid inessential timing constraints and thus makes programs more modular.

計算モデルの構造化において物理世界の私達の認知を一致させることをより進めることは可能です。世界の中のオブジェクトは一時に 1 つが順に変わることはありません。そうではなく、私達はそれらが *concurrently*（並行）に一同時に行動することを知覚します。そのためシステムを並行に実行する計算処理の集合であるとモデル化することは多くにおいて自然です。分離された局所状態を持つオブジェクトを用いてモデルを体系化することにより私達のプログラムをモジュラー化するのと同様に、計算モデルを別々に、並行に発展する部分に分割することは多くの場合に適切です。例えプログラムが逐次的な計算機により実行されるとしてもプログラムを並行に実行される前提で書くことを練習することはプログラマに不必要的制約を防ぐことを強いるため、プログラムをよりモジュール式にします。

In addition to making programs more modular, concurrent computation can provide a speed advantage over sequential computation. Sequential computers execute only one operation at a time, so the amount of time it takes to perform a

task is proportional to the total number of operations performed.<sup>65</sup> However, if it is possible to decompose a problem into pieces that are relatively independent and need to communicate only rarely, it may be possible to allocate pieces to separate computing processors, producing a speed advantage proportional to the number of processors available.

プログラムをよりモジュール式にするのに加えて、並行演算は逐次的演算に対し速度上の利点を与えることが可能です。逐次的演算は一時に1つの命令のみを実行するためタスクの実行にかかる時間量は実行される命令の総量に比例します。<sup>66</sup> しかもしも問題を相対的に独立した部分に分割することが可能で、稀にしか通信を行う必要が無ければ、それらの部分を異なる計算機に配置し、存在する計算機の数に比例した速さの利点を生じることが可能となるでしょう。

Unfortunately, the complexities introduced by assignment become even more problematic in the presence of concurrency. The fact of concurrent execution, either because the world operates in parallel or because our computers do, entails additional complexity in our understanding of time.

残念なことに、代入により持ち込まれた複雑性は並行性の出現により、より一層難しくなります。並行実行の結果は世界が並列に作動するためか計算機がそれを行うためかによらず、私達の時間の理解にさらなる複雑性をもたらします。

### 3.4.1 The Nature of Time in Concurrent Systems

On the surface, time seems straightforward. It is an ordering imposed on events.<sup>67</sup> For any events  $A$  and  $B$ , either  $A$  occurs before  $B$ ,  $A$  and  $B$  are simultaneous, or  $A$  occurs after  $B$ . For instance, returning to the bank account example, suppose that Peter withdraws \$10 and Paul withdraws \$25

---

<sup>65</sup> Most real processors actually execute a few operations at a time, following a strategy called *pipelining*. Although this technique greatly improves the effective utilization of the hardware, it is used only to speed up the execution of a sequential instruction stream, while retaining the behavior of the sequential program.

<sup>66</sup> 本物のCPUの多くは実際にはいくつかの命令を同時に、*pipelining*(パイプライン)と呼ばれる戦略に従い実行します。このテクニックは大きくハードウェアの実行効率を改善しますが、これは一連の命令ストリームの実行を、逐次的プログラムの振舞を保ちながら高速化するためのみに利用されます。

<sup>67</sup> To quote some graffiti seen on a Cambridge building wall: “Time is a device that was invented to keep everything from happening at once.”

from a joint account that initially contains \$100, leaving \$65 in the account. Depending on the order of the two withdrawals, the sequence of balances in the account is either \$100 → \$90 → \$65 or \$100 → \$75 → \$65. In a computer implementation of the banking system, this changing sequence of balances could be modeled by successive assignments to a variable `balance`.

表面上は時間は簡単に見えます。時間はイベントに課される順序付けです。<sup>68</sup> 任意のイベント *A* と *B* に対し、*A* が *B* の前に起こるか、*A* と *B* が同時に、*A* が *B* の後に起こるかです。例えば、銀行口座の例に戻れば、最初に \$100 を持つ連結口座から Peter が \$10 を引き出し、Paul が \$25 を引き出した場合、口座には \$65 が残ります。二人の引き出し順により、口座の残高の列は \$100 → \$90 → \$65 か \$100 → \$75 → \$65 です。銀行システムの計算機実装においてこの口座の列の変化は連続した変数 `balance` への代入としてモデル化できます。

In complex situations, however, such a view can be problematic. Suppose that Peter and Paul, and other people besides, are accessing the same bank account through a network of banking machines distributed all over the world. The actual sequence of balances in the account will depend critically on the detailed timing of the accesses and the details of the communication among the machines.

複雑な状況ではしかし、そのような見方は問題となります。Peter と Paul に加えて他の人々が同じ銀行口座に世界中に分散された現金自動預け払い機のネットワークを通してアクセスするとします。実際の口座の残高の列は大きく、アクセスタイミングの詳細と機械の間の通信の詳細に依存します。

This indeterminacy in the order of events can pose serious problems in the design of concurrent systems. For instance, suppose that the withdrawals made by Peter and Paul are implemented as two separate processes sharing a common variable `balance`, each process specified by the procedure given in Section 3.1.1:

このイベント順の非決定性は並行システムの設計において深刻な問題を提起します。例えば Peter と Paul の引き出しが共通の変数 `balance` を共有する 2 つの分離した処理だとします。各処理はSection 3.1.1にて与えられた手続により指定されます。

```
(define (withdraw amount)
```

---

<sup>68</sup>ケンブリッジのビルの壁上のある落書きを引用すれば“時間とは全てが同時に起こることを防ぐために発明された仕掛けだ”

```
(if (>= balance amount)
  (begin
    (set! balance (- balance amount)) balance)
  "Insufficient funds"))
```

If the two processes operate independently, then Peter might test the balance and attempt to withdraw a legitimate amount. However, Paul might withdraw some funds in between the time that Peter checks the balance and the time Peter completes the withdrawal, thus invalidating Peter's test.

もし2つの処理が独立に動作するなら、Peterは残高を確認し、正当な額面を引き出そうとします。しかしPaulがPeterが残高を確認した時点とPeterが引き出しを完了する時点の間にいくらかの資金を引き出すかもしれません。従ってPeterの確認を無効にするかもしれません。

Things can be worse still. Consider the expression

さらに悪くなります。以下の式について考えてみましょう。

```
(set! balance (- balance amount))
```

executed as part of each withdrawal process. This consists of three steps: (1) accessing the value of the `balance` variable; (2) computing the new balance; (3) setting `balance` to this new value. If Peter and Paul's withdrawals execute this statement concurrently, then the two withdrawals might interleave the order in which they access `balance` and set it to the new value.

この式は各引き出し処理の部分として実行されます。これは3つのステップから成り立ちます。(1) 変数 `balance` の値にアクセスする。(2) 新しい残高を計算する。(3) `balance` に新しい値を設定する。もしPeterとPaulの引き出しがこの命令を並行に実行した場合、二人の引き出しは `balance` にアクセスし、それに新しい値を設定する順を交互に配置するかもしれません。

The timing diagram in [Figure 3.29](#) depicts an order of events where `balance` starts at 100, Peter withdraws 10, Paul withdraws 25, and yet the final value of `balance` is 75. As shown in the diagram, the reason for this anomaly is that Paul's assignment of 75 to `balance` is made under the assumption that the value of `balance` to be decremented is 100. That assumption, however, became invalid when Peter changed `balance` to 90. This is a catastrophic failure for the banking system, because the total amount of money in the system is not conserved. Before the transactions, the total amount of money was \$100. Afterwards, Peter has \$10, Paul has \$25, and the bank has \$75.<sup>69</sup>

---

<sup>69</sup>An even worse failure for this system could occur if the two `set!` operations at-

Figure 3.29のタイミング図はbalanceが100で開始し、Peterが10を引き出し、Paulが25を引き出し、それでもbalanceの最終の値が75である場合のイベントの順を描写しています。図に示されるとおり、この異例の理由はPaulのbalanceへの75の代入が減算されるべきbalanceの値が100であるという前提の下で行われているためです。しかしこの前提はPeterがbalanceを90に変更した時に無効になります。これは銀行システムにとって最悪な失敗です。なぜならシステム中のお金の総量が保存されていません。取引前にお金の総額は100でした。その後、Peterは\$10を持ち、Paulは\$25を持ち、銀行は\$75を持っています。<sup>70</sup>

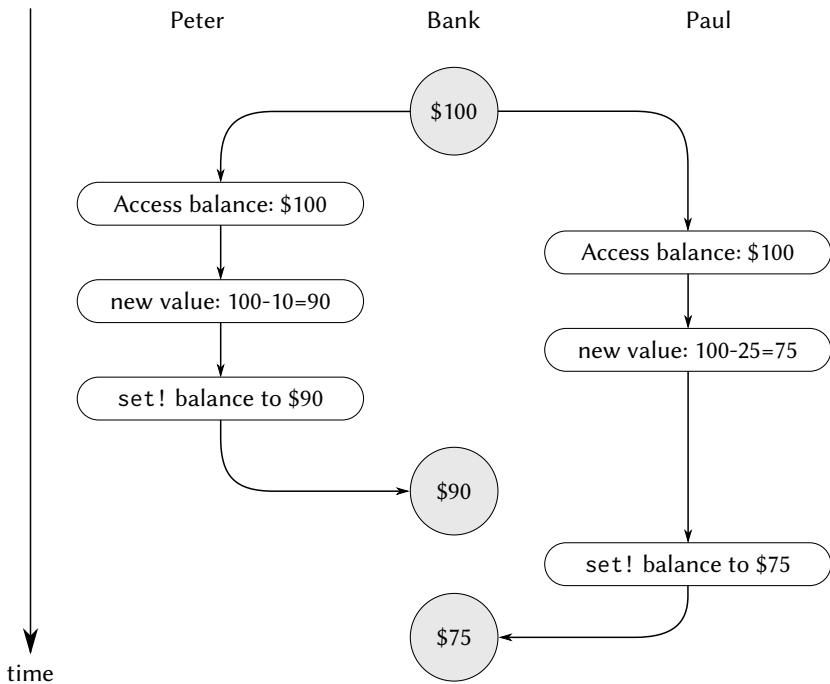
The general phenomenon illustrated here is that several processes may share a common state variable. What makes this complicated is that more than one process may be trying to manipulate the shared state at the same time. For the bank account example, during each transaction, each customer should be able to act as if the other customers did not exist. When a customer changes the balance in a way that depends on the balance, he must be able to assume that, just before the moment of change, the balance is still what he thought it was.

ここに描かれた一般的な現象は、いくつかのプロセスが共通な状態変数を共有していることです。このことを複雑にしているのは複数のプロセスが共有

---

tempt to change the balance simultaneously, in which case the actual data appearing in memory might end up being a random combination of the information being written by the two processes. Most computers have interlocks on the primitive memory-write operations, which protect against such simultaneous access. Even this seemingly simple kind of protection, however, raises implementation challenges in the design of multiprocessor computers, where elaborate *cache-coherence* protocols are required to ensure that the various processors will maintain a consistent view of memory contents, despite the fact that data may be replicated (“cached”) among the different processors to increase the speed of memory access.

<sup>70</sup> このシステムにより悪い失敗が2つのset!命令が残高を同時に変更しようとした場合に起こります。このような場合にはメモリ中に現れる実際のデータは2つの処理により書かれる情報の不作為な組み合わせに最後にはなるかもしれません。多くのコンピュータはプリミティブなメモリライト命令上に内部ロックを持つため、そのような同時アクセスを防ぎます。しかし、この見たところ簡単な種類のプロジェクトでさえマルチプロセスのコンピュータの設計においては実装上の課題を提起します。多様なプロセッサが、データが異なるプロセッサの間でメモリアクセスのスピードを向上するためにレプリケート（“キャッシュとして保存”）が行われるかもしれないという事実に係らず、静的なメモリ内容の見かけを得ることを保証するには、複雑な*cache-coherence*（キャッシュ一貫性）プロトコルが必要となります。



**Figure 3.29:** Timing diagram showing how interleaving the order of events in two banking withdrawals can lead to an incorrect final balance.

された状態を同時に操作しようと試みていることです。銀行口座の例では、各取引の間に、各顧客は他の顧客が存在しないかのように行動できなければなりません。顧客が口座を残高に依存した形で更新する時、その顧客は、変更の瞬間の前に、残高が依然として彼が考えた状態であることを前提とできなければなりません。

## Correct behavior of concurrent programs

The above example typifies the subtle bugs that can creep into concurrent programs. The root of this complexity lies in the assignments to variables that are shared among the different processes. We already know that we must be careful in writing programs that use `set!`, because the results of a computation depend on the order in which the assignments occur.<sup>71</sup> With concurrent processes we must be especially careful about assignments, because we may not be able to control the order of the assignments made by the different processes. If several such changes might be made concurrently (as with two depositors accessing a joint account) we need some way to ensure that our system behaves correctly. For example, in the case of withdrawals from a joint bank account, we must ensure that money is conserved. To make concurrent programs behave correctly, we may have to place some restrictions on concurrent execution.

先の例は並行プログラムに潜みがちな微妙なバグの類型です。この複雑性の根本は異なるプロセスの間で共有される変数への代入に横たわっています。私達は既に `set!` を用いるプログラムを書く場合には気をつけねばならないことを知っています。計算の結果が代入の起こる順に依存するためです。<sup>72</sup> 並行プロセスでは特に代入に気をつけねばなりません。異なるプロセスにより作られる代入の順をコントロールできないかもしれませんためです。もしくはいくつかのそのような変更が（二人の預金者が連結口座にアクセスするように）並行に行われるのであれば私達のシステムの振舞が正しいことを確認するための何らかの方法を必要とします。例えば、連結口座からの引き出しの場合、お金が保管されていることを確認しなければなりません。並行プログラムの振舞を正しくするために、並行実行に何らかの制限を置かねばなりません。

One possible restriction on concurrency would stipulate that no two op-

---

<sup>71</sup>The factorial program in Section 3.1.3 illustrates this for a single sequential process.

<sup>72</sup>Section 3.1.3における指数プログラムはこのことを单一の逐次処理にて説明しました。

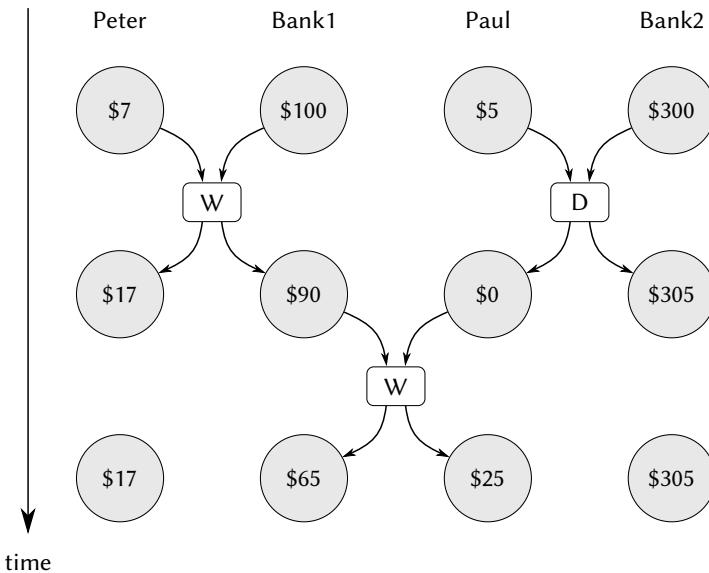
erations that change any shared state variables can occur at the same time. This is an extremely stringent requirement. For distributed banking, it would require the system designer to ensure that only one transaction could proceed at a time. This would be both inefficient and overly conservative. Figure 3.30 shows Peter and Paul sharing a bank account, where Paul has a private account as well. The diagram illustrates two withdrawals from the shared account (one by Peter and one by Paul) and a deposit to Paul's private account.<sup>73</sup> The two withdrawals from the shared account must not be concurrent (since both access and update the same account), and Paul's deposit and withdrawal must not be concurrent (since both access and update the amount in Paul's wallet). But there should be no problem permitting Paul's deposit to his private account to proceed concurrently with Peter's withdrawal from the shared account.

並行性への可能な1つの制限は、任意の共有状態変数を変更するどの2つの命令もどうじには起こり得ないことです。これはとても厳しい制限です。分散銀行システムではシステム設計に対しだけ1つの取引だけが一時に手続できることを保証することを要求します。これは非効率であり、かつ過度に保守的です。Figure 3.30はPeterとPaulが銀行口座を共有し、Paulはまたプライベートな口座を持っていることを示しています。共有口座からの2つの引き出し(1つはPeterによる、もう1つはPaulによるもの)とPaulのプライベート口座への預金を図示しています。<sup>74</sup>共有口座からの2つの引き出しは並行であってはなりません(両方が同じ口座にアクセスと更新を行うため)。またPaulの預金と引き出しは並行であってはなりません(両方がPaulの財布にアクセスと更新を行うため)。しかしPaulによる彼のプライベート口座への預金をPeterの共有アカウントからの引き出しと並行に進行することを許すことは何の問題も起こさないはずです。

A less stringent restriction on concurrency would ensure that a concurrent system produces the same result as if the processes had run sequentially in some order. There are two important aspects to this requirement. First, it does not require the processes to actually run sequentially, but only to produce results

<sup>73</sup>The columns show the contents of Peter's wallet, the joint account (in Bank1), Paul's wallet, and Paul's private account (in Bank2), before and after each withdrawal (W) and deposit (D). Peter withdraws \$10 from Bank1; Paul deposits \$5 in Bank2, then withdraws \$25 from Bank1.

<sup>74</sup>列はPeterの財布、(Bank1内の)共有口座、Paulの財布、(Bank2内の)Paulのプライベート口座の中身を各引き出し(W)と預金(D)の前後にて示しています。Peterは\$10をBank1から引き出し、Paulは\$5をBank2に預金し、次にBank1から\$25を引き出しています。



**Figure 3.30:** Concurrent deposits and withdrawals from a joint account in Bank1 and a private account in Bank2.

that are the same *as if* they had run sequentially. For the example in Figure 3.30, the designer of the bank account system can safely allow Paul’s deposit and Peter’s withdrawal to happen concurrently, because the net result will be the same as if the two operations had happened sequentially. Second, there may be more than one possible “correct” result produced by a concurrent program, because we require only that the result be the same as for *some* sequential order. For example, suppose that Peter and Paul’s joint account starts out with \$100, and Peter deposits \$40 while Paul concurrently withdraws half the money in the account. Then sequential execution could result in the account balance being either \$70 or \$90 (see Exercise 3.38).<sup>75</sup>

<sup>75</sup> A more formal way to express this idea is to say that concurrent programs are inherently *nondeterministic*. That is, they are described not by single-valued functions,

並行性上の比較的厳しくない制限は並行システムがまるでプロセスが同じ順に逐次的に実行されたかのように同じ結果を生成することを保証します。2つの重要な側面がこの制限にはあります。第一にプロセスに対し実際に逐次的に実行することを要求はしませんが、あたかも逐次的に実行された場合と同じ結果を生成することを要求します。[Figure 3.30](#)の例に対して銀行口座システムの設計者は安全に Paul の預金と Peter の引き出しを並行に起こすことを許可できます。なぜなら 2 つの命令が逐次的に起きたのと最終結果が同じになるためです。第二に、複数の可能な“正しい”結果が並行プログラムにより生成されるでしょう。なぜなら結果がある逐次的順序と同じ結果であることを要求しているためです。例えば Peter と Paul の連結口座に \$100 が最初にあるとし、Peter が \$40 を預金し、Paul が並行に口座の半分のお金を取り出しました。すると逐次的実行の口座残高は \$70 か \$90 のどちらかになります ([Exercise 3.38](#) 参照)。<sup>76</sup>

There are still weaker requirements for correct execution of concurrent programs. A program for simulating diffusion (say, the flow of heat in an object) might consist of a large number of processes, each one representing a small volume of space, that update their values concurrently. Each process repeatedly changes its value to the average of its own value and its neighbors' values. This algorithm converges to the right answer independent of the order in which the operations are done; there is no need for any restrictions on concurrent use of the shared values.

並行プログラムの正しい実行のためのより弱い要件はまだ有ります。拡散のシミュレーションのプログラム（例えば物質内の熱の流れ）は巨大な数のプロセスから成り、各プロセスは小容量の空間を表し、その値を並行に更新します。各プロセスはその値を、その値と近傍の値の平均へと繰り返し変更します。このアルゴリズムは命令が行われる順から独立して正しい答に収束します。共有値の並行な使用上にどんな制限も必要としません。

**Exercise 3.38:** Suppose that Peter, Paul, and Mary share a joint bank account that initially contains \$100. Concurrently, Peter deposits \$10, Paul withdraws \$20, and Mary withdraws half the

---

but by functions whose results are sets of possible values. In [Section 4.3](#) we will study a language for expressing nondeterministic computations.

<sup>76</sup>この考え方をより形式的に表す方法は、並行プログラムは本質的に *nondeterministic*(非決定的)であると述べることです。つまり、それらは单一の値を持つ関数ではなく、結果が起こり得る値の集合となる関数により説明されます。[Section 4.3](#)では非決定的演算について学びます。

money in the account, by executing the following commands:

Peter, Paul, Mary が初めに \$100 を持つ連結銀行口座を共有すると仮定する。並行に、Peter が \$10 の預金、Paul が \$20 の引き出し、Mary は口座の半分のお金の引き出しを以下のコマンドにより実行した。

```
Peter: (set! balance (+ balance 10))
Paul: (set! balance (- balance 20))
Mary: (set! balance (/ balance 2)))
```

- a List all the different possible values for `balance` after these three transactions have been completed, assuming that the banking system forces the three processes to run sequentially in some order.

これらの 3 つの取引が完了した後に、全ての異なる `balance` の起り得る値を並べよ。ただし銀行システムはこの 3 つのプロセスが何らかの順にて逐次的に実行する前提とする。

- b What are some other values that could be produced if the system allows the processes to be interleaved? Draw timing diagrams like the one in [Figure 3.29](#) to explain how these values can occur.

もしシステムがプロセスにインターリーブ(相互配置)を認めた場合に生成される他の値は何か? [Figure 3.29](#)の様なタイミング図を描きこれらの値がどのように起り得るのか説明せよ。

### 3.4.2 Mechanisms for Controlling Concurrency

We've seen that the difficulty in dealing with concurrent processes is rooted in the need to consider the interleaving of the order of events in the different processes. For example, suppose we have two processes, one with three ordered events  $(a, b, c)$  and one with three ordered events  $(x, y, z)$ . If the two processes run concurrently, with no constraints on how their execution is interleaved, then there are 20 different possible orderings for the events that are consistent with the individual orderings for the two processes:

並行プロセスの取扱における困難は異なるプロセスのイベント順の交互配置について考える必要性に原因があることを学びました。例えば 2 つのプロセスがあり 1 つは 3 つの順序付けられたイベント  $(a, b, c)$  で、もう 1 つは 3 つの順序付けられたイベント  $(x, y, z)$  であるとします。もし 2 つのプロセスが、それらの実行がどのように相互配置されるのかについて制約無しで並行に実行された時、2 つのプロセスの個々の順は変わらないとしても、20 の異なる起こり得るイベントの順が存在します。

$(a, b, c, x, y, z)$	$(a, x, b, y, c, z)$	$(x, a, b, c, y, z)$	$(x, a, y, z, b, c)$
$(a, b, x, c, y, z)$	$(a, x, b, y, z, c)$	$(x, a, b, y, c, z)$	$(x, y, a, b, c, z)$
$(a, b, x, y, c, z)$	$(a, x, y, b, c, z)$	$(x, a, b, y, z, c)$	$(x, y, a, b, z, c)$
$(a, b, x, y, z, c)$	$(a, x, y, b, z, c)$	$(x, a, y, b, c, z)$	$(x, y, a, z, b, c)$
$(a, x, b, c, y, z)$	$(a, x, y, z, b, c)$	$(x, a, y, b, z, c)$	$(x, y, z, a, b, c)$

As programmers designing this system, we would have to consider the effects of each of these 20 orderings and check that each behavior is acceptable. Such an approach rapidly becomes unwieldy as the numbers of processes and events increase.

プログラマがこのシステムを設計するにつれ、これらの 20 種の順序のそれぞれの結果について考慮して、各振舞が受け入れられるか確認する必要があるでしょう。そのような取り組み方はプロセスとイベントの数が増加するにつれ、急速に手に負えない物となるでしょう。

A more practical approach to the design of concurrent systems is to devise general mechanisms that allow us to constrain the interleaving of concurrent processes so that we can be sure that the program behavior is correct. Many mechanisms have been developed for this purpose. In this section, we describe one of them, the *serializer*.

並行システムの設計に対するより現実的なアプローチはプログラムの振舞が正しいことを確認できるよう並行プロセスのインターリーブを制約できる一般的な仕組みを工夫することです。多くの仕組みがこの目的のため開発されてきました。この節ではそれらの 1 つ、*serializer*(シリализア、並列直列変換器)について学びます。

## Serializing access to shared state

Serialization implements the following idea: Processes will execute concurrently, but there will be certain collections of procedures that cannot be ex-

ecuted concurrently. More precisely, serialization creates distinguished sets of procedures such that only one execution of a procedure in each serialized set is permitted to happen at a time. If some procedure in the set is being executed, then a process that attempts to execute any procedure in the set will be forced to wait until the first execution has finished.

直列化 (serialization) は次の考えを実装します。プロセスは並行に実行します。しかし幾つかの手続の集合が存在し、それらは並行には実行できません。もっと正確に言えば直列化は各直列化された集合内のただ 1 つの手続の実行が一時に許されるような複数の区別された手続の集合を作成します。もし 1 つの集合内のいくつかの手続が実行されるなら、集合内の任意の手続を実行しようとするプロセスは最初の実行が完了するまで待つことを強制されます。

We can use serialization to control access to shared variables. For example, if we want to update a shared variable based on the previous value of that variable, we put the access to the previous value of the variable and the assignment of the new value to the variable in the same procedure. We then ensure that no other procedure that assigns to the variable can run concurrently with this procedure by serializing all of these procedures with the same serializer. This guarantees that the value of the variable cannot be changed between an access and the corresponding assignment.

直列化を用いて共有変数へのアクセスをコントロールできます。例えばもし共有変数をその変数の前の値に応じて変更したい時、同じ手続内でその変数の以前の値にアクセスし、その変数に新しい値を代入します。それからその変数に代入するどの他の手続もこの手続とは並行には実行でなきないことを、同じシリアルライザを持つこれらの手続の全てを直列化することにより確実にします。これはその変数の値がアクセスとそれに対応する代入の間に変更されることができないことを保証します。

## Serializers in Scheme

To make the above mechanism more concrete, suppose that we have extended Scheme to include a procedure called `parallel-execute`:

上記の仕組みをより確実に行うために、`parallel-execute`(並列実行) と呼ばれる手続を含む拡張 Scheme を持っていると仮定しましょう。

`(parallel-execute <p1> <p2> ... <pk>)`

Each  $\langle p \rangle$  must be a procedure of no arguments. **Parallel-execute** creates a separate process for each  $\langle p \rangle$ , which applies  $\langle p \rangle$  (to no arguments). These processes all run concurrently.<sup>77</sup>

各  $\langle p \rangle$  は引数無しの手続でなければなりません。**parallel-execute** は分離されたプロセスを各  $\langle p \rangle$  に対し作り、それらのプロセスは  $\langle p \rangle$  を（引数無しで）適用します。これらのプロセスは全て並行に実行されます。<sup>78</sup>

As an example of how this is used, consider

これがどのように利用されるかの例として、以下について考えてみて下さい。

```
(define x 10)
(parallel-execute
  (lambda () (set! x (* x x)))
  (lambda () (set! x (+ x 1))))
```

This creates two concurrent processes— $P_1$ , which sets  $x$  to  $x$  times  $x$ , and  $P_2$ , which increments  $x$ . After execution is complete,  $x$  will be left with one of five possible values, depending on the interleaving of the events of  $P_1$  and  $P_2$ :

これ2つの並行プロセス— $x$ に  $x$ かける  $x$ を設定する  $P_1$  と、 $x$ に1を足す  $P_2$ を作成します。実行完了後に、 $P_1$ と  $P_2$  のイベントのインターリーブに依存するため、 $x$ は5つの起こり得る値の内1つに成ります。

- 101:  $P_1$  sets  $x$  to 100 and then  $P_2$  increments  $x$  to 101.
- 101:  $P_1$  が  $x$ に 100 を設定し、次に  $P_2$  が  $x$ を 101 に増やす
- 121:  $P_2$  increments  $x$  to 11 and then  $P_1$  sets  $x$  to  $x * x$ .
- 121:  $P_2$  が  $x$ を 11 に増やし、次に  $P_1$  が  $x$ を  $x * x$ に設定
- 110:  $P_2$  changes  $x$  from 10 to 11 between the two times that
- 110:  $P_2$  が  $x$ を 10 から 11 に以下の2度のアクセスの間に変化させる

---

<sup>77</sup>Parallel-execute is not part of standard Scheme, but it can be implemented in MIT Scheme. In our implementation, the new concurrent processes also run concurrently with the original Scheme process. Also, in our implementation, the value returned by parallel-execute is a special control object that can be used to halt the newly created processes.

<sup>78</sup>parallel-execute は標準 Scheme の一部ではありません。しかし MIT Scheme で実装することが可能です。私達の実装においては新しい並行プロセスはまたオリジナルの Scheme プロセスと共に並行に実行できます。また私達の実装では parallel-execute により返される値は特別なコントロールオブジェクトであり新しく作成されたプロセスを停止するために使用できます。

$P_1$  accesses the value of  $x$  during the evaluation of  $(* \ x \ x)$ .

$P_1$  が  $x$  の値に  $(* \ x \ x)$  の評価の間にアクセスする

11:  $P_2$  accesses  $x$ , then  $P_1$  sets  $x$  to 100, then  $P_2$  sets  $x$ .

11:  $P_2$  が  $x$  にアクセスし、次に  $P_1$  が  $x$  に 100 を設定し、 $P_2$  が  $x$  を設定

100:  $P_1$  accesses  $x$  (twice), then  $P_2$  sets  $x$  to 11, then  $P_1$  sets  $x$ .

100:  $P_1$  が  $x$  に(二度) アクセスし、次に  $P_2$  が  $x$  を 11 に設定、次に  $P_1$  が  $x$  を設定

We can constrain the concurrency by using serialized procedures, which are created by *serializers*. Serializers are constructed by `make-serializer`, whose implementation is given below. A serializer takes a procedure as argument and returns a serialized procedure that behaves like the original procedure. All calls to a given serializer return serialized procedures in the same set.

並行性を *serializers*(シリアライザ) により作成された直列化された手続を用いることで抑制することができます。シリアライザは `make-serializer` により構築され、この実装は後程与えられます。シリアライザは手続を引数として取り、元の手続の様に振る舞う *serialized*(被直列化) 手続を返します。与えられたシリアライザへの全ての呼出は同じ集合に属する被直列化手続を返します。

Thus, in contrast to the example above, executing

従って上の例とは異なり、以下の実行は

```
(define x 10)
(define s (make-serializer))
(parallel-execute
 (s (lambda () (set! x (* x x))))
 (s (lambda () (set! x (+ x 1)))))
```

can produce only two possible values for  $x$ , 101 or 121. The other possibilities are eliminated, because the execution of  $P_1$  and  $P_2$  cannot be interleaved.

$x$  に対しだけ 2 つの起こり得る値、101 と 121 を返します。他の可能性は  $P_1$  と  $P_2$  の実行がインターリーブ(相互配置) されないため排除されました。

Here is a version of the `make-account` procedure from Section 3.1.1, where the deposits and withdrawals have been serialized:

以下に Section 3.1.1 の `make-account` 手続を預け入れと引き出しが直列化された版を示します。

```
(define (make-account balance)
  (define (withdraw amount)
```

```

(if (>= balance amount)
  (begin (set! balance (- balance amount))
         balance)
  "Insufficient funds"))
(define (deposit amount)
  (set! balance (+ balance amount))
  balance)
(let ((protected (make-serializer)))
  (define (dispatch m)
    (cond ((eq? m 'withdraw) (protected withdraw))
          ((eq? m 'deposit) (protected deposit))
          ((eq? m 'balance) balance)
          (else (error "Unknown request: MAKE-ACCOUNT"
                       m)))))

dispatch))

```

With this implementation, two processes cannot be withdrawing from or depositing into a single account concurrently. This eliminates the source of the error illustrated in Figure 3.29, where Peter changes the account balance between the times when Paul accesses the balance to compute the new value and when Paul actually performs the assignment. On the other hand, each account has its own serializer, so that deposits and withdrawals for different accounts can proceed concurrently.

この実装により、2つのプロセスは单一の口座に並行に預け入れと引き出しを行うことはできなくなりました。これによりFigure 3.29で図示されたエラーの原因、Paul の新しい値を求めるための残高へのアクセスと、Paul が実際に代入を行う時の間に、Peter が口座残高を変更する場合は排除されます。一方で、各口座はそれ自身のシリアルライザを持つので、異なる口座への預金と引き出しは並行に行うことができます。

**Exercise 3.39:** Which of the five possibilities in the parallel execution shown above remain if we instead serialize execution as follows:

上で示された並行実行における 5 つの可能性の内、もし変わりに以下のような実行を起こなった場合にどれが残るか?

```

(define x 10)
(define s (make-serializer))

```

```
(parallel-execute
  (lambda () (set! x ((s (lambda () (* x x))))))
  (s (lambda () (set! x (+ x 1)))))
```

**Exercise 3.40:** Give all possible values of  $x$  that can result from executing

以下を実行した場合に  $x$  の起こり得る値の全てを上げよ。

```
(define x 10)
(parallel-execute (lambda () (set! x (* x x)))
  (lambda () (set! x (* x x x))))
```

Which of these possibilities remain if we instead use serialized procedures:

これらの内もし代わりに以下の直列化手続を用いた場合どれが残るか?

```
(define x 10)
(define s (make-serializer))
(parallel-execute (s (lambda () (set! x (* x x))))
  (s (lambda () (set! x (* x x x)))))
```

**Exercise 3.41:** Ben Bitdiddle worries that it would be better to implement the bank account as follows (where the commented line has been changed):

Ben Bitdiddle は以下のように銀行口座を実行すればより良くなるのではないかと心配している。(コメントの有る行が変更されている)。

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance
                      (- balance amount))
              balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)))
```

```

        balance)
(let ((protected (make-serializer)))
(define (dispatch m)
  (cond ((eq? m 'withdraw) (protected withdraw))
        ((eq? m 'deposit) (protected deposit))
        ((eq? m 'balance)
         ((protected
           (lambda () balance)))) ; serialized
        (else
         (error "Unknown request: MAKE-ACCOUNT"
               m))))
  dispatch))

```

because allowing unserialized access to the bank balance can result in anomalous behavior. Do you agree? Is there any scenario that demonstrates Ben's concern?

心配の理由は非直列化アクセスを銀行口座に許すと得意な振舞が起こり得るためだ。同意するか? Ben の懸念を実演するシナリオは存在するか?

**Exercise 3.42:** Ben Bitdiddle suggests that it's a waste of time to create a new serialized procedure in response to every `withdraw` and `deposit` message. He says that `make-account` could be changed so that the calls to `protected` are done outside the `dispatch` procedure. That is, an account would return the same serialized procedure (which was created at the same time as the account) each time it is asked for a withdrawal procedure.

Ben Bitdiddle は全ての `withdraw` と `deposit` メッセージに対して新しい被直列化手続を作成することは時間の無駄であると提案した。彼は `protected` への呼出が `dispatch` 手続の外で行われるよう `make-account` を変更することができると述べた。つまり `withdrawal` 手続が呼ばれる度に、口座が(口座が作成されたと同時に作成された)同じ被直列化手続を返すことになるだろう。

```

(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))

```

```

        balance)
      "Insufficient funds"))
(define (deposit amount)
  (set! balance (+ balance amount))
  balance)
(let ((protected (make-serializer)))
  (let ((protected-withdraw (protected withdraw))
        (protected-deposit (protected deposit)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) protected-withdraw)
            ((eq? m 'deposit) protected-deposit)
            ((eq? m 'balance) balance)
            (else
              (error "Unknown request: MAKE-ACCOUNT"
                     m))))
    dispatch)))

```

Is this a safe change to make? In particular, is there any difference in what concurrency is allowed by these two versions of `make-account`?

これは行うことが安全な変更だろうか? 具体的には、これらの2つの版の `make-account` により許される並行性に違いは存在するだろうか?

## Complexity of using multiple shared resources

Serializers provide a powerful abstraction that helps isolate the complexities of concurrent programs so that they can be dealt with carefully and (hopefully) correctly. However, while using serializers is relatively straightforward when there is only a single shared resource (such as a single bank account), concurrent programming can be treacherously difficult when there are multiple shared resources.

シリアルライザは並行プログラムの複雑性の分離を手助けすることで、注意深く(願わくは)正しく取り扱えるようにする強力な抽象化を与えます。しかしシリアルライザの使用は(単一の銀行口座のような)ただ単一の共有リソースが存在する場合には相対的に簡単ですが、並行プログラミングは複数の共有リソースがある場合に、裏切るかのように難しくなります。

To illustrate one of the difficulties that can arise, suppose we wish to swap the balances in two bank accounts. We access each account to find the balance, compute the difference between the balances, withdraw this difference from one account, and deposit it in the other account. We could implement this as follows:<sup>79</sup>

提起できる困難さの内 1 つを説明するために、2 つの銀行口座の残高を交換したいと考えます。各口座にアクセスし残高を見つけ、残高間の差を計算し、一方の口座からこの差を引き出し、もう一方の口座へ預け入れます。これ以下のように実装することができます。<sup>80</sup>

```
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                        (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))
```

This procedure works well when only a single process is trying to do the exchange. Suppose, however, that Peter and Paul both have access to accounts  $a_1$ ,  $a_2$ , and  $a_3$ , and that Peter exchanges  $a_1$  and  $a_2$  while Paul concurrently exchanges  $a_1$  and  $a_3$ . Even with account deposits and withdrawals serialized for individual accounts (as in the `make-account` procedure shown above in this section), `exchange` can still produce incorrect results. For example, Peter might compute the difference in the balances for  $a_1$  and  $a_2$ , but then Paul might change the balance in  $a_1$  before Peter is able to complete the exchange.<sup>81</sup> For correct behavior, we must arrange for the `exchange` procedure to lock out any other concurrent accesses to the accounts during the entire time of the exchange.

この手続は单一のプロセスのみが交換を試みる場合にはうまく働きます。しかし Peter と Paul が二人共口座  $a_1$ ,  $a_2$ ,  $a_3$  にアクセスし、そして Peter が  $a_1$  と  $a_2$  を交換している間に Paul が並行に  $a_1$  と  $a_3$  を交換している場合を考えて

---

<sup>79</sup>We have simplified `exchange` by exploiting the fact that our `deposit` message accepts negative amounts. (This is a serious bug in our banking system!)

<sup>80</sup>`deposit` メッセージが負の額面を受け入れるという事実を利用することで `exchange` を簡略化しました。(これは私達の銀行システムの深刻なバグです！)

<sup>81</sup>If the account balances start out as \$10, \$20, and \$30, then after any number of concurrent exchanges, the balances should still be \$10, \$20, and \$30 in some order. Serializing the deposits to individual accounts is not sufficient to guarantee this. See Exercise 3.43.

みて下さい。例え口座の預け入れと引き出しが個別の口座に対して(この節の上で示された `make-account` 手続のように)直列化されたとしても、`exchange` は依然として不正確な結果を生じることができます。例えば Peter が  $a_1$  と  $a_2$  の残高の差を求める時、Paul が Peter が交換を完了する前に  $a_1$  の残高を変更するかもしれません。<sup>82</sup> 正しい振舞のためには、`exchange` 手続を、交換の全体の時間の間、口座へのどの他の並行アクセスもロックアウト(締め出し)するように準備をしなければなりません。

One way we can accomplish this is by using both accounts' serializers to serialize the entire `exchange` procedure. To do this, we will arrange for access to an account's serializer. Note that we are deliberately breaking the modularity of the bank-account object by exposing the serializer. The following version of `make-account` is identical to the original version given in Section 3.1.1, except that a serializer is provided to protect the balance variable, and the serializer is exported via message passing:

これを達成する 1 つの方法は両方の口座のシリアルライザを用いて `exchange` 手続全体を直列化します。これを行うためには、口座のシリアルライザへのアクセスに準備を行います。シリアルライザを露出することで、銀行口座オブジェクトのモジュール化を意図的に破っていることに注意して下さい。`make-account` の以下の版はSection 3.1.1 で与えられた元の版とシリアルライザが `balance` 変数を守るため提供されていることを除けば同じです。そしてシリアルライザはメッセージパッシングを通して転送されます。

```
(define (make-account-and-serializer balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((balance-serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) withdraw)
```

---

<sup>82</sup>もし口座残高が \$10, \$20, \$30 で始めた場合、任意の回数の交換の後に、残高はどちらかの順にて依然として \$10, \$20, \$30 にならねばなりません。個別の口座への預け入れの直列化はこれを保証するのに十分ではありません。Exercise 3.43 を参照して下さい。

```

((eq? m 'deposit) deposit)
((eq? m 'balance) balance)
((eq? m 'serializer) balance-serializer)
(else
  (error "Unknown request: MAKE-ACCOUNT"
         m))))
dispatch))

```

We can use this to do serialized deposits and withdrawals. However, unlike our earlier serialized account, it is now the responsibility of each user of bank-account objects to explicitly manage the serialization, for example as follows:<sup>83</sup>

これを用いて直列化された預け入れと引き出しを行うことができます。しかし最初の直列化された口座とは異なり、直列化を明示的に管理することは銀行口座オブジェクトの各ユーザーの責任です。例えば以下の様にです。<sup>84</sup>

```

(define (deposit account amount)
  (let ((s (account 'serializer))
        (d (account 'deposit)))
    ((s d) amount)))

```

Exporting the serializer in this way gives us enough flexibility to implement a serialized exchange program. We simply serialize the original `exchange` procedure with the serializers for both accounts:

シリアルライザをこの方法で外出しすることは私達に直列化された交換プログラムを実装するのに十分な柔軟性を与えます。単純に元の `exchange` 手続を両方の口座のシリアルライザにて直列化します。

```

(define (serialized-exchange account1 account2)
  (let ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer)))
    ((serializer1 (serializer2 exchange))
     account1
     account2)))

```

<sup>83</sup>Exercise 3.45 investigates why deposits and withdrawals are no longer automatically serialized by the account.

<sup>84</sup>Exercise 3.45にてなぜ預け入れと引き出しがもはや自動的に口座により直列化されないのかについて調査します。

**Exercise 3.43:** Suppose that the balances in three accounts start out as \$10, \$20, and \$30, and that multiple processes run, exchanging the balances in the accounts. Argue that if the processes are run sequentially, after any number of concurrent exchanges, the account balances should be \$10, \$20, and \$30 in some order. Draw a timing diagram like the one in Figure 3.29 to show how this condition can be violated if the exchanges are implemented using the first version of the account-exchange program in this section. On the other hand, argue that even with this `exchange` program, the sum of the balances in the accounts will be preserved. Draw a timing diagram to show how even this condition would be violated if we did not serialize the transactions on individual accounts.

3つの口座の残高が \$10, \$20, \$30 で始まり、複数のプロセスが実行され口座の残高を交換すると考える。プロセスが逐次的に実行されるなら、任意の数の並行な交換の後に、口座残高がある順序において \$10, \$20, \$30 になると主張する。Figure 3.29 のようなタイミング図を描き、交換がこの節の `account-exchange` の最初の版を用いて実装された場合にこの前提がどのように破られるかについて示せ。一方で、例えこの `exchange` プログラムを用いても口座の残高の合計は保存されると主張する。タイミング図を描き、個別の口座上の取引を直列化しない場合には例えこの前提でもどのように破られるかについて示せ。

**Exercise 3.44:** Consider the problem of transferring an amount from one account to another. Ben Bitdiddle claims that this can be accomplished with the following procedure, even if there are multiple people concurrently transferring money among multiple accounts, using any account mechanism that serializes deposit and withdrawal transactions, for example, the version of `make-account` in the text above.

ある口座から別の口座への振込の問題について考える。Ben Bitdiddle は例え複数の人々が並行にお金を複数の口座間で転送をしても、以下の手続を用いることで、預金と引き出しの取引を直列化する任意の口座の仕組み、例え上記の `make-account` の版を用いながら振込を達成できると主張する。

```
(define (transfer from-account to-account amount))
```

```
((from-account 'withdraw) amount)
((to-account 'deposit) amount))
```

Louis Reasoner claims that there is a problem here, and that we need to use a more sophisticated method, such as the one required for dealing with the exchange problem. Is Louis right? If not, what is the essential difference between the transfer problem and the exchange problem? (You should assume that the balance in `from-account` is at least `amount`.)

Louis Reasoner はここにも問題があると主張した。交換問題を取り扱うのに必要とされた様なより洗練された手法が必要であるとも述べた。Louis は正しいだろうか? もし正しくないのならば振込問題と交換問題の間の本質的な違いは何か? (`from-account` の残高は少くとも `amount` であると考えること)。

**Exercise 3.45:** Louis Reasoner thinks our bank-account system is unnecessarily complex and error-prone now that deposits and withdrawals aren't automatically serialized. He suggests that `make-account-and-serializer` should have exported the serializer (for use by such procedures as `serialized-exchange`) in addition to (rather than instead of) using it to serialize accounts and deposits as `make-account` did. He proposes to redefine accounts as follows: Louis Reasoner は私達の銀行口座システムは不必要に複雑、かつエラーを起こしやすく、預け入れと引き出しも自動的に直列化されないと考えた。彼は `make-account` が行ったように口座と預け入れを直列化するためにそれを用いることに加えて(その代わりにではなく) `make-account-and-serializer` はシリアルライザを(`serialized-exchange` のような手続にて利用するために) 露出させるべきだったと主張した。彼は口座を以下のように再定義することを提案した。

```
(define (make-account-and-serializer balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
  (let ((balance-serializer (make-serializer)))
```

```

(define (dispatch m)
  (cond ((eq? m 'withdraw) (balance-serializer withdraw))
        ((eq? m 'deposit) (balance-serializer deposit))
        ((eq? m 'balance) balance)
        ((eq? m 'serializer) balance-serializer)
        (else (error "Unknown request: MAKE-ACCOUNT" m))))
  dispatch))

```

Then deposits are handled as with the original `make-account`:

すると `deposit(預け入れ)` は元々の `make-account` で用いたように扱われる。

```

(define (deposit account amount)
  ((account 'deposit) amount))

```

Explain what is wrong with Louis's reasoning. In particular, consider what happens when `serialized-exchange` is called.

Louis の推論の何が間違っているか説明せよ。具体的には `serialized-exchange` が呼ばれた時に何が起こるかについて考えよ。

## Implementing serializers

We implement serializers in terms of a more primitive synchronization mechanism called a *mutex*. A mutex is an object that supports two operations—the mutex can be *acquired*, and the mutex can be *released*. Once a mutex has been acquired, no other acquire operations on that mutex may proceed until the mutex is released.<sup>85</sup> In our implementation, each serializer has an associated mutex. Given a procedure `p`, the serializer returns a procedure that acquires

---

<sup>85</sup>The term “mutex” is an abbreviation for *mutual exclusion*. The general problem of arranging a mechanism that permits concurrent processes to safely share resources is called the mutual exclusion problem. Our mutex is a simple variant of the *semaphore* mechanism (see [Exercise 3.47](#)), which was introduced in the “THE” Multiprogramming System developed at the Technological University of Eindhoven and named for the university’s initials in Dutch ([Dijkstra 1968a](#)). The acquire and release operations were originally called P and V, from the Dutch words *passeren* (to pass) and *vrijgeven* (to release), in reference to the semaphores used on railroad systems. Dijkstra’s classic exposition (Section 1968b) was one of the first to clearly present the issues of concurrency control, and showed how to use semaphores to handle a variety of concurrency problems.

the mutex, runs p, and then releases the mutex. This ensures that only one of the procedures produced by the serializer can be running at once, which is precisely the serialization property that we need to guarantee.

私達はシリアルライザを *mutex*(ミューテックス、相互排除) と呼ばれるよりプリミティブな同期の仕組みを用いて実装します。mutex は 2 つの命令をサポートするオブジェクトです。1 つは mutex が *acquired*(獲得) でき、もう 1 つは mutex が *released*(解放) できます。一度 mutex が獲得されれば、他のその mutex に対する獲得命令はその mutex が解放されるまで続行することができません。

<sup>86</sup> 私たちの実装では、各シリアルライザは関連付けられた mutex を持ります。手続 p を与えられた場合、シリアルライザは mutex を獲得する手続を返し、p を実行し、それから mutex を解放します。これがシリアルライザにより生成された手続の 1 つのみが一度に実行できることを保証します。これがまさに私たちが保証する必要のある、直列化の特性です。

```
(define (make-serializer)
  (let ((mutex (make-mutex)))
    (lambda (p)
      (define (serialized-p . args)
        (mutex 'acquire)
        (let ((val (apply p args)))
          (mutex 'release)
          val))
      serialized-p)))
```

The mutex is a mutable object (here we'll use a one-element list, which we'll refer to as a *cell*) that can hold the value true or false. When the value is false, the mutex is available to be acquired. When the value is true, the mutex is

---

<sup>86</sup> “mutex” という用語は *mutual exclusion*(相互排除) の省略形です。並行処理が安全に資源を共有することを可能にする仕組みの準備における一般的な問題は相互排除問題と呼ばれます。私達の mutex は *semaphore*(セマフォ) という仕組みの簡単な改良型です。(Exercise 3.47 参照)。これはアントホーフエン技術大学にて開発された “THE” Multiprogramming system(訳注: THE は究極のとかこれぞとか唯一の等の意味になる)にて導入され、大学のオランダ語でのイニシャルから名付けられました (Dijkstra 1968a)。acquire と release の命令は元々はオランダ語の単語 *passeren*(渡す) と *vrijgeven*(解放する) から P と V と呼ばれ、鉄道システムにて用いられた semaphores(信号装置) を参照しています。Dijkstra(ダイクストラ) の古典的解説 (Dijkstra 1968b) は明確に並行コントロールの問題を表した最も初期の 1 つであり、多様な並行問題をどのようにセマフォを用いて扱うかについて示しました。

unavailable, and any process that attempts to acquire the mutex must wait.

mutex はミュータブルなオブジェクト (ここで私達は 1 要素のリストを使用し、*cell*(セル) と参照します。) であり、true か false の値を保持します。値が false の時、mutex は獲得可能です。値が true の時、mutex は使用不可であり、この mutex を獲得しようと試みるプロセスは待たなければいけません。

Our mutex constructor `make-mutex` begins by initializing the cell contents to false. To acquire the mutex, we test the cell. If the mutex is available, we set the cell contents to true and proceed. Otherwise, we wait in a loop, attempting to acquire over and over again, until we find that the mutex is available.<sup>87</sup> To release the mutex, we set the cell contents to false.

私達の mutex コンストラクタ `make-mutex` はセルの中身を false に初期化することから始めます。mutex を獲得するためにはセルを確認します。もし mutex が使用可能であれば、セルの中身を true にして続行します。そうでなければループの中で待ち、mutex が使用可能になるまで何度も獲得を試みます。<sup>88</sup> mutex を解放するためにはセルの中身に false を設定します。

```
(define (make-mutex)
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell)
                 (the-mutex 'acquire))) ; retry
            ((eq? m 'release) (clear! cell))))
    the-mutex))
(define (clear! cell) (set-car! cell false))
```

`Test-and-set!` tests the cell and returns the result of the test. In addition, if the test was false, `test-and-set!` sets the cell contents to true before returning false. We can express this behavior as the following procedure:

---

<sup>87</sup>In most time-shared operating systems, processes that are blocked by a mutex do not waste time “busy-waiting” as above. Instead, the system schedules another process to run while the first is waiting, and the blocked process is awakened when the mutex becomes available.

<sup>88</sup>多くの時分割 OS では mutex でロックされるプロセスは上記のように “busy-waiting”(占有待ち) にて時間を無駄にはしません。その代わりにシステムは他のプロセスを最初のプロセスが待っている間に実行するようスキジュールし、ロックされたプロセスは mutex が使用可能になると起こされます。

`test-and-set!` はセルをテストし、テストの結果を返します。さらに、もしテストが`false`であれば `test-and-set!` はセルの中身に `false` を返す前に `true` を設定します。この振舞は以下の手続のように表現できます。

```
(define (test-and-set! cell)
  (if (car cell) true (begin (set-car! cell true) false)))
```

However, this implementation of `test-and-set!` does not suffice as it stands. There is a crucial subtlety here, which is the essential place where concurrency control enters the system: The `test-and-set!` operation must be performed *atomically*. That is, we must guarantee that, once a process has tested the cell and found it to be `false`, the cell contents will actually be set to `true` before any other process can test the cell. If we do not make this guarantee, then the mutex can fail in a way similar to the bank-account failure in Figure 3.29. (See Exercise 3.46.)

しかし、この `test-and-set!` の実装は現状では十分ではありません。致命的な機微がここに存在し、ここが並行性コントロールがシステムに入る本質的な場所です。`test-and-set!` 命令は *atomically*(不可分に、アトミックに) 実行されなければなりません。つまり、一度プロセスがセルをテストし `false` であると知ったならば、セルの中身が実際にセルをテストできるどの他のプロセスよりも先に `true` と設定されることを保証せねばなりません。もしこの保証をしなければ mutex は Figure 3.29 における銀行口座の失敗と似た失敗をします。(Exercise 3.46 参照)。

The actual implementation of `test-and-set!` depends on the details of how our system runs concurrent processes. For example, we might be executing concurrent processes on a sequential processor using a time-slicing mechanism that cycles through the processes, permitting each process to run for a short time before interrupting it and moving on to the next process. In that case, `test-and-set!` can work by disabling time slicing during the testing and setting.

`test-and-set!` の実際の実装は私達のシステムが並行プロセスをどのように実行するかの詳細に依存します。例えば私達は並行プロセスを逐次的なプロセッサ上に時分割のメカニズムを用いて複数のプロセスを循環させることで実行するかもしれません。各プロセスに少ない時間の間割り込みが発生するまで実行することを許し次のプロセスを開始します。このような場合には `test-and-set!` はテストと設定の間は時分割を停止することでうまく行きます。<sup>89</sup> Alternatively, multiprocessing computers provide instructions that

---

<sup>89</sup>In MIT Scheme for a single processor, which uses a time-slicing model, `test-and-`

support atomic operations directly in hardware.<sup>90</sup>

代替法として、マルチプロセスのコンピュータはアトミックな命令を直接ハードウェアにてサポートします。<sup>91</sup>

---

`set!` can be implemented as follows:

シングルプロセッサ向けの MIT Scheme は時分割モデルを使うので `test-and-set!` は以下の様に実装できます。

```
(define (test-and-set! cell)
  (without-interrupts
    (lambda ()
      (if (car cell)
          true
          (begin (set-car! cell true)
                 false)))))
```

`Without-interrupts` disables time-slicing interrupts while its procedure argument is being executed.

`without-interrupts` は時分割割り込みをその引数である手続が実行されている間、無効にします。

<sup>90</sup>There are many variants of such instructions—including test-and-set, test-and-clear, swap, compare-and-exchange, load-reserve, and store-conditional—whose design must be carefully matched to the machine’s processor-memory interface. One issue that arises here is to determine what happens if two processes attempt to acquire the same resource at exactly the same time by using such an instruction. This requires some mechanism for making a decision about which process gets control. Such a mechanism is called an *arbiter*. Arbiters usually boil down to some sort of hardware device. Unfortunately, it is possible to prove that one cannot physically construct a fair arbiter that works 100% of the time unless one allows the arbiter an arbitrarily long time to make its decision. The fundamental phenomenon here was originally observed by the fourteenth-century French philosopher Jean Buridan in his commentary on Aristotle’s *De caelo*. Buridan argued that a perfectly rational dog placed between two equally attractive sources of food will starve to death, because it is incapable of deciding which to go to first.

<sup>91</sup>そのような命には `test-and-set`, `test-and-clear`, `swap`, `compare-and-exchange`, `load-reserve`, `store-conditional` 等様々なものが存在し、その設計は注意深くマシンのプロセッサ-メモリ間インターフェイスに合わせなければいけません。ここで起こる1つの問題にはそのような命を用いて完全に同時に同じリソースを2つのプロセスが獲得しようと試みた場合に何が起こるかを決定することです。これはどのプロセスがコントロールを握るのかについて決定するための何らかの仕組みを要求します。そのような仕組みは *arbiter* (アービタ、調停者) と呼ばれます。アービタは通常ある種のハードウェアデバイスにまとめられます。残念なことに、アービタに対し自由裁量の長さの時間を決定を行うのに許さない限り 100% の時間を働く公平なアービタを構築することは物理的に

**Exercise 3.46:** Suppose that we implement `test-and-set!` using an ordinary procedure as shown in the text, without attempting to make the operation atomic. Draw a timing diagram like the one in Figure 3.29 to demonstrate how the mutex implementation can fail by allowing two processes to acquire the mutex at the same time.

`test-and-set!` をテキストに示される通常の手続を用い、命令をアトミックにする試み無しで実装すると仮定する。Figure 3.29 の様なタイミング図を描き、2つのプロセスが同時に mutex を獲得するのを許可した場合に mutex の実装がどのように失敗するのか説明せよ。

**Exercise 3.47:** A semaphore (of size  $n$ ) is a generalization of a mutex. Like a mutex, a semaphore supports acquire and release operations, but it is more general in that up to  $n$  processes can acquire it concurrently. Additional processes that attempt to acquire the semaphore must wait for release operations. Give implementations of semaphores

(サイズ  $n$  の) セマフォは mutex の一般化である。mutex のように、セマフォは acquire と release 命令をサポートするが、最大  $n$  プロセスまでが並行に獲得できることではより一般的である。セマフォを獲得しようとする追加のプロセスは解放命令を待たなければならない。セマフォの実装を以下の条件で行え。

a in terms of mutexes

mutex を用いる

b in terms of atomic `test-and-set!` operations.

アトミックな `test-and-set!` 命令を用いる

---

不可能であることが証明できます。ここでの根本的な現象は元々14世紀のフランス人哲学者 Jean Buridan(ジャンビュリダン)により Aristotle(アリストテレス)の *De caelo*(天体論)への注釈において観察されています。ビュリダンは2つの等しく魅力的な食事の情報源の間に置かれた完全に理性的な犬は飢えて死ぬと主張しました。最初にどちらに行くのか決めることができないのです。

## Deadlock

Now that we have seen how to implement serializers, we can see that account exchanging still has a problem, even with the `serialized-exchange` procedure above. Imagine that Peter attempts to exchange  $a_1$  with  $a_2$  while Paul concurrently attempts to exchange  $a_2$  with  $a_1$ . Suppose that Peter's process reaches the point where it has entered a serialized procedure protecting  $a_1$  and, just after that, Paul's process enters a serialized procedure protecting  $a_2$ . Now Peter cannot proceed (to enter a serialized procedure protecting  $a_2$ ) until Paul exits the serialized procedure protecting  $a_2$ . Similarly, Paul cannot proceed until Peter exits the serialized procedure protecting  $a_1$ . Each process is stalled forever, waiting for the other. This situation is called a *deadlock*. Deadlock is always a danger in systems that provide concurrent access to multiple shared resources.

シリアルライザをどのように実装するべきかについて学習したため、例え上記の `serialized-exchange` を用いても口座の交換が依然として問題を持つことを理解することができます。Peter が  $a_1$  と  $a_2$  を交換しようとすると、Paul が並行に  $a_2$  を  $a_1$  と交換しようと試みていると想像してみて下さい。Peter のプロセスが  $a_1$  を守る直列化された手続に入った時点に届いたとします。その後に、Paul のプロセスが  $a_2$  を守る直列化された手続に入りました。さて Peter は ( $a_2$  を守っている直列化された手続に入ること) を進めることは Paul が  $a_2$  を守る直列化された手続から抜けるまでできません。同様に、Paul もまた Peter が  $a_1$  を守る直列化された手続を抜けるまで進めることができません。この状況は *deadlock*(デッドロック) と呼ばれます。デッドロックは並行なアクセスを複数の共有リソースに対し提供するシステムでは常に存在する危険性です。

One way to avoid the deadlock in this situation is to give each account a unique identification number and rewrite `serialized-exchange` so that a process will always attempt to enter a procedure protecting the lowest-numbered account first. Although this method works well for the exchange problem, there are other situations that require more sophisticated deadlock-avoidance techniques, or where deadlock cannot be avoided at all. (See [Exercise 3.48](#) and [Exercise 3.49](#)).<sup>92</sup>

---

<sup>92</sup>The general technique for avoiding deadlock by numbering the shared resources and acquiring them in order is due to [Havender \(1968\)](#). Situations where deadlock cannot be avoided require *deadlock-recovery* methods, which entail having processes “back out” of the deadlocked state and try again. Deadlock-recovery mechanisms are widely used in database management systems, a topic that is treated in detail in [Gray](#)

この状況におけるデッドロックを防ぐ1つの方法は各口座に固有の識別番号を与え、`serialized-exchange` を書き換えることでプロセスが常に最も小さな番号の口座を守る手続を最初に入るよう試みるようにします。この方法は交換問題に対してうまく行きますが、より洗練されたデッドロック防止技術を必要とする他の状況が存在します。またはデッドロックが全く防げない状況も存在します。[\(Exercise 3.48 と Exercise 3.49を参照\)](#)<sup>93</sup>

**Exercise 3.48:** Explain in detail why the deadlock-avoidance method described above, (i.e., the accounts are numbered, and each process attempts to acquire the smaller-numbered account first) avoids deadlock in the exchange problem. Rewrite `serialized-exchange` to incorporate this idea. (You will also need to modify `make-account` so that each account is created with a number, which can be accessed by sending an appropriate message.)

なぜ上で説明されたデッドロック防止手法(即ち口座に番号を付け各プロセスが最も小さな番号の口座を最初に獲得する)が交換問題のデッドロックを防ぐのか詳細に説明せよ。`serialized-exchange`をこの考えを組込むように書き直せ。`(make-account` も変更する必要があり、そうすることで各口座が番号と共に作られ、その番号が適切なメッセージを送ることによりアクセスできるようにしなければならない。)

**Exercise 3.49:** Give a scenario where the deadlock-avoidance mechanism described above does not work. (Hint: In the exchange problem, each process knows in advance which accounts it will need to get access to. Consider a situation where a process must get access to some shared resources before it can know which additional shared resources it will require.)

上で説明されたデッドロック防止の仕組みがうまく行かない場合のシナリオを示せ。

---

and Reuter 1993.

<sup>93</sup>デッドロックを共有リソースに番号を付け、順に獲得する一般的なテクニックはHavender (1968)によります。デッドロックが防げない状況ではdeadlock-recovery(デッドロックリカバリ(復帰)) 手法を必要とし、それはプロセスにデッドロック状態の“back out”(取消)と再試行を引き起します。デッドロックリカバリの仕組みは広くデータベース管理システムにて使用され、Gray and Reuter 1993に詳細が取り上げられています。

## Concurrency, time, and communication

We've seen how programming concurrent systems requires controlling the ordering of events when different processes access shared state, and we've seen how to achieve this control through judicious use of serializers. But the problems of concurrency lie deeper than this, because, from a fundamental point of view, it's not always clear what is meant by "shared state."

並行システムのプログラミングが異なるプロセスが共有状態にアクセスする時にイベントの順序をコントロールすることをどうして必要とするかについて学びました。そして賢明なシリアルライザの使用を通してこのコントロールをどのように達成するかについても学びました。しかし根本的な視点から、常に“共有状態”が何を意味するのかが明らかでないために、並行性の問題はそれよりも深く位置します。

Mechanisms such as **test-and-set!** require processes to examine a global shared flag at arbitrary times. This is problematic and inefficient to implement in modern high-speed processors, where due to optimization techniques such as pipelining and cached memory, the contents of memory may not be in a consistent state at every instant. In contemporary multiprocessing systems, therefore, the serializer paradigm is being supplanted by new approaches to concurrency control.<sup>94</sup>

**test-and-set!** のような仕組みはプロセスに対し任意の時間にグローバルな共有フラグの試験を要求します。これは解決が難しく、現在の高速なCPUにおいて実装するのに非効率的です。パイプラインやキャッシュメモリの様な最適化の仕組みのためメモリの中身は各瞬間ににおいて静的な状態にはありません。現代のマルチプロセスシステムにおいては、従ってシリアルライザのパラダイムは並行性コントロールの新しい取り組みにより取って代わられてきています。<sup>95</sup>

---

<sup>94</sup>One such alternative to serialization is called *barrier synchronization*. The programmer permits concurrent processes to execute as they please, but establishes certain synchronization points ("barriers") through which no process can proceed until all the processes have reached the barrier. Modern processors provide machine instructions that permit programmers to establish synchronization points at places where consistency is required. The POWERPC, for example, includes for this purpose two instructions called SYNC and EIEIO (Enforced In-order Execution of Input/Output).

<sup>95</sup>そのような直列化の代替法の1つは**barrier synchronization**(バリア同期)と呼ばれます。プログラマは並行プロセスにそれらが気に入るよう実行することを許可します。しかしどのプロセスも全てのプロセスがバリアに着くまでは先に進むことができない

The problematic aspects of shared state also arise in large, distributed systems. For instance, imagine a distributed banking system where individual branch banks maintain local values for bank balances and periodically compare these with values maintained by other branches. In such a system the value of “the account balance” would be undetermined, except right after synchronization. If Peter deposits money in an account he holds jointly with Paul, when should we say that the account balance has changed---when the balance in the local branch changes, or not until after the synchronization? And if Paul accesses the account from a different branch, what are the reasonable constraints to place on the banking system such that the behavior is “correct”? The only thing that might matter for correctness is the behavior observed by Peter and Paul individually and the “state” of the account immediately after synchronization. Questions about the “real” account balance or the order of events between synchronizations may be irrelevant or meaningless.<sup>96</sup>

共有状態の問題となる側面は巨大な分散システムにおいても生じます。例として、分散銀行システムを想像して下さい。個別の銀行支店は銀行残高のローカル値を保持し繰り返しそれらを他の支店により保存されている値と比較します。そのようなシステムにおいては“口座残高”は同期直後を除いて不確定になるでしょう。もしPeterがお金をPaulと連結する口座に預け入れした時に、いつ口座残高が変更されたと言うべきか---地元の支店が残高を変更した時か、または同期の後までは言えないのか? そしてもしPaulが異なる支店から口座にアクセスした場合、振舞が“正しい”銀行システム上に設置する妥当な制約とは何か? 正確性に対し問題となるものはPeterとPaulが独立して観察する振舞と同期直後の口座の“状態”的みでしょう。“本当の”口座残高に関わる質問や同期の間のイベントの順は準用ではないか、意味がないでしょう。<sup>97</sup>

---

いいいくつかの同期点(“バリア”)を設置します。現代のプロセッサはプログラマに一貫性が要求される場所に同期点を設置することを可能にする機械語命令を提供します。例えばPOWERPCはこの目的のためSYNC(同期)とEIEIO(Enforced In-order Execution of Input/Output, I/Oの強制順序実行)と呼ばれる2つの命令を含んでいます。

<sup>96</sup>This may seem like a strange point of view, but there are systems that work this way. International charges to credit-card accounts, for example, are normally cleared on a per-country basis, and the charges made in different countries are periodically reconciled. Thus the account balance may be different in different countries.

<sup>97</sup>これはおかしな見方のように見えるかもしれません。しかしこのように動くシステムは存在します。例えばクレジットカードの口座への国際課金は通常国毎の拠点上で精算され異なる国での課金は繰り返し消し込みされます。従って口座残高は異なる国では異なります。

The basic phenomenon here is that synchronizing different processes, establishing shared state, or imposing an order on events requires communication among the processes. In essence, any notion of time in concurrency control must be intimately tied to communication.<sup>98</sup> It is intriguing that a similar connection between time and communication also arises in the Theory of Relativity, where the speed of light (the fastest signal that can be used to synchronize events) is a fundamental constant relating time and space. The complexities we encounter in dealing with time and state in our computational models may in fact mirror a fundamental complexity of the physical universe.

ここでの基本的な現象は異なるプロセスの同期、共有状態の設置、またはイベントの順を強いることはプロセス間通信を必要とします。本質的に、並行性コントロールにおける任意の時間の概念は緊密に通信に結びつけられねばなりません。<sup>99</sup>面白いことに似たような時間と通信の間の繋がりが相対論にも生じています。光速(イベントの同期に使用可能な最も高速な信号)は基本的に時間と空間に関連して一定です。私達の計算モデルの時間と状態を取り扱うために遭遇した複雑性は実際に物理的宇宙の根本的な複雑性を映しているのかもしれません。

## 3.5 Streams

We've gained a good understanding of assignment as a tool in modeling, as well as an appreciation of the complex problems that assignment raises. It is time to ask whether we could have gone about things in a different way, so as to avoid some of these problems. In this section, we explore an alternative approach to modeling state, based on data structures called *streams*. As we shall see, streams can mitigate some of the complexity of modeling state.

モデリングにおけるツールとしての代入について、また代入が生じる複雑な問題の認識についても良い理解を得ることができました。次は我々が行ってきたことを異なる方法で行えたのか、そうすることでこれらの問題を回避でき

---

<sup>98</sup>For distributed systems, this perspective was pursued by Lampert (1978), who showed how to use communication to establish “global clocks” that can be used to establish orderings on events in distributed systems.

<sup>99</sup>分散システムに対するこの視点はLampert (1978)により追求されました。彼は分散システムにおいてイベントの順序付けを成立させるのに使用できる“グローバルな時計”を設立するためにどのように通信を用いるかについて示しました。

たのかについて尋ねる番です。この節では状態をモデル化する代替となる取り組み方について、*streams*(ストリーム)と呼ばれるデータ構造を基にして探求します。私達が学ぶにつれて、ストリームは状態のモデル化の複雑性のいくつかを和らげることができます。

Let's step back and review where this complexity comes from. In an attempt to model real-world phenomena, we made some apparently reasonable decisions: We modeled real-world objects with local state by computational objects with local variables. We identified time variation in the real world with time variation in the computer. We implemented the time variation of the states of the model objects in the computer with assignments to the local variables of the model objects.

一旦戻って、この複雑性がどこから来たのか再検討してみましょう。実際の世界の現象をモデル化する試みにおいて、私達は幾らかの恐らく適切な決定をしました。私達は実際の世界のオブジェクトを局所状態を用いて、ローカル変数を持つ計算オブジェクトによってモデル化しました。私達はコンピュータ内の時間変化により実際の世界の時間変化を判断しました。私達はコンピュータ内のモデルオブジェクトの状態の時系列変化をモデルオブジェクトのローカル変数への代入を用いて実装しました。

Is there another approach? Can we avoid identifying time in the computer with time in the modeled world? Must we make the model change with time in order to model phenomena in a changing world? Think about the issue in terms of mathematical functions. We can describe the time-varying behavior of a quantity  $x$  as a function of time  $x(t)$ . If we concentrate on  $x$  instant by instant, we think of it as a changing quantity. Yet if we concentrate on the entire time history of values, we do not emphasize change--the function itself does not change.<sup>100</sup>

他に取り組み方があるでしょうか? コンピュータ内の時間をモデル化された世界の時間を用いて判断することを避けられるでしょうか? 変わり行く世界の事象をモデル化するためにモデルを時間と共に変化させなければならないのでしょうか? 問題を数学の関数を用いて考えましょう。数量  $x$  の時間的に変化する振舞を時間の関数  $x(t)$  として説明できます。もし瞬間毎に  $x$  に集中すれば変化する数量だと考えることができます。けれどももし値の歴史全体の時間

---

<sup>100</sup>Physicists sometimes adopt this view by introducing the “world lines” of particles as a device for reasoning about motion. We've also already mentioned (Section 2.2.3) that this is the natural way to think about signal-processing systems. We will explore applications of streams to signal processing in Section 3.5.3.

集中すれば私達は変化を重要視しません。関数それ自体は変化しません。<sup>101</sup>

If time is measured in discrete steps, then we can model a time function as a (possibly infinite) sequence. In this section, we will see how to model change in terms of sequences that represent the time histories of the systems being modeled. To accomplish this, we introduce new data structures called *streams*. From an abstract point of view, a stream is simply a sequence. However, we will find that the straightforward implementation of streams as lists (as in Section 2.2.1) doesn't fully reveal the power of stream processing. As an alternative, we introduce the technique of *delayed evaluation*, which enables us to represent very large (even infinite) sequences as streams.

もし時間が不連続なステップにより測られるのであれば、(無限に成りうる)列として時間関数をモデル化できます。この節では変化をモデル化されたシステムの時刻歴 (time history) を表す列を用いてどのように変化をモデル化するかについて学びます。これを達成するために、*streams*(ストリーム)と呼ばれる新しいデータ構造を導入します。抽象的な視点からはストリームは単に列です。しかし私達はストリームの (Section 2.2.1にあるような) リストによる簡単な実装はストリーム処理の力を完全に明かすことができないことを知るでしょう。代替法として、*delayed evaluation*(遅延評価)のテクニックを導入します。遅延評価は巨大な(例え無限でも)列をストリームして表現することを可能にします。

Stream processing lets us model systems that have state without ever using assignment or mutable data. This has important implications, both theoretical and practical, because we can build models that avoid the drawbacks inherent in introducing assignment. On the other hand, the stream framework raises difficulties of its own, and the question of which modeling technique leads to more modular and more easily maintained systems remains open.

ストリーム処理は状態を持つシステムを代入やミュータブルなデータを用いずにモデル化することを可能にします。これは重要な意味合いを倫理的、物理的両方で持ります。なぜなら代入の導入による固有の欠陥を防ぐモデルを構築できるためです。一方で、ストリームフレームワークはそれ自身の困難を持ちます。そしてどのモデリングテクニックがよりモジュラでより簡単にシステムを保守できるかの疑問が残ります。

<sup>101</sup>物理学者は時折粒子の “world lines”(世界線) を運動に関する推測のための手段として導入することでこの見方を受け入れます。私達もまた既に (Section 2.2.3) においてこれが信号処理システムについて考える自然な方法であると説明しました。Section 3.5.3にて信号処理に対するストリームの適用について探求します。

### 3.5.1 Streams Are Delayed Lists

As we saw in Section 2.2.3, sequences can serve as standard interfaces for combining program modules. We formulated powerful abstractions for manipulating sequences, such as `map`, `filter`, and `accumulate`, that capture a wide variety of operations in a manner that is both succinct and elegant.

Section 2.2.3で学んだように、列はプログラムモジュールを組み合わせるための標準的なインターフェイスの役割を果たすことができます。列を操作するための強力な抽象化を形式化しました。例えば `map`, `filter`, `accumulate` であり、簡潔であり、かつ洗練された作法にて広範囲の操作を獲得します。

Unfortunately, if we represent sequences as lists, this elegance is bought at the price of severe inefficiency with respect to both the time and space required by our computations. When we represent manipulations on sequences as transformations of lists, our programs must construct and copy data structures (which may be huge) at every step of a process.

残念なことに、列をリストとして表現するとこの洗練さは演算により必要とされる時間と記憶域に関する深刻な非効率性を犠牲にして得ることになります。列上の操作をリストの変形として表現した時、私達のプログラムは(大きくなりえる)データ構造を処理の各ステップにおいて構築とコピーをせねばなりません。

To see why this is true, let us compare two programs for computing the sum of all the prime numbers in an interval. The first program is written in standard iterative style:<sup>102</sup>

なぜこれが正しいのか知るために、ある区間の全ての素数の和を求めるための2つのプログラムを比較してみましょう。最初のプログラムは標準的な繰り返しのスタイルを用います。<sup>103</sup>

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count)
           (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
```

---

<sup>102</sup>Assume that we have a predicate `prime?` (e.g., as in Section 1.2.6) that tests for primality.

<sup>103</sup>素数性をテストする (Section 1.2.6のような) 述語 `prime?` を持っていると仮定します。

```
(iter a 0))
```

The second program performs the same computation using the sequence operations of [Section 2.2.3](#):

2つ目のプログラムは同じ演算を [Section 2.2.3](#) の列命令を用いて実行します。

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime?
                      (enumerate-interval a b))))
```

In carrying out the computation, the first program needs to store only the sum being accumulated. In contrast, the filter in the second program cannot do any testing until `enumerate-interval` has constructed a complete list of the numbers in the interval. The filter generates another list, which in turn is passed to `accumulate` before being collapsed to form a sum. Such large intermediate storage is not needed by the first program, which we can think of as enumerating the interval incrementally, adding each prime to the sum as it is generated.

演算の実行において、最初のプログラムは蓄積される合計のみを格納する必要があります。逆に、2つ目のプログラムのフィルタは `enumerate-interval` が区間の数の完全なリストを構築するまで一度もテストを行うことができません。フィルタは別のリストを生成し、合計を形成するため畳み込まれる前に順に `accumulate` に渡されます。そのような大きな中間ストレージは最初のプログラムでは必要ありません。最初のプログラムは区間を昇順に列挙し、各素数が生成されるにつれ合計に足していくと考えることができます。

The inefficiency in using lists becomes painfully apparent if we use the sequence paradigm to compute the second prime in the interval from 10,000 to 1,000,000 by evaluating the expression

リスト使用における非効率性は、以下の式を評価して 10,000 から 1,000,000 の区間にて 2つ目の素数を求めるのに列パラダイムを用いると、悲痛な程、明らかです。

```
(car (cdr (filter prime?
                     (enumerate-interval 10000 1000000))))
```

This expression does find the second prime, but the computational overhead is outrageous. We construct a list of almost a million integers, filter this list by testing each element for primality, and then ignore almost all of the result. In a more traditional programming style, we would interleave the enumeration and the filtering, and stop when we reached the second prime.

この式は2つ目の素数を確かに見つけました。しかし計算上のコストは酷過ぎます。ほとんど百万の整数のリストを構築し、このリストを各要素の素数性をテストすることで選別し、ほとんど全ての結果を無視します。より伝統的なプログラミングスタイルにおいては列挙とフィルタリングを交互に配置し、2つ目の素数を見つけたら停止します。

Streams are a clever idea that allows one to use sequence manipulations without incurring the costs of manipulating sequences as lists. With streams we can achieve the best of both worlds: We can formulate programs elegantly as sequence manipulations, while attaining the efficiency of incremental computation. The basic idea is to arrange to construct a stream only partially, and to pass the partial construction to the program that consumes the stream. If the consumer attempts to access a part of the stream that has not yet been constructed, the stream will automatically construct just enough more of itself to produce the required part, thus preserving the illusion that the entire stream exists. In other words, although we will write programs as if we were processing complete sequences, we design our stream implementation to automatically and transparently interleave the construction of the stream with its use.

ストリームは列をリストとして扱うコストを負担することなく列操作を用いることが可能な賢明な考えです。ストリームを用いると2つの世界の良い所取りができます。プログラムを列操作のように優雅に定式化できます。繰り返し演算の効率も獲得できます。基本的なアイデアはストリームを部分的にのみ構築する準備を行い、部分的な構築物をストリームを消費するプログラムに渡します。もし消費プログラムがまだ構築されていないストリームの部分にアクセスしようと試みた場合、ストリームは要求された部分を生成するために自動的にそれ自身の十分な追加を構築します。従ってストリーム全体が存在するという錯覚を維持することができます。言い替えれば、私達は完全な列を処理するようなプログラムを書きますが、私達のストリーム実装に自動的に、透過的にストリームの構築とその使用を相互配置するように設計します。

On the surface, streams are just lists with different names for the procedures that manipulate them. There is a constructor, `cons-stream`, and two selectors, `stream-car` and `stream-cdr`, which satisfy the constraints

表面上では、ストリームはそれを操作するための異なった名前を持つただのリストです。コンストラクタ `cons-stream` と以下の制約を満たす 2 つのセレクタ `stream-car` と `stream-cdr` が存在します。

```
(stream-car (cons-stream x y)) = x  
(stream-cdr (cons-stream x y)) = y
```

There is a distinguishable object, `the-empty-stream`, which cannot be the result of any `cons-stream` operation, and which can be identified with the predicate `stream-null?`.<sup>104</sup> Thus we can make and use streams, in just the same way as we can make and use lists, to represent aggregate data arranged in a sequence. In particular, we can build stream analogs of the list operations from Chapter 2, such as `list-ref`, `map`, and `for-each`:<sup>105</sup>

判別可能なオブジェクト `the-empty-stream` が存在し、これはどんな `cons-stream` 命令の結果にはなりえず、述語 `stream-null?` にて識別できます。<sup>106</sup> 従ってストリームを作成し、使用して、リストの作成と使用と同様に、準備された列のデータの集約を表現することができます。具体的には、Chapter 2 からストリーム用のリスト命令の類似手続、例えば `list-ref`, `map`, `for-each` を構築できます。<sup>107</sup>

```
(define (stream-ref s n)  
  (if (= n 0)  
      (stream-car s)
```

---

<sup>104</sup>In the MIT implementation, `the-empty-stream` is the same as the empty list '(), and `stream-null?` is the same as `null?`.

<sup>105</sup>This should bother you. The fact that we are defining such similar procedures for streams and lists indicates that we are missing some underlying abstraction. Unfortunately, in order to exploit this abstraction, we will need to exert finer control over the process of evaluation than we can at present. We will discuss this point further at the end of Section 3.5.4. In Section 4.2, we'll develop a framework that unifies lists and streams.

<sup>106</sup>MIT の実装では `the-empty-stream` は空のリスト'()'と同じで、`stream-null?` は `null?` と同じです。

<sup>107</sup>これはあなたを困惑させるでしょう。そのような似た手続をストリームとリストに定義するという事実は、私達がその根底にある抽象を見逃していることを示します。残念なことに、この抽象を利用するためには、現在可能なものよりより細かな評価過程に対するコントロールを行使する必要があります。この点についてはSection 3.5.4の終わりにより詳細に議論します。Section 4.2ではリストとストリームを統合するフレームワークを開発します。

```

(stream-ref (stream-cdr s) (- n 1)))
(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc (stream-cdr s)))))

(define (stream-for-each proc s)
  (if (stream-null? s)
      'done
      (begin (proc (stream-car s))
             (stream-for-each proc (stream-cdr s)))))


```

`Stream-for-each` is useful for viewing streams:

`stream-for-each` はストリームを見るのに便利です。

```

(define (display-stream s)
  (stream-for-each display-line s))
(define (display-line x) (newline) (display x))


```

To make the stream implementation automatically and transparently interleave the construction of a stream with its use, we will arrange for the `cdr` of a stream to be evaluated when it is accessed by the `stream-cdr` procedure rather than when the stream is constructed by `cons-stream`. This implementation choice is reminiscent of our discussion of rational numbers in [Section 2.1.2](#), where we saw that we can choose to implement rational numbers so that the reduction of numerator and denominator to lowest terms is performed either at construction time or at selection time. The two rational-number implementations produce the same data abstraction, but the choice has an effect on efficiency. There is a similar relationship between streams and ordinary lists. As a data abstraction, streams are the same as lists. The difference is the time at which the elements are evaluated. With ordinary lists, both the `car` and the `cdr` are evaluated at construction time. With streams, the `cdr` is evaluated at selection time.

ストリームの実装に自動的、かつ透過的にストリームの構築とその使用を相互配置させるためには、ストリームの `cdr` が、ストリームが `cons-stream` により構築された時でなく、`stream-cdr` 手続によりアクセスされた時に評価されるように手筈を整えます。この実装の選択は[Section 2.1.2](#)での分数の議論を思い出させます。その場合は分子と分母の最小の項への約分を構築時または選択時に実行されるよう実装を選択できることを学びました。2つの分数実装は

同じデータ抽象化を生成しますが、選択が効率に影響を与えました。似た関係がストリームと通常のリストの間にも存在します。データ抽象化としては、ストリームはリストと同じです。違いは要素が評価されるタイミングです。通常のリストでは `car` と `cdr` の両方は構築時に評価されます。ストリームでは `cdr` は選択時に評価されます。

Our implementation of streams will be based on a special form called `delay`. Evaluating (`delay <exp>`) does not evaluate the expression `<exp>`, but rather returns a so-called *delayed object*, which we can think of as a “promise” to evaluate `<exp>` at some future time. As a companion to `delay`, there is a procedure called `force` that takes a delayed object as argument and performs the evaluation—in effect, forcing the `delay` to fulfill its promise. We will see below how `delay` and `force` can be implemented, but first let us use these to construct streams.

私達のストリームの実装は `delay(遅延)` と呼ばれる特殊形式を基にします。`(delay <exp>)` の評価は式 `<exp>` を評価しません。しかしその代わりに所謂 *delayed object*(遅延オブジェクト)を返します。これはある将来の時点での `<exp>` を評価する“promise”(プロミス、約束)として考えることができます。`delay` の相方として `force(強いる)` と呼ばれる手続が存在し、遅延オブジェクトを引数として取り、評価を実行します。実際に `delay` にその約束を果たさせることを強要します。以下で `delay` と `force` がどのように実装できるかについて学びますが、最初にこれらを用いてストリームを構築しましょう。

`Cons-stream` is a special form defined so that

`cons-stream` は特殊形式で、

`(cons-stream (a) (b))`

is equivalent to

上が以下と同じになるよう設計されています。

`(cons (a) (delay (b)))`

What this means is that we will construct streams using pairs. However, rather than placing the value of the rest of the stream into the `cdr` of the pair we will put there a promise to compute the rest if it is ever requested. `Stream-car` and `stream-cdr` can now be defined as procedures:

この意味する所は、私達はペアを用いてストリームを構築します。しかし、ペアの `cdr` にストリームの残りの値を置くのではなく、そこにプロミスを置き要求された時点での残りを計算します。これで `stream-car` と `stream-cdr` が手続として定義できます。

```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```

`Stream-car` selects the `car` of the pair; `stream-cdr` selects the `cdr` of the pair and evaluates the delayed expression found there to obtain the rest of the stream.<sup>108</sup>

`stream-car` はペアの `car` を選択します。`stream-cdr` はペアの `cdr` を選択し、そこに見つかった遅延表現を評価し、ストリームの残りを得ます。<sup>109</sup>

## The stream implementation in action

To see how this implementation behaves, let us analyze the “outrageous” prime computation we saw above, reformulated in terms of streams:

この実装がどのように振る舞うのかを見るために、先に見た“法外な”素数演算をストリームを用いて再定式化したものを作成してみましょう。

```
(stream-car
  (stream-cdr
    (stream-filter prime?
      (stream-enumerate-interval
        10000 1000000))))
```

We will see that it does indeed work efficiently.

これが本当に効率的に働くことを見るでしょう。

We begin by calling `stream-enumerate-interval` with the arguments 10,000 and 1,000,000. `Stream-enumerate-interval` is the stream analog of `enumerate-interval` (Section 2.2.3):

---

<sup>108</sup>Although `stream-car` and `stream-cdr` can be defined as procedures, `cons-stream` must be a special form. If `cons-stream` were a procedure, then, according to our model of evaluation, evaluating `(cons-stream <a> <b>)` would automatically cause `<b>` to be evaluated, which is precisely what we do not want to happen. For the same reason, `delay` must be a special form, though `force` can be an ordinary procedure.

<sup>109</sup>`stream-car` と `stream-cdr` が手続として定義できるにも係わらず、`cons-stream` は特殊形式でなければなりません。もし `cons-stream` が手続であるのならば、私達の評価モデルに従い、`(cons-stream <a> <b>)` の評価は自動的に `<b>` の評価を起こします。これは明らかに私たちにとって起こって欲しくないことです。同じ理由から `delay` も特殊形式でなければなりません。しかし `force` は通常の手続になります。

`stream-enumerate-interval` を引数 10,000 と 1,000,000 と共に呼び出すことから始めます。`Stream-enumerate-interval` は `enumerate-interval` (Section 2.2.3) のストリーム版同等品です。

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
        low
        (stream-enumerate-interval (+ low 1) high))))
```

and thus the result returned by `stream-enumerate-interval`, formed by the `cons-stream`, is<sup>110</sup>

従って `stream-enumerate-interval` で返される結果は、`cons-stream` で形成された<sup>111</sup>

```
(cons 10000
      (delay (stream-enumerate-interval 10001 1000000)))
```

That is, `stream-enumerate-interval` returns a stream represented as a pair whose `car` is 10,000 and whose `cdr` is a promise to enumerate more of the interval if so requested. This stream is now filtered for primes, using the stream analog of the `filter` procedure (Section 2.2.3):

つまり `stream-enumerate-interval` はペアとして表現されたストリームを返し、その `car` は 10,000 で、その `cdr` はプロミスでありもし要求されれば区間のより多くを列挙します。このストリームはここでフィルタをかけ素数を残します。`filter` 手続 (Section 2.2.3) のストリーム版同等品を用います。

```
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
```

---

<sup>110</sup>The numbers shown here do not really appear in the delayed expression. What actually appears is the original expression, in an environment in which the variables are bound to the appropriate numbers. For example, `(+ low 1)` with `low` bound to 10,000 actually appears where 10001 is shown.

<sup>111</sup>ここで示されている数値は遅延オブジェクトの中には実際には現れません。実際に現れるのは元の式であり、環境の中で変数は適切な数値に束縛されています。例えば `low` が 10,000 に束縛されながら `(+ low 1)` が 10001 が表示されている場所に現れます。

```

(stream-filter
 pred
 (stream-cdr stream))))
(else (stream-filter pred (stream-cdr stream))))))

```

`Stream-filter` tests the `stream-car` of the stream (the `car` of the pair, which is 10,000). Since this is not prime, `stream-filter` examines the `stream-cdr` of its input stream. The call to `stream-cdr` forces evaluation of the delayed `stream-enumerate-interval`, which now returns

`stream-filter` はストリームの `stream-car`(ペアの `car` であり 10,000) をテストします。これは素数ではないので `stream-filter` は入力ストリームの `stream-cdr` を調査します。`stream-cdr` の呼出は遅延化された `stream-enumerate-interval` の評価を強制します。これは今、以下を返します。

```

(cons 10001
      (delay (stream-enumerate-interval 10002 1000000)))

```

`Stream-filter` now looks at the `stream-car` of this stream, 10,001, sees that this is not prime either, forces another `stream-cdr`, and so on, until `stream-enumerate-interval` yields the prime 10,007, whereupon `stream-filter`, according to its definition, returns

`stream-filter` は今このストリームの `stream-car` である 10,001 を見て、これもまた素数ではないことを確認し、もう一度 `stream-cdr` を強制します。これを `stream-enumerate-interval` が素数 10,007 を生じるまで繰り返し、すると直ぐに `stream-filter` はその定義に従い以下を返します。

```

(cons-stream (stream-car stream)
             (stream-filter pred (stream-cdr stream)))

```

which in this case is

これはこの場合以下のようになります。

```

(cons 10007
      (delay (stream-filter
              prime?
              (cons 10008
                    (delay (stream-enumerate-interval
                            10009
                            1000000))))))

```

This result is now passed to `stream-cdr` in our original expression. This forces the delayed `stream-filter`, which in turn keeps forcing the delayed `stream-enumerate-interval` until it finds the next prime, which is 10,009. Finally, the result passed to `stream-car` in our original expression is

これでこの結果は元の式の `stream-cdr` に渡されます。これにより遅延された `stream-filter` が強制され、それが順に遅延された `stream-enumerate-interval` を次の素数、10,009を見つけるまで強制します。最終的に、結果が私達の元の式の `stream-car` に渡された物が以下です。

```
(cons 10009
      (delay (stream-filter
                  prime?
                  (cons 10010
                        (delay (stream-enumerate-interval
                                    10011
                                    1000000))))))
```

`Stream-car` returns 10,009, and the computation is complete. Only as many integers were tested for primality as were necessary to find the second prime, and the interval was enumerated only as far as was necessary to feed the prime filter.

`stream-car` が 10,009 を返し計算が完了します。2つ目の素数を見つけるのに必要なだけの整数が素数性のテストを受け、区間は素数フィルタに入力するのに必要なだけ列挙されました。

In general, we can think of delayed evaluation as “demand-driven” programming, whereby each stage in the stream process is activated only enough to satisfy the next stage. What we have done is to decouple the actual order of events in the computation from the apparent structure of our procedures. We write procedures as if the streams existed “all at once” when, in reality, the computation is performed incrementally, as in traditional programming styles.

一般的に、遅延評価は“demand-driven”(要求駆動) プログラミングだと考えることができ、ストリーム処理の各ステージは次のステージを満たすのに十分な場合にのみ稼動されます。私達がここで行ったことは手続の見掛け上の構造から実際のイベントの順を分断することです。手続をストリームが“一度に揃って”存在するかのように書くが、実際には演算は漸増的に伝統的なプログラミングスタイルのように実行されます。

## Implementing delay and force

Although `delay` and `force` may seem like mysterious operations, their implementation is really quite straightforward. `Delay` must package an expression so that it can be evaluated later on demand, and we can accomplish this simply by treating the expression as the body of a procedure. `Delay` can be a special form such that

`delay` と `force` はミステリアスな命令に見えるかもしれません、それらの実装は本当にとても簡単です。`delay` は式を梱包して要求に応じて評価できるようにせねばなりません。私達はこれを手続のボディのように式を扱うことで簡単に達成できます。`delay` は以下のよう特殊形式です。

```
(delay <exp>)
```

is syntactic sugar for

これは以下の構文糖になります。

```
(lambda () <exp>)
```

`Force` simply calls the procedure (of no arguments) produced by `delay`, so we can implement `force` as a procedure:

`force` は単純に `delay` により生成された(引数無しの)手続を呼び出します。従って `force` は手続として実装可能です。

```
(define (force delayed-object) (delayed-object))
```

This implementation suffices for `delay` and `force` to work as advertised, but there is an important optimization that we can include. In many applications, we end up forcing the same delayed object many times. This can lead to serious inefficiency in recursive programs involving streams. (See [Exercise 3.57.](#)) The solution is to build delayed objects so that the first time they are forced, they store the value that is computed. Subsequent forcings will simply return the stored value without repeating the computation. In other words, we implement `delay` as a special-purpose memoized procedure similar to the one described in [Exercise 3.27](#). One way to accomplish this is to use the following procedure, which takes as argument a procedure (of no arguments) and returns a memoized version of the procedure. The first time the memoized procedure is run, it saves the computed result. On subsequent evaluations, it simply returns the result.

この実装は `delay` と `force` が広報通りに動く程度には十分です。しかし導入可能な重要な最適化が存在します。多くのアプリケーションにおいては同じ

遅延オブジェクトを何度も強制することになります。これがストリームを利用する再帰プログラムにおいて深刻な非効率の原因となります (Exercise 3.57 参照)。解決方法は遅延オブジェクトが初めて強制された時に計算された値を保存するように遅延オブジェクトを構築します。続く強制は格納された値の計算を繰り返さずに、単純に格納された値を返します。言い替えれば、`delay` を特別な目的のメモ化手続として Exercise 3.27 にて説明された物と同様に実装します。これを達成する 1 つの方法は以下の手続を用います。これは引数として (引数の無い) 手続を取りその手続のメモ化された版を返します。メモ化された手続が最初に実行される時、計算結果を格納します。以降の評価では単純に結果を返します。

```
(define (memo-proc proc)
  (let ((already-run? false) (result false))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result)
      result))))
```

`Delay` is then defined so that (`delay <exp>`) is equivalent to

`delay` はすると (`delay <exp>`) が以下と等価になるように定義されます。

```
(memo-proc (lambda () <exp>))
```

and `force` is as defined previously.<sup>112</sup>

そして `force` は以前の定義と同じです。<sup>113</sup>

---

<sup>112</sup>There are many possible implementations of streams other than the one described in this section. Delayed evaluation, which is the key to making streams practical, was inherent in Algol 60's *call-by-name* parameter-passing method. The use of this mechanism to implement streams was first described by Landin (1965). Delayed evaluation for streams was introduced into Lisp by Friedman and Wise (1976). In their implementation, `cons` always delays evaluating its arguments, so that lists automatically behave as streams. The memoizing optimization is also known as *call-by-need*. The Algol community would refer to our original delayed objects as *call-by-name thunks* and to the optimized versions as *call-by-need thunks*.

<sup>113</sup>この節で説明された物以外にも多くのストリームの実装が存在します。遅延評価はストリームを現実的にする鍵ですが、Algol 60 の *call-by-name*(コールバイネーム、名前による呼出) パラメタパッシング法固有の物でした。ストリームの実装にこの仕組みを

**Exercise 3.50:** Complete the following definition, which generalizes `stream-map` to allow procedures that take multiple arguments, analogous to `map` in Section 2.2.1, Footnote 12.

以下の定義を完成させよ。これは `stream-map` を複数の引数を取ることができるようにする Section 2.2.1 の `map`, Footnote 12 の同等品である

```
(define (stream-map proc . argstreams)
  (if (⟨??⟩) (car argstreams))
    the-empty-stream
  (⟨??⟩
    (apply proc (map ⟨??⟩ argstreams))
    (apply stream-map
      (cons proc (map ⟨??⟩ argstreams))))))
```

**Exercise 3.51:** In order to take a closer look at delayed evaluation, we will use the following procedure, which simply returns its argument after printing it:

遅延評価のより詳細を見るために、単純に引数を表示した後に引数を返すだけの以下の手続を使用する。

```
(define (show x)
  (display-line x)
  x)
```

What does the interpreter print in response to evaluating each expression in the following sequence?<sup>114</sup>

---

使用することは最初に Landin (1965) により説明されました。ストリームに対する遅延評価は Friedman and Wise (1976) により Lisp に導入されました。彼等の実装では `cons` は常にその引数の評価を遅延するので、リストは自動的にストリームとして振舞いました。メモ化最適化は *call-by-need* (必要による呼出) としても知られています。Alogol コミュニティは私達の元の遅延オブジェクトを *call-by-name thunks* (コールバイネームサンク) と呼び最適化された版を *call-by-need thunks* (コールバイニードサンク) と呼ぶでしょう。

<sup>114</sup>Exercises such as Exercise 3.51 and Exercise 3.52 are valuable for testing our understanding of how `delay` works. On the other hand, intermixing delayed evaluation with printing—and, even worse, with assignment—is extremely confusing, and instructors of courses on computer languages have traditionally tormented their stu-

インタプリタが以下の一連の式のそれぞれを評価した時に何を表示するだろうか?<sup>115</sup>

```
(define x
  (stream-map show
               (stream-enumerate-interval 0 10)))
(stream-ref x 5)
(stream-ref x 7)
```

**Exercise 3.52:** Consider the sequence of expressions  
以下の一連の式について考える。

```
(define sum 0)
(define (accum x) (set! sum (+ x sum)) sum)
(define seq
  (stream-map accum
               (stream-enumerate-interval 1 20)))
(define y (stream-filter even? seq))
(define z
  (stream-filter (lambda (x) (= (remainder x 5) 0))
                seq))
(stream-ref y 7)
(display-stream z)
```

---

dents with examination questions such as the ones in this section. Needless to say, writing programs that depend on such subtleties is odious programming style. Part of the power of stream processing is that it lets us ignore the order in which events actually happen in our programs. Unfortunately, this is precisely what we cannot afford to do in the presence of assignment, which forces us to be concerned with time and change.

<sup>115</sup>Exercise 3.51やExercise 3.52のような課題は `delay` がどのように働くかについての私達の理解を試すために価値有るもので。一方で、遅延評価を表示—そしてさらに悪いことに代入と—混ぜることは大きな混乱要因であり、コンピュータ言語の授業のインストラクタ達はこの節にあるような試験問題で学生達を苦しめてきました。言うまでもありませんが、そのような微妙さに依存するプログラムを書くことは醜悪なプログラミングスタイルです。ストリーム処理の力の一部は私達にイベントが実際にプログラムの中で起こる順について忘れさせてくれることです。残念なことにこれは明らかに代入が存在する場合にはできない事です。代入は私達に時間と変更に関して心配することを強いるのです。

What is the value of `sum` after each of the above expressions is evaluated? What is the printed response to evaluating the `stream-ref` and `display-stream` expressions? Would these responses differ if we had implemented `(delay <exp>)` simply as `(lambda () <exp>)` without using the optimization provided by `memo-proc`? Explain  
上記の各式が評価された後の `sum` の値はいくつか? 式 `stream-ref` と `display-stream` を評価した時表示される応答は何か? これらの応答はもし `(delay <exp>)` を単純に `(lambda () <exp>)` と実装し `memo-proc` により提供される最適化を使用しなかった場合に異なるだろうか? 説明せよ。

### 3.5.2 Infinite Streams

We have seen how to support the illusion of manipulating streams as complete entities even though, in actuality, we compute only as much of the stream as we need to access. We can exploit this technique to represent sequences efficiently as streams, even if the sequences are very long. What is more striking, we can use streams to represent sequences that are infinitely long. For instance, consider the following definition of the stream of positive integers:

実際にはアクセスに必要な分のストリームしか計算していないのにストリームを完全な要素の集合として扱うイリュージョンをどのようにサポートするのかについて学びました。このテクニックを利用して例え例が実際にはとても長くても列を効率的にストリームとして表現することができます。より印象的なことに、ストリームを無限に長い列を表現するために使用することができます。例として以下の正の整数のストリームの定義について考えてみましょう。

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))
(define integers (integers-starting-from 1))
```

This makes sense because `integers` will be a pair whose `car` is 1 and whose `cdr` is a promise to produce the integers beginning with 2. This is an infinitely long stream, but in any given time we can examine only a finite portion of it. Thus, our programs will never know that the entire infinite stream is not there.

`integers` が `car` が 1 で `cdr` が 2 で始まる整数を生成するプロミスになるためこれは理にかなっています。これは無限に長いストリームです。しかし任意の与えた時間にはその有限な一部しか検討することはできません。従って

私達のプログラムは無限のストリーム全体がそこにあることを知ることはできません。

Using `integers` we can define other infinite streams, such as the stream of integers that are not divisible by 7:

`integers` を用いて他の無限のストリームを定義できます。例えば 7 で割ることのできない整数のストリームです。

```
(define (divisible? x y) (= (remainder x y) 0))
(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7)))
    integers))
```

Then we can find integers not divisible by 7 simply by accessing elements of this stream:

すると 7 で割り切れない整数をこのストリームの要素にアクセスするだけで見つけることができます。

```
(stream-ref no-sevens 100)
117
```

In analogy with `integers`, we can define the infinite stream of Fibonacci numbers:

`integers` と同様に、フィボナッチ数の無限ストリームも定義できます。

```
(define (fibgen a b) (cons-stream a (fibgen b (+ a b))))
(define fibs (fibgen 0 1))
```

`Fibs` is a pair whose `car` is 0 and whose `cdr` is a promise to evaluate (`fibgen 1 1`). When we evaluate this delayed (`fibgen 1 1`), it will produce a pair whose `car` is 1 and whose `cdr` is a promise to evaluate (`fibgen 1 2`), and so on.

`fibs` はその `car` が 0 で、その `cdr` は (`fibgen 1 1`) を評価するプロミスであるペアです。この遅延化した (`fibgen 1 1`) を評価すると、`car` が 1 で `cdr` が (`fibgen 1 2`) を評価するプロミスであるペアを生成します。以下、その繰り返しです。

For a look at a more exciting infinite stream, we can generalize the `no-sevens` example to construct the infinite stream of prime numbers, using a method known as the *sieve of Eratosthenes*.<sup>116</sup> We start with the integers beginning with 2, which is the first prime. To get the rest of the primes, we start

---

<sup>116</sup>Eratosthenes, a third-century B.C. Alexandrian Greek philosopher, is famous for giving the first accurate estimate of the circumference of the Earth, which he computed

by filtering the multiples of 2 from the rest of the integers. This leaves a stream beginning with 3, which is the next prime. Now we filter the multiples of 3 from the rest of this stream. This leaves a stream beginning with 5, which is the next prime, and so on. In other words, we construct the primes by a sieving process, described as follows: To sieve a stream  $S$ , form a stream whose first element is the first element of  $S$  and the rest of which is obtained by filtering all multiples of the first element of  $S$  out of the rest of  $S$  and sieving the result. This process is readily described in terms of stream operations:

より刺激的な無限ストリームの調査のために、`no-sevens` の例を一般化し、素数の無限ストリームを *sieve of Eratosthenes*(エラトステネスの篩) として知られる手法を用いて構築します。<sup>117</sup> 私達は整数を最初の素数である 2 で始めます。残りの素数を得るために、整数の残りから 2 の倍数をフィルタリングすることから始めます。これは 3 で始まるストリームを残し、3 は次の素数です。ここで 3 の倍数をこのストリームの残りからフィルタリングします。これは 5 で始まるストリームを残し、5 は次の素数です。以下これを繰り返します。言い換えれば、素数を次の様に説明する篩にかける処理により構築します。まずストリーム  $S$  に篩をかけるために、最初の要素が  $S$  の最初の要素であり、残りは  $S$  の残りから  $S$  の最初の要素の倍数をフィルタリングすることで得られるストリームを形成します。そして結果をさらに篩にかけます。この処理は容易にストリーム命令を用いて記述できます。

```
(define (sieve stream)
  (cons-stream
    (stream-car stream)
    (sieve (stream-filter
      (lambda (x)
        (not (divisible? x (stream-car stream)))))))
```

---

by observing shadows cast at noon on the day of the summer solstice. Eratosthenes's sieve method, although ancient, has formed the basis for special-purpose hardware "sieves" that, until recently, were the most powerful tools in existence for locating large primes. Since the 70s, however, these methods have been superseded by outgrowths of the probabilistic techniques discussed in Section 1.2.6.

<sup>117</sup> エラトステネスは紀元前 3 世紀のアレキサンドリア学派のギリシャ人学者で、地球の外周を最初に正しく推測したとして有名です。彼は夏至の日の正午の影を観察することでこれを求めました。エラトステネスの篩は古典ですが、特殊用途のハードウェア“篩”的基礎をなしており、最近まで巨大な素数を突き止める最も強力なツールでした。しかし 70 年代からこれらの手法は Section 1.2.6 で議論された確率的な技術の成長により取って代わられました。

```
(stream-cdr stream))))  
(define primes (sieve (integers-starting-from 2)))
```

Now to find a particular prime we need only ask for it:

これで特定の素数を見つけるのには以下のように尋ねるだけです。

```
(stream-ref primes 50)  
233
```

It is interesting to contemplate the signal-processing system set up by `sieve`, shown in the “Henderson diagram” in Figure 3.31.<sup>118</sup> The input stream feeds into an “unconser” that separates the first element of the stream from the rest of the stream. The first element is used to construct a divisibility filter, through which the rest is passed, and the output of the filter is fed to another sieve box. Then the original first element is `consed` onto the output of the internal sieve to form the output stream. Thus, not only is the stream infinite, but the signal processor is also infinite, because the sieve contains a sieve within it.

Figure 3.31の“ヘンダーソン図”に示されるように `sieve` により設定された信号処理システムを熟考することは面白いです。<sup>119</sup> 入力ストリームは“unconser”に流し込まれ、ストリームの最初の要素をストリームの残りから分離します。最初の要素は可分性フィルタを構築するのに用いられ、残りはそれに渡され通ります。フィルタの出力はもう1つの篩の箱に流し込まれます。次に元の最初の用途は内側の篩の出力上に `cons` され出力ストリームを形成します。従ってストリームのみが無限ではなく、信号処理器もまた無限です。なぜなら篩がその中に篩を持っているからです。

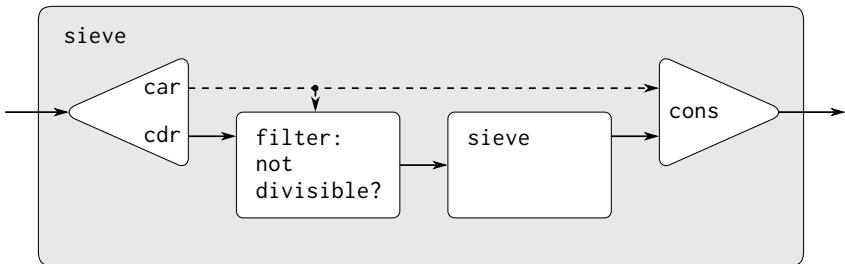
## Defining streams implicitly

The `integers` and `fibs` streams above were defined by specifying “generating” procedures that explicitly compute the stream elements one by one. An

---

<sup>118</sup>We have named these figures after Peter Henderson, who was the first person to show us diagrams of this sort as a way of thinking about stream processing. Each solid line represents a stream of values being transmitted. The dashed line from the `car` to the `cons` and the `filter` indicates that this is a single value rather than a stream.

<sup>119</sup>私達はこれらの図を Peter Henderson にちなんで名付けました。彼はこの種の図をストリーム処理について考える方法として示した最初の人物です。各実線は送信される値のストリームを表しています。`car` から `cons` と `filter` への点線はこれがストリームではなく単一の値であることを示します。



**Figure 3.31:** The prime sieve viewed as a signal-processing system.

alternative way to specify streams is to take advantage of delayed evaluation to define streams implicitly. For example, the following expression defines the stream `ones` to be an infinite stream of ones:

上記の `integers` と `fibs` のストリームは明示的にストリーム要素を 1 つづつ計算する“生成”手続を指定することにより定義されました。ストリームを指定する代替法として遅延評価の利点を用いて暗黙的にストリームを定義することが上げられます。例えば以下の式はストリーム `ones` を 1 の無限ストリームとして定義します。

```
(define ones (cons-stream 1 ones))
```

This works much like the definition of a recursive procedure: `ones` is a pair whose `car` is 1 and whose `cdr` is a promise to evaluate `ones`. Evaluating the `cdr` gives us again a 1 and a promise to evaluate `ones`, and so on.

これは再帰手続の定義そっくりに動きます。`ones` はペアでその `car` は 1 でその `cdr` は `ones` を評価するプロミスです。`cdr` の評価は再び 1 と `ones` を評価するプロミスを与えます。以下、繰り返しです。

We can do more interesting things by manipulating streams with operations such as `add-streams`, which produces the elementwise sum of two given streams.<sup>120</sup>

より面白いこととしてストリームを `add-streams` のような命令で操作することができます。`add-streams` は 2 つの与えられたストリームのエレメント同

---

<sup>120</sup>This uses the generalized version of `stream-map` from Exercise 3.50.

士の和を生成します。

```
(define (add-streams s1 s2) (stream-map + s1 s2))
```

Now we can define the integers as follows:

これで整数を以下のように定義できます。

```
(define integers
  (cons-stream 1 (add-streams ones integers)))
```

This defines **integers** to be a stream whose first element is 1 and the rest of which is the sum of **ones** and **integers**. Thus, the second element of **integers** is 1 plus the first element of **integers**, or 2; the third element of **integers** is 1 plus the second element of **integers**, or 3; and so on. This definition works because, at any point, enough of the **integers** stream has been generated so that we can feed it back into the definition to produce the next integer.

これは **integers** が最初の要素は 1 で残りは **ones** と **integers** の和になります。従って **integers** の 2 つ目の要素は 1 足す **integers** の最初の要素、つまり 2 になります。**integers** の 3 つ目の要素は 1 足す **integers** の 2 つ目の要素、つまり 3 です。以下繰り返します。この定義は任意の時点で十分な **integers** ストリームが生成されているので次の整数を生成するために定義にフィードバックすることができるためうまく行くのです。

We can define the Fibonacci numbers in the same style:

フィボナッチ数も同じスタイルで定義できます。

```
(define fibs
  (cons-stream
    0
    (cons-stream 1 (add-streams (stream-cdr fibs) fibs))))
```

This definition says that **fibs** is a stream beginning with 0 and 1, such that the rest of the stream can be generated by adding **fibs** to itself shifted by one place:

この定義は **fibs** は 0 と 1 で始まるストリームであり残りのストリームは **fibs** を自身に 1 つずらして足することで生成することができると述べています。

1	1	2	3	5	8	13	21	...	=	(stream-cdr fibs)		
0	1	1	2	3	5	8	13	...	=	fibs		
0	1	1	2	3	5	8	13	21	34	...	=	fibs

`Scale-stream` is another useful procedure in formulating such stream definitions. This multiplies each item in a stream by a given constant:

`scale-stream` はまた別の、そのようなストリーム定義を形成するのに便利な手続です。これはストリームの各要素に与えられた定数を掛けます。

```
(define (scale-stream stream factor)
  (stream-map (lambda (x) (* x factor))
              stream))
```

For example,

例として

```
(define double (cons-stream 1 (scale-stream double 2)))
```

produces the stream of powers of 2: 1, 2, 4, 8, 16, 32, ....

は 2 の幕乗のストリームを生成します: 1, 2, 4, 8, 16, 32, ...

An alternate definition of the stream of primes can be given by starting with the integers and filtering them by testing for primality. We will need the first prime, 2, to get started:

素数ストリームの代替定義は整数で始まり、それらの素数性をテストすることでフィルタリングすることでも与えられます。最初の素数、2 を開始に必要とします。

```
(define primes
  (cons-stream
    2
    (stream-filter prime? (integers-starting-from 3))))
```

This definition is not so straightforward as it appears, because we will test whether a number  $n$  is prime by checking whether  $n$  is divisible by a prime (not by just any integer) less than or equal to  $\sqrt{n}$ :

この定義はあまり見かけほど簡単ではありません。 $n$  が (任意の整数でなく)  $\sqrt{n}$  以下の素数で割り切れるかどうかをチェックすることにより数  $n$  が素数であるかを決めるためです。

```
(define (prime? n)
  (define (iter ps)
    (cond ((> (square (stream-car ps)) n) true)
          ((divisible? n (stream-car ps)) false)
          (else (iter (stream-cdr ps))))))
```

```
(iter primes))
```

This is a recursive definition, since `primes` is defined in terms of the `prime?` predicate, which itself uses the `primes` stream. The reason this procedure works is that, at any point, enough of the `primes` stream has been generated to test the primality of the numbers we need to check next. That is, for every  $n$  we test for primality, either  $n$  is not prime (in which case there is a prime already generated that divides it) or  $n$  is prime (in which case there is a prime already generated—i.e., a prime less than  $n$ —that is greater than  $\sqrt{n}$ ).<sup>121</sup>

これは再帰定義であり、`primes` が `primes` を用いる `prime?` 述語を用いて定義されています。この手続がうまく行く訳は、任意の時点で、十分な `primes` ストリームが生成されており、次にチェックするのに必要な数の素数性をテストできるからです。全ての  $n$  に対して素数性をテストします。例え  $n$  が素数でなくとも（この場合、それを割り切れる素数が既に生成されています。）、例え  $n$  が素数（この場合、素数が既に生成されています—言い換えれば、 $\sqrt{n}$  より大きく  $n$  未満の素数）であってもです。<sup>122</sup>

**Exercise 3.53:** Without running the program, describe the elements of the stream defined by

プログラムを実行すること無しに以下により定義されたストリームの要素について説明せよ。

```
(define s (cons-stream 1 (add-streams s s)))
```

---

<sup>121</sup>This last point is very subtle and relies on the fact that  $p_{n+1} \leq p_n^2$ . (Here,  $p_k$  denotes the  $k^{\text{th}}$  prime.) Estimates such as these are very difficult to establish. The ancient proof by Euclid that there are an infinite number of primes shows that  $p_{n+1} \leq p_1 p_2 \dots p_n + 1$ , and no substantially better result was proved until 1851, when the Russian mathematician P. L. Chebyshev established that  $p_{n+1} \leq 2p_n$  for all  $n$ . This result, originally conjectured in 1845, is known as *Bertrand's hypothesis*. A proof can be found in section 22.3 of [Hardy and Wright 1960](#).

<sup>122</sup>この最後の点はとても微妙で  $p_{n+1} \leq p_n^2$  という事実に依存しています。（ここで  $p_k$  は  $k$  番目の素数を示します）。このような予測を立証するのはとても難しいです。ユーハードによる太古の証明による、ある素数が無限に存在することが  $p_{n+1} \leq p_1 p_2 \dots p_n + 1$  を示しています。そして実質的にはより良い結果が証明されることには 1851 年までありませんでした。この年、ロシア人の数学者 P. L. Chebyshev(パフヌティ・チエビシェフ) は全ての  $n$  に対し  $p_{n+1} \leq 2p_n$  であることを証明しました。最初に 1845 年に予想されたこの結果は *Bertrand's hypothesis*(ベルトランの仮説) として知られています。証明は [Hardy and Wright 1960](#) の節 22.3 に見つかります。

**Exercise 3.54:** Define a procedure `mul-streams`, analogous to `add-streams`, that produces the elementwise product of its two input streams. Use this together with the stream of `integers` to complete the following definition of the stream whose  $n^{\text{th}}$  element (counting from 0) is  $n + 1$  factorial:

`add-streams` と類似の手続 `mul-streams` を定義せよ。これは 2 つの入力ストリームの要素同士の積を生成する。これを `integers` ストリームと共に用いて以下のストリームの定義を完成させよ。これの  $n$  番目の要素 (0 で開始) は  $n + 1$  の階乗である。

```
(define factorials (cons-stream 1 (mul-streams (λ() (λ())))))
```

**Exercise 3.55:** Define a procedure `partial-sums` that takes as argument a stream  $S$  and returns the stream whose elements are  $S_0, S_0 + S_1, S_0 + S_1 + S_2, \dots$ . For example, `(partial-sums integers)` should be the stream  $1, 3, 6, 10, 15, \dots$ .

手続 `partial-sums` を定義せよ。これはストリーム  $S$  を引数として取り、要素が  $S_0, S_0 + S_1, S_0 + S_1 + S_2, \dots$  であるストリームを返す。例えば `(partial-sums integers)` は  $1, 3, 6, 10, 15, \dots$  のストリームにならねばならない。

**Exercise 3.56:** A famous problem, first raised by R. Hamming, is to enumerate, in ascending order with no repetitions, all positive integers with no prime factors other than 2, 3, or 5. One obvious way to do this is to simply test each integer in turn to see whether it has any factors other than 2, 3, and 5. But this is very inefficient, since, as the integers get larger, fewer and fewer of them fit the requirement. As an alternative, let us call the required stream of numbers `S` and notice the following facts about it.

R. Hamming(リチャードハミング)により取り上げられた有名な問題に、昇順に、重複無く、2, 3, 5 以外の素因数を持たない正の整数を列挙せよというものがある。これを行う 1 つの明らかな方法は単純に各整数を順に 2, 3, 5 以外の素因数を持つかどうかテストする方法です。しかしこれはとても非効率です。整数が大きくなる程に要求に合う数はより少なくなるためです。代替法として、要求された数のストリームを `s` と呼び、以下の事実について注目してみましょう。

- S begins with 1.  
S は 1 で始まる。
- The elements of (scale-stream S 2) are also elements of S.  
(scale-stream S 2) の要素もまた S の要素である。
- The same is true for (scale-stream S 3) and (scale-stream 5 S).  
同じことが (scale-stream S 3) と (scale-stream 5 S) に  
対しても真である。
- These are all the elements of S.  
これらは全て S の要素である。

Now all we have to do is combine elements from these sources.  
For this we define a procedure `merge` that combines two ordered streams into one ordered result stream, eliminating repetitions:

さて私達が行わなければならないこと全てはこれらの情報から要素を結合することである。このために 2 つの順序有りストリームを重複を省き 1 つの順序付けられた結果のストリームに結合する手続 `merge` を定義する。

```
(define (merge s1 s2)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
          (let ((s1car (stream-car s1))
                (s2car (stream-car s2)))
            (cond ((< s1car s2car)
                  (cons-stream
                    s1car
                    (merge (stream-cdr s1) s2)))
                  ((> s1car s2car)
                    (cons-stream
                      s2car
                      (merge s1 (stream-cdr s2))))
                  (else
                    (cons-stream
```

```

s1car
(merge (stream-cdr s1)
        (stream-cdr s2))))))))

```

Then the required stream may be constructed with `merge`, as follows:

次に要求されたストリームが `merge` を用いて以下のように構築されるだろう。

```
(define S (cons-stream 1 (merge ⟨??⟩ ⟨??⟩)))
```

Fill in the missing expressions in the places marked ⟨??⟩ above.

上で ⟨??⟩ とマークされた箇所の欠けた式を埋めよ。

**Exercise 3.57:** How many additions are performed when we compute the  $n^{\text{th}}$  Fibonacci number using the definition of `fibs` based on the `add-streams` procedure? Show that the number of additions would be exponentially greater if we had implemented `(delay <exp>)` simply as `(lambda () <exp>)`, without using the optimization provided by the `memo-proc` procedure described in [Section 3.5.1](#).<sup>123</sup>

$n$  番目のフィボナッチ数を `add-streams` 手続を基にした `fibs` の定義を用いて計算した場合に加算は何回実行されるか? 加算回数が `(delay <exp>)` を単純に [Section 3.5.1](#)で説明した `memo-proc` 手続により提供される最適化を用いずに、`(lambda () <exp>)` として実装した場合に指数関数的に増加することを示せ。<sup>124</sup>

**Exercise 3.58:** Give an interpretation of the stream computed by the following procedure:

以下の手続により計算されるストリームの解説を与える。

---

<sup>123</sup>This exercise shows how call-by-need is closely related to ordinary memoization as described in [Exercise 3.27](#). In that exercise, we used assignment to explicitly construct a local table. Our call-by-need stream optimization effectively constructs such a table automatically, storing values in the previously forced parts of the stream.

<sup>124</sup>この課題は call-by-need が [Exercise 3.27](#)で説明された通常のメモ化に密接に関係していることを示します。その課題では代入を明示的にローカルの表の構築に用いました。私達の call-by-need ストリームの最適化は効果的にそのようなテーブルを自動的に構築し、ストリームの以前に強制された部分の値を格納します。

```
(define (expand num den radix)
  (cons-stream
    (quotient (* num radix) den)
    (expand (remainder (* num radix) den) den radix))))
```

(Quotient is a primitive that returns the integer quotient of two integers.) What are the successive elements produced by (expand 1 7 10)? What is produced by (expand 3 8 10)?

(quotient はプリミティブであり、2つの整数の、整数の商を返す)。(expand 1 7 10) により生成される一連の要素は何か? (expand 3 8 10) では何が生成されるか?

**Exercise 3.59:** In Section 2.5.3 we saw how to implement a polynomial arithmetic system representing polynomials as lists of terms. In a similar way, we can work with *power series*, such as

Section 2.5.3にて多項式を項のリストとして表現する多項式数値演算システムをどのように実装するかについて学んだ。同様な方法で以下のような *power series*(べき級数)についても扱うことができる。

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2} + \frac{x^3}{3 \cdot 2} + \frac{x^4}{4 \cdot 3 \cdot 2} + \dots, \\ \cos x &= 1 - \frac{x^2}{2} + \frac{x^4}{4 \cdot 3 \cdot 2} - \dots, \\ \sin x &= x - \frac{x^3}{3 \cdot 2} + \frac{x^5}{5 \cdot 4 \cdot 3 \cdot 2} - \dots \end{aligned}$$

represented as infinite streams. We will represent the series  $a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$  as the stream whose elements are the coefficients  $a_0, a_1, a_2, a_3, \dots$ .

これらは無限ストリームとして表現されている。数列  $a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$  を要素が係数  $a_0, a_1, a_2, a_3, \dots$  のストリームとして表すこととする。

- a The integral of the series  $a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$  is the series

級数  $a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$  の積分は次の級数になる。

$$c + a_0x + \frac{1}{2}a_1x^2 + \frac{1}{3}a_2x^3 + \frac{1}{4}a_3x^4 + \dots,$$

where  $c$  is any constant. Define a procedure `integrate-series` that takes as input a stream  $a_0, a_1, a_2, \dots$  representing a power series and returns the stream  $a_0, \frac{1}{2}a_1, \frac{1}{3}a_2, \dots$  of coefficients of the non-constant terms of the integral of the series. (Since the result has no constant term, it doesn't represent a power series; when we use `integrate-series`, we will `cons` on the appropriate constant.)

ここで  $c$  は任意の定数である。累級数を表すストリーム  $a_0, a_1, a_2, \dots$  を入力として取り、その級数の積分の非定数項の係数のストリーム  $a_0, \frac{1}{2}a_1, \frac{1}{3}a_2, \dots$  を返す手続 `integrate-series` を定義せよ。(結果が定数項を持たないため、それは累級数では無い。`integrate-series` を使う時、後で適切な定数を `cons` する。)

- b The function  $x \mapsto e^x$  is its own derivative. This implies that  $e^x$  and the integral of  $e^x$  are the same series, except for the constant term, which is  $e^0 = 1$ . Accordingly, we can generate the series for  $e^x$  as

関数  $x \mapsto e^x$  はそれ自身導関数である。これは  $e^x$  と  $e^x$  の不定積分が定数項を除いて同じ級数になることを暗示する。定数項は  $e^0 = 1$  である。結果的に、 $e^x$  の級数を次のように生成できる。

```
(define exp-series
  (cons-stream 1 (integrate-series exp-series)))
```

Show how to generate the series for sine and cosine, starting from the facts that the derivative of sine is cosine and the derivative of cosine is the negative of sine:

$\sin$  と  $\cos$  の級数をどのように生成するか示せ。 $\sin$  の導関数が  $\cos$  であり、 $\cos$  の導関数が負の  $\sin$  であることから始めよ。

```
(define cosine-series (cons-stream 1 (?))
  (define sine-series (cons-stream 0 (?))))
```

**Exercise 3.60:** With power series represented as streams of coefficients as in [Exercise 3.59](#), adding series is implemented by `add-`

**streams.** Complete the definition of the following procedure for multiplying series:

Exercise 3.59における係数ストリームとして表現された冪級数を用いて、級数の加算は `add-streams` により実装される。級数を乗算するための以下の手続の定義を完成させよ。

```
(define (mul-series s1 s2)
  (cons-stream ?? (add-streams ?? ??)))
```

You can test your procedure by verifying that  $\sin^2 x + \cos^2 x = 1$ , using the series from Exercise 3.59.

手続ができたらExercise 3.59の級数を用いて  $\sin^2 x + \cos^2 x = 1$  を確認せよ。

**Exercise 3.61:** Let  $S$  be a power series (Exercise 3.59) whose constant term is 1. Suppose we want to find the power series  $1/S$ , that is, the series  $X$  such that  $SX = 1$ . Write  $S = 1 + S_R$  where  $S_R$  is the part of  $S$  after the constant term. Then we can solve for  $X$  as follows:

$S$  が定数項が 1 の冪級数 (Exercise 3.59) であるとする。冪級数  $1/S$  を見つけたいとする。つまり  $SX = 1$  となるような級数  $X$  である。 $S_R$  が  $S$  の定数項の後の部分である場合に  $S = 1 + S_R$  を書け。そうすれば  $X$  を以下のようにして求めることができる。

$$\begin{aligned} S \cdot X &= 1, \\ (1 + S_R) \cdot X &= 1, \\ X + S_R \cdot X &= 1, \\ X &= 1 - S_R \cdot X. \end{aligned}$$

In other words,  $X$  is the power series whose constant term is 1 and whose higher-order terms are given by the negative of  $S_R$  times  $X$ . Use this idea to write a procedure `invert-unit-series` that computes  $1/S$  for a power series  $S$  with constant term 1. You will need to use `mul-series` from Exercise 3.60.

言い換えれば、 $X$  は定数項が 1 であり高次項が負の  $S_R$  と  $X$  の積により与えられる冪級数である。この考えを用いて定数項 1 を持つ冪級数  $S$  に対する  $1/S$  を求める手続 `invert-unit-series` を書け。Exercise 3.60の `mul-series` を用いる必要がある。

**Exercise 3.62:** Use the results of Exercise 3.60 and Exercise 3.61 to define a procedure `div-series` that divides two power series. `Div-series` should work for any two series, provided that the denominator series begins with a nonzero constant term. (If the denominator has a zero constant term, then `div-series` should signal an error.) Show how to use `div-series` together with the result of Exercise 3.59 to generate the power series for tangent.

Exercise 3.60とExercise 3.61の結果を用いて 2 つの幂級数を割る手続 `div-series` を定義せよ。`div-series` は任意の 2 つの級数に対して利用できねばならず分母の級数は非ゼロな定数項で始まらねばならない。(もし分母がゼロの定数項を持つならば `div-series` はエラーを発すこと)。`div-series` を Exercise 3.59 の結果と一緒にどのように用いて  $\tan$  の幂級数を生成するか示せ。

### 3.5.3 Exploiting the Stream Paradigm

Streams with delayed evaluation can be a powerful modeling tool, providing many of the benefits of local state and assignment. Moreover, they avoid some of the theoretical tangles that accompany the introduction of assignment into a programming language.

遅延評価を伴なうストリームは強力なモデリングツールにすることができ、局所状態と代入の利点の多くを提供する。さらにプログラミング言語への代入の導入に伴う、いくつかの理論的な混乱を防ぎます。

The stream approach can be illuminating because it allows us to build systems with different module boundaries than systems organized around assignment to state variables. For example, we can think of an entire time series (or signal) as a focus of interest, rather than the values of the state variables at individual moments. This makes it convenient to combine and compare components of state from different moments.

ストリームのアプローチは私達に、状態変数への代入の周りに体系化されたシステムよりも、異なるモジュール境界を伴なうシステムを構築することを可能にするため、啓発的です。例えば私達は個別の瞬間における状態変数の値としてではなく、時系列(または信号)全体を興味の中心として考えることができます。このことが異なる瞬間の状態のコンポーネントの比較と接続を行うのにより便利にします。

## Formulating iterations as stream processes

In Section 1.2.1, we introduced iterative processes, which proceed by updating state variables. We know now that we can represent state as a “timeless” stream of values rather than as a set of variables to be updated. Let’s adopt this perspective in revisiting the square-root procedure from Section 1.1.7. Recall that the idea is to generate a sequence of better and better guesses for the square root of  $x$  by applying over and over again the procedure that improves guesses:

Section 1.2.1において、反復プロセスを紹介しました。これは状態変数を更新することで進行されます。私達は今、状態を更新される変数の集合としてではなく、“永遠”的の値のストリームとして表すことができます。Section 1.1.7の平方根手続への再訪問にこの視点を導入しましょう。考え方方は推測値を改善する手続を何度も適用することで  $x$  の平方根の推測値をより良い値の列を生成するということを思い出して下さい。

```
(define (sqrt-improve guess x)
  (average guess (/ x guess)))
```

In our original `sqrt` procedure, we made these guesses be the successive values of a state variable. Instead we can generate the infinite stream of guesses, starting with an initial guess of 1:<sup>125</sup>

私達の元の `sqrt` 手続では、これらの推測値を状態変数の一連の値にしました。代わりに推測値の無限ストリームを作ることができます。推測値の初期値は 1 で始めます。<sup>126</sup>

```
(define (sqrt-stream x)
  (define guesses
    (cons-stream
      1.0
      (stream-map (lambda (guess) (sqrt-improve guess x))
                  guesses)))
  guesses)
```

---

<sup>125</sup>We can’t use `let` to bind the local variable `guesses`, because the value of `guesses` depends on `guesses` itself. Exercise 3.63 addresses why we want a local variable here.

<sup>126</sup>`let` をローカル変数 `guesses` を束縛するのに使うことはできません。`guesses` の値は `guesses` 自身に依存するためです。Exercise 3.63はなぜここで局所変数を欲しがるのかを扱います。

```
(display-stream (sqrt-stream 2))
1.
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
...
```

We can generate more and more terms of the stream to get better and better guesses. If we like, we can write a procedure that keeps generating terms until the answer is good enough. (See [Exercise 3.64](#).)

より多くのストリームの項を生成することでより良い推測値を得ることができます。もし望むなら、解答が十分に良くなるまで項の生成を続ける手続を書くことも可能です。[\(Exercise 3.64参照\)](#)。

Another iteration that we can treat in the same way is to generate an approximation to  $\pi$ , based upon the alternating series that we saw in [Section 1.3.1](#):

同じ方法で扱えるもう 1 つの反復は  $\pi$  の近似値を [Section 1.3.1](#)で見た交項級数(交代級数)を基にして生成することが可能です。

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

We first generate the stream of summands of the series (the reciprocals of the odd integers, with alternating signs). Then we take the stream of sums of more and more terms (using the `partial-sums` procedure of [Exercise 3.55](#)) and scale the result by 4:

最初に級数の加数(符号が交互に代わる奇数の逆数)のストリームを生成します。次に([Exercise 3.55](#)の `partial-sums` 手続を用いてより多くの項の和のストリームを取り、結果を 4 倍します。

```
(define (pi-summands n)
  (cons-stream (/ 1.0 n)
              (stream-map - (pi-summands (+ n 2)))))

(define pi-stream
  (scale-stream (partial-sums (pi-summands 1)) 4))

(display-stream pi-stream)
```

4.  
2.6666666666666667  
3.4666666666666667  
2.8952380952380956  
3.3396825396825403  
2.9760461760461765  
3.2837384837384844  
3.017071817071818  
...

This gives us a stream of better and better approximations to  $\pi$ , although the approximations converge rather slowly. Eight terms of the sequence bound the value of  $\pi$  between 3.284 and 3.017.

これはより良い  $\pi$  の近似値のストリームを提供します。しかし、近似値の収束はとても遅いです。列の 8 個の項は  $\pi$  の値を 3.284 から 3.017 の間に束縛されます。

So far, our use of the stream of states approach is not much different from updating state variables. But streams give us an opportunity to do some interesting tricks. For example, we can transform a stream with a *sequence accelerator* that converts a sequence of approximations to a new sequence that converges to the same value as the original, only faster.

今の所、状態のストリームを使用する取り組みは状態変数を更新する物から大きくは異なりません。しかしストリームはある面白いトリックを行う機会を提供します。例えば、近似値の列を、同じ値に、ただしそれほど速く収束する列に変換する *sequence accelerator*(列アクセラレータ) を用いてストリームを変換することができます。

One such accelerator, due to the eighteenth-century Swiss mathematician Leonhard Euler, works well with sequences that are partial sums of alternating series (series of terms with alternating signs). In Euler's technique, if  $S_n$  is the  $n^{\text{th}}$  term of the original sum sequence, then the accelerated sequence has terms

18世紀のスイスの数学者 Leonhard Euler(レオンハルト・オイラー)によるそのようなアクセラレータの1つは交項級数(符号を互い違いにする項の列)の部分和である列とうまく働きます。オイラーの手法においては、もし  $S_n$  が元の和の列の  $n$  番目の項であるなら、加速された列は以下の項を持ちます。

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}.$$

Thus, if the original sequence is represented as a stream of values, the transformed sequence is given by

従って元の列が値のストリームとして表現されるならば、変換された列は以下により与えられる。

```
(define (euler-transform s)
  (let ((s0 (stream-ref s 0))) ;  $S_{n-1}$ 
    (s1 (stream-ref s 1))) ;  $S_n$ 
    (s2 (stream-ref s 2))) ;  $S_{n+1}$ 
  (cons-stream (- s2 (/ (square (- s2 s1))
                           (+ s0 (* -2 s1) s2)))
               (euler-transform (stream-cdr s)))))
```

We can demonstrate Euler acceleration with our sequence of approximations to  $\pi$ :

オイラーによる加速を私達の  $\pi$  の近似値の列を用いて実演できます。

```
(display-stream (euler-transform pi-stream))
3.166666666666667
3.133333333333337
3.1452380952380956
3.13968253968254
3.1427128427128435
3.1408813408813416
3.142071817071818
3.1412548236077655
...
...
```

Even better, we can accelerate the accelerated sequence, and recursively accelerate that, and so on. Namely, we create a stream of streams (a structure we'll call a *tableau*) in which each stream is the transform of the preceding one:

さらに良くなるよう、加速された列を加速でき、そして再帰的にその加速を繰り返すことが可能です。すなわち、ストリームのストリーム (*tableau*(タブロー)と呼ぶ構造)を作り、その中では各ストリームは1つ前の変換です。

```
(define (make-tableau transform s)
  (cons-stream s (make-tableau transform (transform s))))
```

The tableau has the form

タブローは以下の形を取ります。

$s_{00}$	$s_{01}$	$s_{02}$	$s_{03}$	$s_{04}$	$\dots$
$s_{10}$	$s_{11}$	$s_{12}$	$s_{13}$	$\dots$	
$s_{20}$	$s_{21}$	$s_{22}$	$\dots$		
$\dots$					

Finally, we form a sequence by taking the first term in each row of the tableau:  
最後にタブローの各行の最初の項を取ることで列を形成します。

```
(define (accelerated-sequence transform s)
  (stream-map stream-car (make-tableau transform s)))
```

We can demonstrate this kind of “super-acceleration” of the  $\pi$  sequence:  
 $\pi$  の列のこの種の“超加速”を実演することができます。

```
(display-stream
  (accelerated-sequence euler-transform pi-stream))
4.
3.166666666666667
3.142105263157895
3.141599357319005
3.1415927140337785
3.1415926539752927
3.1415926535911765
3.141592653589778
...
```

The result is impressive. Taking eight terms of the sequence yields the correct value of  $\pi$  to 14 decimal places. If we had used only the original  $\pi$  sequence, we would need to compute on the order of  $10^{13}$  terms (i.e., expanding the series far enough so that the individual terms are less than  $10^{-13}$ ) to get that much accuracy!

結果は感動的です。列の 8 つの項を得ることで  $\pi$  の小数点以下 14 桁の正しい値がもたらせられます。もし元の  $\pi$  の列のみを使用したなら、 $10^{13}$  のオーダーの演算をする必要が(すなわち列の個々の項が  $10^{-13}$  よりも小さくなるまで十分に長く展開する必要が)同じ程度の正確さを得るために必要です！

We could have implemented these acceleration techniques without using streams. But the stream formulation is particularly elegant and convenient

because the entire sequence of states is available to us as a data structure that can be manipulated with a uniform set of operations.

これらの加速テクニックをストリームを用いずに実装することもできました。しかしストリームによる定式化はとりわけエレガントで便利です。状態の列全体が統一された命令の集合により操作可能なデータ構造として使用できるからです。

**Exercise 3.63:** Louis Reasoner asks why the `sqrt-stream` procedure was not written in the following more straightforward way, without the local variable `guesses`:

Louis Reasoner はなぜ `sqrt-stream` 手続が以下のように簡単な方法で、局所変数 `guesses` 無しで実装されていないのか尋ねた。

```
(define (sqrt-stream x)
  (cons-stream 1.0 (stream-map
    (lambda (guess)
      (sqrt-improve guess x))
    (sqrt-stream x))))
```

Alyssa P. Hacker replies that this version of the procedure is considerably less efficient because it performs redundant computation. Explain Alyssa's answer. Would the two versions still differ in efficiency if our implementation of `delay` used only `(lambda () <exp>)` without using the optimization provided by `memo-proc` ([Section 3.5.1](#))?

Alyssa P. Hacker が問題の手続のこの版は冗長な演算を行うため、かなり非効率であるからと答えた。Alyssa の答を説明せよ。もし `delay` の実装が `memo-proc` ([Section 3.5.1](#)) で提供された最適化を用いずに `(lambda () <exp>)` のみを使用したならば 2 つの版の間に依然として効率上の違いは存在するだろうか?

**Exercise 3.64:** Write a procedure `stream-limit` that takes as arguments a stream and a number (the tolerance). It should examine the stream until it finds two successive elements that differ in absolute value by less than the tolerance, and return the second of the two elements. Using this, we could compute square roots up to a given tolerance by

引数としてストリームと許容値の数値を取る手続 `stream-limit` を書け。差の絶対値が許容値未満である 2 つの連続する要素を見つけるまでストリームを検査し、その 2 つの要素の 2 番目を返す。これを用いて与えられた許容誤差以内の平方根を求めることができるだろう。

```
(define (sqrt x tolerance)
  (stream-limit (sqrt-stream x) tolerance))
```

**Exercise 3.65:** Use the series

以下の級数を用いて、

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots$$

to compute three sequences of approximations to the natural logarithm of 2, in the same way we did above for  $\pi$ . How rapidly do these sequences converge?

$\pi$  に対して上で行ったのと同様に、2 の自然対数の近似値を 3 種の近似値の列を求めよ。これらの列はどれだけ早く収束するか？

## Infinite streams of pairs

In Section 2.2.3, we saw how the sequence paradigm handles traditional nested loops as processes defined on sequences of pairs. If we generalize this technique to infinite streams, then we can write programs that are not easily represented as loops, because the “looping” must range over an infinite set.

Section 2.2.3において列パラダイムがどのように伝統的な入れ子ループをペアの列上に定義された手続として扱うかについて学びました。もしこのテクニックを無限ストリームに対しても一般化すれば簡単には繰り返しとしては表現されないプログラムを書くことができます。なぜなら“ループ”を無限集合の範囲にも渡らせなければなりません。

For example, suppose we want to generalize the `prime-sum-pairs` procedure of Section 2.2.3 to produce the stream of pairs of *all* integers  $(i, j)$  with  $i \leq j$  such that  $i + j$  is prime. If `int-pairs` is the sequence of all pairs of integers  $(i, j)$  with  $i \leq j$ , then our required stream is simply<sup>127</sup>

---

<sup>127</sup>As in Section 2.2.3, we represent a pair of integers as a list rather than a Lisp pair.

例えばSection 2.2.3の prime-sum-pairs 手続を一般化して、整数全てのペア  $(i, j)$ 、但し  $i \leq j$  で  $i + j$  が素数である場合のストリームを生成します。もし int-pairs が  $i \leq j$  における全ての整数のペア  $(i, j)$  の列であるのならば、私達が必要とするストリームは単純に以下のように定義されます。<sup>128</sup>

```
(stream-filter
  (lambda (pair) (prime? (+ (car pair) (cadr pair))))
  int-pairs)
```

Our problem, then, is to produce the stream `int-pairs`. More generally, suppose we have two streams  $S = (S_i)$  and  $T = (T_j)$ , and imagine the infinite rectangular array すると問題は `int-pairs` ストリームを生成することになります。より一般的には、2つのストリーム、 $S = (S_i)$  と  $T = (T_j)$  を持っているとした場合に、無限の長方形の配列を想像してみて下さい。

$(S_0, T_0)$	$(S_0, T_1)$	$(S_0, T_2)$	$\dots$
$(S_1, T_0)$	$(S_1, T_1)$	$(S_1, T_2)$	$\dots$
$(S_2, T_0)$	$(S_2, T_1)$	$(S_2, T_2)$	$\dots$
$\dots$			

We wish to generate a stream that contains all the pairs in the array that lie on or above the diagonal, i.e., the pairs

配列内の、対角線上かその上部の全てのペアを含むストリームを生成したいと考えます。つまり、以下のペアです。

$(S_0, T_0)$	$(S_0, T_1)$	$(S_0, T_2)$	$\dots$
$(S_1, T_1)$	$(S_1, T_2)$	$\dots$	
$(S_2, T_2)$	$\dots$		
$\dots$			

(If we take both  $S$  and  $T$  to be the stream of integers, then this will be our desired stream `int-pairs`.)

(もし  $S$  と  $T$  の両方を整数のストリームとして取るなら、これが望んだストリーム `int-pairs` です。)

Call the general stream of pairs (`pairs S T`), and consider it to be composed of three parts: the pair  $(S_0, T_0)$ , the rest of the pairs in the first row, and the remaining pairs.<sup>129</sup>

<sup>128</sup>Section 2.2.3にもある通り、私達は整数のペアを Lisp のペアではなく、リストにて表現します。

<sup>129</sup>See Exercise 3.68 for some insight into why we chose this decomposition.

一般的なペアのストリームを (`pairs S T`) と呼び、それが 3 つの部分から組み立てられていると考えます。ペア  $(S_0, T_0)$ 、最初の行の残りのペア、残りのペアです。<sup>130</sup>

$(S_0, T_0)$	$(S_0, T_1)$	$(S_0, T_2)$	$\dots$
	$(S_1, T_1)$	$(S_1, T_2)$	$\dots$
		$(S_2, T_2)$	$\dots$
			$\dots$

Observe that the third piece in this decomposition (pairs that are not in the first row) is (recursively) the pairs formed from (`stream-cdr S`) and (`stream-cdr T`). Also note that the second piece (the rest of the first row) is

この分解の 3 つ目の断片(最初の行にないペア)は(再帰的に) (`stream-cdr S`) と (`stream-cdr T`) から形成されることに注意して下さい。また 2 番目の断片(最初の行の残り)は以下により求められます。

```
(stream-map (lambda (x) (list (stream-car s) x))
            (stream-cdr t))
```

Thus we can form our stream of pairs as follows:

従って私達のペアのストリームは以下により形成できます。

```
(define (pairs s t)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (⟨combine-in-some-way⟩
      (stream-map (lambda (x) (list (stream-car s) x))
                  (stream-cdr t)))
    (pairs (stream-cdr s) (stream-cdr t)))))
```

In order to complete the procedure, we must choose some way to combine the two inner streams. One idea is to use the stream analog of the `append` procedure from Section 2.2.1:

手続を完成させるためには、2つの内部ストリームを接続する何らかの方法を選択せねばなりません。アイデアの1つはSection 2.2.1の `append` 手続の類似ストリームを用いる方法です。

```
(define (stream-append s1 s2)
```

---

<sup>130</sup>なぜ私達が分解を選ぶのかについての見識についてはExercise 3.68を参照して下さい。

```
(if (stream-null? s1)
    s2
    (cons-stream (stream-car s1)
                 (stream-append (stream-cdr s1) s2))))
```

This is unsuitable for infinite streams, however, because it takes all the elements from the first stream before incorporating the second stream. In particular, if we try to generate all pairs of positive integers using

しかし、これは無限ストリームには不適切です。なぜならばこれは最初のストリームからの要素を全て、2つ目のストリームとの合併前に取ります。具体的には、もし全ての正の整数のペアを以下のようにして生成しようとすると、

```
(pairs integers integers)
```

our stream of results will first try to run through all pairs with the first integer equal to 1, and hence will never produce pairs with any other value of the first integer.

結果のストリームは最初に 1 番目の整数が 1 の場合の全てのペアを通して実行しようとします。そしてそれ故に 1 番目の整数が他の値のペアを全く生成することができません。

To handle infinite streams, we need to devise an order of combination that ensures that every element will eventually be reached if we let our program run long enough. An elegant way to accomplish this is with the following `interleave` procedure.<sup>131</sup>

無限ストリームを扱うためには、プログラムを十分に長く実行したならば全ての要素がいつかは得られることを保証する組み合わせの順を工夫する必要があります。これを達成する洗練された方法は以下の `interleave`(相互配置)手続を用います。<sup>132</sup>

---

<sup>131</sup>The precise statement of the required property on the order of combination is as follows: There should be a function  $f$  of two arguments such that the pair corresponding to element  $i$  of the first stream and element  $j$  of the second stream will appear as element number  $f(i, j)$  of the output stream. The trick of using `interleave` to accomplish this was shown to us by David Turner, who employed it in the language KRC (Turner 1981).

<sup>132</sup>組み合わせの順に要求された属性を正しく上げると次によくなります。2つの引数を取る関数が必須であり、最初のストリームの要素  $i$  と 2つ目のストリームの要素  $j$  に対応するペアは出力ストリームの  $f(i, j)$  番目として現れます。`interleave` を用いてこれを達成するトリックは、KRC 言語にこれを採用した David Turner により示されました。(Turner 1981)

```
(define (interleave s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (interleave s2 (stream-cdr s1)))))
```

Since `interleave` takes elements alternately from the two streams, every element of the second stream will eventually find its way into the interleaved stream, even if the first stream is infinite.

`interleave` は 2 つのストリームから交代に要素を得るため、2 つ目のストリームの各要素がいつかは相互配置ストリームへ入ることが、例え最初のストリームが無限でもわかります。

We can thus generate the required stream of pairs as

従って要求されたペアのストリームを以下のように生成できます。

```
(define (pairs s t)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (interleave
      (stream-map (lambda (x) (list (stream-car s) x))
                  (stream-cdr t))
      (pairs (stream-cdr s) (stream-cdr t))))))
```

**Exercise 3.66:** Examine the stream (`pairs integers integers`). Can you make any general comments about the order in which the pairs are placed into the stream? For example, approximately how many pairs precede the pair (1, 100)? the pair (99, 100)? the pair (100, 100)? (If you can make precise mathematical statements here, all the better. But feel free to give more qualitative answers if you find yourself getting bogged down.)

ストリーム (`pairs integers integers`) を試験せよ。`pairs` がストリーム内に配置する順について全体的なコメントを行え。例えればペア (1, 100) の前にはおよそどれだけの数のペアが先行するか? (99, 100) と (100, 100) の場合についても答えよ。(もし正確な数学上の説明ができるなら、なおさら良い。しかし行き詰まつた感じるのならばより程度的な回答を気楽に上げて欲しい。)

**Exercise 3.67:** Modify the `pairs` procedure so that `(pairs integers integers)` will produce the stream of *all* pairs of integers  $(i, j)$  (without the condition  $i \leq j$ ). Hint: You will need to mix in an additional stream.

`pairs` 手続を変更し、`(pairs integers integers)` が全ての整数のペア  $(i, j)$  を ( $i \leq j$  という条件無しで) 生成するようにならね。ヒント：追加のストリームを混ぜ合わせる必要がある。

**Exercise 3.68:** Louis Reasoner thinks that building a stream of pairs from three parts is unnecessarily complicated. Instead of separating the pair  $(S_0, T_0)$  from the rest of the pairs in the first row, he proposes to work with the whole first row, as follows:

Louis Reasoner は 3 つの部分からペアストリームを構築することは不必要に複雑なのではないかと考えた。最初の行のペア  $(S_0, T_0)$  を残りのペアから分離する代わりに、以下のように最初の行全体を用いて行うことを探した。

```
(define (pairs s t)
  (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                t)
    (pairs (stream-cdr s) (stream-cdr t))))
```

Does this work? Consider what happens if we evaluate `(pairs integers integers)` using Louis's definition of `pairs`.

これはうまく行くだろうか? `(pairs integers integers)` を Louis の `pairs` の定義を用いて評価した場合に何が起こるか考えよ。

**Exercise 3.69:** Write a procedure `triples` that takes three infinite streams,  $S$ ,  $T$ , and  $U$ , and produces the stream of triples  $(S_i, T_j, U_k)$  such that  $i \leq j \leq k$ . Use `triples` to generate the stream of all Pythagorean triples of positive integers, i.e., the triples  $(i, j, k)$  such that  $i \leq j$  and  $i^2 + j^2 = k^2$ .

3 つの無限ストリーム、 $S$ ,  $T$ ,  $U$  を取り、三つ組  $(S_i, T_j, U_k)$  のストリームを生成する手続 `triples` を書け。但し  $i \leq j \leq k$  とする。`triples` を用いて全ての正の整数のピタゴラス数の 3 つ組のストリームを生成せよ。すなわち三つ組  $(i, j, k)$  は  $i \leq j$ 、かつ  $i^2 + j^2 = k^2$  である。

**Exercise 3.70:** It would be nice to be able to generate streams in which the pairs appear in some useful order, rather than in the order that results from an *ad hoc* interleaving process. We can use a technique similar to the `merge` procedure of Exercise 3.56, if we define a way to say that one pair of integers is “less than” another. One way to do this is to define a “weighting function”  $W(i, j)$  and stipulate that  $(i_1, j_1)$  is less than  $(i_2, j_2)$  if  $W(i_1, j_1) < W(i_2, j_2)$ . Write a procedure `merge-weighted` that is like `merge`, except that `merge-weighted` takes an additional argument `weight`, which is a procedure that computes the weight of a pair, and is used to determine the order in which elements should appear in the resulting merged stream.<sup>133</sup> Using this, generalize `pairs` to a procedure `weighted-pairs` that takes two streams, together with a procedure that computes a weighting function, and generates the stream of pairs, ordered according to weight. Use your procedure to generate

アドホック(その場その場)な相互配置処理の結果の順ではなく、ペアが何らかの便利な順で現れるストリームを生成できれば便利だろう。もし整数の1つのペアが別のペアよりも“小さい”と言える方法を定義できるならば Exercise 3.56 の `merge` 手続に似たテクニックを用いることができる。これを行う1つの方法は“重み関数” $W(i, j)$ を定義し  $W(i_1, j_1) < W(i_2, j_2)$  であるなら  $(i_1, j_1)$  は  $(i_2, j_2)$  未満であると取り決める。`merge` に似た手続 `merge-weighted` を書け。ただし `merge-weighted` は追加の引数 `weight` を取り、`weight` はペアの重みを計る手続でありマージされた結果のストリームの中でどの要素が現れるべきかの順を決定するのに利用される。<sup>134</sup> 重み関数を計算する手続と一緒にこれを用いて、`pairs` を2つのストリームを取る手続 `weighted-pairs` に一般化し、重みに従った順のペアのストリームを生成する。作成した手続を用いて以下を生成せよ。

- a the stream of all pairs of positive integers  $(i, j)$  with  $i \leq j$   
ordered according to the sum  $i + j$ ,

---

<sup>133</sup>We will require that the weighting function be such that the weight of a pair increases as we move out along a row or down along a column of the array of pairs.

<sup>134</sup>ペアの重みがペアの配列の中で行に沿って外へ動くか、列に沿って下った場合に増えるように重み関数に対して要求するだろう。

全ての正の整数のペア  $(i, j)$  のストリームを  $i \leq j$  の条件で、和  $i + j$  に従った順で生成する

- b the stream of all pairs of positive integers  $(i, j)$  with  $i \leq j$ , where neither  $i$  nor  $j$  is divisible by 2, 3, or 5, and the pairs are ordered according to the sum  $2i + 3j + 5ij$ .

全ての正の整数のペア  $(i, j)$  のストリームを  $i \leq j$ かつ  $i$  と  $j$  がどちらも 2, 3, 5 で割り切れない条件で、和  $2i+3j+5ij$  に従う順序で生成する

**Exercise 3.71:** Numbers that can be expressed as the sum of two cubes in more than one way are sometimes called *Ramanujan numbers*, in honor of the mathematician Srinivasa Ramanujan.<sup>135</sup> Ordered streams of pairs provide an elegant solution to the problem of computing these numbers. To find a number that can be written as the sum of two cubes in two different ways, we need only generate the stream of pairs of integers  $(i, j)$  weighted according to the sum  $i^3 + j^3$  (see Exercise 3.70), then search the stream for two consecutive pairs with the same weight. Write a procedure to generate the Ramanujan numbers. The first such number is 1,729. What are the next five?

2つの立方数の和で表す方法が複数ある数は時折 *Ramanujan numbers* (ラマヌジャン数) と呼ばれる。これは数学者 Srinivasa Ramanujan (シュリニヴァーサ・ラマヌジャン) に敬意を表している。  
<sup>136</sup> ペアの順序有リストリームはこれらの数を計算する問題に対

---

<sup>135</sup>To quote from G. H. Hardy's obituary of Ramanujan (Hardy 1921): "It was Mr. Littlewood (I believe) who remarked that 'every positive integer was one of his friends.' I remember once going to see him when he was lying ill at Putney. I had ridden in taxi-cab No. 1729, and remarked that the number seemed to me a rather dull one, and that I hoped it was not an unfavorable omen. 'No,' he replied, 'it is a very interesting number; it is the smallest number expressible as the sum of two cubes in two different ways.' The trick of using weighted pairs to generate the Ramanujan numbers was shown to us by Charles Leiserson.

<sup>136</sup>G. H. Hardy(ゴッドフレイ・ハロルド・ハーディ)によるラマヌジャンの死亡告知 (Hardy 1921) から引用すれば、「Mr. Littlewood こそが‘全ての自然数が彼の友達であった’と述べられた (私が信じる) 人だ。私は彼が Putney で病気で倒れた時に 1 度会いに行つた。その時私は車番 1729 のタクシーに乗つたのでとてもつまらない数に見えたと告げ、それが良くない前触れでないことを祈ると伝えた。‘いいえ’と彼が答えた。‘それ

し洗練された解法を提供する。2つの立方数の和として表現する方法が2つある数を見付けるためには、 $i^3 + j^3$  の和に従い重み付けられた整数のペア  $(i, j)$  のストリームを生成し (Exercise 3.70 参照)、次にストリームから同じ重みを持つ連続したペアを探すだけで良い。ラマヌジャン数を生成する手續を書け。そのような最初の数は 1,729 である。次の 5 つは何か?

**Exercise 3.72:** In a similar way to Exercise 3.71 generate a stream of all numbers that can be written as the sum of two squares in three different ways (showing how they can be so written).

Exercise 3.71 と同様な方法で 2 つの平方数の和として 3 つの異なる方法で書ける全ての数のストリームを生成せよ。(それらがどのようにして、そう書けるのか示せ)。

## Streams as signals

We began our discussion of streams by describing them as computational analogs of the “signals” in signal-processing systems. In fact, we can use streams to model signal-processing systems in a very direct way, representing the values of a signal at successive time intervals as consecutive elements of a stream. For instance, we can implement an *integrator* or *summer* that, for an input stream  $x = (x_i)$ , an initial value  $C$ , and a small increment  $dt$ , accumulates the sum

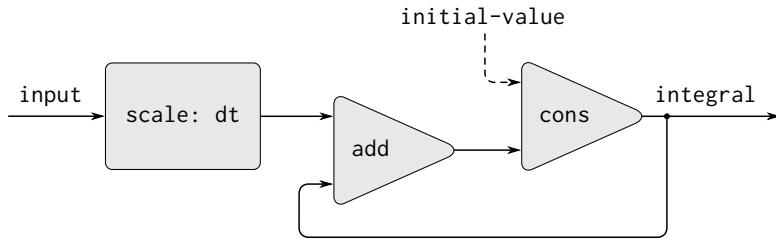
ストリームの議論を信号処理システムにおける“信号”的計算可能な同類であると説明することから始めました。実際に、ストリームを用いて信号処理システムをとても直接的な方法でモデル化することができ、連続する時間区間の信号の値をストリームの連続する要素として表現します。例えば *integrator*(積分器)、つまり *summer*(アナログ加算器) を実装し、入力ストリーム  $x = (x_i)$  と初期値  $C$ 、小さな増分  $dt$  に対し、以下の合計を累算し、

$$S_i = C + \sum_{j=1}^i x_j dt$$

and returns the stream of values  $S = (S_i)$ . The following `integral` procedure is reminiscent of the “implicit style” definition of the stream of integers (Section 3.5.2):

---

はとても面白い数です。それは 2 つの立方数の和により表現する方法が 2 つ有る最小の数です” ラマヌジャン数を生成する重み付けられたペアのトリックは Charles Leiserson により私達に示されました。



**Figure 3.32:** The `integral` procedure viewed as a signal-processing system.

値  $S = (S_i)$  のストリームを返します。以下の `integral` 手続は (Section 3.5.2) の整数ストリームの“暗黙的なスタイル”的定義を思い出させます。

```

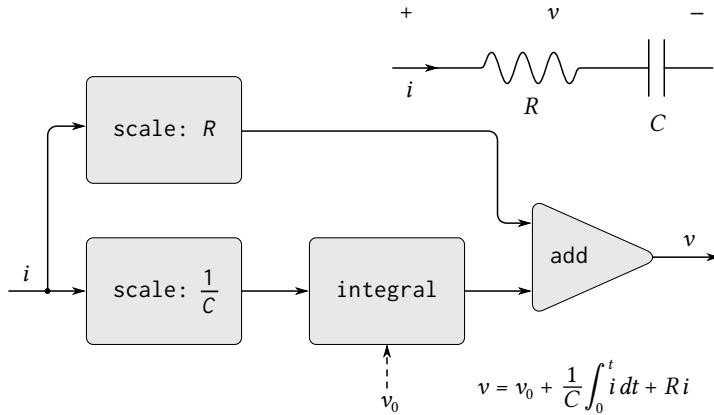
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (add-streams (scale-stream integrand dt)
                              int))))
int)

```

Figure 3.32 is a picture of a signal-processing system that corresponds to the `integral` procedure. The input stream is scaled by  $dt$  and passed through an adder, whose output is passed back through the same adder. The self-reference in the definition of `int` is reflected in the figure by the feedback loop that connects the output of the adder to one of the inputs.

Figure 3.32は `integral` 手続に相当する信号処理システムの絵です。入力ストリームは  $dt$  によりスケール(拡大)され加算器を通して渡され、その出力は同じ加算器に戻されます。`int` の定義内の自己参照が図では加算器の出力が入力の 1 つに接続されるフィードバックループにより反映されています。

**Exercise 3.73:** We can model electrical circuits using streams to represent the values of currents or voltages at a sequence of times. For instance, suppose we have an *RC circuit* consisting of a resistor of resistance  $R$  and a capacitor of capacitance  $C$  in series. The voltage response  $v$  of the circuit to an injected current  $i$  is deter-



**Figure 3.33:** An RC circuit and the associated signal-flow diagram.

mined by the formula in Figure 3.33, whose structure is shown by the accompanying signal-flow diagram.

電子回路をストリームを用いて一連の時間の時系列電流や電圧の値を表すことでモデル化できる。例えば、抵抗値  $R$  の抵抗と静電容量  $C$  のコンデンサから成る  $RC$  circuit(  $RC$  回路)を連続して持っているとする。入力された電流  $i$ に対する回路の電圧レスポンス  $v$ は Figure 3.33 の式により決定し、その構造は添付の信号フロー図により示される。

Write a procedure `RC` that models this circuit. `RC` should take as inputs the values of  $R$ ,  $C$ , and  $dt$  and should return a procedure that takes as inputs a stream representing the current  $i$  and an initial value for the capacitor voltage  $v_0$  and produces as output the stream of voltages  $v$ . For example, you should be able to use `RC` to model an  $RC$  circuit with  $R = 5$  ohms,  $C = 1$  farad, and a 0.5-second time step by evaluating (`define RC1 (RC 5 1 0.5)`). This defines `RC1` as a procedure that takes a stream representing the time sequence of currents and an initial capacitor voltage and produces the output stream of voltages.

この回路をモデル化する手続 `RC`を書け。`RC`は入力として  $R$ ,  $C$ ,  $dt$

を取り、手続を返さねばならない。返り値の手続は入力として電流  $i$  を表すストリームとコンデンサの初期電圧  $v_0$  を取り、出力として電圧  $v$  のストリームを生成する。例えば RC を用いて  $R$  が  $5[\Omega]$ 、 $C$  が  $1[F]$ 、タイムステップが 0.5 秒の RC 回路を (`define RC1 (RC 5 1 0.5)`) を評価することでモデル化できなければならない。これは RC1 を電流の時系列を表すストリームとコンデンサの初期電圧を取り電圧の出力ストリームを生成する。

**Exercise 3.74:** Alyssa P. Hacker is designing a system to process signals coming from physical sensors. One important feature she wishes to produce is a signal that describes the *zero crossings* of the input signal. That is, the resulting signal should be  $+1$  whenever the input signal changes from negative to positive,  $-1$  whenever the input signal changes from positive to negative, and  $0$  otherwise. (Assume that the sign of a  $0$  input is positive.) For example, a typical input signal with its associated zero-crossing signal would be

Alyssa P. Hacker は物理センサから来る信号を処理するシステムを設計している。彼女が作りたい重要な機能は入力信号の *zero crossings*(ゼロ交差) を記録する信号である。結果の信号は入力信号が負から正に変わった時に  $+1$ 、正から負に変わった時に  $-1$ 、その他の場合は  $0$  である。(入力が  $0$  の場合の符号は正とする)。例えば典型的な入力信号とその関連するゼロ交差信号は以下のようになる。

```
... 1 2 1.5 1 0.5 -0.1 -2 -3 -2 -0.5 0.2 3 4 ...
... 0 0 0 0 -1 0 0 0 0 1 0 0 ...
```

In Alyssa's system, the signal from the sensor is represented as a stream `sense-data` and the stream `zero-crossings` is the corresponding stream of zero crossings. Alyssa first writes a procedure `sign-change-detector` that takes two values as arguments and compares the signs of the values to produce an appropriate  $0$ ,  $1$ , or  $-1$ . She then constructs her zero-crossing stream as follows:

Alyssa のシステムではセンサからの信号はストリーム `sense-data` で表され、ストリーム `zero-crossings` が関連するゼロ交差のスト

リームである。Alyssa は最初に手続 `sign-change-detector` を書いた。これは 2 つの値を引数として取り値の符号を比べ値に対応した 0, 1, -1 を生成する。次にゼロ交差ストリームを以下のように構築した。

```
(define (make-zero-crossings
    input-stream last-value)
  (cons-stream
    (sign-change-detector
      (stream-car input-stream)
      last-value)
    (make-zero-crossings
      (stream-cdr input-stream)
      (stream-car input-stream)))))

(define zero-crossings
  (make-zero-crossings sense-data 0))
```

Alyssa's boss, Eva Lu Ator, walks by and suggests that this program is approximately equivalent to the following one, which uses the generalized version of `stream-map` from [Exercise 3.50](#):

Alyssa の上司、Eva Lu Ator が歩み寄り、このプログラムは以下の、[Exercise 3.50](#)の `stream-map` を一般化した版を使用した物とほぼ同じであると提案した。

```
(define zero-crossings
  (stream-map sign-change-detector
              sense-data
              (expression)))
```

Complete the program by supplying the indicated `(expression)`.  
`(expression)` で示された部分を与えてプログラムを完成させよ。

**Exercise 3.75:** Unfortunately, Alyssa's zero-crossing detector in [Exercise 3.74](#) proves to be insufficient, because the noisy signal from the sensor leads to spurious zero crossings. Lem E. Tweakit, a hardware specialist, suggests that Alyssa smooth the signal to filter out the noise before extracting the zero crossings. Alyssa takes his advice and decides to extract the zero crossings from the signal constructed by averaging each value of the sense data with the

previous value. She explains the problem to her assistant, Louis Reasoner, who attempts to implement the idea, altering Alyssa's program as follows:

残念なことに、Exercise 3.74の Alyssa のゼロ交差判別器は十分でないことが証明された。センサからのノイズの多い信号が誤ったゼロ交差へと導くためである。ハードウェアのスペシャリスト、Lem E. Tweakit は Alyssa にゼロ交差を試験する前にノイズを排除するために信号を滑らかにすることを提案した。Alyssa は彼のアドバイスを受け入れ、センサのデータの各値を前の値との平均を取ることで構築された信号からゼロ交差を抽出することを決めた。彼女は問題を彼女のアシスタント、Louis Reasoner に伝えた。彼はその考えを実装しようと試み、Alyssa のプログラムを以下のように変更した。

```
(define (make-zero-crossings
    input-stream last-value)
  (let ((avpt (/ (+ (stream-car input-stream)
    last-value)
    2)))
    (cons-stream
      (sign-change-detector avpt last-value)
      (make-zero-crossings
        (stream-cdr input-stream) avpt)))))
```

This does not correctly implement Alyssa's plan. Find the bug that Louis has installed and fix it without changing the structure of the program. (Hint: You will need to increase the number of arguments to `make-zero-crossings`.)

これは Alyssa の計画を正しく実装していない。Louis が入れてしまつたバグを見つけプログラムの構造を変更せずに直せ。(ヒント：`make-zero-crossings` の引数の数を増やす必要がある。)

**Exercise 3.76:** Eva Lu Ator has a criticism of Louis's approach in Exercise 3.75. The program he wrote is not modular, because it intermixes the operation of smoothing with the zero-crossing extraction. For example, the extractor should not have to be changed if Alyssa finds a better way to condition her input signal. Help Louis by writing a procedure `smooth` that takes a stream as input and

produces a stream in which each element is the average of two successive input stream elements. Then use `smooth` as a component to implement the zero-crossing detector in a more modular style.

Eva Lu Ator は Exercise 3.75 における Louis の取り組み方を批判した。彼が書いたプログラムはモジュラ化されていない。滑らかにする操作とゼロ交差抽出が混ざってしまっているためである。例えば抽出器は Alyssa が入力信号を調整するより良い手段を見つければ変更する必要が無かった。Louis を手助けし、入力としてストリームを取り、2つの連続する入力ストリームの要素の平均を要素とするストリームを生成する手続 `smooth` を書け。次に `smooth` をゼロ交差判定機を実装するためのコンポーネントとしてよりモジュラー化スタイルにて用いよ。

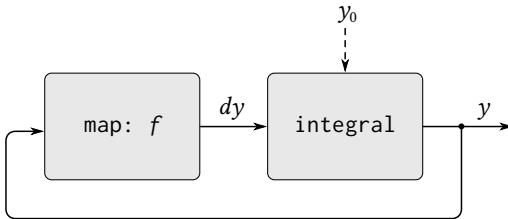
### 3.5.4 Streams and Delayed Evaluation

The `integral` procedure at the end of the preceding section shows how we can use streams to model signal-processing systems that contain feedback loops. The feedback loop for the adder shown in Figure 3.32 is modeled by the fact that `integral`'s internal stream `int` is defined in terms of itself:

先の節の終わりの `integral` 手續はどのようにストリームを用いてフィードバックループを持つ信号処理システムをモデル化できるかを示しています。Figure 3.32 に示される加算器のフィードバックループは `integral` の内部ストリーム `int` がそれ自身を用いて定義されている事実によりモデル化されています。

```
(define int
  (cons-stream
    initial-value
    (add-streams (scale-stream integrand dt)
                 int)))
```

The interpreter's ability to deal with such an implicit definition depends on the `delay` that is incorporated into `cons-stream`. Without this `delay`, the interpreter could not construct `int` before evaluating both arguments to `cons-stream`, which would require that `int` already be defined. In general, `delay` is crucial for using streams to model signal-processing systems that contain loops. Without `delay`, our models would have to be formulated so that the inputs to



**Figure 3.34:** An “analog computer circuit” that solves the equation  $dy/dt = f(y)$ .

any signal-processing component would be fully evaluated before the output could be produced. This would outlaw loops.

暗黙的定義のような物を扱うインタプリタの能力は `cons-stream` に組込まれている `delay` に依存している。この `delay` 無しではインタプリタは、`int` が既に定義されていることを要件とする `cons-stream`への引数両方を評価する前には `int` を構築することができませんでした。一般的に、`delay` はストリームを用いてループを含む信号処理システムをモデル化するのに不可欠です。`delay` 無しでは任意の信号処理コンポーネントへの入力が、出力を生成する前に完全に評価されるように、私達のモデルが定式化されなければなりません。

Unfortunately, stream models of systems with loops may require uses of `delay` beyond the “hidden” `delay` supplied by `cons-stream`. For instance, Figure 3.34 shows a signal-processing system for solving the differential equation  $dy/dt = f(y)$  where  $f$  is a given function. The figure shows a mapping component, which applies  $f$  to its input signal, linked in a feedback loop to an integrator in a manner very similar to that of the analog computer circuits that are actually used to solve such equations.

残念なことに、ループを伴うシステムのストリームモデルは `cons-stream` により提供される“隠れた”`delay` を越えて、`delay` の使用を要求します。例えば Figure 3.34 は  $f$  が与えられた関数である場合に微分方程式信号  $dy/dt = f(y)$  を解く処理システムを示しています。図は  $f$  をその入力信号に適用するマッピングコンポーネント (map) を示しています。map はフィードバックループの中に積分器へ向けて実際にそのような方程式を解くために利用されているアナログ計算機回路にとても似た作法で接続されています。

Assuming we are given an initial value  $y_0$  for  $y$ , we could try to model this

system using the procedure

$y$  に対し初期値  $y_0$  を与えられたとした時、このシステムを以下の手続を用いてモデル化を試みることができるでしょう。

```
(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (stream-map f y)))
y)
```

This procedure does not work, because in the first line of `solve` the call to `integral` requires that the input `dy` be defined, which does not happen until the second line of `solve`.

この手続はうまく行きません。`solve` の最初の行にて `integral` の呼出は入力 `dy` が定義されることを要求します。これは `solve` の二行目までは起こり得ません。

On the other hand, the intent of our definition does make sense, because we can, in principle, begin to generate the `y` stream without knowing `dy`. Indeed, `integral` and many other stream operations have properties similar to those of `cons-stream`, in that we can generate part of the answer given only partial information about the arguments. For `integral`, the first element of the output stream is the specified `initial-value`. Thus, we can generate the first element of the output stream without evaluating the integrand `dy`. Once we know the first element of `y`, the `stream-map` in the second line of `solve` can begin working to generate the first element of `dy`, which will produce the next element of `y`, and so on.

一方で、私達の定義の意図もつじつまが合いません。原理上は、`y` ストリームを `dy` を知らずに生成し始めることができます。再に `integral` や他の多くのストリーム命令は `cons-stream` に、引数に関する部分的な情報を与えられただけで応答の部分を生成できるという点で似た性質を持っています。`integral` では出力ストリームの最初の要素は `initial-value` で与えられます。従って出力ストリームの最初の要素を被積分関数 `dy` を評価せずに生成することができます。一度 `y` の最初の要素を知ることができれば、`solve` の 2 行目の `stream-map` は `dy` の最初の要素を生成する仕事を開始できます。これにより `y` の次の要素を生成することもでき、以下繰り返しとなります。

To take advantage of this idea, we will redefine `integral` to expect the integrand stream to be a *delayed argument*. `Integral` will force the integrand to be evaluated only when it is required to generate more than the first element

of the output stream:

この考え方の利点を得るために、`integral` を再定義し、被積分関数ストリームが *delayed argument*(遅延引数) を要求するようにします。`Integral` は出力ストリームの最初の要素より多くを生成することを要求された時のみ、被積分関数を `force` し評価させます。

```
(define (integral delayed-integrand initial-value dt)
  (define int
    (cons-stream
      initial-value
      (let ((integrand (force delayed-integrand)))
        (add-streams (scale-stream integrand dt)
                     int)))))

int)
```

Now we can implement our `solve` procedure by delaying the evaluation of `dy` in the definition of `y`.<sup>137</sup>

これで `solve` 手続の実装が、`y` の定義内で `dy` を遅延させればできます。<sup>138</sup>

```
(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt)))
  (define dy (stream-map f y))
  y)
```

In general, every caller of `integral` must now `delay` the integrand argument. We can demonstrate that the `solve` procedure works by approximating  $e \approx 2.718$  by computing the value at  $y = 1$  of the solution to the differential equation  $dy/dt = y$  with initial condition  $y(0) = 1$ :

全体的に、`integral` を呼び出す者は今では被積分関数を `delay` しなければなりません。 $e \approx 2.718$  の近似値を微分方程式  $dy/dt = y$  に対する解が  $y = 1$  の場合の値を初期条件  $y(0) = 1$  で求めてことで `solve` 手続がうまく働くことを実演できます。

---

<sup>137</sup>This procedure is not guaranteed to work in all Scheme implementations, although for any implementation there is a simple variation that will work. The problem has to do with subtle differences in the ways that Scheme implementations handle internal definitions. (See [Section 4.1.6](#).)

<sup>138</sup>この手続は全ての Scheme 実装で動くことが保証されていません。とはいっても、任意の実装に対して簡単な変更で動作します。問題は Scheme 実装の内部定義の扱い方に関係があります。([Section 4.1.6](#)参照)

```
(stream-ref (solve (lambda (y) y) 1 0.001) 1000)
2.716924
```

**Exercise 3.77:** The `integral` procedure used above was analogous to the “implicit” definition of the infinite stream of integers in Section 3.5.2. Alternatively, we can give a definition of `integral` that is more like `integers-starting-from` (also in Section 3.5.2):

上で使用された `integral` 手続はSection 3.5.2の整数無限ストリームの“暗示的”定義に似ている。代替的に、より `integers-starting-from` に似た `integral` の定義を与えることができない。(これもSection 3.5.2参照)

```
(define (integral integrand initial-value dt)
  (cons-stream
    initial-value
    (if (stream-null? integrand)
        the-empty-stream
        (integral (stream-cdr integrand)
                  (+ (* dt (stream-car integrand))
                     initial-value)
                  dt)))))
```

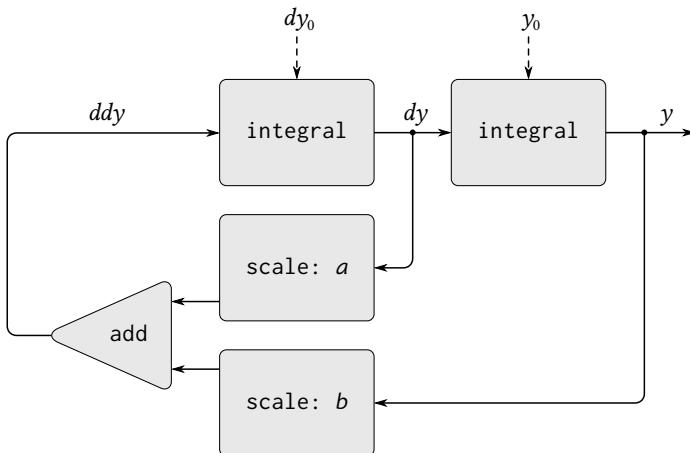
When used in systems with loops, this procedure has the same problem as does our original version of `integral`. Modify the procedure so that it expects the `integrand` as a delayed argument and hence can be used in the `solve` procedure shown above.

ループを持つシステム内で利用された場合、この手続は `integral` の元の版が抱えた問題と同じ問題を持つ。手続を変更して `integrand` に対し遅延された引数を要求するようにし、それ故に上で示されたように `solve` 手続で利用できるようにせよ。

**Exercise 3.78:** Consider the problem of designing a signal-processing system to study the homogeneous second-order linear differential equation

単項二次線形微分方程式を学ぶための信号処理システムの設計問題について考えよ。

$$\frac{d^2y}{dt^2} - a \frac{dy}{dt} - by = 0$$



**Figure 3.35:** Signal-flow diagram for the solution to a second-order linear differential equation.

The output stream, modeling  $y$ , is generated by a network that contains a loop. This is because the value of  $d^2y/dt^2$  depends upon the values of  $y$  and  $dy/dt$  and both of these are determined by integrating  $d^2y/dt^2$ . The diagram we would like to encode is shown in Figure 3.35. Write a procedure `solve-2nd` that takes as arguments the constants  $a$ ,  $b$ , and  $dt$  and the initial values  $y_0$  and  $dy_0$  for  $y$  and  $dy/dt$  and generates the stream of successive values of  $y$ .

$y$  をモデル化する出力ストリームはループを含むネットワークにより生成される。これは  $d^2y/dt^2$  の値が  $y$  と  $dy/dt$  の値に依存し、これらの両方が  $d^2y/dt^2$  を積分することにより決定されるからである。Figure 3.35 に示される図の符号化を行いたい。定数  $a$ ,  $b$ ,  $dt$  と初期値、 $y$  に対する  $y_0$  と  $dy_0$  を引数として取り、 $y$  の一連の値のストリームを生成する手続 `solve-2nd` を書け。

**Exercise 3.79:** Generalize the `solve-2nd` procedure of Exercise 3.78 so that it can be used to solve general second-order differential

equations  $d^2y/dt^2 = f(dy/dt, y)$ .

**Exercise 3.78**の `solve-2nd` 手続を一般化し、一般的な二次微分方程式  $d^2y/dt^2 = f(dy/dt, y)$  を解くのに使用できるようにせよ。

**Exercise 3.80:** A *series RLC circuit* consists of a resistor, a capacitor, and an inductor connected in series, as shown in Figure 3.36. If  $R$ ,  $L$ , and  $C$  are the resistance, inductance, and capacitance, then the relations between voltage ( $v$ ) and current ( $i$ ) for the three components are described by the equations

*series RLC circuit*(連続 RLC 回路) は抵抗、コンデンサ、インダクタンスが Figure 3.36 に示されるように連結されている。 $R$ ,  $L$ ,  $C$  が抵抗、インダクタンス、コンデンサである場合、3 つのコンポーネントに対する電圧 ( $v$ ) と電流 ( $i$ ) の間の関係は以下の方程式により説明される。

$$v_R = i_R R, \quad v_L = L \frac{di_L}{dt}, \quad i_C = C \frac{dv_C}{dt},$$

and the circuit connections dictate the relations

そして回路の接続が以下の関係を決定する。

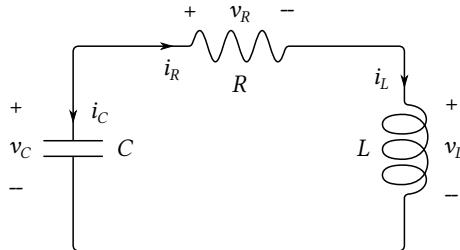
$$i_R = i_L = -i_C, \quad v_C = v_L + v_R.$$

Combining these equations shows that the state of the circuit (summarized by  $v_C$ , the voltage across the capacitor, and  $i_L$ , the current in the inductor) is described by the pair of differential equations  
これらの方程式の組み合わせは(コンデンサに渡る電圧  $v_C$  とインダクタンスの電流  $i_L$  にてまとめれば)回路の状態が以下の微分方程式のペアで説明されることを示している。

$$\frac{dv_C}{dt} = -\frac{i_L}{C}, \quad \frac{di_L}{dt} = \frac{1}{L}v_C - \frac{R}{L}i_L.$$

The signal-flow diagram representing this system of differential equations is shown in Figure 3.37.

この微分方程式のシステムを表す信号フロー図は Figure 3.37 に示される。



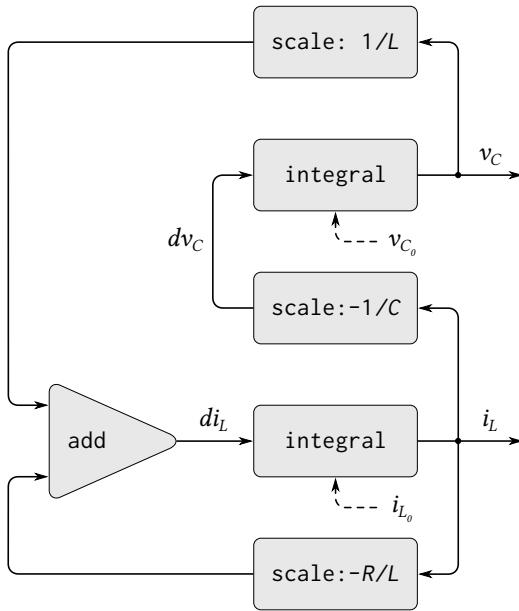
**Figure 3.36:** A series RLC circuit.

Write a procedure `RLC` that takes as arguments the parameters  $R$ ,  $L$ , and  $C$  of the circuit and the time increment  $dt$ . In a manner similar to that of the `RC` procedure of [Exercise 3.73](#), `RLC` should produce a procedure that takes the initial values of the state variables,  $v_{C_0}$  and  $i_{L_0}$ , and produces a pair (using `cons`) of the streams of states  $v_C$  and  $i_L$ . Using `RLC`, generate the pair of streams that models the behavior of a series RLC circuit with  $R = 1$  ohm,  $C = 0.2$  farad,  $L = 1$  henry,  $dt = 0.1$  second, and initial values  $i_{L_0} = 0$  amps and  $v_{C_0} = 10$  volts.

引数として回路のパラメタ  $R$ ,  $L$ ,  $C$  と時間の増分  $dt$  を取る手続 `RLC` を書け。ある意味では [Exercise 3.73](#) の `RC` 手続のそれに似ているが、`RLC` は状態変数の初期値  $v_{C_0}$  と  $i_{L_0}$  を取り、(`cons` を用いて)  $v_C$  と  $i_L$  の状態のストリームのペアを生成する手続を生成せねばならない。`RLC` を用いて、連結 RLC 回路の振舞をモデル化するストリームのペアを生成せよ。ただし  $R = 1[\Omega]$ ,  $C = 0.2[F]$ ,  $L = 1$  henry,  $dt = 0.1[s]$ , sorenに 初期値  $i_{L_0} = 0[A]$ ,  $v_{C_0} = 10[V]$  とする。

## Normal-order evaluation

The examples in this section illustrate how the explicit use of `delay` and `force` provides great programming flexibility, but the same examples also show how this can make our programs more complex. Our new `integral` procedure, for instance, gives us the power to model systems with loops, but we must now remember that `integral` should be called with a delayed integrand, and



**Figure 3.37:** A signal-flow diagram for the solution to a series RLC circuit.

every procedure that uses `integral` must be aware of this. In effect, we have created two classes of procedures: ordinary procedures and procedures that take delayed arguments. In general, creating separate classes of procedures forces us to create separate classes of higher-order procedures as well.<sup>139</sup>

---

<sup>139</sup>This is a small reflection, in Lisp, of the difficulties that conventional strongly typed languages such as Pascal have in coping with higher-order procedures. In such languages, the programmer must specify the data types of the arguments and the result of each procedure: number, logical value, sequence, and so on. Consequently, we could not express an abstraction such as “map a given procedure `proc` over all the elements in a sequence” by a single higher-order procedure such as `stream-map`. Rather, we would need a different mapping procedure for each different combination of argument and result data types that might be specified for a `proc`. Maintaining a

この節の例は明示的な `delay` と `force` の使用がどのようにして大きなプログラミングの柔軟性を与えるかについて説明します。しかし同じ例がまたこのことがどのようにして私達のプログラムをより複雑にするかについても示します。例として私達の新しい `integral` 手続はループを伴うシステムをモデル化するための力を与えます。しかし今では `integral` が遅延化された被積分関数と共に呼び出されなければならないことを忘れてはならなくなりました。そして `integral` を使用する全ての手続はこのことについて注意しなければなりません。実際には、手続の 2 つの組を作りました。通常の手續と遅延化された引数を取る手續です。一般的に、分離された手續の組を作ることは私達に分離された高階手續の組もまた作ることを強います。<sup>140</sup>

One way to avoid the need for two different classes of procedures is to make all procedures take delayed arguments. We could adopt a model of evaluation in which all arguments to procedures are automatically delayed and arguments are forced only when they are actually needed (for example, when they are required by a primitive operation). This would transform our language to use normal-order evaluation, which we first described when we introduced the substitution model for evaluation in Section 1.1.5. Converting to normal-order evaluation provides a uniform and elegant way to simplify the use of delayed evaluation,

---

practical notion of “data type” in the presence of higher-order procedures raises many difficult issues. One way of dealing with this problem is illustrated by the language ML (Gordon et al. 1979), whose “polymorphic data types” include templates for higher-order transformations between data types. Moreover, data types for most procedures in ML are never explicitly declared by the programmer. Instead, ML includes a *type-inferencing* mechanism that uses information in the environment to deduce the data types for newly defined procedures.

<sup>140</sup>これは Pascal の様な旧来の強い方の言語が高階手続をこなす場合に持つ困難さの (Lisp にとっては) 小さな反射です。そのような言語ではプログラマは必ず各手続の引数と結果のデータ型を指定せねばなりません。数値、論理値、配列、等です。その結果として“与えられた手続 `proc` を列の全ての要素に `map` する”ような抽象化を `stream-map` のような单一の高階手続にて表すことができませんでした。それどころか `proc` に対して指定されるかもしれない異なる引数と結果のデータ型の組み合わせ全てに対して異なるマッピング手続を必要としました。高階手続の存在における“データ型”的実用的な概念を維持することは多くの困難な問題を提起しました。この問題を処理する 1 つの方法は言語 ML(Gordon et al. 1979)により説明され、その“多層データ型”はデータ型間の高階変換のためのテンプレートを含んでいます。さらに ML のほとんどの手続データ型は明示的にプログラマにより宣言されることはありません。その代わりに ML は *type-inferencing*(型推論) メカニズムを含み、環境の情報を用いて新しく定義された手続のデータ型を推論します。

and this would be a natural strategy to adopt if we were concerned only with stream processing. In [Section 4.2](#), after we have studied the evaluator, we will see how to transform our language in just this way. Unfortunately, including delays in procedure calls wreaks havoc with our ability to design programs that depend on the order of events, such as programs that use assignment, mutate data, or perform input or output. Even the single `delay` in `cons-stream` can cause great confusion, as illustrated by [Exercise 3.51](#) and [Exercise 3.52](#). As far as anyone knows, mutability and delayed evaluation do not mix well in programming languages, and devising ways to deal with both of these at once is an active area of research.

2つの異なる手続の組の必要性を防ぐ1つの方法は全ての手続に対し遅延引数を取らせることです。手続に対する全ての引数が自動的に遅延化され、引数が実際に必要とされる時(例えばプリミティブ命令に要求された時)強制される評価のモデルを受け入れることができるでしょう。これは私達の言語を正規順評価を用いるように変形します。これは私達が[Section 1.1.5](#)の評価の置換モデルを紹介した時、最初に説明しました。正規順評価への変換は統一、洗練された方法で遅延評価の利用を簡易化します。そしてこれはストリーム処理のみについて考慮するのならば受け入れるべき自然な戦略です。[Section 4.2](#)では評価機を学んだ後に、私達の言語をどのようにしてこのように変形するのかについて学びます。残念なことに手続呼出しに遅延を導入することはイベント順に依存するプログラムを設計する能力に混乱をもたらします。例えば代入を利用する、データを変更する、入出力を実行するプログラムです。例え1つの`cons-stream`内の`delay`でも[Exercise 3.51](#)や[Exercise 3.52](#)で説明された大きな混乱を招きます。誰もが知っているように、変更可能性と遅延評価はプログラミング言語の中でうまく混ざりません。そしてこれらの両方を一度に取り扱う方法の発明は活発な研究領域です。

### 3.5.5 Modularity of Functional Programs and Modularity of Objects

As we saw in [Section 3.1.2](#), one of the major benefits of introducing assignment is that we can increase the modularity of our systems by encapsulating, or “hiding,” parts of the state of a large system within local variables. Stream models can provide an equivalent modularity without the use of assignment. As an illustration, we can reimplement the Monte Carlo estimation of  $\pi$ , which we examined in [Section 3.1.2](#), from a stream-processing point of view.

Section 3.1.2で学んだように、代入の導入の主な利点の1つは巨大システムの状態の一部をローカル変数の中にカプセル化、または“隠す”ことによりシステムのモジュール化の容易性を増すことができます。ストリームモジュールは同等なモジュールの容易性を代入の使用成しに提供可能です。例として $\pi$ のモンテカルロ推定を再実装してみましょう。Section 3.1.2にてこれをストリーム処理の視点から試しました。

The key modularity issue was that we wished to hide the internal state of a random-number generator from programs that used random numbers. We began with a procedure `rand-update`, whose successive values furnished our supply of random numbers, and used this to produce a random-number generator:

モジュール化容易性の鍵となる問題は、乱数生成器の内部状態を乱数を使用するプログラムから隠したいと願ったことです。手続 `rand-update` から始めました。これの連続する値が私達の乱数を供給し、そしてこれを乱数生成器を作り出すに使用しました。

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

In the stream formulation there is no random-number generator *per se*, just a stream of random numbers produced by successive calls to `rand-update`:

ストリームの定式化においては乱数生成器が単体では存在しません。乱数のストリームがただ `rand-update` を連続して呼ぶことで生成されます。

```
(define random-numbers
  (cons-stream random-init
               (stream-map rand-update random-numbers)))
```

We use this to construct the stream of outcomes of the Cesàro experiment performed on consecutive pairs in the `random-numbers` stream:

これを用いて乱数ストリームにおける連続したペア上で行われた Cesàro(チエザロ)の実験の結果のストリームを構築します。

```
(define cesaro-stream
  (map-successive-pairs
   (lambda (r1 r2) (= (gcd r1 r2) 1)))
```

```

    random-numbers))
(define (map-successive-pairs f s)
  (cons-stream
    (f (stream-car s) (stream-car (stream-cdr s)))
    (map-successive-pairs f (stream-cdr (stream-cdr s))))))

```

The `cesaro-stream` is now fed to a `monte-carlo` procedure, which produces a stream of estimates of probabilities. The results are then converted into a stream of estimates of  $\pi$ . This version of the program doesn't need a parameter telling how many trials to perform. Better estimates of  $\pi$  (from performing more experiments) are obtained by looking farther into the `pi` stream:

`cesaro-stream` が次に `monte-carlo` 手続に与えられます。これは確率の推測のストリームを生成します。するとその結果は  $\pi$  の推測値のストリームへと変換されます。このプログラムのこの版は何回試行を行うかのパラメタが必要ありません。より良い  $\pi$  の推測値(より多くの試行からの)はより多くの `pi` ストリームを見ることで得られます。

```

(define (monte-carlo experiment-stream passed failed)
  (define (next passed failed)
    (cons-stream
      (/ passed (+ passed failed))
      (monte-carlo
        (stream-cdr experiment-stream) passed failed)))
  (if (stream-car experiment-stream)
      (next (+ passed 1) failed)
      (next passed (+ failed 1)))))
(define pi
  (stream-map
    (lambda (p) (sqrt (/ 6 p)))
    (monte-carlo cesaro-stream 0 0)))

```

There is considerable modularity in this approach, because we still can formulate a general `monte-carlo` procedure that can deal with arbitrary experiments. Yet there is no assignment or local state.

考慮すべきモジュール化容易性がこの取り組み方には存在します。なぜなら依然として任意の実験を取り扱うことが可能な一般的な `monte-carlo` 手続を定式化できるためです。その上、代入やローカル変数が存在しません。

**Exercise 3.81:** Exercise 3.6 discussed generalizing the random-number generator to allow one to reset the random-number sequence so as to produce repeatable sequences of “random” numbers. Produce a stream formulation of this same generator that operates on an input stream of requests to `generate` a new random number or to `reset` the sequence to a specified value and that produces the desired stream of random numbers. Don’t use assignment in your solution.

Exercise 3.6は乱数生成器に乱数列のリセットを許可することで“ランダム”な数の列を繰り返し生成させる一般化について議論した。入力ストリームの要求に従い操作するこれと同じ生成器のストリーム定式化を実現せよストリームの要素が `generate` なら新しい乱数を生成し、また `reset` なら指定された値に列をリセットすることで希望の乱数列を生成する。代入は使用しないこと。

**Exercise 3.82:** Redo Exercise 3.5 on Monte Carlo integration in terms of streams. The stream version of `estimate-integral` will not have an argument telling how many trials to perform. Instead, it will produce a stream of estimates based on successively more trials.

Exercise 3.5のモンテカルロ積分をストリームを用いて再度行え。ストリーム版の `estimate-integral` は何度試行を行うのか伝える引数は持たない。その代わり連続するより多くの試行を基に推測値のストリームを生成する。

## A functional-programming view of time

Let us now return to the issues of objects and state that were raised at the beginning of this chapter and examine them in a new light. We introduced assignment and mutable objects to provide a mechanism for modular construction of programs that model systems with state. We constructed computational objects with local state variables and used assignment to modify these variables. We modeled the temporal behavior of the objects in the world by the temporal behavior of the corresponding computational objects.

さて、この章の始めに提起されたオブジェクトと状態の問題に戻り新しい光の下で調査しましょう。私達は代入とミュータブルオブジェクトを導入し状態を持つシステムのモデル化を行うプログラムのモジュラー方式の構築のため

の仕組みを提供しました。ローカル状態変数を持つ計算オブジェクトを構築し、代入を用いてこれらの変数を変更しました。世界のオブジェクトの一時的な振舞を相当する計算オブジェクトの一時的な振舞によりモデル化しました。

Now we have seen that streams provide an alternative way to model objects with local state. We can model a changing quantity, such as the local state of some object, using a stream that represents the time history of successive states. In essence, we represent time explicitly, using streams, so that we decouple time in our simulated world from the sequence of events that take place during evaluation. Indeed, because of the presence of `delay` there may be little relation between simulated time in the model and the order of events during the evaluation.

今までストリームが局所状態を持つオブジェクトのモデル化する代替法を提供することを学んできました。何らかのオブジェクトの局所状態のような変化する数量を、連続する状態の時刻歴を表現するストリームを用いてモデル化できます。本質的に、私達はストリームを用いることで時間を明示的に表現しています。そうすることで私達のシミュレートされた世界の時間を評価の間に現れる一連のイベントから分断しています。実際に、`delay` の存在のため、モデルのシミュレートされた時間と評価中のイベントの順の間には何の関係も無いでしょう。

In order to contrast these two approaches to modeling, let us reconsider the implementation of a “withdrawal processor” that monitors the balance in a bank account. In Section 3.1.3 we implemented a simplified version of such a processor:

これらの 2 つのモデル化の取り組み方を対比するために、“銀行の引き出し機”的実装について再考してみましょう。これは銀行口座の残高を監視します。Section 3.1.3ではそのような処理機の単純化された版を実装しました。

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

Calls to `make-simplified-withdraw` produce computational objects, each with a local state variable `balance` that is decremented by successive calls to the object. The object takes an `amount` as an argument and returns the new balance. We can imagine the user of a bank account typing a sequence of inputs to such an object and observing the sequence of returned values shown on a display

screen.

`make-simplified-withdraw`への呼出は計算オブジェクトを生成します。各計算オブジェクトは局所状態変数 `balance` を個別に持ち、そのオブジェクトを続けて呼ぶと `balance` は減少します。銀行口座のユーザが連續したそのようなオブジェクトに対する入力を打ち、ディスプレイの画面に表われる一連の返り値を観察するのを想像できます。

Alternatively, we can model a withdrawal processor as a procedure that takes as input a balance and a stream of amounts to withdraw and produces the stream of successive balances in the account:

代替的に、引き出し処理機を入力として残高と引き出す金額のストリームを取り口座の一連の残高のストリームを生成する手続としてモデル化することができます。

```
(define (stream-withdraw balance amount-stream)
  (cons-stream
    balance
    (stream-withdraw (- balance (stream-car amount-stream))
                     (stream-cdr amount-stream))))
```

`Stream-withdraw` implements a well-defined mathematical function whose output is fully determined by its input. Suppose, however, that the input `amount-stream` is the stream of successive values typed by the user and that the resulting stream of balances is displayed. Then, from the perspective of the user who is typing values and watching results, the stream process has the same behavior as the object created by `make-simplified-withdraw`. However, with the stream version, there is no assignment, no local state variable, and consequently none of the theoretical difficulties that we encountered in [Section 3.1.3](#). Yet the system has state!

`stream-withdraw` は明確な数学上の関数を実装し、関数の出力はその入力のみにより完全に決定します。しかし入力 `amount-stream` がユーザにより打鍵された一連の値のストリームであり結果の残高ストリームが表示されたと考えてみて下さい。すると、値を入力し結果を見ているユーザの視点からはストリーム処理が `make-simplified-withdraw` により作成されたかのように、同じ振舞をしています。しかしそストリーム版では代入が無く、局所状態変数が無く、それ故に [Section 3.1.3](#)で遭遇した論理的な困難に存在しません。それにもかかわらずシステムは状態を持っています！

This is really remarkable. Even though `stream-withdraw` implements a

well-defined mathematical function whose behavior does not change, the user's perception here is one of interacting with a system that has a changing state. One way to resolve this paradox is to realize that it is the user's temporal existence that imposes state on the system. If the user could step back from the interaction and think in terms of streams of balances rather than individual transactions, the system would appear stateless.<sup>141</sup>

これは本当に驚くべきことです。`stream-withdraw` は明確な数学上の関数を実装しその振舞は代わらないのに、ここでのユーザの知覚はシステムとの相互作用の 1 つであり変化する状態を持ちます。このパラドックスを解決する 1 つの方法はユーザの一時的な存在がシステムに状態を与えると認識することです。もしユーザが相互作用から一歩離れて個々の取引ではなく、残高のストリームに関して考えれば、システムはステートレス（状態が無く）として現われるでしょう。<sup>142</sup>

From the point of view of one part of a complex process, the other parts appear to change with time. They have hidden time-varying local state. If we wish to write programs that model this kind of natural decomposition in our world (as we see it from our viewpoint as a part of that world) with structures in our computer, we make computational objects that are not functional—they must change with time. We model state with local state variables, and we model the changes of state with assignments to those variables. By doing this we make the time of execution of a computation model time in the world that we are part of, and thus we get “objects” in our computer.

複雑な処理のある部分の視点からは、他の部分は時間と共に変化するよう見えます。それらは隠された時間と共に変化する局所状態を持ちます。もし私達がこの種の自然な分解を私達の世界において（世界の一部である私達の視点から見たままに）計算機内のモデル化したプログラムを計算機内の構造を用いて書きたいのならば、関数型でない計算オブジェクトを作成します。それらは時間と共に変化します。状態を局所状態変数でモデル化し、そして状態の変化をそれらの変数への代入を用いてモデル化します。これを行うことにより計算の実行時間を、私達がその一部である世界の時間のモデルとし、従って私達は“オブジェクト”を計算機の中に得ることになります。

---

<sup>141</sup>Similarly in physics, when we observe a moving particle, we say that the position (state) of the particle is changing. However, from the perspective of the particle's world line in space-time there is no change involved.

<sup>142</sup>物理でも同様に、私達が移動点を観察する時、点の位置（状態）は変化していると言えます。しかし、移動点の時空の世界線の視点からは何の変化も起こってはいません。

Modeling with objects is powerful and intuitive, largely because this matches the perception of interacting with a world of which we are part. However, as we've seen repeatedly throughout this chapter, these models raise thorny problems of constraining the order of events and of synchronizing multiple processes. The possibility of avoiding these problems has stimulated the development of *functional programming languages*, which do not include any provision for assignment or mutable data. In such a language, all procedures implement well-defined mathematical functions of their arguments, whose behavior does not change. The functional approach is extremely attractive for dealing with concurrent systems.<sup>143</sup>

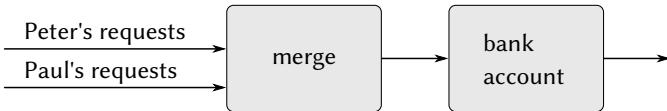
オブジェクトを用いるモデリングは強力、かつ直感的です。その理由の多くはこれが私達がその一部である世界との相互作用の視点に合うためです。しかしこの章を通して繰り返し学んできた様に、これらのモデルは悩ましいイベント順の制約と複数の処理間の同期の問題を提起します。これらの問題を防ぐ可能性から*functional programming languages*(関数型プログラミング言語)の開発が促進されてきました。これは代入や変更可能なデータを提供しません。そのような言語では全ての手続は引数の明確な数学的関数を実装し、その振舞は変化しません。関数型の取り組み方は並行システムを扱うのにとても魅力的です。<sup>144</sup>

On the other hand, if we look closely, we can see time-related problems creeping into functional models as well. One particularly troublesome area arises when we wish to design interactive systems, especially ones that model interactions between independent entities. For instance, consider once more the implementation a banking system that permits joint bank accounts. In a conventional system using assignment and objects, we would model the fact that Peter and Paul share an account by having both Peter and Paul send their transaction requests to the same bank-account object, as we saw in Section 3.1.3. From the stream point of view, where there are no "objects" *per se*, we have already indicated that a bank account can be modeled as a process that

---

<sup>143</sup>John Backus, the inventor of Fortran, gave high visibility to functional programming when he was awarded the ACM Turing award in 1978. His acceptance speech (Backus 1978) strongly advocated the functional approach. A good overview of functional programming is given in Henderson 1980 and in Darlington et al. 1982.

<sup>144</sup>Fortran の開発者である John Backus は 1978 年に ACM のチューリング賞を授与された時に関数型プログラミングに高い知名度を与えました。彼の受賞スピーチ (Backus 1978) は関数型のアプローチを強く支持しました。関数型プログラミングの良い概観はHenderson 1980とDarlington et al. 1982で与えられます。



**Figure 3.38:** A joint bank account, modeled by merging two streams of transaction requests.

operates on a stream of transaction requests to produce a stream of responses. Accordingly, we could model the fact that Peter and Paul have a joint bank account by merging Peter’s stream of transaction requests with Paul’s stream of requests and feeding the result to the bank-account stream process, as shown in Figure 3.38.

一方で、もしきっちりと見てみれば時間に関係する問題が関数型のモデルにも潜んでいることが見えます。ある特に厄介な領域がインタラクティブシステム(応答システム)を設計したい時に、特に独立した要素の間で相互作用を行うシステムにおいて提起されます。例として、もう1度連結銀行口座を許可する銀行システムについて考えてみましょう。代入とオブジェクトを用いる保守的なシステムでは、PeterとPaulが口座を共有しているという事実をモデル化します。共有はSection 3.1.3で見たように、PeterとPaulの両者が彼等の取引要求を同じ銀行口座オブジェクトに送ることにより行われます。ストリームの視点からは、“オブジェクト”それ自身は無いため、銀行口座を取引要求の操作を行う処理としてモデル化し、応答のストリームを生成できることが既に示されています。従って、PeterとPaulが連結銀行口座を持つことが、Figure 3.38で示すようにPeterの取引要求ストリームとPaulの取引要求リクエストをマージし、その結果を銀行口座ストリーム処理へ渡すという事実をモデル化することができるでしょう。

The trouble with this formulation is in the notion of *merge*. It will not do to merge the two streams by simply taking alternately one request from Peter and one request from Paul. Suppose Paul accesses the account only very rarely. We could hardly force Peter to wait for Paul to access the account before he could issue a second transaction. However such a merge is implemented, it must interleave the two transaction streams in some way that is constrained by “real time” as perceived by Peter and Paul, in the sense that, if Peter and Paul meet, they can agree that certain transactions were processed before the meeting, and

other transactions were processed after the meeting.<sup>145</sup> This is precisely the same constraint that we had to deal with in [Section 3.4.1](#), where we found the need to introduce explicit synchronization to ensure a “correct” order of events in concurrent processing of objects with state. Thus, in an attempt to support the functional style, the need to merge inputs from different agents reintroduces the same problems that the functional style was meant to eliminate.

この定式化に伴う問題は *merge*(マージ) という概念にあります。これは 2 つのストリームを単純に交互に Peter の要求を 1 つ、Paul の要求を 1 つと取りマージすることはしません。Paul が口座にとても稀にしかアクセスしないと考えてみましょう。Peter に対し、彼が 2 つ目の取引を発行できる前に Paul が口座にアクセスするのを待つよう強いることはできません。しかしそのようなマージが実装された場合、Peter と Paul により知覚される “実時間” に制約された何らかの方法で 2 つの取引ストリームを相互配置しなければなりません。何らかの方法とはもし Peter と Paul が会えば、いくつかの取引が会う前に処理され、他の取引が会った後に処理されることに合意できるという意味です。<sup>146</sup> これは正確に、[Section 3.4.1](#)で扱わねばならなかったのと同じ制約です。そこでは状態を持つオブジェクトの並行処理におけるイベントの “正しい” 順を保証する明示的な同期の導入の必要性が見つかりました。従って、関数型のスタイルをサポートする取り組みにおいて、異なる要因からの入力のマージの必要性は関数型のスタイルが排除するはずだった同じ問題を再び導入します。

We began this chapter with the goal of building computational models whose structure matches our perception of the real world we are trying to model. We can model the world as a collection of separate, time-bound, interacting objects with state, or we can model the world as a single, timeless, stateless unity. Each view has powerful advantages, but neither view alone is

---

<sup>145</sup>Observe that, for any two streams, there is in general more than one acceptable order of interleaving. Thus, technically, “merge” is a relation rather than a function—the answer is not a deterministic function of the inputs. We already mentioned ([Footnote 39](#)) that nondeterminism is essential when dealing with concurrency. The merge relation illustrates the same essential nondeterminism, from the functional perspective. In [Section 4.3](#), we will look at nondeterminism from yet another point of view.

<sup>146</sup>任意の 2 つのストリームに対し一般的に複数の受け入れ可能な相互配置の順が存在することに注意して下さい。従って技術的には “マージ” は関数ではなく関係です。その答は入力の決定的な関数ではありません。私達は既に ([Footnote 39](#)) で非決定論が並行の扱いの本質であると述べました。マージの関係は同じ本質的な非決定論を関数型の視点から説明します。[Section 4.3](#) では非決定論をまた別の視点から見ることになります。

completely satisfactory. A grand unification has yet to emerge.<sup>147</sup>

私達はこの章を、私達がモデル化しようとする実際の世界の私達の視点に合う構造を持つ計算モデルの構築をゴールとして始めました。世界を分離した、時間制約のある、相互応答する、状態を持つオブジェクトでモデル化できます。または世界を単一の、時間制約のない、状態の無い、個体によりモデル化できます。それぞれの視点が強力な利点を持ちますが、どちらの視点も単独では十分ではありません。大統一は未だ現われてはいません。<sup>148</sup>

---

<sup>147</sup>The object model approximates the world by dividing it into separate pieces. The functional model does not modularize along object boundaries. The object model is useful when the unshared state of the “objects” is much larger than the state that they share. An example of a place where the object viewpoint fails is quantum mechanics, where thinking of things as individual particles leads to paradoxes and confusions. Unifying the object view with the functional view may have little to do with programming, but rather with fundamental epistemological issues.

<sup>148</sup>オブジェクトモデルは世界を分割し分離した部分にすることで近似します。関数型モデルはオブジェクト境界に従ってモジュラ化はしません。オブジェクトモデルは“オブジェクト”的非共有下の状態が共有されている状態よりもとても大きい場合に便利です。オブジェクトの視点が失敗する場合の例は量子力学です。そこでは物を個別の点として考えることは逆説と混乱を招きます。オブジェクトの視点を関数型の視点と統一することはプログラミングとはあまり関係が無いかもしれません。しかしより根本的な認識論の問題と関係するのです。

# 4

## Metalinguistic Abstraction

... It's in words that the magic is—Abracadabra, Open Sesame, and the rest—but the magic words in one story aren't magical in the next. The real magic is to understand which words work, and when, and for what; the trick is to learn the trick.

... 魔法とは言葉の中にある—アブラカダブラ、開けゴマ、その他 もろもろ—しかしあるお話の魔法の言葉は次のお話では魔法では ない。真の魔法とはどの言葉が、いつ、何に対して働くかを知るこ とだ。トリックを学ぶことがトリックなんだ。

... And those words are made from the letters of our alphabet: a couple-dozen squiggles we can draw with the pen. This is the key! And the treasure, too, if we can only get our hands on it! It's as if—as if the key to the treasure *is* the treasure!

... そしてそれらの言葉は私達のアルファベットの文字から出来て いる。ペンで書ける 2, 3 ダースの走り書きだ。これが鍵なんだ！ そして宝もある、もしそれに手をつけることさえできれば！ そ れはまるで—まるで宝の鍵こそが宝のようだ！

—John Barth, *Chimera*

In our study of program design, we have seen that expert programmers control the complexity of their designs with the same general techniques used by

designers of all complex systems. They combine primitive elements to form compound objects, they abstract compound objects to form higher-level building blocks, and they preserve modularity by adopting appropriate large-scale views of system structure. In illustrating these techniques, we have used Lisp as a language for describing processes and for constructing computational data objects and processes to model complex phenomena in the real world. However, as we confront increasingly complex problems, we will find that Lisp, or indeed any fixed programming language, is not sufficient for our needs. We must constantly turn to new languages in order to express our ideas more effectively. Establishing new languages is a powerful strategy for controlling complexity in engineering design; we can often enhance our ability to deal with a complex problem by adopting a new language that enables us to describe (and hence to think about) the problem in a different way, using primitives, means of combination, and means of abstraction that are particularly well suited to the problem at hand.<sup>1</sup>

プログラム設計の学習において、エキスパートなプログラマが設計の複雑さを全ての複雑なシステムの設計者が用いるのと同じ一般的な技術を用いてコントロールすることを学んできました。彼等はプリミティブな要素を接続して複合オブジェクトを形成し、複合オブジェクトを抽象化することでより高いレベルの建築ブロックを形成しそして適切な大規模のシステム構造の見方を受け入れることでモジュール化方式は維持しました。これらのテクニックの説明に

<sup>1</sup>The same idea is pervasive throughout all of engineering. For example, electrical engineers use many different languages for describing circuits. Two of these are the language of electrical *networks* and the language of electrical *systems*. The network language emphasizes the physical modeling of devices in terms of discrete electrical elements. The primitive objects of the network language are primitive electrical components such as resistors, capacitors, inductors, and transistors, which are characterized in terms of physical variables called voltage and current. When describing circuits in the network language, the engineer is concerned with the physical characteristics of a design. In contrast, the primitive objects of the system language are signal-processing modules such as filters and amplifiers. Only the functional behavior of the modules is relevant, and signals are manipulated without concern for their physical realization as voltages and currents. The system language is erected on the network language, in the sense that the elements of signal-processing systems are constructed from electrical networks. Here, however, the concerns are with the large-scale organization of electrical devices to solve a given application problem; the physical feasibility of the parts is assumed. This layered collection of languages is another example of the stratified design technique illustrated by the picture language of Section 2.2.4.

おいて、私達は Lisp をプロセスを記述するための言語として用い、また計算データオブジェクトと実世界の複雑な現象をモデル化する処理を構築するためにも用いてきました。しかし、複雑さを増す問題に取り組むにつれ、Lisp、またはどのような固定されたプログラミング言語も、我々の必要には十分でないことを知ることでしょう。私達は、私達の考えをより効果的に表現するために、耐えず新しい言語に向かわねばなりません。新しい言語を定めることは工学上の設計の複雑さをコントロールするための強力な戦略です。私達は良く、問題を異なった方法で記述できる（そしてそれ故に考えることができる）新しい言語を受け入れることで、複雑な問題への対処能力を拡張することができます。プリミティブな、組み合わせの手段や抽象化の手段を、目前の問題に特によく合つたものを用います。<sup>2</sup>

Programming is endowed with a multitude of languages. There are physical languages, such as the machine languages for particular computers. These languages are concerned with the representation of data and control in terms of individual bits of storage and primitive machine instructions. The machine-language programmer is concerned with using the given hardware to erect systems and utilities for the efficient implementation of resource-limited computations. High-level languages, erected on a machine-language substrate, hide concerns about the representation of data as collections of bits and the representation of programs as sequences of primitive instructions. These languages have means of combination and abstraction, such as procedure definition, that are appropriate to the larger-scale organization of systems.

プログラミングは数多くの言語により生じます。特定のコンピュータのた

<sup>2</sup>同じ考えが工学全てに渡り普及しています。例えば電子工学は多くの異なる言語を回路の記述に用います。これらの内 2 つは電子ネットワークの言語と電子システムの言語です。ネットワーク言語は別個の電子素子に関する装置の物理モデリングを重視します。ネットワーク言語のプリミティブなオブジェクトはプリミティブな抵抗や、キャパシタ（コンデンサ）、コイルやトランジスタ等の電子コンポーネントであり電圧と電流と呼ばれる物理的変数を用いて特徴付けられます。回路をネットワーク言語で記述する時、技術者は設計の物理特性に関心を持ちます。逆に、システム言語のプリミティブなオブジェクトはフィルタやアンプのような信号処理モジュールです。モジュールの機能上の振舞のみが関係し、信号はそれらの電圧や電流のような物理的な認識に関心を持ちません。信号処理システムの要素が電子ネットワークから構築される意味の上ではシステム言語はネットワーク言語の上に組み立てられます。しかしここでは関心事は与えられた応用問題を解くための大規模な電子装置の編成にあります。パートの物理的実現可能性は当然と考えられています。この階層化された言語の集合は Section 2.2.4 のピクチャーランゲージにて説明された階層化された設計テクニックのまた別の例になります。

めの機械語のような物理言語も存在します。これらの言語は個別のストレージの断片とプリミティブな機械命令を用いてデータとコントロールの表現に関係します。機械語プログラマは与えられたハードウェアの使用に関心を持つことでリソースに限りある演算の効率的な実装のためのシステムとユーティリティを組み立てます。高級言語は機械語の素地の上にありますが、データをビットの集合として表したり、プログラムをプリミティブな命令の列で表すという懸念を隠します。これらの言語は手続定義のような組み合わせと抽象化の手段を持ち大規模なシステム構成に適しています。

*Metalinguistic abstraction*—establishing new languages—plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing evaluators. An *evaluator* (or *interpreter*) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression.

*Metalinguistic abstraction*(メタ言語抽象化)—新しい言語を構築すること—が工学設計の全ての部門にて重要な役割を果たします。これは計算機プログラミングでとても重要です。プログラミングでは新しい言語を形成するだけでなく、これらの言語を評価機を構築することで実装することもできるからです。プログラミング言語の *evaluator*(評価機)(または *interpreter*(インタプリタ)) は手続であり、言語の式に対して適用された時、その式を評価するために要求される行動を実行します。

It is no exaggeration to regard this as the most fundamental idea in programming:

プログラミングにおける考え方で最も根本的な物と見做すことに何の誇張もありません。

The evaluator, which determines the meaning of expressions in a programming language, is just another program.

評価機はプログラミング言語の評価手段を決定するが、それ自体は別のプログラムである。

To appreciate this point is to change our images of ourselves as programmers. We come to see ourselves as designers of languages, rather than only users of languages designed by others.

この点を理解することはプログラマとしての私達自身のイメージを変更することです。私達は私達自身を、他人が設計した言語のユーザとしてのみではなく、言語の設計者として見る時点に辿りつきました。

In fact, we can regard almost any program as the evaluator for some language. For instance, the polynomial manipulation system of Section 2.5.3 embodies the rules of polynomial arithmetic and implements them in terms of operations on list-structured data. If we augment this system with procedures to read and print polynomial expressions, we have the core of a special-purpose language for dealing with problems in symbolic mathematics. The digital-logic simulator of Section 3.3.4 and the constraint propagator of Section 3.3.5 are legitimate languages in their own right, each with its own primitives, means of combination, and means of abstraction. Seen from this perspective, the technology for coping with large-scale computer systems merges with the technology for building new computer languages, and computer science itself becomes no more (and no less) than the discipline of constructing appropriate descriptive languages.

実際に、私達はほとんど全てのプログラムをある言語の評価機だと見做すことができます。例えば、Section 2.5.3の多項式操作システムは多項式の数値演算のルールを具象化し、リスト構造データ上の命令を用いて実装しました。もし私達がこのシステムを多項式を読み込み、表示する手続と共に拡張したなら、記号数学の問題を扱う特定目的言語のコア(核)を持つことになります。Section 3.3.4のデジタル論理シミュレータとSection 3.3.5の制約伝播はそれら自身の正しさにおける論理的言語であり、それぞれがそれ自身のプリミティブと組み合わせの手段、抽象化的手段を持ちます。この視点から見れば大規模計算機システムをこなす技術は新しい計算機言語を構築する技術と結合し、計算機科学それ自身が適切な記述言語を構築する分野それ以上でも以下でもなくなります。

We now embark on a tour of the technology by which languages are established in terms of other languages. In this chapter we shall use Lisp as a base, implementing evaluators as Lisp procedures. Lisp is particularly well suited to this task, because of its ability to represent and manipulate symbolic expressions. We will take the first step in understanding how languages are implemented by building an evaluator for Lisp itself. The language implemented by our evaluator will be a subset of the Scheme dialect of Lisp that we use in this book. Although the evaluator described in this chapter is written for a particular dialect of Lisp, it contains the essential structure of an evaluator for any expression-oriented language designed for writing programs for a sequential machine. (In fact, most language processors contain, deep within them, a little “Lisp” evaluator.) The evaluator has been simplified for the purposes of

illustration and discussion, and some features have been left out that would be important to include in a production-quality Lisp system. Nevertheless, this simple evaluator is adequate to execute most of the programs in this book.<sup>3</sup>

私達は今から他言語を用いて言語が構築される技術を巡り始めます。この章では Lisp を基盤として用い、評価機を Lisp の手続として実装します。Lisp はこの任務にとても良く合います。記号式を表現し、操作する能力がその理由です。私達は Lisp 自身の評価機を構築することで言語がどのように実装されているかを理解することから最初の一歩を踏み出します。私達の評価機により実装される言語はこの本で用いる Lips の Scheme 方言の部分集合となります。この章で説明される評価機が Lisp の特定の方言に向けて書かれていても、逐次式計算機のプログラムを書くために設計された任意の式指向言語のための評価機の本質的な構造を含みます。(実際に、多くの言語処理機がそれらの奥深くに小さな “Lisp” 評価機を含んでいるのです。) 評価機は説明と議論のために簡略化されており、製品品質の Lisp システムに含まれるべき重要な機能が省略されています。それにもかかわらず、この単純な評価機はこの本に現われる多くのプログラムを実行するのに適しています。<sup>4</sup>

An important advantage of making the evaluator accessible as a Lisp program is that we can implement alternative evaluation rules by describing these as modifications to the evaluator program. One place where we can use this power to good effect is to gain extra control over the ways in which computational models embody the notion of time, which was so central to the discussion in [Chapter 3](#). There, we mitigated some of the complexities of state and assignment by using streams to decouple the representation of time in the world from time in the computer. Our stream programs, however, were sometimes cumbersome, because they were constrained by the applicative-order evaluation of Scheme. In [Section 4.2](#), we'll change the underlying language to provide for a more elegant approach, by modifying the evaluator to provide for *normal-order evaluation*.

評価機を Lisp プログラムとして利用可能にする重要な利点は代替となる評

---

<sup>3</sup>The most important features that our evaluator leaves out are mechanisms for handling errors and supporting debugging. For a more extensive discussion of evaluators, see [Friedman et al. 1992](#), which gives an exposition of programming languages that proceeds via a sequence of evaluators written in Scheme.

<sup>4</sup>私達の評価機が取り除いた最も重要な機能はエラーを扱う仕組みとデバッグのサポートです。評価機のより広範囲の議論については[Friedman et al. 1992](#)を参照して下さい。これは Scheme で書かれた一連の評価機を通して進められたプログラミング言語の解説を与えます。

価ルールを評価機プログラムへの変更として記述することで実装できます。この力を良い効果として用いることが可能な箇所として、Chapter 3の議論のまさに中心であった、計算モデルが時間の概念を統合する方法に対し特別なコントロールを得ることができます。そこではストリームを用いて世界の時間表現を計算機の時間から分離することで、状態と代入の複雑さのいくらかを緩和しました。しかし、私達のストリームプログラムは時々扱いにくい物でした。Schemeの評価の適用順により制約されていたためです。Section 4.2ではより洗練された取り組み方を準備するために、*normal-order evaluation*(正規順評価)に対応する様に評価機を変更することで基盤となる言語を変更します。

Section 4.3 implements a more ambitious linguistic change, whereby expressions have many values, rather than just a single value. In this language of *nondeterministic computing*, it is natural to express processes that generate all possible values for expressions and then search for those values that satisfy certain constraints. In terms of models of computation and time, this is like having time branch into a set of “possible futures” and then searching for appropriate time lines. With our nondeterministic evaluator, keeping track of multiple values and performing searches are handled automatically by the underlying mechanism of the language.

Section 4.3では式が単一の値のみでなく多くの値を持つ場合において、より野心的な言語の変更を実装します。この*nondeterministic computing*(非決定的演算)の言語においては、式の全ての可能な値を生成する過程を生成し、次にそれらの値からいくつかの制約を充足する値を探索することが自然に表現できます。計算と時間のモデルに用いれば、これは“可能な未来”的な集合を成す時間の分岐を持ち、次に適切な時系列を探すような物です。私達の非決定的評価機を用いる複数の値の追跡と探索の実行は、根底に存在する言語の仕組みにより自動的に取り扱われます。

In Section 4.4 we implement a *logic-programming* language in which knowledge is expressed in terms of relations, rather than in terms of computations with inputs and outputs. Even though this makes the language drastically different from Lisp, or indeed from any conventional language, we will see that the logic-programming evaluator shares the essential structure of the Lisp evaluator.

Section 4.4では*logic-programming*(論理プログラミング)言語を実装します。それにより知識が入出力を伴なう計算を用いてではなく、関係性を用いて表現されます。これは言語を Lisp から、または本当に全ての従来の言語から大幅に異なる物にしますが、論理プログラミング評価機が Lisp 評価機の本質的な構

造を共有することを学びます。

## 4.1 The Metacircular Evaluator

Our evaluator for Lisp will be implemented as a Lisp program. It may seem circular to think about evaluating Lisp programs using an evaluator that is itself implemented in Lisp. However, evaluation is a process, so it is appropriate to describe the evaluation process using Lisp, which, after all, is our tool for describing processes.<sup>5</sup> An evaluator that is written in the same language that it evaluates is said to be *metacircular*.

私達の Lisp 評価機は Lisp プログラムとして実装されます。Lisp プログラムを Lisp で実装された評価機を用いて評価することについて考えることは循環論に見えるかもしれません。しかし評価はプロセス(処理、過程)であり、從って評価過程を Lisp を用いて説明することは適切です。Lisp は結局の所、プロセスを記述するためのツールなのです。<sup>6</sup>評価する対象と同じ言語で書かれた評価機は *metacircular*(メタ循環)と呼ばれます。

The metacircular evaluator is essentially a Scheme formulation of the environment model of evaluation described in [Section 3.2](#). Recall that the model has two basic parts:

メタ循環評価機は本質的には[Section 3.2](#)で説明された評価の環境モデルの Scheme 形式化です。モデルには以下の 2 つの基本的パートがあることを思い出して下さい

1. To evaluate a combination (a compound expression other than a special form), evaluate the subexpressions and then apply the value of the operator subexpression to the values of the operand subexpressions.

---

<sup>5</sup>Even so, there will remain important aspects of the evaluation process that are not elucidated by our evaluator. The most important of these are the detailed mechanisms by which procedures call other procedures and return values to their callers. We will address these issues in [Chapter 5](#), where we take a closer look at the evaluation process by implementing the evaluator as a simple register machine.

<sup>6</sup>例えそうだとしても、私達の評価機により説明されない評価プロセスの重要な側面が残ります。これらの最も重要なことは手続が他の手続を呼び出し、そしてそれらを呼び出した物に値を返す原因となる詳細な仕組みです。これらの問題は[Chapter 5](#)で解明します。そこで私達は評価機を簡単なレジスタマシンとして実装することで評価プロセスにより詳細に調べます。

組み合わせ(特殊な形式を除く複合式)を評価するためには、部分式を評価し、次にオペレータ部分式をオペランド部分式の値に適用する。

2. To apply a compound procedure to a set of arguments, evaluate the body of the procedure in a new environment. To construct this environment, extend the environment part of the procedure object by a frame in which the formal parameters of the procedure are bound to the arguments to which the procedure is applied.

複合手続を引数の集合に適用するためには、手続のボディを新しい環境で評価する。この環境を構築するためには、手続オブジェクトの環境部分をフレームにより拡張する。フレームの中ではその手続の形式パラメタが、その手続が適用される引数に対して束縛される

These two rules describe the essence of the evaluation process, a basic cycle in which expressions to be evaluated in environments are reduced to procedures to be applied to arguments, which in turn are reduced to new expressions to be evaluated in new environments, and so on, until we get down to symbols, whose values are looked up in the environment, and to primitive procedures, which are applied directly (see [Figure 4.1](#)).

これら2つのルールが評価プロセスの本質を説明します。環境の中で式が評価される基本的なサイクルは引数に適用される手続に簡約され、引数は順に新しい環境で評価される新しい式へと簡約され、以下、値がその環境の中で見つかるシンボルか直接適用されるプリミティブな手続([Figure 4.1](#)参照)に辿り着くまで繰り返されます。<sup>7</sup>

---

<sup>7</sup>If we grant ourselves the ability to apply primitives, then what remains for us to implement in the evaluator? The job of the evaluator is not to specify the primitives of the language, but rather to provide the connective tissue—the means of combination and the means of abstraction—that binds a collection of primitives to form a language. Specifically:

もし私達自身にプリミティブを適用する能力を与えるのであれば、評価機の実装には何が残っているのでしょうか? 評価機の仕事は言語のプリミティブを指定することではなく、結合組織—組み合わせと抽象化の手段—を提供することであり、それがプリミティブの集合を言語を形成するために束縛します。具体的には、

- The evaluator enables us to deal with nested expressions. For example, although simply applying primitives would suffice for evaluating the expression  $(+ 1 6)$ , it is not adequate for handling  $(+ 1 (* 2 3))$ . As far as the primitive procedure `+` is concerned, its arguments must be numbers, and it would choke if we passed it the expression  $(* 2 3)$  as an argument. One important role of the evaluator is to choreograph procedure composition so that  $(* 2 3)$  is reduced to 6 before being passed as an argument to `+`.

This evaluation cycle will be embodied by the interplay between the two critical procedures in the evaluator, `eval` and `apply`, which are described in Section 4.1.1 (see Figure 4.1).

この評価サイクルは評価機内の 2 つの重大な手続、`eval` と `apply` の間の相互作用により具体化されます。これらの手続はSection 4.1.1にて説明されます。(Figure 4.1参照)

The implementation of the evaluator will depend upon procedures that define the *syntax* of the expressions to be evaluated. We will use data abstraction to make the evaluator independent of the representation of the language. For example, rather than committing to a choice that an assignment is to be represented by a list beginning with the symbol `set!` we use an abstract predicate `assignment?` to test for an assignment, and we use abstract selectors `assignment-variable` and `assignment-value` to access the parts of an assignment. Implementation of expressions will be described in detail in Section 4.1.2. There are also operations, described in Section 4.1.3, that specify the representation of procedures and environments. For example, `make-procedure`

---

- 評価機は入れ子の式の取扱を許可します。例えば単純にプリミティブを適用することは式  $(+ 1 6)$  を評価するのに十分ですが、 $(+ 1 (* 2 3))$  を取り扱うには十分ではありません。プリミティブな手続 `+` が対象である限り、その引数は数値でなければならず、もし式  $(* 2 3)$  を引数として渡せば失敗します。評価機の重要な役割の 1 つは手続合成を演出することで、 $(* 2 3)$  を `+` に引数として渡す前に 6 に簡約します。

- The evaluator allows us to use variables. For example, the primitive procedure for addition has no way to deal with expressions such as  $(+ x 1)$ . We need an evaluator to keep track of variables and obtain their values before invoking the primitive procedures.

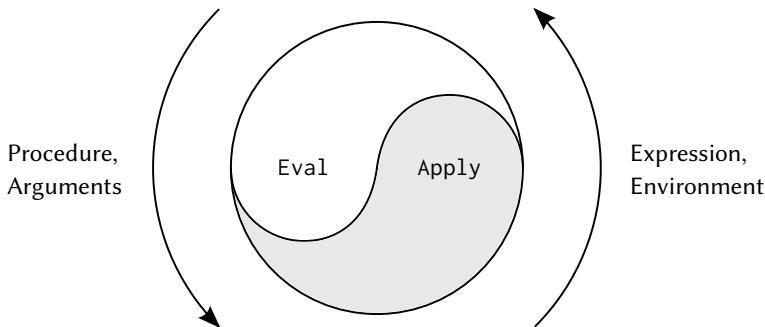
- 評価機は変数の使用を許可します。例えば加算のためのプリミティブな手続は  $(+ x 1)$  のような式に対応する手段を持ちません。私達は評価機に変数を追跡しその値をプリミティブな手続を実行する前に得るようにする必要があります。

- The evaluator allows us to define compound procedures. This involves keeping track of procedure definitions, knowing how to use these definitions in evaluating expressions, and providing a mechanism that enables procedures to accept arguments.

- 評価機は複合手続の定義を許可します。これは手続定義の追跡を含み、これらの手続を式評価においてどのように使用するかを知っています。そして手続に引数を受け入れることを許可する仕組みを提供します。

- The evaluator provides the special forms, which must be evaluated differently from procedure calls.

- 評価機は特殊形式を提供します。これは手続呼出と異なった形で評価されねばなりません。



**Figure 4.1:** The eval-apply cycle exposes the essence of a computer language.

constructs compound procedures, `lookup-variable-value` accesses the values of variables, and `apply-primitive-procedure` applies a primitive procedure to a given list of arguments.

評価機の実装は評価される式の *syntax*(構文)を定義する手続に依存します。私達はデータ抽象化を用いて評価機を言語の表現に非依存にします。例えば代入はシンボル `set!` で始まるリストにより表現されるべきという選択に委ねるのではなく、代入のためのテストに抽象述語 `assignment?` を用い、そして代入の部品にアクセスするために抽象セレクタ `assignment-variable` と `assignment-value` を用います。式の実装については [Section 4.1.2](#) で詳細に説明されます。また [Section 4.1.3](#) で説明される“命令”もあり、これは手続と環境の表現を指定します。例えば `make-procedure` は複合手続を構築し、`lookup-variable-value` は変数の値にアクセスし、`apply-primitive-procedure` はプリミティブな手續を与えられた引数のリストに対し適用します。

#### 4.1.1 The Core of the Evaluator

The evaluation process can be described as the interplay between two procedures: `eval` and `apply`.

評価プロセスは 2 つの手続 `eval` と `apply` の相互作用であると説明可能です。

## **Eval**

`Eval` takes as arguments an expression and an environment. It classifies the expression and directs its evaluation. `Eval` is structured as a case analysis of the syntactic type of the expression to be evaluated. In order to keep the procedure general, we express the determination of the type of an expression abstractly, making no commitment to any particular representation for the various types of expressions. Each type of expression has a predicate that tests for it and an abstract means for selecting its parts. This *abstract syntax* makes it easy to see how we can change the syntax of the language by using the same evaluator, but with a different collection of syntax procedures.

`eval` は引数として式と環境を取ります。これは式と分類しその評価を監督します。`eval` は評価される式の構文上の方の事例分析として構造化されます。手続的一般性を保つため、式の型の決定を抽象的に表現し、多種の式に対するどんな特定の表現にも委託しません。式の各型はそれをテストする述語と、その部分を選択する抽象手段を持ちます。この*abstract syntax*(抽象構文) は同じ評価機を用いつつ、異なる構文手続の集合と合わせることで、言語の文法をどのようにして変更できるかについて知ることを簡単にします。

### **Primitive expressions(プリミティブな式)**

- For self-evaluating expressions, such as numbers, `eval` returns the expression itself.  
数値のような自己評価式に対しては `eval` は式それ自身を返す。
- `Eval` must look up variables in the environment to find their values.  
`eval` は環境の中で変数をその値を見つけるために探さなければならぬ。

### **Special forms(特殊形式)**

- For quoted expressions, `eval` returns the expression that was quoted.  
クオートされた式に対しては `eval` はクオートされた式を返す。
- An assignment to (or a definition of) a variable must recursively call `eval` to compute the new value to be associated with the variable. The environment must be modified to change (or create) the binding of the variable.  
変数への代入(または定義)は再帰的に `eval` を呼び出し変数に関連付けられる新しい値を計算しなければならない。環境は変数の束縛を変更(または作成)しなければならない。

- An `if` expression requires special processing of its parts, so as to evaluate the consequent if the predicate is true, and otherwise to evaluate the alternative.  
`if` 式はその部品に対し特別な処理を要求する。もし述語が真であれば consequent(結果) を評価し、そうでなければ alternative(代替) を評価するためである。
- A `lambda` expression must be transformed into an applicable procedure by packaging together the parameters and body specified by the `lambda` expression with the environment of the evaluation.  
`lambda`(ラムダ) 式は適用可能な手続に変形しなければならない。変形は ラムダ式により指定されたパラメタとボディを評価の環境と共にパッケージ化することにより行う。
- A `begin` expression requires evaluating its sequence of expressions in the order in which they appear.  
`begin` 式はその一連の式をそれらが現れる順で評価する必要がある。
- A case analysis (`cond`) is transformed into a nest of `if` expressions and then evaluated.  
事例分析 (`cond`) は入れ子の `if` 式に変形し、それから評価する。

### Combinations(組み合わせ)

- For a procedure application, `eval` must recursively evaluate the operator part and the operands of the combination. The resulting procedure and arguments are passed to `apply`, which handles the actual procedure application.  
手続の適用に対して、`eval` は再帰的に組み合わせの演算子とオペランドの部分を評価しなければならない。結果となる手続と引数は `apply` に渡す。これは実際の手続適用を取り扱う。

Here is the definition of `eval`:

以下に `eval` の定義を示します。

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
```

```

((definition? exp) (eval-definition exp env))
((if? exp) (eval-if exp env))
((lambda? exp)
  (make-procedure (lambda-parameters exp)
                  (lambda-body exp)
                  env))
((begin? exp)
  (eval-sequence (begin-actions exp) env))
((cond? exp) (eval (cond->if exp) env))
((application? exp)
  (apply (eval (operator exp) env)
         (list-of-values (operands exp) env)))
(else
  (error "Unknown expression type: EVAL" exp)))

```

For clarity, `eval` has been implemented as a case analysis using `cond`. The disadvantage of this is that our procedure handles only a few distinguishable types of expressions, and no new ones can be defined without editing the definition of `eval`. In most Lisp implementations, dispatching on the type of an expression is done in a data-directed style. This allows a user to add new types of expressions that `eval` can distinguish, without modifying the definition of `eval` itself. (See [Exercise 4.3](#).)

明快さのために、`eval` は `cond` を用いた条件分岐として実装されています。これの欠点は手続がいくつかの判別可能な式の型のみを取り扱い、`eval` の定義を編集すること無しに新しい式が定義できることです。多くの Lisp 実装では式の型に従う呼出はデータ適従スタイルにより行われています。これはユーザに `eval` が判別可能な新しい式の型の追加を許可します。`eval` 自身の定義の変更は必要有りません。（[Exercise 4.3](#) 参照）

## Apply

`Apply` takes two arguments, a procedure and a list of arguments to which the procedure should be applied. `Apply` classifies procedures into two kinds: It calls `apply-primitive-procedure` to apply primitives; it applies compound procedures by sequentially evaluating the expressions that make up the body of the procedure. The environment for the evaluation of the body of a compound procedure is constructed by extending the base environment carried by the

procedure to include a frame that binds the parameters of the procedure to the arguments to which the procedure is to be applied. Here is the definition of `apply`:

`apply` は 2 つの引数、手続と手続が適用されるべき引数のリストを取ります。`apply` は手続を 2 つ種類に分類します。プリミティブの適用には `apply-primitive-procedure` を呼びます。複合手続の適用には手続のボディを作る式を連続して評価することにより行います。複合手続のボディの評価のための環境は手続により運ばれた基礎環境を拡張することで構築し、手続のパラメタを手続が適用される引数に束縛するフレームを含めます。以下が `apply` の定義です。

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type: APPLY" procedure))))
```

## Procedure arguments

When `eval` processes a procedure application, it uses `list-of-values` to produce the list of arguments to which the procedure is to be applied. `List-of-values` takes as an argument the operands of the combination. It evaluates each operand and returns a list of the corresponding values:<sup>8</sup>

---

<sup>8</sup>We could have simplified the `application?` clause in `eval` by using `map` (and stipulating that `operands` returns a list) rather than writing an explicit `list-of-values` procedure. We chose not to use `map` here to emphasize the fact that the evaluator can be implemented without any use of higher-order procedures (and thus could be written in a language that doesn't have higher-order procedures), even though the language that it supports will include higher-order procedures.

`eval` が手続適用を処理する時、`list-of-values` を用いて手続が適用される引数のリストを生成します。`list-of-values` は引数として組み合わせのオペランドを取ります。各オペランドを評価し対応する値のリストを返します。<sup>9</sup>

```
(define (list-of-values exps env)
  (if (no-operands? exps)
    '()
    (cons (eval (first-operand exps) env)
          (list-of-values (rest-operands exps) env))))
```

## Conditionals

`Eval-if` evaluates the predicate part of an `if` expression in the given environment. If the result is true, `eval-if` evaluates the consequent, otherwise it evaluates the alternative:

`eval-if` は与えられた環境において `if` 式の述語部分を評価します。もし結果が真なら `eval-if` は consequent(結果) を評価し、そうでなければ alternative(代替) を評価します。

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
    (eval (if-consequent exp) env)
    (eval (if-alternative exp) env)))
```

The use of `true?` in `eval-if` highlights the issue of the connection between an implemented language and an implementation language. The `if-predicate` is evaluated in the language being implemented and thus yields a value in that language. The interpreter predicate `true?` translates that value into a value that can be tested by the `if` in the implementation language: The metacircular representation of truth might not be the same as that of the underlying

---

<sup>9</sup>`eval` の `application?` 節は明示的に `list-of-values` 手続を書くのではなく、`map` を用いることで(そして `operands` がリストを返すよう規定することで)より単純にすることができます。ここでは `map` を用いないことを選択することで高階手続を用いなくとも、例え評価機がサポートする言語が高階手続をサポートすることになっても、評価機が実装できることを強調しました。(従って高階手続を持たない言語で評価機を書くことも可能です)。

Scheme.<sup>10</sup>

`eval-if` 内での `true?` の使用は実装言語と被実装言語の間の接続の問題を強調します。`if-predicate` は被実装言語にて評価されるのでその言語の値を生じます。インタプリタの述語 `true?` はその値を実装言語の `if` でテストできる値に翻訳します。真実性のメタ循環表現は根底をなす Scheme のそれとは同じではないかもしれません。<sup>11</sup>

## Sequences

`Eval-sequence` is used by `apply` to evaluate the sequence of expressions in a procedure body and by `eval` to evaluate the sequence of expressions in a `begin` expression. It takes as arguments a sequence of expressions and an environment, and evaluates the expressions in the order in which they occur. The value returned is the value of the final expression.

`eval-sequence` は `apply` により用いられ手続のボディの中にある連続した式を評価します。また `eval` でも使用され `begin` 式の中の一連の式を評価します。引数として一連の式と環境を取り、式が現われる順で評価します。返り値は最後の式の値です。

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps)
          (eval (first-exp exps) env))
        (else
          (eval (first-exp exps) env)
          (eval-sequence (rest-exp exps) env))))
```

## Assignments and definitions

The following procedure handles assignments to variables. It calls `eval` to find the value to be assigned and transmits the variable and the resulting value to `set-variable-value!` to be installed in the designated environment.

---

<sup>10</sup>In this case, the language being implemented and the implementation language are the same. Contemplation of the meaning of `true?` here yields expansion of consciousness without the abuse of substance.

<sup>11</sup>今回は実装言語と被実装言語は同じです。ここで `true?` の意味に対する熟考は本質を誤解することなく理解の発展を促します。

以下の手続は変数への代入を扱います。`eval` を呼び代入される値を見つけ 値と結果となる変数を `set-variable-value!` へ転送することで指定された環 境へ設定されるようにします。

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
    (eval (assignment-value exp) env)
    env)
  'ok)
```

Definitions of variables are handled in a similar manner.<sup>12</sup>

変数の定義は同様の方法で扱われます。<sup>13</sup>

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (eval (definition-value exp) env)
    env)
  'ok)
```

We have chosen here to return the symbol `ok` as the value of an assignment or a definition.<sup>14</sup>

ここで代入、または定義の値としてシンボル `ok` を返すことを選択しました。<sup>15</sup>

**Exercise 4.1:** Notice that we cannot tell whether the metacircular evaluator evaluates operands from left to right or from right to left. Its evaluation order is inherited from the underlying Lisp: If the arguments to `cons` in `list-of-values` are evaluated from left to right, then `list-of-values` will evaluate operands from left to

---

<sup>12</sup>This implementation of `define` ignores a subtle issue in the handling of internal definitions, although it works correctly in most cases. We will see what the problem is and how to solve it in [Section 4.1.6](#).

<sup>13</sup>この `define` の実装は内部定義の扱いの微妙な問題を無視します。しかし多くの場合では正しく動きます。問題が何か、どのようにして解決するかについては[Section 4.1.6](#)で学びます。

<sup>14</sup>As we said when we introduced `define` and `set!`, these values are implementation-dependent in Scheme—that is, the implementor can choose what value to return.

<sup>15</sup>`define` と `set!` を導入した時に述べたように、これらの値は Scheme の実装依存です—つまり、実装者がどんな値を返すのか選択できます。

right; and if the arguments to `cons` are evaluated from right to left, then `list-of-values` will evaluate operands from right to left.

メタ循環評価機がオペランドを左から右へ評価するのか、右から左へなのか判断が付かないことに注意せよ。評価順は下位に横たわる Lisp から継承する。もし `list-of-values` 内の `cons` の引数が左から右へ評価されるなら、`list-of-values` はオペランドを左から右へと評価する。もし `cons` の引数が右から左へ評価されるなら、`list-of-values` は右から左へ評価する。

Write a version of `list-of-values` that evaluates operands from left to right regardless of the order of evaluation in the underlying Lisp. Also write a version of `list-of-values` that evaluates operands from right to left.

オペランドを左から右へと下位に横たわる Lisp の評価順に係らず評価する `list-of-values` の版を書け。またオペランドを右から左へ評価する `list-of-values` の版も書け。

### 4.1.2 Representing Expressions

The evaluator is reminiscent of the symbolic differentiation program discussed in [Section 2.3.2](#). Both programs operate on symbolic expressions. In both programs, the result of operating on a compound expression is determined by operating recursively on the pieces of the expression and combining the results in a way that depends on the type of the expression. In both programs we used data abstraction to decouple the general rules of operation from the details of how expressions are represented. In the differentiation program this meant that the same differentiation procedure could deal with algebraic expressions in prefix form, in infix form, or in some other form. For the evaluator, this means that the syntax of the language being evaluated is determined solely by the procedures that classify and extract pieces of expressions.

評価機は[Section 2.3.2](#)で議論された記号微分プログラムを思い出させます。双方のプログラムが記号式を操作します。両方のプログラムにおいて、複合式上の操作の結果は式の断片を再帰的に操作し、式の型に依存した方法で結合することにより決定します。両方のプログラムにおいて、私達はデータ抽象化を用いて式がどのように表現されるかの詳細から命令の一般的なルールを分離します。微分プログラムではこのことが、同じ微分手続が接頭辞形式、接中辞形

式、またはいくつかの他の形式の代数式を扱えることを意味しました。評価機にとっては、これは評価される言語の構文がもっぱら式を分類し、断片を抽出する手続により決定されることを意味します。

Here is the specification of the syntax of our language:

以下に私達の言語の構文の仕様を示します。

- The only self-evaluating items are numbers and strings:  
自己評価アイテムは数値と文字列のみです。

```
(define (self-evaluating? exp)
  (cond ((number? exp) true)
        ((string? exp) true)
        (else false)))
```

- Variables are represented by symbols:  
変数はシンボルにより表現されます。

```
(define (variable? exp) (symbol? exp))
```

- Quotations have the form (quote <text-of-quotation>):<sup>16</sup>  
引用は (quote <text-of-quotation>) の形式を持ちます。<sup>17</sup>

```
(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))
```

Quoted? is defined in terms of the procedure tagged-list?, which identifies lists beginning with a designated symbol:

quoted? は手続 tagged-list? を用いて定義されます。これはリストが指定されたシンボルで開始するかを判断します。

```
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
```

---

<sup>16</sup>As mentioned in Section 2.3.1, the evaluator sees a quoted expression as a list beginning with quote, even if the expression is typed with the quotation mark. For example, the expression 'a would be seen by the evaluator as (quote a). See Exercise 2.55.

<sup>17</sup>Section 2.3.1で述べたとおり、評価機は引用 (quote) された式を quote で始まるリストだと見ます。例え式がクオーテーションマークで入力されていてもです。例え式'aはこの評価機では (quote a) と見られます。Exercise 2.55を参照して下さい。

- Assignments have the form (`set! <var> <value>`):  
代入は (`set! <var> <value>`) の形式を取ります。

```
(define (assignment? exp) (tagged-list? exp 'set!))
(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))
```

- Definitions have the form  
定義は以下の形式を取ります。

```
(define <var> <value>)
```

or the form

または以下の形式になります。

```
(define (<var> <parameter1> ... <parametern>)
       <body>)
```

The latter form (standard procedure definition) is syntactic sugar for  
後者の形式 (標準手続き定義) は以下に対する構文糖です。

```
(define <var>
  (lambda (<parameter1> ... <parametern>)
         <body>))
```

The corresponding syntax procedures are the following:  
対応する構文手続は以下となります。

```
(define (definition? exp) (tagged-list? exp 'define))
(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)           ; formal parameters
                   (cddr exp))))    ; body
```

- Lambda expressions are lists that begin with the symbol `lambda`:  
`lambda` 式はシンボル `lambda` で始まるリストです。

```
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))
```

We also provide a constructor for `lambda` expressions, which is used by `definition-value`, above:

また `lambda` 式に対するコンストラクタも提供します。これは上記の `definition-value` で使用されます。

```
(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

- Conditionals begin with `if` and have a predicate, a consequent, and an (optional) alternative. If the expression has no alternative part, we provide `false` as the alternative.<sup>18</sup>

条件式は `if` で始まり述語、結果式を持ち、(任意で) 代替式を持ちます。もし式が代替式の部分を持たないのであれば代替式として `false` を与えます。<sup>19</sup>

```
(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))
```

---

<sup>18</sup>The value of an `if` expression when the predicate is false and there is no alternative is unspecified in Scheme; we have chosen here to make it `false`. We will support the use of the variables `true` and `false` in expressions to be evaluated by binding them in the global environment. See Section 4.1.4.

<sup>19</sup>述語が `false` になり代替式が存在しない場合の `if` 式の値は Scheme では未定義です。ここでは私達は `false` にすることを選択しました。私達は変数 `true` と `false` の式内での利用をサポートし、グローバル環境でのそれらの束縛により評価されるようにします。Section 4.1.4 参照。

We also provide a constructor for `if` expressions, to be used by `cond->if` to transform `cond` expressions into `if` expressions:

また `if` 式に対するコンストラクタも提供します。これは `cond->if` により `cond` 式を `if` 式に変換するのに用いられます。

```
(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))
```

- `Begin` packages a sequence of expressions into a single expression. We include syntax operations on `begin` expressions to extract the actual sequence from the `begin` expression, as well as selectors that return the first expression and the rest of the expressions in the sequence.<sup>20</sup>

`begin` は一連の式を 1 つの式へとまとめます。`begin` 式から実際の列を取り出す命令と同時に、列の最初の式とその残りの式を返すセレクタも含まれます。<sup>21</sup>

```
(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions exp) (cdr exp))
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))
```

We also include a constructor `sequence->exp` (for use by `cond->if`) that transforms a sequence into a single expression, using `begin` if necessary: また `cond->if` で用いるコンストラクタ `sequence->exp` も含めます。これは列を单一の式に、必要ならば `begin` を用いて、変換します。

```
(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq)))
```

---

<sup>20</sup>These selectors for a list of expressions—and the corresponding ones for a list of operands—are not intended as a data abstraction. They are introduced as mnemonic names for the basic list operations in order to make it easier to understand the explicit-control evaluator in Section 5.4.

<sup>21</sup>式のリストに対するこれらのセレクタ — それに対応するオペランドのリスト向けのものも含めて — はデータ抽象化を意図するものではありません。それらは基本的なリスト命令のための mnemonic(ニーモニック)名として Section 5.4 にて明示的コントロール評価機を理解することを易しくするために導入されます。

```
(else (make-begin seq)))
(define (make-begin seq) (cons 'begin seq))
```

- A procedure application is any compound expression that is not one of the above expression types. The `car` of the expression is the operator, and the `cdr` is the list of operands:

手続の適用は上記の式の型ではない任意の複合式です。その式の `car` はオペレータであり、`cdr` はオペランドのリストです。

```
(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))
```

## Derived expressions

Some special forms in our language can be defined in terms of expressions involving other special forms, rather than being implemented directly. One example is `cond`, which can be implemented as a nest of `if` expressions. For example, we can reduce the problem of evaluating the expression

いくつかの私達の言語内の特殊形式は直接実装されるのではなく、他の特殊形式を含む式を用いて定義できます。例の 1 つは `cond` です。これは入れ子の `if` 式として実装できます。例えば以下の式の評価上の問題を、

```
(cond ((> x 0) x)
      ((= x 0) (display 'zero) 0)
      (else (- x)))
```

to the problem of evaluating the following expression involving `if` and `begin` expressions:

次の `if` と `begin` の式を含む式の評価問題へと簡約することができます。

```
(if (> x 0)
    x
    (if (= x 0)
        (begin (display 'zero) 0)
        (- x)))
```

Implementing the evaluation of `cond` in this way simplifies the evaluator because it reduces the number of special forms for which the evaluation process must be explicitly specified.

`cond` の評価をこのように実装することは評価機を簡略化します。評価過程が明示的に指定されねばならない特殊形式の数を減らすことができるからです。

We include syntax procedures that extract the parts of a `cond` expression, and a procedure `cond->if` that transforms `cond` expressions into `if` expressions. A case analysis begins with `cond` and has a list of predicate-action clauses. A clause is an `else` clause if its predicate is the symbol `else`.<sup>22</sup>

`cond` 式の部分を抽出する構文手続と `cond` 式を `if` 式に変形する式 `cond->if` を含めます。事例分析は `cond` で始まり述語-行動節のリストを持ちます。節はもしその述語がシンボル `else` ならば `else` 節です。<sup>23</sup>

```
(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))
(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))
(define (cond-predicate clause) (car clause))
(define (cond-actions clause) (cdr clause))
(define (cond->if exp) (expand-clauses (cond-clauses exp)))
(define (expand-clauses clauses)
  (if (null? clauses)
      'false
      ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last: COND->IF"
                      clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first)))))))
```

---

<sup>22</sup>The value of a `cond` expression when all the predicates are false and there is no `else` clause is unspecified in Scheme; we have chosen here to make it false.

<sup>23</sup>全ての述語が false で `else` 節が存在しない場合の `cond` 式の値は Scheme では未定義です。ここではそれを false にしました。

```
(expand-clauses rest))))))
```

Expressions (such as `cond`) that we choose to implement as syntactic transformations are called *derived expressions*. `Let` expressions are also derived expressions (see [Exercise 4.6](#)).<sup>24</sup>

文法上の変形を実装することを選択した `cond` のような式は *derived expressions*(派生式) と呼ばれます。`let` 式もまた派生式です。([Exercise 4.6](#) 参照)<sup>25</sup>

**Exercise 4.2:** Louis Reasoner plans to reorder the `cond` clauses in `eval` so that the clause for procedure applications appears before the clause for assignments. He argues that this will make the interpreter more efficient: Since programs usually contain more applications than assignments, definitions, and so on, his modified `eval` will usually check fewer clauses than the original `eval` before identifying the type of an expression.

Louis Reasoner は `eval` の `cond` の順を変えて手続適用の節が代入のための節の前になるようにする計画を立てた。彼はこうすることでインタプリタをより効率良くできると主張した。プログラムは通常代入、定義等より適用を含んでいるためだ、と。彼の変更した `eval` は元の `eval` よりも通常より少ない節を式の型が判明する前にチェックするだろうという主張だ

- a What is wrong with Louis's plan? (Hint: What will Louis's evaluator do with the expression `(define x 3)`?)

<sup>24</sup>Practical Lisp systems provide a mechanism that allows a user to add new derived expressions and specify their implementation as syntactic transformations without modifying the evaluator. Such a user-defined transformation is called a *macro*. Although it is easy to add an elementary mechanism for defining macros, the resulting language has subtle name-conflict problems. There has been much research on mechanisms for macro definition that do not cause these difficulties. See, for example, [Kohlbecker 1986](#), [Clinger and Rees 1991](#), and [Hanson 1991](#).

<sup>25</sup>実用的な Lisp システムはユーザに対し新しい派生式を追加し、評価機の変更無しに文法上の変形としての実装を指定できる仕組みを提供します。そのようなユーザ定義変形は *macro* と呼ばれます。マクロ定義の初步的な仕組みを追加することは簡単なのですが、結果的にその言語は微妙な名前衝突の問題を持ちます。これらの困難をもたらさないマクロ定義の仕組みに関する多くの研究が存在します。例えば [Kohlbecker 1986](#), [Clinger and Rees 1991](#), [Hanson 1991](#) を参照して下さい。

Louis の計画の何が間違っているか? (ヒント : Louis の評価機は式 (define x 3) に対し何を行うか?)

- b Louis is upset that his plan didn't work. He is willing to go to any lengths to make his evaluator recognize procedure applications before it checks for most other kinds of expressions. Help him by changing the syntax of the evaluated language so that procedure applications start with `call`. For example, instead of `(factorial 3)` we will now have to write `(call factorial 3)` and instead of `(+ 1 2)` we will have to write `(call + 1 2)`.

Louis は彼の計画がうまく行かないことに激昂した。彼は他の多くの型の式をチェックする前に彼の評価機をいくらでも長くして手続適用を認識させようとしている。評価される言語を変更し手続適用が `call` で始まるようにすることで彼の手助けをせよ。例えば `(factorial 3)` の代わりに変更後は `(call factorial 3)` と書かねばならず、`(+ 1 2)` は `(call + 1 2)` と書かねばならない。

**Exercise 4.3:** Rewrite `eval` so that the dispatch is done in data-directed style. Compare this with the data-directed differentiation procedure of [Exercise 2.73](#). (You may use the `car` of a compound expression as the type of the expression, as is appropriate for the syntax implemented in this section.)

`eval` を書き直し呼出がデータ適従スタイルにて行われるようにせよ。これを [Exercise 2.73](#) のデータ適従型微分手続と比較せよ。この節で実装された文法に適切であるとおり、(複合式の `car` を式の型として用いてよい。

**Exercise 4.4:** Recall the definitions of the special forms `and` and `or` from [Chapter 1](#):

[Chapter 1](#)の特殊形式 `and` と `or` の定義を思い出せ。

- `and`: The expressions are evaluated from left to right. If any expression evaluates to false, false is returned; any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then true is returned.

**and:** 式は左から右へと評価される。もし任意の式が false と評価されるなら false が返される。残りの式全ては評価されない。もし全ての式が true の値に評価されるなら最後の式の評価値が返される。もし式が全く存在しないなら true が返される。

- **or:** The expressions are evaluated from left to right. If any expression evaluates to a true value, that value is returned; any remaining expressions are not evaluated. If all expressions evaluate to false, or if there are no expressions, then false is returned.

**or:** 式は左から右へと評価される。もし任意の式が true と評価されるのならその値が返される。残りの式全ては評価されない。もし全ての式が false と評価されるのなら、またはもし式が全く存在しないなら、false が返される。

Install **and** and **or** as new special forms for the evaluator by defining appropriate syntax procedures and evaluation procedures **eval-and** and **eval-or**. Alternatively, show how to implement **and** and **or** as derived expressions.

**and** と **or** を評価機に対する新しい特殊形式として適切な構文手続と評価手続 **eval-and** と **eval-or** を定義することで導入せよ。代替法として、**and** と **or** を派生式として実装する方法を示せ。

**Exercise 4.5:** Scheme allows an additional syntax for **cond** clauses, (**<test>** => **<recipient>**). If **<test>** evaluates to a true value, then **<recipient>** is evaluated. Its value must be a procedure of one argument; this procedure is then invoked on the value of the **<test>**, and the result is returned as the value of the **cond** expression. For example

Scheme は **cond** の節に追加の文法、(**<test>** => **<recipient>**) を認めている。もし **<test>** が true として評価されるなら、**<recipient>** が評価される。その値は 1 引数の手続でなければならない。そしてこの手続が **<test>** の値で起動され、その結果が **cond** 式の値として返される。例えば、

```
(cond ((assoc 'b '((a 1) (b 2))) => cadr)
      (else false))
```

returns 2. Modify the handling of `cond` so that it supports this extended syntax.

は 2 を返す。`cond` を変更してこの拡張文法をサポートするようにせよ。

**Exercise 4.6:** `Let` expressions are derived expressions, because `let` 式は派生式である。なぜなら、

```
(let ((⟨var1⟩ ⟨exp1⟩) ... (⟨varn⟩ ⟨expn⟩))  
  ⟨body⟩)
```

is equivalent to

は以下と等価である。

```
((lambda (⟨var1⟩ ... ⟨varn⟩)  
  ⟨body⟩)  
  ⟨exp1⟩  
  ...  
  ⟨expn⟩)
```

Implement a syntactic transformation `let->combination` that reduces evaluating `let` expressions to evaluating combinations of the type shown above, and add the appropriate clause to `eval` to handle `let` expressions.

文法上の変形 `let->combination` を実装せよ。これは `let` 式の評価を上記で示された型の組み合わせの評価へと簡約する。そして `let` 式を扱うために `eval` に適切な節を追加する。

**Exercise 4.7:** `Let*` is similar to `let`, except that the bindings of the `let*` variables are performed sequentially from left to right, and each binding is made in an environment in which all of the preceding bindings are visible. For example

`let*` は `let` に似ているが、`let*` の変数の束縛が左から右へと続けて実行され、全ての先行する束縛が可視となるよう各束縛が環境へ追加されていく。例えば、

```
(let* ((x 3) (y (+ x 2)) (z (+ x y 5)))  
  (* x z))
```

returns 39. Explain how a `let*` expression can be rewritten as a set of nested `let` expressions, and write a procedure `let*->nested-lets` that performs this transformation. If we have already implemented `let` (Exercise 4.6) and we want to extend the evaluator to handle `let*`, is it sufficient to add a clause to `eval` whose action is

は 39 を返す。`let*` 式が入れ子の `let` 式の集合としてどのように書き直すことができるか説明せよ。そしてこの変形を実行する手続 `let*->nested-lets` を書け。もし私達が既に `let` を実装していて (Exercise 4.6)、評価機を拡張し `let*` を扱いたいとしたら、以下の処理を行う節を `eval` に追加することは十分であろうか?

```
(eval (let*->nested-lets exp) env)
```

or must we explicitly expand `let*` in terms of non-derived expressions?

または私達は明示的に `let*` を非派生式を用いて拡張するべきであろうか?

**Exercise 4.8:** “Named `let`” is a variant of `let` that has the form  
“名前付き `let`” は `let` の変種であり以下の形式を持つ。

```
(let <var> <bindings> <body>)
```

The `<bindings>` and `<body>` are just as in ordinary `let`, except that `<var>` is bound within `<body>` to a procedure whose body is `<body>` and whose parameters are the variables in the `<bindings>`. Thus, one can repeatedly execute the `<body>` by invoking the procedure named `<var>`. For example, the iterative Fibonacci procedure (Section 1.2.2) can be rewritten using named `let` as follows:

`<bindings>` と `<body>` は通常の `let` と同様である。しかし `<var>` が `<body>` 内部で束縛される手続であり、ボディが `<body>` であり、かつパラメタが `<bindings>` の変数である点が異なる。従って `<var>` で名付けられた手続を呼び出すことで繰り返し `<body>` を実行することができる。例えば、反復ファイボナッチ手続 (Section 1.2.2) は名前付き `let` を用いて以下のように書き直すことができる。

```
(define (fib n)
  (let fib-iter ((a 1)
                 (b 0)
                 (count n))
    (if (= count 0)
        b
        (fib-iter (+ a b) a (- count 1)))))
```

Modify `let->combination` of Exercise 4.6 to also support named `let`.

Exercise 4.6の `let->combination` を変更して名前付き `let` もサポートするようにせよ。

**Exercise 4.9:** Many languages support a variety of iteration constructs, such as `do`, `for`, `while`, and `until`. In Scheme, iterative processes can be expressed in terms of ordinary procedure calls, so special iteration constructs provide no essential gain in computational power. On the other hand, such constructs are often convenient. Design some iteration constructs, give examples of their use, and show how to implement them as derived expressions.

多くの言語が `do`, `for`, `while`, `until` のような多様な反復構造をサポートする。Scheme では反復処理が通常の手続呼出を用いて表現できるため特別な反復構造が演算能力に対し本質的な利益を与えることはない。一方でそのような構造は時折便利でもある。いくつかの反復構造を設計せよ。それらの使用の例を与えどのように派生式として実装するかについて示せ。

**Exercise 4.10:** By using data abstraction, we were able to write an `eval` procedure that is independent of the particular syntax of the language to be evaluated. To illustrate this, design and implement a new syntax for Scheme by modifying the procedures in this section, without changing `eval` or `apply`.

データ抽象化を用いることで、評価されるべき言語の特定の文法から独立した `eval` 手続を書くことができる。これを説明するために `eval` と `apply` を変更することなくこの節の手続を変更することで Scheme の新しい文法を設計し、実装せよ。

### 4.1.3 Evaluator Data Structures

In addition to defining the external syntax of expressions, the evaluator implementation must also define the data structures that the evaluator manipulates internally, as part of the execution of a program, such as the representation of procedures and environments and the representation of true and false.

式の外側の文法を定義するのに加えて、評価機の実装は評価機が内部的に操作するデータ構造もプログラムの実行の一部として、手続と環境の表現や true と false の表現を定義しなければなりません。

#### Testing of predicates

For conditionals, we accept anything to be true that is not the explicit `false` object.

条件節に対しては真になるものは全て受け入れます。真とは明示的な `false` オブジェクトでは無いものです。

```
(define (true? x)  (not (eq? x false)))
(define (false? x) (eq? x false))
```

#### Representing procedures

To handle primitives, we assume that we have available the following procedures:

プリミティブを扱うために、以下の手続が利用可能であると仮定します。

- `(apply-primitive-procedure <proc> <args>)`  
applies the given primitive procedure to the argument values in the list `<args>` and returns the result of the application.  
与えられたプリミティブな手続をリスト `<args>` 中の引数の値に適用し、適用の結果を返します。
- `(primitive-procedure? <proc>)`  
tests whether `<proc>` is a primitive procedure.  
`<proc>` がプリミティブな手続であるか確認します。

These mechanisms for handling primitives are further described in [Section 4.1.4](#).

プリミティブを扱うこれらの仕組みは[Section 4.1.4](#)でさらに説明されます。

Compound procedures are constructed from parameters, procedure bodies, and environments using the constructor `make-procedure`:

複合手続はパラメタ、手続のボディ、環境からコンストラクタ `make-procedure` を用いて構築されます。

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? p)
  (tagged-list? p 'procedure))
(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (caddar p))
```

## Operations on Environments

The evaluator needs operations for manipulating environments. As explained in [Section 3.2](#), an environment is a sequence of frames, where each frame is a table of bindings that associate variables with their corresponding values. We use the following operations for manipulating environments:

評価機は環境を操作する命令を必要とします。[Section 3.2](#)で説明された通り、環境は連続するフレームであり、各フレームは変数をその対応する値に関連付ける束縛のテーブルです。以下の命令を用いて環境を操作します。

- `(lookup-variable-value <var> <env>)`  
returns the value that is bound to the symbol `<var>` in the environment `<env>`, or signals an error if the variable is unbound.  
環境 `<env>` 内でシンボル `<var>` に束縛された値を返します。または変数が束縛されていない場合エラーを発します。
- `(extend-environment <variables> <values> <base-env>)`  
returns a new environment, consisting of a new frame in which the symbols in the list `<variables>` are bound to the corresponding elements in the list `<values>`, where the enclosing environment is the environment `<base-env>`.

新しいフレームから成る環境を返します。フレームの中ではリスト $\langle\text{variables}\rangle$ 中のシンボルがリスト中 $\langle\text{values}\rangle$ の対応する要素に束縛されます。取り囲む環境は環境 $\langle\text{base-env}\rangle$ です。

- `(define-variable! <var> <value> <env>)`  
adds to the first frame in the environment  $\langle\text{env}\rangle$  a new binding that associates the variable  $\langle\text{var}\rangle$  with the value  $\langle\text{value}\rangle$ .  
環境 $\langle\text{env}\rangle$ の最初のフレームに変数 $\langle\text{var}\rangle$ を値 $\langle\text{value}\rangle$ に関連付ける新しい束縛を追加します。
- `(set-variable-value! <var> <value> <env>)`  
changes the binding of the variable  $\langle\text{var}\rangle$  in the environment  $\langle\text{env}\rangle$  so that the variable is now bound to the value  $\langle\text{value}\rangle$ , or signals an error if the variable is unbound.  
環境 $\langle\text{env}\rangle$ 中の変数 $\langle\text{var}\rangle$ の束縛を変更し、その変数が新しく値 $\langle\text{value}\rangle$ に束縛されるようにします。またはもし変数が束縛されていない場合にはエラーを発します。

To implement these operations we represent an environment as a list of frames. The enclosing environment of an environment is the `cdr` of the list. The empty environment is simply the empty list.

これらの命令を実装するためには環境をフレームのリストとして表現します。環境を内包する環境はリストの `cdr` です。空の環境は単純に空リストです。

```
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())
```

Each frame of an environment is represented as a pair of lists: a list of the variables bound in that frame and a list of the associated values.<sup>26</sup>

環境の各フレームはリストのペアとして表現されます。フレームに束縛される変数のリストと対応する値のリストです。<sup>27</sup>

<sup>26</sup>Frames are not really a data abstraction in the following code: `Set-variable-value!` and `define-variable!` use `set-car!` to directly modify the values in a frame. The purpose of the frame procedures is to make the environment-manipulation procedures easy to read.

<sup>27</sup>フレームは実際には以下のコードにおいてデータ抽象化されていません。`Set-variable-value!` と `define-variable!` は `set-car!` を用いて直接フレームの値を変更しています。フレーム手続の目的は環境操作手続を読み易くすることです。

```
(define (make-frame variables values)
  (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame)))))
```

To extend an environment by a new frame that associates variables with values, we make a frame consisting of the list of variables and the list of values, and we adjoin this to the environment. We signal an error if the number of variables does not match the number of values.

環境を変数を値に関連付ける新しいフレームにより拡張するために、変数のリストと値のリストから成るフレームを作成します。そしてその環境に隣接させます。もし変数の数が値の数に合わない場合にはエラーを発します。

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```

To look up a variable in an environment, we scan the list of variables in the first frame. If we find the desired variable, we return the corresponding element in the list of values. If we do not find the variable in the current frame, we search the enclosing environment, and so on. If we reach the empty environment, we signal an “unbound variable” error.

環境内の変数を探すためには、最初のフレームの変数のリストを走査します。希望の変数を見つければ対応する値リスト内の要素を返します。もし現在のフレーム内にその変数が見つからなければ内包する環境を探します。以下、繰り返しです。もし空環境まで辿り着いたならば“束縛されていない変数”的エラーを発します。

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
```

```

        (env-loop (enclosing-environment env)))
        ((eq? var (car vars)) (car vals))
        (else (scan (cdr vars) (cdr vals))))))
(if (eq? env the-empty-environment)
(error "Unbound variable" var)
(let ((frame (first-frame env)))
  (scan (frame-variables frame)
    (frame-values frame)))))

(env-loop env))

```

To set a variable to a new value in a specified environment, we scan for the variable, just as in `lookup-variable-value`, and change the corresponding value when we find it.

変数に新しい値を指定された環境にて設定するには、`lookup-variable-value` と同様に変数を走査し、対応する見つかった場合には対応する値を変更します。

```

(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))

      (if (eq? env the-empty-environment)
          (error "Unbound variable: SET!" var)
          (let ((frame (first-frame env)))
            (scan (frame-variables frame)
              (frame-values frame)))))

    (env-loop env)))

```

To define a variable, we search the first frame for a binding for the variable, and change the binding if it exists (just as in `set-variable-value!`). If no such binding exists, we adjoin one to the first frame.

変数を定義するには、最初にその変数の束縛を最初のフレームにて探します。束縛が存在すれば変更を行います。`(set-variable-value! と同様です)`。そのような束縛がなければ最初のフレームに追加します。

```

(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
              (add-binding-to-frame! var val frame))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
      (scan (frame-variables frame) (frame-values frame))))

```

The method described here is only one of many plausible ways to represent environments. Since we used data abstraction to isolate the rest of the evaluator from the detailed choice of representation, we could change the environment representation if we wanted to. (See [Exercise 4.11](#).) In a production-quality Lisp system, the speed of the evaluator's environment operations—especially that of variable lookup—has a major impact on the performance of the system. The representation described here, although conceptually simple, is not efficient and would not ordinarily be used in a production system.<sup>28</sup>

ここで記述された手法は環境を表現する多くのもととなる方法の 1 つでしかありません。データ抽象化を用いて評価機の他の部分を表現の詳細な選択から分離したので、もし望めば環境の表現を変更することが可能です。[\(Exercise 4.11 参照\)](#) 実運用品質の Lisp システムでは評価機の環境向け命令のスピードが—特に変数探索の物が—システムのパフォーマンスに主に影響を与えます。ここで説明された表現は概念上シンプルではありますが、効率的ではなく通常は実運用システムでは用いられません。<sup>29</sup>

**Exercise 4.11:** Instead of representing a frame as a pair of lists, we can represent a frame as a list of bindings, where each binding is

---

<sup>28</sup>The drawback of this representation (as well as the variant in [Exercise 4.11](#)) is that the evaluator may have to search through many frames in order to find the binding for a given variable. (Such an approach is referred to as *deep binding*.) One way to avoid this inefficiency is to make use of a strategy called *lexical addressing*, which will be discussed in [Section 5.5.6](#).

<sup>29</sup>この表現の欠点は ([Exercise 4.11](#) の通りも同様に) 評価機が与えられた変数を見つけるために数多くのフレームを探索しなければならないかもしだれない点です。(このような取り組み方は *deep binding*(深い束縛) と参照されます)。この非効率性を防ぐ 1 つの方法は *lexical addressing*(レキシカルアドレッシング) と呼ばれ [Section 5.5.6](#) にて議論されます。

a name-value pair. Rewrite the environment operations to use this alternative representation.

フレームをリストのペアと表現する代わりに、フレームを束縛のリストとして表現可能である。この場合、各束縛は名前と値のペアだ。環境の命令を書き換えこの代替表現を用いるようにせよ。

**Exercise 4.12:** The procedures `set-variable-value!`, `define-variable!` and `lookup-variable-value` can be expressed in terms of more abstract procedures for traversing the environment structure. Define abstractions that capture the common patterns and redefine the three procedures in terms of these abstractions.

手続 `set-variable-value!`, `define-variable!`, `lookup-variable-value` は環境の構造を縦断するためのより抽象的な手続を用いて表現することができる。共通なパターンを捕える抽象化を定義し、3つの手続をこれらの抽象化を用いて再定義せよ。

**Exercise 4.13:** Scheme allows us to create new bindings for variables by means of `define`, but provides no way to get rid of bindings. Implement for the evaluator a special form `make-unbound!` that removes the binding of a given symbol from the environment in which the `make-unbound!` expression is evaluated. This problem is not completely specified. For example, should we remove only the binding in the first frame of the environment? Complete the specification and justify any choices you make.

Scheme は `define` を用いて新しい束縛を作成することができる。しかし束縛を取り除く手段は提供しない。評価機に特殊形式 `make-unbound!` を実装せよ。これは `make-unbound!` が評価された環境から与えられたシンボルの束縛を削除する。この問題は完全には指示されていない。例えば環境の最初のフレームの束縛のみを削除するべきだろうか？仕様を完成させあなたが行った選択について理由を述べよ。

#### 4.1.4 Running the Evaluator as a Program

Given the evaluator, we have in our hands a description (expressed in Lisp) of the process by which Lisp expressions are evaluated. One advantage of expressing the evaluator as a program is that we can run the program. This gives

us, running within Lisp, a working model of how Lisp itself evaluates expressions. This can serve as a framework for experimenting with evaluation rules, as we shall do later in this chapter.

評価機を与えられたことで、Lisp 式が評価されるプロセスの (Lisp で表現された) 記述を手中にしました。評価機をプログラムとして表現することの利点の 1 つはプログラムを実行できることです。これにより Lisp の中で実行することで Lisp 自身がどのように式を評価するのかについての実行モデルを得ることができました。これは評価ルールを検証するフレームワークの役割を果たします。実際にこの章の後の方で行います。

Our evaluator program reduces expressions ultimately to the application of primitive procedures. Therefore, all that we need to run the evaluator is to create a mechanism that calls on the underlying Lisp system to model the application of primitive procedures.

評価機プログラムは式を究極的にはプリミティブな手続の適用まで簡約します。従って評価機を実行するのに必要なもの全ては基盤を無す Lisp システムを呼び出す仕組みを作成することでプリミティブ手続の適用をモデル化することです。

There must be a binding for each primitive procedure name, so that when `eval` evaluates the operator of an application of a primitive, it will find an object to pass to `apply`. We thus set up a global environment that associates unique objects with the names of the primitive procedures that can appear in the expressions we will be evaluating. The global environment also includes bindings for the symbols `true` and `false`, so that they can be used as variables in expressions to be evaluated.

各プリミティブな手続の名前の束縛が存在しなければなりません。そのため `eval` がプリミティブの適用の命令を評価する時、`apply` に渡すオブジェクトを見つけます。従って私達は評価しようとする式の中に現れることが可能なプリミティブな手続の名前と独自のオブジェクトを関連付けするグローバル環境を設定します。グローバル環境はまたシンボル `true` と `false` のための束縛も含めます。そうすることでそれらが評価される式の中で変数として利用することができます。

```
(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
```

```
(define-variable! 'true true initial-env)
(define-variable! 'false false initial-env)
initial-env))
(define the-global-environment (setup-environment))
```

It does not matter how we represent the primitive procedure objects, so long as `apply` can identify and apply them by using the procedures `primitive-procedure?` and `apply-primitive-procedure`. We have chosen to represent a primitive procedure as a list beginning with the symbol `primitive` and containing a procedure in the underlying Lisp that implements that primitive.

どのようにプリミティブ手続オブジェクトを評価するかは、`apply` がそれらを手続 `primitive-procedure?` と `apply-primitive-procedure` を用いて判別できる限り問題ではありません。私達はプリミティブな手続をシンボル `primitive` で始まり、そのプリミティブを実装する低層の Lisp の手続を含むリストとして表現することを選択しました。

```
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))
(define (primitive-implementation proc) (cadr proc))
```

`Setup-environment` will get the primitive names and implementation procedures from a list:<sup>30</sup>

`setup-environment` はプリミティブの名前と実装手続をリストから得ます。  
31

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)))
```

<sup>30</sup> Any procedure defined in the underlying Lisp can be used as a primitive for the metacircular evaluator. The name of a primitive installed in the evaluator need not be the same as the name of its implementation in the underlying Lisp; the names are the same here because the metacircular evaluator implements Scheme itself. Thus, for example, we could put (list 'first car) or (list 'square (lambda (x) (\* x x))) in the list of `primitive-procedures`.

<sup>31</sup> 低層の Lisp で定義される任意の手続はメタ循環評価機のプリミティブとして使用できます。評価機にインストールされるプリミティブの名前は低層の Lisp における実装の名前と同じである必要はありません。ここで名前が同じなのはメタ循環評価機が Scheme それ自身を実装するためです。従って例えば (list 'first car) や (list 'square (lambda (x) (\* x x))) を `primitive-procedures` に入れることもできたでしょう。

```

(list 'cons cons)
(list 'null? null?)
⟨more primitives⟩ ))
(define (primitive-procedure-names)
  (map car primitive-procedures))
(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
    primitive-procedures))

```

To apply a primitive procedure, we simply apply the implementation procedure to the arguments, using the underlying Lisp system:

プリミティブ手続を適用するためには単純に実装手続を引数に対して低層の Lisp システムを用いて適用します。<sup>32</sup>

```

(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
    (primitive-implementation proc) args))

```

For convenience in running the metacircular evaluator, we provide a *driver loop* that models the read-eval-print loop of the underlying Lisp system. It prints a *prompt*, reads an input expression, evaluates this expression in the

---

<sup>32</sup>Apply-in-underlying-scheme is the apply procedure we have used in earlier chapters. The metacircular evaluator's apply procedure (Section 4.1.1) models the working of this primitive. Having two different things called apply leads to a technical problem in running the metacircular evaluator, because defining the metacircular evaluator's apply will mask the definition of the primitive. One way around this is to rename the metacircular apply to avoid conflict with the name of the primitive procedure. We have assumed instead that we have saved a reference to the underlying apply by doing

apply-in-underlying-scheme は前の章で使用した apply 手続です。メタ循環評価機の apply 手続 (Section 4.1.1) はこのプリミティブの動き方をモデルにしています。2つの異なる apply と呼ばれる物を持つことはメタ循環評価機を実行するにおいて問題へと導きます。メタ循環評価機の apply を定義することがプリミティブの定義を隠してしまうためです。これを回避する1つの方法はメタ循環の apply をリネームすることでプリミティブ手続の名前との衝突を避けることです。私達はその代わりに下層の apply への参照をメタ循環の apply を定義する前に以下のようにすることで保存しました。

```
(define apply-in-underlying-scheme apply)
```

before defining the metacircular apply. This allows us to access the original version of apply under a different name.

これで元の版の apply に異なる名前でアクセスできるようになりました。

global environment, and prints the result. We precede each printed result by an *output prompt* so as to distinguish the value of the expression from other output that may be printed.<sup>33</sup>

メタ循環評価機実行時の利便性のために、低層の Lisp システムの read-eval-print loop (REPL: レプレル) をモデルにした *driver loop* を提供します。これは *prompt*(プロンプト) を表示し、入力式を読み込み、この式をグローバル環境の中で評価し、結果を表示します。私達は各表示された結果の前に *output prompt*(出力プロンプト) を置きます。そうすることで式の値を他の表示されるかもしれない出力から判別するためです。<sup>34</sup>

```
(define input-prompt ";; M-Eval input:")
(define output-prompt ";; M-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))
(define (announce-output string)
  (newline) (display string) (newline))
```

We use a special printing procedure, `user-print`, to avoid printing the environment part of a compound procedure, which may be a very long list (or may even contain cycles).

私達は特別なプリント手続、`user-print` を使用します。これは複合手続の環境部分を表示するのを防ぐためです。これはとても長いリストに成り得ます。

---

<sup>33</sup>The primitive procedure `read` waits for input from the user, and returns the next complete expression that is typed. For example, if the user types `(+ 23 x)`, `read` returns a three-element list containing the symbol `+`, the number 23, and the symbol `x`. If the user types `'x`, `read` returns a two-element list containing the symbol `quote` and the symbol `x`.

<sup>34</sup>プリミティブな手続 `read` はユーザからの入力を待ち、次の入力された完全な式を返します。例えばもしユーザが `(+ 23 x)` と入力した場合、`read` は 3 つの要素、シンボル `+`、数値 23、シンボル `x` を含むリストを返します。もしユーザが `'x` と入力したなら `read` は 2 つの要素、シンボル `quote` とシンボル `x` を含むリストを返します。

(またはさらにループを含んでいるかもしれません。)

```
(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))
```

Now all we need to do to run the evaluator is to initialize the global environment and start the driver loop. Here is a sample interaction:

これで評価機を実行するのに必要なことはグローバル環境の初期化とドライバーループの開始のみです。以下がサンプルの応答です。

```
(define the-global-environment (setup-environment))
(driver-loop)
;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
;; M-Eval value:
ok
;; M-Eval input:
(append '(a b c) '(d e f))
;; M-Eval value:
(a b c d e f)
```

**Exercise 4.14:** Eva Lu Ator and Louis Reasoner are each experimenting with the metacircular evaluator. Eva types in the definition of `map`, and runs some test programs that use it. They work fine. Louis, in contrast, has installed the system version of `map` as a primitive for the metacircular evaluator. When he tries it, things go terribly wrong. Explain why Louis's `map` fails even though Eva's works.

Eva Lu Ator と Louis Reasoner はそれぞれ評価機を検証している。Eva は `map` の定義を入力しきつつかそれを用いるテストプログラ

ム実行している。それらはうまく動いた。Louis は逆に `map` のシステム版をメタ循環評価機のプリミティブとして導入した。彼がそれを確かめた時、全くうまく動かなかった。なぜ Eva はうまく行ったのに Louis の `map` は失敗するのか。説明せよ。

### 4.1.5 Data as Programs

In thinking about a Lisp program that evaluates Lisp expressions, an analogy might be helpful. One operational view of the meaning of a program is that a program is a description of an abstract (perhaps infinitely large) machine. For example, consider the familiar program to compute factorials:

Lisp 式を評価する Lisp プログラムについて考えることにおいて、例えはとても良い手助けになるでしょう。プログラムの意味についての命令上の視点の1つに、プログラムは(恐らく無限に大きな)抽象機械の記述であるという物があります。例えば階乗を計算する親しみのあるプログラムについて考えてみましょう。

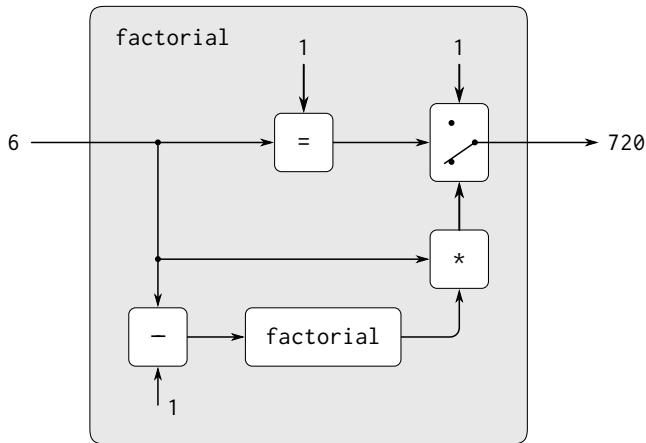
```
(define (factorial n)
  (if (= n 1) 1 (* (factorial (- n 1)) n)))
```

We may regard this program as the description of a machine containing parts that decrement, multiply, and test for equality, together with a two-position switch and another factorial machine. (The factorial machine is infinite because it contains another factorial machine within it.) [Figure 4.2](#) is a flow diagram for the factorial machine, showing how the parts are wired together.

私達はこのプログラムを減算、乗算、等価試験の部品と一緒に2つの位置を持つスイッチと他の階乗機械を含める機械の説明であると見做すことができるでしょう。(階乗機械は無限です。他の階乗機械をその中に含んでいるのです。[Figure 4.2](#)は階乗機械の流れ図でありその部品がどのようにお互いに接続されているかを示しています。

In a similar way, we can regard the evaluator as a very special machine that takes as input a description of a machine. Given this input, the evaluator configures itself to emulate the machine described. For example, if we feed our evaluator the definition of `factorial`, as shown in [Figure 4.3](#), the evaluator will be able to compute factorials.

同様な方法で、評価機を入力として機械の説明書を取るとても特殊な機械だと見做すことができます。この入力を与えられると、評価機はそれ自身を記



**Figure 4.2:** The factorial program, viewed as an abstract machine.

述された機械を真似するように設定します。例えばもし評価機にFigure 4.3で示される `factorial` の記述を与えれば、評価機は階乗の計算ができるようになります。

From this perspective, our evaluator is seen to be a *universal machine*. It mimics other machines when these are described as Lisp programs.<sup>35</sup> This is striking.

---

<sup>35</sup>The fact that the machines are described in Lisp is inessential. If we give our evaluator a Lisp program that behaves as an evaluator for some other language, say C, the Lisp evaluator will emulate the C evaluator, which in turn can emulate any machine described as a C program. Similarly, writing a Lisp evaluator in C produces a C program that can execute any Lisp program. The deep idea here is that any evaluator can emulate any other. Thus, the notion of “what can in principle be computed” (ignoring practicalities of time and memory required) is independent of the language or the computer, and instead reflects an underlying notion of *computability*. This was first demonstrated in a clear way by Alan M. Turing (1912–1954), whose 1936 paper laid the foundations for theoretical computer science. In the paper, Turing presented a simple computational model—now known as a *Turing machine*—and argued that any “effective process” can be formulated as a program for such a machine. (This argument is known as the *Church-Turing thesis*.) Turing then implemented a universal machine,

Try to imagine an analogous evaluator for electrical circuits. This would be a circuit that takes as input a signal encoding the plans for some other circuit, such as a filter. Given this input, the circuit evaluator would then behave like a filter with the same description. Such a universal electrical circuit is almost unimaginably complex. It is remarkable that the program evaluator is a rather simple program.<sup>36</sup>

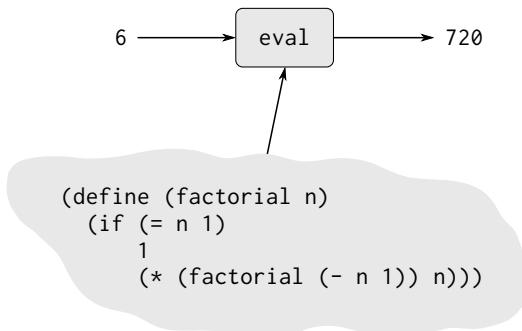
この視点からは、私達の評価機は*universal machine*(万能機械)であると見えます。他の機械がLispにて説明される時、それを真似します。<sup>37</sup>これは特筆

---

i.e., a Turing machine that behaves as an evaluator for Turing-machine programs. He used this framework to demonstrate that there are well-posed problems that cannot be computed by Turing machines (see [Exercise 4.15](#)), and so by implication cannot be formulated as “effective processes.” Turing went on to make fundamental contributions to practical computer science as well. For example, he invented the idea of structuring programs using general-purpose subroutines. See [Hodges 1983](#) for a biography of Turing.

<sup>36</sup>Some people find it counterintuitive that an evaluator, which is implemented by a relatively simple procedure, can emulate programs that are more complex than the evaluator itself. The existence of a universal evaluator machine is a deep and wonderful property of computation. *Recursion theory*, a branch of mathematical logic, is concerned with logical limits of computation. Douglas Hofstadter’s beautiful book *Gödel, Escher, Bach* explores some of these ideas ([Hofstadter 1979](#)).

<sup>37</sup>機械がLispで記述されるということは本質ではありません。もし私達の評価機にC言語の様な他の言語のための評価機として振る舞うLispプログラムを与えた場合、Lisp評価機はC評価機の真似をします。それは順に、C言語で記述された任意の機械の真似が可能です。同様にCで書かれたLisp評価機は任意のLispプログラムを実行できるCのプログラムを生成します。ここで深い意図は評価機は任意の他の物を真似できることです。従って“原理上、何が計算できるのか”という概念(必要な時間とメモリの実現性は無視)は言語や計算機に非依存です。その代わりに根底を成す概念である*computability*(計算可能性)を反映します。これは最初にAlan M. Turing (1912-1954)により明確に証明されました。彼の1936年の論文は計算機科学理論の基礎を導きました。この論文でチューリングは簡素な計算モデル—今日、*Turing machine*(チューリングマシン)として知られる—を公開し、任意の“実効的な処理”はそのような機械のプログラムとして定式化できると主張しました。(この論拠は*Church-Turing thesis*(チャーチ・チューリングのテーゼ、または提唱)として知られます)。チューリングは次に万能機械、即ちチューリングマシン向けプログラムの評価機として振る舞うチューリングマシンを実装しました。彼はこのフレームワークを用いてチューリングマシンでは計算できない上手く設定された問題が存在することを証明しました。([Exercise 4.15](#)参照)。そのため暗に“実効的な処理”として定式化できない問題の存在も示したのです。チューリングは実用的な計算機科学への基礎的な貢献の行いも続けました。例えば彼は汎用目的サブルーチンを用いて構造化プログラミングの考え方を発明しました。チューリングの経歴について



**Figure 4.3:** The evaluator emulating a factorial machine.

すべきことです。電子回路向けの同等な評価機について想像することを試してみて下さい。入力としてフィルタのようなある他の回路の計画を符号化した信号を取る回路になるでしょう。この入力を与えられて、回路評価機はそこで記述と同じフィルタのように振る舞うでしょう。そのような万能電子回路はほとんど想像不可能なほど複雑です。プログラム評価機がとても簡単なプログラムであることは特筆に値します。<sup>38</sup>

Another striking aspect of the evaluator is that it acts as a bridge between the data objects that are manipulated by our programming language and the programming language itself. Imagine that the evaluator program (implemented in Lisp) is running, and that a user is typing expressions to the evaluator and observing the results. From the perspective of the user, an input expression such as `(* x x)` is an expression in the programming language, which the evaluator should execute. From the perspective of the evaluator, however, the expression is simply a list (in this case, a list of three symbols: `*`, `x`, and `x`) that is to be manipulated according to a well-defined set of rules.

---

はHodges 1983を参照して下さい。

<sup>38</sup>ある人々は比較的単純な手続により実装された評価機が評価機それ自身より複雑なプログラムの真似ができることが直感的でないと感じました。万能評価機械の存在は深く、そして素晴らしい演算処理の特性です。Recursion theory(再帰理論)は数理論理学の1部門であり、演算処理の論理上の制約に関係します。Douglas Hofstadter(ダグラスホフスタッター)の美しい本 *Gödel, Escher, Bach*(邦題: ゲーデルエッシャーバッハ)はこれらの考え方のいくつかについて探求します。(Hofstadter 1979)

もう 1 つの特筆すべき評価機の側面はそれがプログラミング言語により操作されるデータオブジェクトとプログラミング言語それ自身との間のブリッジ(橋)として働くことです。(Lisp で実装された) 評価機プログラムが実行中であり、ユーザが式をその評価機に入力し結果を観察していると想像してみて下さい。ユーザの視点からは  $(* x x)$  の様な入力式はプログラミング言語による式であり、評価機が実行すべき物です。しかし、評価機の視点からは式は単純なリスト(この場合ではシンボル  $*$ ,  $x$ ,  $x$  のリスト)であり、これは明確なルール集合に従って操作されねばならぬ物です。

That the user's programs are the evaluator's data need not be a source of confusion. In fact, it is sometimes convenient to ignore this distinction, and to give the user the ability to explicitly evaluate a data object as a Lisp expression, by making `eval` available for use in programs. Many Lisp dialects provide a primitive `eval` procedure that takes as arguments an expression and an environment and evaluates the expression relative to the environment.<sup>39</sup> Thus,

ユーザのプログラムが評価機のデータだということは混乱の元となる必要はありません。実際に、時々はこの区別は無視したほうが便利です。そしてユーザに対し明示的にデータオブジェクトを Lisp の式として評価する能力を `eval` 手続をプログラム内で使用できるようにすることで与えることもまた便利なことです。多くの Lisp 方言は引数として式と環境を取り、その環境に関連してその式を評価するプリミティブな `eval` 手続を与えます。<sup>40</sup> 従って、

```
(eval '(* 5 5) user-initial-environment)
```

と

---

<sup>39</sup>Warning: This `eval` primitive is not identical to the `eval` procedure we implemented in Section 4.1.1, because it uses *actual* Scheme environments rather than the sample environment structures we built in Section 4.1.3. These actual environments cannot be manipulated by the user as ordinary lists; they must be accessed via `eval` or other special operations. Similarly, the `apply` primitive we saw earlier is not identical to the metacircular `apply`, because it uses actual Scheme procedures rather than the procedure objects we constructed in Section 4.1.3 and Section 4.1.4.

<sup>40</sup>警告: この `eval` プリミティブは私達がSection 4.1.1で実装した `eval` 手続とは異なります。それは私達がSection 4.1.3で構築したサンプルの環境構造ではなく、実際の Scheme 環境を用いるためです。これらの実際の環境はユーザにより通常のリストとして操作することはできません。それらは `eval` によりアクセスされるか、他の特別な命令を用います。同様に以前に見た `apply` プリミティブもメタ循環 `apply` とは異なります。それが私達がSection 4.1.3とSection 4.1.4で構築した手続オブジェクトではなく、実際の Scheme 手続を用いるからです。

```
(eval (cons '* (list 5 5)) user-initial-environment)
```

will both return 25.<sup>41</sup>

の両方は 25 を返します。<sup>42</sup>

**Exercise 4.15:** Given a one-argument procedure  $p$  and an object  $a$ ,  $p$  is said to “halt” on  $a$  if evaluating the expression  $(p\ a)$  returns a value (as opposed to terminating with an error message or running forever). Show that it is impossible to write a procedure  $\text{halts?}$  that correctly determines whether  $p$  halts on  $a$  for any procedure  $p$  and object  $a$ . Use the following reasoning: If you had such a procedure  $\text{halts?}$ , you could implement the following program:

1引数手続  $p$  とオブジェクト  $a$  を与えられた時、式  $(p\ a)$  が(エラー メッセージや無限に停止しない場合とは対照的に) 値を返す場合に、 $p$  は  $a$  に対して“halt”(停止) すると呼ばれる。 $p$  が  $a$  に対して停止するかどうかを任意の手続  $p$  と任意のオブジェクト  $a$  に対して正確に決定する手続  $\text{halts?}$  を書くことは不可能であることを示せ。以下の推測を用いろ：もしそのような手続  $\text{halts?}$  が存在するなら以下のプログラムを実装できるだろう。

```
(define (run-forever) (run-forever))
(define (try p)
  (if (halts? p p) (run-forever) 'halted))
```

Now consider evaluating the expression  $(\text{try}\ \text{try})$  and show that any possible outcome (either halting or running forever) violates the intended behavior of  $\text{halts?}$ .<sup>43</sup>

---

<sup>41</sup>The MIT implementation of Scheme includes `eval`, as well as a symbol `user-initial-environment` that is bound to the initial environment in which the user's input expressions are evaluated.

<sup>42</sup>Scheme の MIT 実装は `eval` と同様にユーザの入力式が評価される初期環境に束縛されるシンボル `user-initial-environment` も含みます。

<sup>43</sup>Although we stipulated that  $\text{halts?}$  is given a procedure object, notice that this reasoning still applies even if  $\text{halts?}$  can gain access to the procedure's text and its environment. This is Turing's celebrated *Halting Theorem*, which gave the first clear example of a *non-computable* problem, i.e., a well-posed task that cannot be carried out as a computational procedure.

ここで式 (`try try`) の評価について考え、どんな可能な結果(停止するか、無限に実行するか)も `halts?` の意図した振舞に違反することを示せ。<sup>44</sup>

#### 4.1.6 Internal Definitions

Our environment model of evaluation and our metacircular evaluator execute definitions in sequence, extending the environment frame one definition at a time. This is particularly convenient for interactive program development, in which the programmer needs to freely mix the application of procedures with the definition of new procedures. However, if we think carefully about the internal definitions used to implement block structure (introduced in Section 1.1.8), we will find that name-by-name extension of the environment may not be the best way to define local variables.

私達の評価の環境モデルとメタ循環評価機は定義を順に実行し、環境のフレームを1度に1定義づつ拡張します。これはインタラクティブなプログラム開発に対して特に便利です。その場合にはプログラマは自由に手続の適用を新しい手続の定義に混ぜる必要があります。しかし、(Section 1.1.8で紹介された) ブロック構造を実装するために用いられた内部定義について注意深く考えてみれば、環境の名前毎の拡張はローカル変数の定義に最良の方法ではないのではと気付くのではないでしょうか。

Consider a procedure with internal definitions, such as  
内部定義を伴なう以下のような手続について考えてみます。

```
(define (f x)
  (define (even? n) (if (= n 0) true (odd? (- n 1))))
  (define (odd? n) (if (= n 0) false (even? (- n 1))))
  (rest of body of f))
```

Our intention here is that the name `odd?` in the body of the procedure `even?` should refer to the procedure `odd?` that is defined after `even?`. The scope of the name `odd?` is the entire body of `f`, not just the portion of the body of `f` starting

---

<sup>44</sup>`halts?` が手続オブジェクトを与えられたと規定したが、この推測が例え `halts?` が手続のテキストとその環境へのアクセスを得ることが出来るとしても依然として適用できることに注意せよ。これはチューリングの著名な *Halting Theorem*(停止性問題)であり、*non-computable*(計算不可能)な問題の最初の明確な例を与える。言い換えれば、計算手続として実行不可能なうまく設定された課題である。

at the point where the `define` for `odd?` occurs. Indeed, when we consider that `odd?` is itself defined in terms of `even?`—so that `even?` and `odd?` are mutually recursive procedures—we see that the only satisfactory interpretation of the two `defines` is to regard them as if the names `even?` and `odd?` were being added to the environment simultaneously. More generally, in block structure, the scope of a local name is the entire procedure body in which the `define` is evaluated.

ここでの意図は手続 `even?` のボディ内の名前 `odd?` は `even?` の後に定義された手続 `odd?` を参照しなければなりません。名前 `odd?` のスコープは `f` のボディ全体であり、`odd?` の定義が起きた箇所から始まる `f` のボディの一部分ではありません。実際に `odd?` がそれ自身 `even?` を用いて定義されていることについて考えると—`even?` と `odd?` は相互再帰手続であり—2つの `define` を満足させる解釈はそれらを名前 `even?` と `odd?` が環境に同時に追加されたと見做すことのみだとわかります。より一般的には、ブロック構造において、ローカルな名前のスコープは `define` が評価された手続のボディ全体だということです。

As it happens, our interpreter will evaluate calls to `f` correctly, but for an “accidental” reason: Since the definitions of the internal procedures come first, no calls to these procedures will be evaluated until all of them have been defined. Hence, `odd?` will have been defined by the time `even?` is executed. In fact, our sequential evaluation mechanism will give the same result as a mechanism that directly implements simultaneous definition for any procedure in which the internal definitions come first in a body and evaluation of the value expressions for the defined variables doesn’t actually use any of the defined variables. (For an example of a procedure that doesn’t obey these restrictions, so that sequential definition isn’t equivalent to simultaneous definition, see Exercise 4.19.)<sup>45</sup>

偶然にも私達のインタプリタは `f` の呼出を正確に評価します。しかし“予想外”的理由のためです。内部手続の定義が最初に来るため、これらの手続への

---

<sup>45</sup>Wanting programs to not depend on this evaluation mechanism is the reason for the “management is not responsible” remark in [Footnote 28 of Chapter 1](#). By insisting that internal definitions come first and do not use each other while the definitions are being evaluated, the IEEE standard for Scheme leaves implementors some choice in the mechanism used to evaluate these definitions. The choice of one evaluation rule rather than another here may seem like a small issue, affecting only the interpretation of “badly formed” programs. However, we will see in [Section 5.5.6](#) that moving to a model of simultaneous scoping for internal definitions avoids some nasty difficulties that would otherwise arise in implementing a compiler.

呼出はそれらの全てが定義されるまで起こりません。従って `odd?` は `even?` が実行された時に定義されるのです。内部定義がボディの最初に来て定義された変数の値の式の評価が實際にはどの定義された変数も用いない任意の手続に対し直接、同時定義を実装する仕組みと、私達の逐次的な評価システムの仕組みは實際に同じ結果を与えます。(これらの制限に従わず、その結果逐次定義が同時定義と等価でない手続の例に対しては [Exercise 4.19](#)を参照して下さい)。<sup>46</sup>

There is, however, a simple way to treat definitions so that internally defined names have truly simultaneous scope—just create all local variables that will be in the current environment before evaluating any of the value expressions. One way to do this is by a syntax transformation on `lambda` expressions. Before evaluating the body of a `lambda` expression, we “scan out” and eliminate all the internal definitions in the body. The internally defined variables will be created with a `let` and then set to their values by assignment. For example, the procedure

しかし内部定義の名前が真に同時にスコープを持つようになる簡単な定義の扱い方が存在します。単に現在の環境に入ることになる全てのローカル変数をどの値の式が評価されるよりも早く作成することです。これを行う1つの方法は `lambda` 式上の構文変形によります。`lambda` 式のボディを評価する前に、ボディの中の全ての内部定義を走査し、削除します。内部で定義された変数は `let` を用いて作成され、次に代入を用いてそれらの値に設定されます。例えば、以下の手続は、

```
(lambda <vars>
  (define u <e1>)
  (define v <e2>)
  <e3>)
```

would be transformed into

以下の形式に変形されます。

---

<sup>46</sup> プログラムにこの評価の仕組みに依存して欲しくないというのがChapter 1のFootnote 28での見解、“管理は責任を取れない”に対する理由です。これを主張することで内部定義は最初に来て、定義中でお互いを定義が評価されている間に使用はしません。Scheme の IEEE 標準は実装者にこれらの定義の評価に用いられる仕組みについて幾つかの選択を残します。別のルールではなくある評価ルールを選択することはここでは“悪い形式”的なプログラムの解釈のみに影響する小さな問題に見えるかもしれません。しかしSection 5.5.6では同時に内部定義を行うモデルへの移行が、そうしなければコンパイラの実装にて起こり得る意地の悪い問題を防ぐことを学びます。

```
(lambda <vars>
  (let ((u '*unassigned*)
        (v '*unassigned*))
    (set! u <e1>)
    (set! v <e2>)
    <e3>))
```

where `*unassigned*` is a special symbol that causes looking up a variable to signal an error if an attempt is made to use the value of the not-yet-assigned variable.

ここで `*unassigned*` は特別なシンボルであり、変数が調べられた時にもしまだ値が割り当てられていない変数を使用しようとしたならばエラーを発せさせます。

An alternative strategy for scanning out internal definitions is shown in [Exercise 4.18](#). Unlike the transformation shown above, this enforces the restriction that the defined variables' values can be evaluated without using any of the variables' values.<sup>47</sup>

内部定義を全て走査する方法の代替となる戦略は[Exercise 4.18](#)にて示されます。上で示された変形とは異なり、これは定義された変数の値がその変数のどんな値も用いずに評価できるという制約を強制します。<sup>48</sup>

**Exercise 4.16:** In this exercise we implement the method just described for interpreting internal definitions. We assume that the evaluator supports `let` (see [Exercise 4.6](#)).

この課題では内部定義を逐次実行するためについて先程説明された手法を実装する。評価機は `let` をサポートすると仮定する。  
([Exercise 4.6](#) 参照)

- a Change `lookup-variable-value` ([Section 4.1.3](#)) to signal an error if the value it finds is the symbol `*unassigned*`.

---

<sup>47</sup>The IEEE standard for Scheme allows for different implementation strategies by specifying that it is up to the programmer to obey this restriction, not up to the implementation to enforce it. Some Scheme implementations, including MIT Scheme, use the transformation shown above. Thus, some programs that don't obey this restriction will in fact run in such implementations.

<sup>48</sup>Scheme の IEEE 標準はこの制約を強制する実装にまかせるのではなく、プログラマに対してこの制約に従うかをまかせると指定することで、異なる実装戦略を許しています。MIT Scheme を含むいくつかの Scheme 実装は上で示された変形を用いています。従ってこの制約に従わないプログラムは実際にはそのような実装の下では動作します。

`lookup-variable-value` (Section 4.1.3) を変更してもし見つけた値がシンボル `*unassigned*` ならエラーを発するようにする。

- b Write a procedure `scan-out-defines` that takes a procedure body and returns an equivalent one that has no internal definitions, by making the transformation described above.

手続のボディを取り内部手続を持たない同等な手続を返す手続 `scan-out-defines` を上で説明された変形を作成することにより、書け。

- c Install `scan-out-defines` in the interpreter, either in `make-procedure` or in `procedure-body` (see Section 4.1.3). Which place is better? Why?

`scan-out-defines` をインタプリタの `make-procedure` または `procedure-body` (see Section 4.1.3) の中に導入せよ。どちらの場所が良いか? それは何故か?

**Exercise 4.17:** Draw diagrams of the environment in effect when evaluating the expression  $\langle e3 \rangle$  in the procedure in the text, comparing how this will be structured when definitions are interpreted sequentially with how it will be structured if definitions are scanned out as described. Why is there an extra frame in the transformed program? Explain why this difference in environment structure can never make a difference in the behavior of a correct program. Design a way to make the interpreter implement the “simultaneous” scope rule for internal definitions without constructing the extra frame.

この本の手続の式  $\langle e3 \rangle$  の評価を実施している時の環境図を書くことで、定義が逐次的に翻訳された時にどのように構築されるかと、定義が説明されたように走査された場合にどのように構築されるかとの違いを比較せよ。変形されたプログラムにはなぜ余分なフレームが存在するのか? 環境構造内のこの違いが正しいプログラムの振舞に違いを起こさないのか説明せよ。インタプリタに内部定義の“同時”スコープのルールを余分なフレームの構築成しに実装させる方法を設計せよ。

**Exercise 4.18:** Consider an alternative strategy for scanning out definitions that translates the example in the text to

テキストの例を以下の様に変形する定義の走査に対する代替となる戦略を考えよ。

```
(lambda (vars)
  (let ((u '*unassigned*)
        (v '*unassigned*))
    (let ((a <e1>) (b <e2>))
      (set! u a)
      (set! v b))
    (e3)))
```

Here **a** and **b** are meant to represent new variable names, created by the interpreter, that do not appear in the user's program. Consider the `solve` procedure from [Section 3.5.4](#):

ここで **a** と **b** は新しい変数の名前を表現することを意味し、インタプリタにより作成され、ユーザのプログラムには現れない。[Section 3.5.4](#) の `solve` 手続について考える。

```
(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)
```

Will this procedure work if internal definitions are scanned out as shown in this exercise? What if they are scanned out as shown in the text? Explain.

この手続はこの課題に示されたように内部定義が走査された場合にうまく動くだろうか? テキストに示されたように走査された場合には動くだろうか? 説明せよ。

**Exercise 4.19:** Ben Bitdiddle, Alyssa P. Hacker, and Eva Lu Ator are arguing about the desired result of evaluating the expression Ben Bitdiddle, Alyssa P. Hacker, Eva Lu Ator の 3 人は以下の式を評価した場合の望まれた結果について議論している。

```
(let ((a 1))
  (define (f x)
    (define b (+ a x)))
  (define a 5))
```

```
(+ a b))  
(f 10))
```

Ben asserts that the result should be obtained using the sequential rule for `define`: `b` is defined to be 11, then `a` is defined to be 5, so the result is 16. Alyssa objects that mutual recursion requires the simultaneous scope rule for internal procedure definitions, and that it is unreasonable to treat procedure names differently from other names. Thus, she argues for the mechanism implemented in [Exercise 4.16](#). This would lead to `a` being unassigned at the time that the value for `b` is to be computed. Hence, in Alyssa's view the procedure should produce an error. Eva has a third opinion. She says that if the definitions of `a` and `b` are truly meant to be simultaneous, then the value 5 for `a` should be used in evaluating `b`. Hence, in Eva's view `a` should be 5, `b` should be 15, and the result should be 20. Which (if any) of these viewpoints do you support? Can you devise a way to implement internal definitions so that they behave as Eva prefers?<sup>49</sup>

Benは結果は `define` に対する逐次的実行のルールを用いて得られるべきだと主張した。`b` は 11 に定義され、`a` は 5 に定義される。従って結果は 16 である。Alyssa は相互再帰は同時スコープのルールが内部手続定義に要求されるとして異議を唱えた。手続の名前を他の名前から異なって扱うのは不合理だ。従って彼女は [Exercise 4.16](#) で実装された仕組みに賛成した。これは `a` が `b` の値が計算される時点では割り当てられていないという結論に導くだろう。従って Alyssa の視点では手続はエラーを生じなければならない。Eva は 3 つ目の立ち位置を取る。彼女はもし `a` と `b` の定義が真に同時にすることを意味するのであれば、`a` に対する値 5 は `b` の評価にて用いられるべきであると述べた。従って Eva の視点では `a` は 5 でなければならず、`b` は 15 でなければならぬ。そして結果は 20 にならなければならない。(もし同意するなら)3 人の視点あなたは

---

<sup>49</sup>The MIT implementors of Scheme support Alyssa on the following grounds: Eva is in principle correct--the definitions should be regarded as simultaneous. But it seems difficult to implement a general, efficient mechanism that does what Eva requires. In the absence of such a mechanism, it is better to generate an error in the difficult cases of simultaneous definitions (Alyssa's notion) than to produce an incorrect answer (as Ben would have it).

どれを支持するのか? あなたは Eva が好んだように振る舞う内部定義を実装する方法を考案できるか?<sup>50</sup>

**Exercise 4.20:** Because internal definitions look sequential but are actually simultaneous, some people prefer to avoid them entirely, and use the special form `letrec` instead. `Letrec` looks like `let`, so it is not surprising that the variables it binds are bound simultaneously and have the same scope as each other. The sample procedure `f` above can be written without internal definitions, but with exactly the same meaning, as

内部定義は逐次的に見えるが実際には同時であるため、いくらかの人々はこれを完全に回避するほうを好むだろう。そして特殊形式 `letrec` を代わりに用いる。`letrec` は `let` に似ているため、それが束縛する変数が同時に束縛されお互いに同じスコープを持つことは不思議ではないだろう。上記のサンプル手続 `f` は内部手続を用いずに、しかし全く同じ意味を持つように書くことができる。

```
(define (f x)
  (letrec
    ((even? (lambda (n)
              (if (= n 0) true (odd? (- n 1)))))
     (odd? (lambda (n)
              (if (= n 0) false (even? (- n 1))))))
    (rest-of-body-of-f)))
  (rest-of-body-of-f)))
```

`Letrec` expressions, which have the form

`letrec` 式は以下の形式を持つ。

```
(letrec ((<var1> <exp1>) ... (<varn> <expn>))
  <body>)
```

are a variation on `let` in which the expressions  $\langle exp_k \rangle$  that provide the initial values for the variables  $\langle var_k \rangle$  are evaluated in an

---

<sup>50</sup>MIT Scheme の実装者達は次の根拠に従って Alyssa を支持する。Eva は原理上は正しい。定義は同時だと見做されるべきだ。しかし Eva が要求することを行う一般的でかつ効率的な仕組みを実装することは難しく見える。そのような仕組みが不足している状況では、同時定義の難しい場合についてエラーを生成するほうが (Alyssa の意見)、正しくない答を生成するよりも (Ben の様に)、より良いだろう。

environment that includes all the `letrec` bindings. This permits recursion in the bindings, such as the mutual recursion of `even?` and `odd?` in the example above, or the evaluation of 10 factorial with

`letrec` 式は `let` の亜種であり、変数  $\langle var_k \rangle$  に初期値を与える式  $\langle exp_k \rangle$  は、全ての `letrec` の束縛を含む環境にて評価される。これは上の例においての `even?` と `odd?` の相互再帰のような束縛の中での再帰を許す。または以下の様な 10 の階乗の評価も可能である。

```
(letrec
  ((fact (lambda (n)
            (if (= n 1) 1 (* n (fact (- n 1)))))))
  (fact 10))
```

- a Implement `letrec` as a derived expression, by transforming a `letrec` expression into a `let` expression as shown in the text above or in [Exercise 4.18](#). That is, the `letrec` variables should be created with a `let` and then be assigned their values with `set!`.

`letrec` を派生式として実装せよ。`letrec` 式を `let` を上で示したように、または[Exercise 4.18](#)の様に変形することで行え。即ち、`letrec` の変数は `let` を用いて作成しなければならず、そして次にそれらの値を `set!` で代入すること。

- b Louis Reasoner is confused by all this fuss about internal definitions. The way he sees it, if you don't like to use `define` inside a procedure, you can just use `let`. Illustrate what is loose about his reasoning by drawing an environment diagram that shows the environment in which the  $\langle rest\ of\ body\ of\ f \rangle$  is evaluated during evaluation of the expression  $(f\ 5)$ , with  $f$  defined as in this exercise. Draw an environment diagram for the same evaluation, but with `let` in place of `letrec` in the definition of  $f$ .

Louis Reasoner は内部定義に関するこの全ての空騒ぎにより混乱してしまった。彼の見解は、もし手続の中での `define` の使用を好まないのであれば、単に `let` を使えるのではないかである。彼の reasoning(推測) の何が緩いのかを、この課題と

同様に定義された `f` を用いて、式 (f 5) の評価の間に `(rest of body of f)` が評価された環境を示す環境図を書くことによって説明せよ。同じ環境の、ただし `f` の定義中の `letrec` の場所に `let` を用いた場合の環境図を書け。

**Exercise 4.21:** Amazingly, Louis's intuition in Exercise 4.20 is correct. It is indeed possible to specify recursive procedures without using `letrec` (or even `define`), although the method for accomplishing this is much more subtle than Louis imagined. The following expression computes 10 factorial by applying a recursive factorial procedure:<sup>51</sup>

驚くべきことに、Exercise 4.20における Louis の直感は正しい。`letrec` を（または `define` すらも）用いずに再帰手続を指定することは本当に可能である。しかしこれを達成する手法は Louis が存在したよりもずっと繊細である。以下の式は 10 の階乗を再帰階乗手続を適用することを求めている。<sup>52</sup>

```
((lambda (n)
  ((lambda (fact) (fact fact n))
   (lambda (ft k)
     (if (= k 1) 1 (* k (ft ft (- k 1)))))))
 10)
```

- a Check (by evaluating the expression) that this really does compute factorials. Devise an analogous expression for computing Fibonacci numbers.

(式を評価することで) これが実際に階乗を計算することを確認せよ。フィボナッチ数を計算する同様な式を工夫せよ。

---

<sup>51</sup>This example illustrates a programming trick for formulating recursive procedures without using `define`. The most general trick of this sort is the *Y operator*, which can be used to give a “pure  $\lambda$ -calculus” implementation of recursion. (See Stoy 1977 for details on the  $\lambda$ -calculus, and Gabriel 1988 for an exposition of the *Y* operator in Scheme.)

<sup>52</sup>この例は再帰手続を `define` を用いずに定式化するためのプログラミング上の技を説明しています。最も一般的なこの種の技は *Y operator*(Y コンビネータ、不動点演算子)です。これは“pure  $\lambda$ -calculus”(純粋ラムダ計算)による再帰の実装を与えます。( $\lambda$  計算の詳細についてはStoy 1977を参照して下さい。また Scheme による Y コンビネータの解説についてはGabriel 1988を参照して下さい。

- b Consider the following procedure, which includes mutually recursive internal definitions:

以下の手続について考える。これは相互再帰内部定義を含む。

```
(define (f x)
  (define (even? n)
    (if (= n 0) true (odd? (- n 1))))
  (define (odd? n)
    (if (= n 0) false (even? (- n 1))))
  (even? x))
```

Fill in the missing expressions to complete an alternative definition of `f`, which uses neither internal definitions nor `letrec`: `f` の代替的な定義を完成するために欠けている式を埋めよ。これは内部定義も `letrec` も使用してはいない。

```
(define (f x)
  ((lambda (even? odd?) (even? even? odd? x))
   (lambda (ev? od? n)
     (if (= n 0) true (od? <??> <??> <??>)))
   (lambda (ev? od? n)
     (if (= n 0) false (ev? <??> <??> <??>))))))
```

#### 4.1.7 Separating Syntactic Analysis from Execution

The evaluator implemented above is simple, but it is very inefficient, because the syntactic analysis of expressions is interleaved with their execution. Thus if a program is executed many times, its syntax is analyzed many times. Consider, for example, evaluating `(factorial 4)` using the following definition of `factorial`:

上で実装された評価機は簡単ですが、非効率です。式の構文上の分析がその実行と相互配置されているためです。従ってもしプログラムが何度も実行された場合、その構文は何度も分析されます。例えば次の `factorial` を用いて `(factorial 4)` を評価することを考えてみて下さい。

```
(define (factorial n)
  (if (= n 1) 1 (* (factorial (- n 1)) n)))
```

Each time `factorial` is called, the evaluator must determine that the body is an `if` expression and extract the predicate. Only then can it evaluate the predicate and dispatch on its value. Each time it evaluates the expression `(* (factorial (- n 1)) n)`, or the subexpressions `(factorial (- n 1))` and `(- n 1)`, the evaluator must perform the case analysis in `eval` to determine that the expression is an application, and must extract its operator and operands. This analysis is expensive. Performing it repeatedly is wasteful.

`factorial` が呼ばれる度に、評価機はボディが `if` 式であることを判断せねばならず、それから述語を取り出します。その後にのみ述語を評価しその値により振り分けが行えます。式 `(* (factorial (- n 1)) n)`、または部分式 `(factorial (- n 1))` と `(- n 1)` を評価する度に、評価機は `eval` にて状況分析を行い式が適用であるかを判断せねばならず、また演算子とオペランドの抽出をせねばなりません。この分析はコストが高いのです。これを繰り返し実行することは無駄が多いでしょう。

We can transform the evaluator to be significantly more efficient by arranging things so that syntactic analysis is performed only once.<sup>53</sup> We split `eval`, which takes an expression and an environment, into two parts. The procedure `analyze` takes only the expression. It performs the syntactic analysis and returns a new procedure, the *execution procedure*, that encapsulates the work to be done in executing the analyzed expression. The execution procedure takes an environment as its argument and completes the evaluation. This saves work because `analyze` will be called only once on an expression, while the execution procedure may be called many times.

評価機を変形し文法上の分析をたった 1 度のみ実行されるように準備することで著しく効率良くすることができます。<sup>54</sup> 私達は式と環境を取る `eval` を 2 つに分けます。手続 `analyze` は式のみを取ります。構文上の分析を行い新しい手続 `execution procedure`(実行手続) を返します。この手続は分析された式を実行するにおいて行われた結果をカプセル化します。実行手続は環境を引数として取り評価を完了します。これは実行手続が何度も呼ばれるのに対し、`analyze`

---

<sup>53</sup>This technique is an integral part of the compilation process, which we shall discuss in Chapter 5. Jonathan Rees wrote a Scheme interpreter like this in about 1982 for the T project (Rees and Adams 1982). Marc Feeley (1986) (see also Feeley and Lapalme 1987) independently invented this technique in his master's thesis.

<sup>54</sup>この技はコンパイル過程に不可欠な要素であり、Chapter 5で議論します。Jonathan Rees は 1982 年頃にこのような Scheme インタプリタを T プロジェクトのために書きました (Rees and Adams 1982)。Marc Feeley (1986) (Feeley and Lapalme 1987も参照) は彼の修士論文にて独力でこの技を発明しました。

が式に対して 1 度しか呼ばれないため作業量を減らせます。

With the separation into analysis and execution, `eval` now becomes  
分析と実行への分離に伴ない、`eval` は以下の様になります。

```
(define (eval exp env) ((analyze exp) env))
```

The result of calling `analyze` is the execution procedure to be applied to the environment. The `analyze` procedure is the same case analysis as performed by the original `eval` of Section 4.1.1, except that the procedures to which we dispatch perform only analysis, not full evaluation:

`analyze` の呼出の結果は環境に適用される実行手続です。`analyze` 手続はSection 4.1.1の元の `eval` により実行されたのと同じ状況分析です。ただし私達が呼び出す手続は完全な評価ではなく分析のみを実行します。

```
(define (analyze exp)
  (cond ((self-evaluating? exp)
         (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp)
         (analyze-sequence (begin-actions exp)))
        ((cond? exp) (analyze (cond->if exp)))
        ((application? exp) (analyze-application exp))
        (else
         (error "Unknown expression type: ANALYZE" exp))))
```

Here is the simplest syntactic analysis procedure, which handles self-evaluating expressions. It returns an execution procedure that ignores its environment argument and just returns the expression:

以下に最も簡単な構文分析手続があります。これは自己評価式です。環境引数を無視し、ただ式を返す実行手続を返します。

```
(define (analyze-self-evaluating exp)
  (lambda (env) exp))
```

For a quoted expression, we can gain a little efficiency by extracting the text of the quotation only once, in the analysis phase, rather than in the execution phase.

クオートされた式に対してはそのテキストの取り出しを実行フェーズでなく、分析フェーズで 1 度だけ行うことでほんの少し効率良くすることができます。

```
(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env) qval)))
```

Looking up a variable value must still be done in the execution phase, since this depends upon knowing the environment.<sup>55</sup>

変数の値の探索は依然として実行フェーズで行わねばなりません。これは環境を知ることに依存するためです。<sup>56</sup>

```
(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))
```

Analyze-assignment also must defer actually setting the variable until the execution, when the environment has been supplied. However, the fact that the assignment-value expression can be analyzed (recursively) during analysis is a major gain in efficiency, because the assignment-value expression will now be analyzed only once. The same holds true for definitions.

analyze-assignment もまた実際の変数の設定を環境の供給が完了する実行時まで遅らせなければなりません。しかし assignment-value 式が分析の間に（再帰的に）分析されることができることは効率を大きく向上します。assignment-value 式は今はまだ 1 度しか分析されないためです。同じことが定義に対しても言えます。

---

<sup>55</sup>There is, however, an important part of the variable search that *can* be done as part of the syntactic analysis. As we will show in Section 5.5.6, one can determine the position in the environment structure where the value of the variable will be found, thus obviating the need to scan the environment for the entry that matches the variable.

<sup>56</sup>しかし、構文上の分析の部分にて終わらせられる変数探索の重要な部分があります。Section 5.5.6 にて示されるように、環境構造の中でどこで変数の値が見つかるか、その位置を決定することができます。従って変数にマッチするエントリのために環境を走査する必要を防ぐことができます。

```

(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env)
      (set-variable-value! var (vproc env) env)
      'ok)))
(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env)
      (define-variable! var (vproc env) env)
      'ok)))

```

For if expressions, we extract and analyze the predicate, consequent, and alternative at analysis time.

if 式に対しては分析時に述語、結果、代替を取り出し分析します。

```

(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env) (if (true? (pproc env))
                      (cproc env)
                      (aproc env)))))

```

Analyzing a lambda expression also achieves a major gain in efficiency: We analyze the lambda body only once, even though procedures resulting from evaluation of the lambda may be applied many times.

lambda 式の分析もまた効率が大きく向上します。lambda のボディは 1 度しか分析しません。例え lambda の評価の結果としての手続が何度適用されてもです。

```

(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env) (make-procedure vars bproc env))))

```

Analysis of a sequence of expressions (as in a begin or the body of a lambda

expression) is more involved.<sup>57</sup> Each expression in the sequence is analyzed, yielding an execution procedure. These execution procedures are combined to produce an execution procedure that takes an environment as argument and sequentially calls each individual execution procedure with the environment as argument.

(`begin` や `lambda` 式のボディの中としての) 式の列の評価の分析はより必要とされます。<sup>58</sup>列の各式は分析され実行手続を生じます。これらの実行手続は環境を引数として取り順番に各個別の実行手続を引数としての環境と共に呼び出す実行手続を生成するために組み合わせれます。

```
(define (analyze-sequence exps)
  (define (sequentially proc1 proc2)
    (lambda (env) (proc1 env) (proc2 env)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs)
                           (cdr rest-procs)))))
  (let ((procs (map analyze exps)))
    (if (null? procs) (error "Empty sequence: ANALYZE"))
    (loop (car procs) (cdr procs)))))
```

To analyze an application, we analyze the operator and operands and construct an execution procedure that calls the operator execution procedure (to obtain the actual procedure to be applied) and the operand execution procedures (to obtain the actual arguments). We then pass these to `execute-application`, which is the analog of `apply` in Section 4.1.1. `Execute-application` differs from `apply` in that the procedure body for a compound procedure has already been analyzed, so there is no need to do further analysis. Instead, we just call the execution procedure for the body on the extended environment.

適用を分析するためには、演算子とオペランドを分析し、演算子の実行手続を(実際に適用される手続を得るために)呼び出し、オペランドの実行手続を(実際の引数を得るために)呼び出す実行手続を構築します。次にこれらを `execute-application` に渡します。これは Section 4.1.1 の `apply` の類似品です。`execute-application` は `apply` とは複合手続のための手続のボディが既

---

<sup>57</sup>See Exercise 4.23 for some insight into the processing of sequences.

<sup>58</sup>列の処理に関する実態についてはExercise 4.23を参照して下さい。

に分析されている点が異なります。そのためさらなる分析の必要性があります。その代わりに、ただ拡張された環境上のボディに対して手続実行を呼び出します。

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application
        (fproc env)
        (map (lambda (aproc) (aproc env))
              aprocs)))))

(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment
            (procedure-parameters proc)
            args
            (procedure-environment proc))))
        (else
         (error "Unknown procedure type:
                EXECUTE-APPLICATION"
               proc))))
```

Our new evaluator uses the same data structures, syntax procedures, and run-time support procedures as in sections [Section 4.1.2](#), [Section 4.1.3](#), and [Section 4.1.4](#).

私達の新しい評価機は節[Section 4.1.2](#), [Section 4.1.3](#), [Section 4.1.4](#)にあるように、同じデータ構造、構文手続、実行時サポート手続を用います。

**Exercise 4.22:** Extend the evaluator in this section to support the special form `let`. (See [Exercise 4.6](#).)

この節の評価機を特殊形式 `let` をサポートするように拡張せよ。  
([Exercise 4.6](#)参照)

**Exercise 4.23:** Alyssa P. Hacker doesn't understand why `analyze-sequence` needs to be so complicated. All the other analysis pro-

cedures are straightforward transformations of the corresponding evaluation procedures (or `eval` clauses) in Section 4.1.1. She expected `analyze-sequence` to look like this:

Alyssa P. Hacker はなぜ `analyze-sequence` がそんなに複雑になるのか理解できなかった。他の分析手続全てはSection 4.1.1の対応する評価手続（または `eval` 節）の簡単な変形である。彼女は `analyze-sequence` は以下のようにになるのではと予想した。

```
(define (analyze-sequence exps)
  (define (execute-sequence procs env)
    (cond ((null? (cdr procs))
           ((car procs) env))
          (else
            ((car procs) env)
            (execute-sequence (cdr procs) env))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence: ANALYZE"))
        (lambda (env) (execute-sequence procs env)))))
```

Eva Lu Ator explains to Alyssa that the version in the text does more of the work of evaluating a sequence at analysis time. Alyssa's sequence-execution procedure, rather than having the calls to the individual execution procedures built in, loops through the procedures in order to call them: In effect, although the individual expressions in the sequence have been analyzed, the sequence itself has not been.

Eva Lu Ator は Alyssa に対し、テキストの版は分析時に列を評価する仕事よりも多くのことを行っていると説明した。Alyssa の逐次実行手続は個別の組み込みの実行手続に対する呼出を行うのではなく、複数の手続を通してそれらを呼び出すためにループする。実際に列内の個別の式は分析されるが、列それ自身は分析されない。

Compare the two versions of `analyze-sequence`. For example, consider the common case (typical of procedure bodies) where the sequence has just one expression. What work will the execution procedure produced by Alyssa's program do? What about the exe-

cution procedure produced by the program in the text above? How do the two versions compare for a sequence with two expressions?

2つの版の `analyze-sequence` を比較せよ。例として、列がただ1つの式を持つ場合において(手続のボディ特有の)共通な場合について考えよ。Alyssaのプログラムにより生成された実行手続はどのような行いをするか?上のテキスト内のプログラムで生成された実行手続についてはどうか?2つの版は2つの式を持つ列に対してはどのように比較されるか?

**Exercise 4.24:** Design and carry out some experiments to compare the speed of the original metacircular evaluator with the version in this section. Use your results to estimate the fraction of time that is spent in analysis versus execution for various procedures.

元のメタ循環評価機とこの節の版のスピードを比較するためのいくつかの実験を設計し実行せよ。あなたの結果を用いて種々の手続に対して分析と実行で消費された時間を概算せよ。

## 4.2 Variations on a Scheme — Lazy Evaluation

Now that we have an evaluator expressed as a Lisp program, we can experiment with alternative choices in language design simply by modifying the evaluator. Indeed, new languages are often invented by first writing an evaluator that embeds the new language within an existing high-level language. For example, if we wish to discuss some aspect of a proposed modification to Lisp with another member of the Lisp community, we can supply an evaluator that embodies the change. The recipient can then experiment with the new evaluator and send back comments as further modifications. Not only does the high-level implementation base make it easier to test and debug the evaluator; in addition, the embedding enables the designer to snarf<sup>59</sup> features from the underlying language, just as our embedded Lisp evaluator uses primitives and

---

<sup>59</sup>Snarf: “To grab, especially a large document or file for the purpose of using it either with or without the owner’s permission.” Snarf down: “To snarf, sometimes with the connotation of absorbing, processing, or understanding.” (These definitions were snarfed from Steele et al. 1983. See also Raymond 1993.)

control structure from the underlying Lisp. Only later (if ever) need the designer go to the trouble of building a complete implementation in a low-level language or in hardware. In this section and the next we explore some variations on Scheme that provide significant additional expressive power.

今や私達は Lisp プログラムとして表現された評価機を得ました。これで言語設計上の代替となる選択を単純に評価機を変更することで試験することができます。実際に新しい言語は良く、最初に既存の高級言語の中に新しい言語を埋め込む評価機を書くことで開発されます。例えばもしわしたちが Lisp に対する変更の提案のある側面について Lisp コミュニティの他のメンバと議論したい時に、変更を組み込んだ評価機を与えることができます。受け手はすると新しい評価機を持ちいて実験を行いさらなる変更としてのコメントを返すことができます。高レベルな実装ベースが評価機のテストとデバッグをより簡単にするだけではありません。加えて組込むことは設計者に対し下層の言語から機能を snarf する<sup>60</sup> ことを可能にします。これは私達の組込 Lisp 評価機が下層の Lisp からプリミティブやコントロール構造を使用するのと同じです。設計者は(もし必要があれば) 後で低レベル言語やハードウェアにて完全な実装を構築するだけです。この節と次では Scheme の、優位な追加の表現力を提供するいくつかのバリエーションについて探求します。

#### 4.2.1 Normal Order and Applicative Order

In Section 1.1, where we began our discussion of models of evaluation, we noted that Scheme is an *applicative-order* language, namely, that all the arguments to Scheme procedures are evaluated when the procedure is applied. In contrast, *normal-order* languages delay evaluation of procedure arguments until the actual argument values are needed. Delaying evaluation of procedure arguments until the last possible moment (e.g., until they are required by a primitive operation) is called *lazy evaluation*.<sup>61</sup> Consider the procedure

---

<sup>60</sup>Snarf: “つかみ取ること、特に巨大な文書やファイルを持ち主の許可を得ても得なくとも使う目的のため” Snarf Down: “snarfすること、稀に吸収する、処理する、または理解するの含意を持つ” (これらの定義は Steele et al. 1983 から snarf した。Raymond 1993 も参照すること)

<sup>61</sup>The difference between the “lazy” terminology and the “normal-order” terminology is somewhat fuzzy. Generally, “lazy” refers to the mechanisms of particular evaluators, while “normal-order” refers to the semantics of languages, independent of any particular evaluation strategy. But this is not a hard-and-fast distinction, and the two terminologies are often used interchangeably.

Section 1.1では評価のモデルについての議論を始めましたが、Schemeは*applicative-order*(適用順序)言語であると記しました。即ち、Schemeの手続への全ての引数は手続が適用される時に評価される、と。逆に、*normal-order*(正規順序)言語は手続引数の評価を実際に引数の値が必要とされるまで遅らせます。手続引数の評価を可能な限り最後の瞬間まで(例えばプリミティブ命令により必要とされるまで)遅らせることは*lazy evaluation*(遅延評価)と呼ばれます。<sup>62</sup>以下の手続について考えてみましょう。

```
(define (try a b) (if (= a 0) 1 b))
```

Evaluating `(try 0 (/ 1 0))` generates an error in Scheme. With lazy evaluation, there would be no error. Evaluating the expression would return 1, because the argument `(/ 1 0)` would never be evaluated.

`(try 0 (/ 1 0))` の評価は Scheme ではエラーを生成します。遅延評価ではエラーは現れません。その式の評価は 1 になります。なぜなら絶対に引数は評価されないためです。

An example that exploits lazy evaluation is the definition of a procedure `unless`

遅延評価を利用した例で、手続 `unless` の定義です。

```
(define (unless condition usual-value exceptional-value)
  (if condition exceptional-value usual-value))
```

that can be used in expressions such as

これは以下のような式で使用できます。

```
(unless (= b 0)
  (/ a b)
  (begin (display "exception: returning 0") 0))
```

This won't work in an applicative-order language because both the usual value and the exceptional value will be evaluated before `unless` is called (compare Exercise 1.6). An advantage of lazy evaluation is that some procedures, such as `unless`, can do useful computation even if evaluation of some of their arguments would produce errors or would not terminate.

---

<sup>62</sup>専門用語“lazy”と“normal-order”的間の違いはいささか曖昧(fuzzy)です。一般的に“lazy”は特定の評価機の仕組みを参照しますが、一方で“normal-order”は言語の意味を参照し、どんな特定の評価戦術からも独立しています。しかしこれは確かな区別ではありません。そして2つの専門用語は良く同義的に用いられています。

これは適用順序の言語では動きません。通常値と例外値の両方が `unless` が呼ばれる前に評価されるためです。(Exercise 1.6 と比較してみて下さい)。遅延評価の利点は `unless` のような手続は例えそれらの引数の幾つかの評価がエラーを発したり、停止しなかったとしても役立つ計算ができます。

If the body of a procedure is entered before an argument has been evaluated we say that the procedure is *non-strict* in that argument. If the argument is evaluated before the body of the procedure is entered we say that the procedure is *strict* in that argument.<sup>63</sup> In a purely applicative-order language, all procedures are strict in each argument. In a purely normal-order language, all compound procedures are non-strict in each argument, and primitive procedures may be either strict or non-strict. There are also languages (see Exercise 4.31) that give programmers detailed control over the strictness of the procedures they define.

引数の評価が完了する前に手続のボディに入ることを、手続がその引数において *non-strict*(非厳密) であると呼びます。もし引数が手続のボディに入る前に評価されたなら手続はその引数に対し *strict*(厳密) であると言います。<sup>64</sup> 純粹適用順序言語では全ての手続が全ての引数に対し厳密です。そしてプリミティブな手続は厳密にも非厳密にもなり得ます。またプログラマに彼等が定義する手続の厳密さに細かなコントロールを提供する言語もあります。(Exercise 4.31 参照)

A striking example of a procedure that can usefully be made non-strict is `cons` (or, in general, almost any constructor for data structures). One can do useful computation, combining elements to form data structures and operating on the resulting data structures, even if the values of the elements are not known. It makes perfect sense, for instance, to compute the length of a list without knowing the values of the individual elements in the list. We will

---

<sup>63</sup>The “strict” versus “non-strict” terminology means essentially the same thing as “applicative-order” versus “normal-order,” except that it refers to individual procedures and arguments rather than to the language as a whole. At a conference on programming languages you might hear someone say, “The normal-order language Hassle has certain strict primitives. Other procedures take their arguments by lazy evaluation.”

<sup>64</sup>“厳密”対“非厳密”的技術用語は本質的には“適用順序”対“正規順序”と同じことを言っています。しかし個別の手続と引数を言及しており、言語全体を言及してはいません。プログラミング言語のカソフアレンスでは誰かがこのようなことを言うのを聞くかもしれません。“正規順序言語の Hassle はいくつか厳密なプリミティブを持っている。他の手続はそれらの引数を遅延評価で取る。”

exploit this idea in Section 4.2.3 to implement the streams of Chapter 3 as lists formed of non-strict `cons` pairs.

実用性のため非厳密にすることができる手続の印象的な例には `cons`(または一般的に、ほとんど全てのデータ構造のコンストラクタが) あります。例えもし要素の値がわからなくても、データ構造を形成する要素を組み立て、結果のデータ構造上で操作する実用的な計算を行えます。例えばリストの長さをリスト内の個々の要素の値を知ること無しに計算することは完璧に意味があります。私達はこの考えを Section 4.2.3 で非厳密な `cons` ペアにより形成されたリストとして Chapter 3 のストリームを実装するために利用します。

**Exercise 4.25:** Suppose that (in ordinary applicative-order Scheme) we define `unless` as shown above and then define `factorial` in terms of `unless` as

(通常の適用順 Schemeにおいて)`unless`を上で示されたように定義し、次に `unless`を用いて以下のように `factorial`を定義する。

```
(define (factorial n)
  (unless (= n 1)
    (* n (factorial (- n 1)))
    1))
```

What happens if we attempt to evaluate `(factorial 5)`? Will our definitions work in a normal-order language?

もし `(factorial 5)` を評価したら何が起こるか? この定義は正規順序言語では動くか?

**Exercise 4.26:** Ben Bitdiddle and Alyssa P. Hacker disagree over the importance of lazy evaluation for implementing things such as `unless`. Ben points out that it's possible to implement `unless` in applicative order as a special form. Alyssa counters that, if one did that, `unless` would be merely syntax, not a procedure that could be used in conjunction with higher-order procedures. Fill in the details on both sides of the argument. Show how to implement `unless` as a derived expression (like `cond` or `let`), and give an example of a situation where it might be useful to have `unless` available as a procedure, rather than as a special form.

Ben Bitdiddle と Alyssa P. Hacker は `unless`のような物を実装するための遅延評価の重要性について意見が分かれた。Ben は

`unless` を適用順序でも特殊形式として実装可能である点を指摘した。Alyssa はもしそれを行えば `unless` はただ単に構文であり高階手続と連動して使用できる手続ではないと反論した。議論の両サイド上の詳細を埋めよ。`unless` をどのようにして派生手続として (`cond` や `let` のように) 実装するかを示せ。そして特殊形式ではなく手続として存在する `unless` を持つことが有効である状況の例を与える。

#### 4.2.2 An Interpreter with Lazy Evaluation

In this section we will implement a normal-order language that is the same as Scheme except that compound procedures are non-strict in each argument. Primitive procedures will still be strict. It is not difficult to modify the evaluator of Section 4.1.1 so that the language it interprets behaves this way. Almost all the required changes center around procedure application.

この節では Scheme と同じですが、複合手続が全ての引数に対して非厳密であることが異なる正規順言語を実装します。プリミティブな手続は依然として厳密です。Section 4.1.1 の評価機を、それが解釈する言語がこのように振る舞うように変更するのは難しくありません。ほとんど全ての必要な変更は手続適用が中心となります。

The basic idea is that, when applying a procedure, the interpreter must determine which arguments are to be evaluated and which are to be delayed. The delayed arguments are not evaluated; instead, they are transformed into objects called *thunks*.<sup>65</sup> The thunk must contain the information required to produce the value of the argument when it is needed, as if it had been evaluated at the time of the application. Thus, the thunk must contain the argument expression and the environment in which the procedure application is being evaluated.

基本的な考えは、手続を適用する時、インタプリタはどの引数が評価されるべきかと、どの引数が遅延されるべきかを決定しなければなりません。遅延化された引数は評価されません。その代わりにそれらは *thunks*(サンク) と呼ばれます。

---

<sup>65</sup>The word *thunk* was invented by an informal working group that was discussing the implementation of call-by-name in Algol 60. They observed that most of the analysis of (“thinking about”) the expression could be done at compile time; thus, at run time, the expression would already have been “thunk” about (Ingerman et al. 1960).

れるオブジェクトに変形されます。<sup>66</sup>サンクは引数の値を生成するために必要な情報を必要な時に含んでなければなりません。それはまるで適用時に評価されたかのようにです。従って、サンクは引数の式と手続適用がその中で評価される環境を持たなければなりません。

The process of evaluating the expression in a thunk is called *forcing*.<sup>67</sup> In general, a thunk will be forced only when its value is needed: when it is passed to a primitive procedure that will use the value of the thunk; when it is the value of a predicate of a conditional; and when it is the value of an operator that is about to be applied as a procedure. One design choice we have available is whether or not to *memoize* thunks, as we did with delayed objects in Section 3.5.1. With memoization, the first time a thunk is forced, it stores the value that is computed. Subsequent forcings simply return the stored value without repeating the computation. We'll make our interpreter memoize, because this is more efficient for many applications. There are tricky considerations here, however.<sup>68</sup>

サンク中の式の評価プロセスは*forcing*(強制)と呼ばれます。<sup>69</sup>一般的にはサンクはその値が必要になった時のみ強制されます。サンクの値を使用するプリミティブな手続に渡された時です。またオペレータの値であり手続として適

---

<sup>66</sup>thunk という単語は非公式な作業部会により考案されました。彼等は Algol 60 にて call-by-name の実装について議論していたのです。彼等は式のほとんどの分析は(“式についての考えは”)コンパイル時に行えることに気付きました。従って実行時には式は既に (Ingerman et al. 1960) に係わる“サンク”を持っていました。

<sup>67</sup>This is analogous to the use of `force` on the delayed objects that were introduced in Chapter 3 to represent streams. The critical difference between what we are doing here and what we did in Chapter 3 is that we are building delaying and forcing into the evaluator, and thus making this uniform and automatic throughout the language.

<sup>68</sup>Lazy evaluation combined with memoization is sometimes referred to as *call-by-need* argument passing, in contrast to *call-by-name* argument passing. (Call-by-name, introduced in Algol 60, is similar to non-memoized lazy evaluation.) As language designers, we can build our evaluator to memoize, not to memoize, or leave this an option for programmers (Exercise 4.31). As you might expect from Chapter 3, these choices raise issues that become both subtle and confusing in the presence of assignments. (See Exercise 4.27 and Exercise 4.29.) An excellent article by Clinger (1982) attempts to clarify the multiple dimensions of confusion that arise here.

<sup>69</sup>これは Chapter 3 でストリームを表現するのに導入された遅延化オブジェクト上で `force` を用いることに類似しています。ここで行っていることと、Chapter 3 で行ったことの重大な違いは、ここでは遅延化と強制を評価機の中に構築していることです。従ってこれを言語を通して同一化し、自動化しています。

用される時です。設計上の 1 つの選択として可能なこととして、Section 3.5.1 にて遅延化オブジェクトに我々がしたように、サンクを *memoize*(メモ化) するか否かがあります。メモ化を用いれば、サンクが初めて強制された時、計算された値が格納されます。続く強制は単純に演算を繰り返すことなく単純に格納された値を返します。私達はインタプリタをメモ化します。これはとても多くのアプリケーションに対して効率的だからです。しかし、これには用心しなければならない考慮点が存在します。<sup>70</sup>

## Modifying the evaluator

The main difference between the lazy evaluator and the one in Section 4.1 is in the handling of procedure applications in `eval` and `apply`.

遅延評価と Section 4.1 の評価との主な違いは `eval` と `apply` における手続適用の取扱に存在します。

The `application?` clause of `eval` becomes  
`eval` の `application?` 節は以下になります。

```
((application? exp)
 (apply (actual-value (operator exp) env)
        (operands exp)
        env))
```

This is almost the same as the `application?` clause of `eval` in Section 4.1.1. For lazy evaluation, however, we call `apply` with the operand expressions, rather than the arguments produced by evaluating them. Since we will need the environment to construct thunks if the arguments are to be delayed, we must pass this as well. We still evaluate the operator, because `apply` needs the actual procedure to be applied in order to dispatch on its type (primitive versus compound) and apply it.

---

<sup>70</sup> メモ化と組み合わされた遅延評価は時々、*call-by-need*(必要時呼出) 引数渡しと呼ばれます。*call-by-name*(名前呼出) 引数渡しと対照的です。(call-by-name は Algol 60 で導入されました。メモ化を行わない遅延評価と同類です)。言語設計者として、私達は評価機をメモ化することも、しないことも、プログラマに任せることも可能です (Exercise 4.31)。Chapter 3 からおわかりかと思いますが、これらの選択は微妙で、かつ混乱を招く問題を代入の存在において提起します。(Exercise 4.27 と Exercise 4.29 を参照)。Clinger (1982) による素晴らしい論文がここで提起される混乱の複数の特徴を明らかにしようと試みています。

これはSection 4.1.1の eval の application? 節とほとんど同じです。遅延評価のためにはしかし、apply をオペランド式と共に呼びます。それらを評価することで生成された引数と共にではありません。もし引数が遅延化されるのであれば環境にサンクを構築させる必要が出るので、これも渡さなければいけません。依然として演算子は評価します。apply は実際の手続が必要です。その型(プリミティブであるか複合であるか)に従って呼出と適用を行なうためです。

Whenever we need the actual value of an expression, we use

式の実際の値が必要になる度に、eval するだけの代わりに以下を用います。

```
(define (actual-value exp env)
  (force-it (eval exp env)))
```

instead of just eval, so that if the expression's value is a thunk, it will be forced.

これでもし式の値がサンクであれば強制されます。

Our new version of apply is also almost the same as the version in Section 4.1.1. The difference is that eval has passed in unevaluated operand expressions: For primitive procedures (which are strict), we evaluate all the arguments before applying the primitive; for compound procedures (which are non-strict) we delay all the arguments before applying the procedure.

また新しい版の apply もほとんどSection 4.1.1の版と同じです。違いは eval が未評価のオペランド式を通り直したことです。(厳密である) プリミティブな手続に対してはプリミティブを適用する前に全ての引数を評価します。(非厳密である) 複合手続に対しては全ての引数を手続に適用する前に遅延化します。

```
(define (apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env))) ;changed
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env) ;changed
           (procedure-environment procedure))))
```

```
(else (error "Unknown procedure type: APPLY"
            procedure))))
```

The procedures that process the arguments are just like `list-of-values` from [Section 4.1.1](#), except that `list-of-delayed-args` delays the arguments instead of evaluating them, and `list-of-arg-values` uses `actual-value` instead of `eval`:

引数を処理する手続は[Section 4.1.1](#)の `list-of-values` そっくりです。しかし `list-of-delayed-args` が引数を評価するのではなく遅延化すること、`list-of-arg-values` が `eval` の代わりに `actual-value` を用いることが違います。

```
(define (list-of-arg-values exps env)
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps)
                           env)
            (list-of-arg-values (rest-operands exps)
                               env))))
(define (list-of-delayed-args exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps)
                      env)
            (list-of-delayed-args (rest-operands exps)
                               env)))))
```

The other place we must change the evaluator is in the handling of `if`, where we must use `actual-value` instead of `eval` to get the value of the predicate expression before testing whether it is true or false:

評価機で変更しなければいけない他の場所は `if` の取扱の中にあります。そこでは `eval` の代わりに `actual-value` を使用して述語が真であるか偽であるかテストする前に、述語式の値を取らねばなりません。

```
(define (eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Finally, we must change the `driver-loop` procedure (Section 4.1.4) to use `actual-value` instead of `eval`, so that if a delayed value is propagated back to the read-eval-print loop, it will be forced before being printed. We also change the prompts to indicate that this is the lazy evaluator:

最後に、Section 4.1.4の `driver-loop` 手続を変更して、`eval` の代わりに `actual-value` を使用せねばなりません。そうすることでもし遅延化された値が REPL に伝播して返った場合に、表示される前に強制されます。またプロンプトも変更してこれが遅延評価であることを示します。

```
(define input-prompt ";;; L-Eval input:")
(define output-prompt ";;; L-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output
          (actual-value
            input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

With these changes made, we can start the evaluator and test it. The successful evaluation of the `try` expression discussed in Section 4.2.1 indicates that the interpreter is performing lazy evaluation:

これらの変更を行うことで、評価機を開始しテストすることができます。Section 4.2.1で議論された `try` 式の評価の成功はインタプリタが遅延評価を実行していることを示しています。

```
(define the-global-environment (setup-environment))
(driver-loop)
;;; L-Eval input:
(define (try a b) (if (= a 0) 1 b))
;;; L-Eval value:
ok
;;; L-Eval input:
(try 0 (/ 1 0))
;;; L-Eval value:
1
```

## Representing thunks

Our evaluator must arrange to create thunks when procedures are applied to arguments and to force these thunks later. A thunk must package an expression together with the environment, so that the argument can be produced later. To force the thunk, we simply extract the expression and environment from the thunk and evaluate the expression in the environment. We use `actual-value` rather than `eval` so that in case the value of the expression is itself a thunk, we will force that, and so on, until we reach something that is not a thunk:

私達の評価機は手続が引数に適用される時にサンクを作成することと、これらのサンクを後で強制することの準備をせねばなりません。サンクは式を環境と一緒にまとめなければなりません。そうすることで引数が後から生成することができます。サンクを強制するためには単純に式と環境をサンクから取り出しその環境の中で式を評価します。`eval`ではなく `actual-value` を用いることで式の値それ自身がサンクである場合にその強制をサンクでない物に辿り着くまで繰り返します。

```
(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))
```

One easy way to package an expression with an environment is to make a list containing the expression and the environment. Thus, we create a thunk as follows:

式と環境をまとめる1つの簡単な方法はその式と環境を含むリストを作成することです。従ってサンクを以下のように作成します。

```
(define (delay-it exp env)
  (list 'thunk exp env))
(define (thunk? obj)
  (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))
```

Actually, what we want for our interpreter is not quite this, but rather thunks that have been memoized. When a thunk is forced, we will turn it into an evaluated thunk by replacing the stored expression with its value and changing the `thunk` tag so that it can be recognized as already evaluated.

実際に私達のインタプリタに欲つしたものはこれではなく、メモ化されたサンクです。サンクが強制された時に格納された式とその値に置き換え、`thunk`タグを変更することでサンクをサンクが評価された物に変更することで、既に評価されたと認識させることができます。<sup>71</sup>

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value (thunk-exp obj)
                                     (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj)
                     result) ;exp をその値で置き換える
           (set-cdr! (cdr obj)
                     '()) ;不要な env を忘れる
           result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj)))
```

---

<sup>71</sup>Notice that we also erase the `env` from the thunk once the expression's value has been computed. This makes no difference in the values returned by the interpreter. It does help save space, however, because removing the reference from the thunk to the `env` once it is no longer needed allows this structure to be *garbage-collected* and its space recycled, as we will discuss in [Section 5.3](#).

一度式の値が計算されればサンクから `env` もまた消去します。これによりインタプリタにより返される値に違いは生じません。しかしこれはスペースを本当に節約します。サンクから `env` への参照が必要が無くなったら直ぐに削除することで、この構造が *garbage-collected*(ガベージコレクション、GC、ゴミ集め)され、そのスペースは [Section 5.3](#)で議論されるようにリサイクルされます。

Similarly, we could have allowed unneeded environments in the memoized delayed objects of [Section 3.5.1](#) to be garbage-collected, by having `memo-proc` do something like `(set! proc '())` to discard the procedure `proc` (which includes the environment in which the `delay` was evaluated) after storing its value.

同様に、[Section 3.5.1](#)のメモ化、遅延化されたオブジェクトの中の不必要的環境を GC されるように許可することも、`memo-proc` に `(set! proc '())` の様な何かを行わせて (`delay` が中で評価される環境を含む) 手続 `proc` をその値を格納した後に捨ててできました。

Notice that the same `delay-it` procedure works both with and without memoization.

同じ `delay-it` 手続がメモ化有りでも無しでも動作することに注意して下さい。

**Exercise 4.27:** Suppose we type in the following definitions to the lazy evaluator:

以下の定義を遅延評価機に入力したとする。

```
(define count 0)
(define (id x) (set! count (+ count 1)) x)
```

Give the missing values in the following sequence of interactions, and explain your answers.<sup>72</sup>

以下の一連の応答の欠けた値を与えよ。そして貴方の回答について説明せよ。<sup>73</sup>

```
(define w (id (id 10)))
;; L-Eval input:
count
;; L-Eval value:
⟨response⟩
;; L-Eval input:
w
;; L-Eval value:
⟨response⟩
;; L-Eval input:
count
;; L-Eval value:
⟨response⟩
```

**Exercise 4.28:** Eval uses `actual-value` rather than `eval` to evaluate the operator before passing it to `apply`, in order to force the

---

<sup>72</sup>This exercise demonstrates that the interaction between lazy evaluation and side effects can be very confusing. This is just what you might expect from the discussion in Chapter 3.

<sup>73</sup>この課題は遅延評価と副作用の間の応答が大きな混乱の原因となりえることを実演している。これこそがChapter 3の議論からあなたが予期するかもしれないことである。

value of the operator. Give an example that demonstrates the need for this forcing.

`eval` は `eval` でなく `actual-value` を用いて `apply` に渡す前に演算子を評価する。演算子の値を強制するためである。この強制の必要性を実演する例を与える。

**Exercise 4.29:** Exhibit a program that you would expect to run much more slowly without memoization than with memoization. Also, consider the following interaction, where the `id` procedure is defined as in [Exercise 4.27](#) and `count` starts at 0:

メモ化しない場合に、メモ化した場合よりも非常に遅く実行されると予測するプログラムを示せ。また以下の応答について考えよ。`id` 手続は [Exercise 4.27](#) と同じに定義され `count` は 0 から始める。

```
(define (square x) (* x x))
;; L-Eval input:
(square (id 10))
;; L-Eval value:
⟨response⟩
;; L-Eval input:
count
;; L-Eval value:
⟨response⟩
```

Give the responses both when the evaluator memoizes and when it does not.

評価機がメモ化された場合とメモ化されない場合の両方について応答を与える。

**Exercise 4.30:** Cy D. Fect, a reformed C programmer, is worried that some side effects may never take place, because the lazy evaluator doesn't force the expressions in a sequence. Since the value of an expression in a sequence other than the last one is not used (the expression is there only for its effect, such as assigning to a variable or printing), there can be no subsequent use of this value (e.g., as an argument to a primitive procedure) that will cause it to be forced. Cy thus thinks that when evaluating sequences, we must force all expressions in the sequence except the final one.

He proposes to modify `eval-sequence` from Section 4.1.1 to use `actual-value` rather than `eval`:

Cy D. Fect は元 C 言語プログラマである。彼はいくつかの side effects(副作用)が起こらないのではないかと心配している。遅延評価が列内の式に強制を行わないためである。最後の 1つ以外の列内の式の値は使用されないため(式は変数への代入や表示等の作用のためだけに存在している)、この値の強制を引き起こす後の使用は存在しない(例えばプリミティブな手続の引数として)。Cy は従って列を評価する時には列内の最後の 1つを除いた全ての式を評価せねばならないと考えた。彼は Section 4.1.1 の `eval-sequence` を変更し `eval` でなく `actual-value` を使用することを提案した。

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (actual-value (first-exp exps) env)
              (eval-sequence (rest-exp exps) env))))
```

a Ben Bitdiddle thinks Cy is wrong. He shows Cy the `for-each` procedure described in Exercise 2.23, which gives an important example of a sequence with side effects:

Ben Bitdiddle は Cy が間違っていると考えた。彼は Cy に Exercise 2.23 で説明された `for-each` 手續を見せた。これは副作用を伴う列の重要な例を与える。

```
(define (for-each proc items)
  (if (null? items)
      'done
      (begin (proc (car items))
             (for-each proc (cdr items)))))
```

He claims that the evaluator in the text (with the original `eval-sequence`) handles this correctly:

彼はテキストの評価機(オリジナルの `eval-sequence` を持つ物)はこれを正しく取り扱うと主張した。

```
; ; ; L-Eval input:
(for-each (lambda (x) (newline) (display x))
          (list 57 321 88))
```

```
57  
321  
88  
;;; L-Eval value:  
done
```

Explain why Ben is right about the behavior of **for-each**.

なぜ Ben が **for-each** の振舞について正しいのか説明せよ。

- b Cy agrees that Ben is right about the **for-each** example, but says that that's not the kind of program he was thinking about when he proposed his change to **eval-sequence**. He defines the following two procedures in the lazy evaluator:

Cy は Ben が **for-each** について正しいのは同意した。しかし彼が **eval-sequence** に対する変更を提案した時、彼が考えていたのはこの種のプログラムではないと言った。彼は以下の 2 つの遅延評価の手続を定義した。

```
(define (p1 x)
  (set! x (cons x '(2)))
  x)
(define (p2 x)
  (define (p e)
    e
    x)
  (p (set! x (cons x '(2)))))
```

What are the values of `(p1 1)` and `(p2 1)` with the original **eval-sequence**? What would the values be with Cy's proposed change to **eval-sequence**?

オリジナルの **eval-sequence** を用いた時、`(p1 1)` と `(p2 1)` の値はいくらか? Cy が提案した **eval-sequence** への変更を用いた時には値はどうなるか?

- c Cy also points out that changing **eval-sequence** as he proposes does not affect the behavior of the example in part a. Explain why this is true.

Cy は彼が提案した通りの eval-sequence への変更は a の例の振舞に影響を与えないと指摘した。なぜこれが正しいのか説明せよ。

- d How do you think sequences ought to be treated in the lazy evaluator? Do you like Cy's approach, the approach in the text, or some other approach?

遅延評価では列はどのように扱われるべきと考えるか? あなたは Cy のアプローチ、テキストのアプローチ、または他のアプローチのどれを好むか?

**Exercise 4.31:** The approach taken in this section is somewhat unpleasant, because it makes an incompatible change to Scheme. It might be nicer to implement lazy evaluation as an *upward-compatible extension*, that is, so that ordinary Scheme programs will work as before. We can do this by extending the syntax of procedure declarations to let the user control whether or not arguments are to be delayed. While we're at it, we may as well also give the user the choice between delaying with and without memoization. For example, the definition

この節で取り上げられた取り組み方は少々、不愉快である。Scheme に対して互換性のない変更を行なうためだ。遅延評価を *upward-compatible extension*(上位互換性のある拡張) として実装するほうがより良いだろう。それは通常の Scheme プログラムが依然と同じように働くということである。これをユーザに引数が遅延されるか、されないかをコントロールするように手続定義の構文を拡張することで可能である。それを行う間、ユーザに遅延をメモ化させるか、させないかの選択も同様に与えることができるだろう。例えば、以下の定義は

```
(define (f a (b lazy) c (d lazy-memo))
  ...)
```

would define f to be a procedure of four arguments, where the first and third arguments are evaluated when the procedure is called, the second argument is delayed, and the fourth argument is both delayed and memoized. Thus, ordinary procedure definitions will produce the same behavior as ordinary Scheme, while adding the

`lazy-memo` declaration to each parameter of every compound procedure will produce the behavior of the lazy evaluator defined in this section. Design and implement the changes required to produce such an extension to Scheme. You will have to implement new syntax procedures to handle the new syntax for `define`. You must also arrange for `eval` or `apply` to determine when arguments are to be delayed, and to force or delay arguments accordingly, and you must arrange for forcing to memoize or not, as appropriate.

`f` は 4 つの引数の手続であり、最初と 3 番目の引数は手続が呼ばれた時に評価され、2 番目は遅延化され、4 番目は遅延化とメモ化が行われる。従って通常の手続定義は通常の Scheme と同じ振舞を行なうが、`lazy-memo` 宣言を各複合手続の各パラメタに追加することでこの節で定義された遅延評価の振舞を行う。この変更の設計と実装は Scheme に対しそのような拡張の生成を必要とする。あなたは `define` に対する新しい構文を取り扱う新しい構文手続を実装しなければならない。また引数がいつ遅延化されるか、そしていつ強制するか、またはそれに応じて引数を遅延化するか決定するため `eval` や `apply` に対して準備もしなければならない。同時に強制に対してメモ化するか、しないかも適切に準備すること。

### 4.2.3 Streams as Lazy Lists

In Section 3.5.1, we showed how to implement streams as delayed lists. We introduced special forms `delay` and `cons-stream`, which allowed us to construct a “promise” to compute the `cdr` of a stream, without actually fulfilling that promise until later. We could use this general technique of introducing special forms whenever we need more control over the evaluation process, but this is awkward. For one thing, a special form is not a first-class object like a procedure, so we cannot use it together with higher-order procedures.<sup>74</sup> Additionally, we were forced to create streams as a new kind of data object similar but not identical to lists, and this required us to reimplement many ordinary list operations (`map`, `append`, and so on) for use with streams.

Section 3.5.1ではどのようにストリームを遅延化されたリストとして実装するかについて示しました。特殊形式 `delay` と `cons-stream` 導入しました。こ

---

<sup>74</sup>This is precisely the issue with the `unless` procedure, as in Exercise 4.26.

のことは私達にストリームの `cdr` を求める “promise”(プロミス、約束) を、実際には後になるまではプロミスを実行すること無しに構築することを可能にしました。

With lazy evaluation, streams and lists can be identical, so there is no need for special forms or for separate list and stream operations. All we need to do is to arrange matters so that `cons` is non-strict. One way to accomplish this is to extend the lazy evaluator to allow for non-strict primitives, and to implement `cons` as one of these. An easier way is to recall ([Section 2.1.3](#)) that there is no fundamental need to implement `cons` as a primitive at all. Instead, we can represent pairs as procedures:<sup>75</sup>

遅延評価ではストリームとリストは同一にできます。そのため特殊形式やリストとストリームの命令を分ける必要はありません。私達が行わなければならぬこと全ては `cons` が非厳密になるよう問題を準備することです。これを達成する 1 つの方法は遅延評価を拡張しプリミティブにも非厳密を許し `cons` をこれらの内の 1 つとすることです。より簡単な方法は ([Section 2.1.3](#)) の `cons` をプリミティブとして実装する必要性は本質的には全く無いということを思い出すことです。その代わりに、ペアは手続として表現可能です。<sup>76</sup>

```
(define (cons x y) (lambda (m) (m x y)))
(define (car z) (z (lambda (p q) p)))
(define (cdr z) (z (lambda (p q) q)))
```

In terms of these basic operations, the standard definitions of the list operations will work with infinite lists (streams) as well as finite ones, and the stream operations can be implemented as list operations. Here are some examples:

これらの基本的な命令において、リスト命令の標準定義は無限リスト (スト

---

<sup>75</sup>This is the procedural representation described in [Exercise 2.4](#). Essentially any procedural representation (e.g., a message-passing implementation) would do as well. Notice that we can install these definitions in the lazy evaluator simply by typing them at the driver loop. If we had originally included `cons`, `car`, and `cdr` as primitives in the global environment, they will be redefined. (Also see [Exercise 4.33](#) and [Exercise 4.34](#).)

<sup>76</sup>これが[Exercise 2.4](#)で説明された手続表現です。本質的にはどんな手続表現(例えばメッセージパッキングによる実装)も同じことが行えるでしょう。これらの定義を遅延評価に単純にドライバーループにて型を付けることのみでインストールできることに注意して下さい。もし私達が元々`cons`, `car`, `cdr`をグローバル環境のプリミティブとして含めていたのならば、それらは再定義されるでしょう。(Exercise 4.33と Exercise 4.34も参照して下さい。

リーム)と同様に有限な物としても働きます。そしてストリーム命令はリスト命令として実装可能です。以下にいくつかの例を示します。

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items)) (map proc (cdr items)))))
(define (scale-list items factor)
  (map (lambda (x) (* x factor)) items))
(define (add-lists list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        (else (cons (+ (car list1) (car list2))
                    (add-lists (cdr list1) (cdr list2))))))
(define ones (cons 1 ones))
(define integers (cons 1 (add-lists ones integers)))
;;; L-Eval input:
(list-ref integers 17)
;;; L-Eval value:
18
```

Note that these lazy lists are even lazier than the streams of Chapter 3: The `car` of the list, as well as the `cdr`, is delayed.<sup>77</sup> In fact, even accessing the `car` or `cdr` of a lazy pair need not force the value of a list element. The value will be forced only when it is really needed—e.g., for use as the argument of a primitive, or to be printed as an answer.

これらの遅延リストはChapter 3のストリームよりもさらに遅延化されています。リストの `car` も `cdr` と同様に遅延化されます。<sup>78</sup> 実際に、遅延化ペアの

---

<sup>77</sup>This permits us to create delayed versions of more general kinds of list structures, not just sequences. Hughes 1990 discusses some applications of “lazy trees.”

<sup>78</sup>このことはより一般的な種類の、ただの列ではない、リスト構造の遅延化版を作ることを可能にします。Hughes 1990は“遅延化木”的いくつかのアプリケーションについて議論しています。

`car` や `cdr` に対するアクセスさえもリスト要素の値を強制する必要がありません。その値はそれが本当に必要になった場合—例えばプリミティブの引数としてや回答として表示される場合に—強制されることになります。

Lazy pairs also help with the problem that arose with streams in Section 3.5.4, where we found that formulating stream models of systems with loops may require us to sprinkle our programs with explicit `delay` operations, beyond the ones supplied by `cons-stream`. With lazy evaluation, all arguments to procedures are delayed uniformly. For instance, we can implement procedures to integrate lists and solve differential equations as we originally intended in Section 3.5.4:

遅延化ペアはまたSection 3.5.4でストリームに対して提起された問題に対する手助けにもなります。その時はループを伴なうシステムのストリームモデルを定式化することは明示的な `delay` 命令を `cons-stream` で提供されるものを越えてプログラムの中に撒き散らすことを必要としました。遅延評価では全ての手続の引数は遅延化に統一されています。例えばリストと統合する手続を実装し微分方程式をSection 3.5.4で人々意図したように実装することが可能です。

```
(define (integral integrand initial-value dt)
  (define int
    (cons initial-value
          (add-lists (scale-list integrand dt) int)))
  int)
(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (map f y))
  y)
;;; L-Eval input:
(list-ref (solve (lambda (x) x) 1 0.001) 1000)
;;; L-Eval value:
2.716924
```

**Exercise 4.32:** Give some examples that illustrate the difference between the streams of Chapter 3 and the “lazier” lazy lists described in this section. How can you take advantage of this extra laziness?

Chapter 3のストリームとこの節で説明された“より遅延化された”

遅延化リストの間の違いを説明する例をいくつか上げよ。この拡張された遅延性の利点をどのように活用するか？

**Exercise 4.33:** Ben Bitdiddle tests the lazy list implementation given above by evaluating the expression: (car '(a b c)).

Ben Bitdiddle は上で与えられた遅延リスト実装を式 (car '(a b c)) を評価することでテストした。

To his surprise, this produces an error. After some thought, he realizes that the “lists” obtained by reading in quoted expressions are different from the lists manipulated by the new definitions of `cons`, `car`, and `cdr`. Modify the evaluator’s treatment of quoted expressions so that quoted lists typed at the driver loop will produce true lazy lists.

驚いたことにこれはエラーを生じる。幾らか考えた後に、彼はクオートされた式を読み込むことで得られた“リスト”が`cons`, `car`, `cdr` の新しい定義で操作されたリストから異なることに気が付いた。評価機のクオートされた式の扱いを修正しドライバーループで型付けされたクオートされたリストが正しい遅延リストを生成するようにせよ。

**Exercise 4.34:** Modify the driver loop for the evaluator so that lazy pairs and lists will print in some reasonable way. (What are you going to do about infinite lists?) You may also need to modify the representation of lazy pairs so that the evaluator can identify them in order to print them.

評価機のドライバーループを変更し遅延化したペアとリストが何らかの妥当な方法で表示を行うようにせよ。(無限リストに対しては何を行なうか?)。遅延化ペアの表現も変更が必要になるだろう。評価機がそれらを表示するためにそれらを判別することができるようにするためである。

## 4.3 Variations on a Scheme — Nondeterministic Computing

In this section, we extend the Scheme evaluator to support a programming paradigm called *nondeterministic computing* by building into the evaluator a facility to support automatic search. This is a much more profound change to the language than the introduction of lazy evaluation in [Section 4.2](#).

この節では*nondeterministic computing*(非決定性演算)と呼ばれるプログラミングパラダイムをサポートするよう、評価機の中に自動的な探索をサポートするための機能を構築することで、Scheme評価機を拡張します。これは[Section 4.2](#)での遅延評価の導入に比べ、とても深い言語への変更です。

Nondeterministic computing, like stream processing, is useful for “generate and test” applications. Consider the task of starting with two lists of positive integers and finding a pair of integers—one from the first list and one from the second list—whose sum is prime. We saw how to handle this with finite sequence operations in [Section 2.2.3](#) and with infinite streams in [Section 3.5.3](#). Our approach was to generate the sequence of all possible pairs and filter these to select the pairs whose sum is prime. Whether we actually generate the entire sequence of pairs first as in [Chapter 2](#), or interleave the generating and filtering as in [Chapter 3](#), is immaterial to the essential image of how the computation is organized.

非決定性演算はストリーム処理のように、“生成してテストする”アプリケーションに対して便利です。正の整数の2つのリストを用いて開始する、整数のペアを見つけるタスクについて考えます 1つは最初のリストから、もう1つは別のリストから取得し、その和は素数となります。私達はこの問題をどのように扱うかについて、[Section 2.2.3](#)では有限列の命令を用いて、[Section 3.5.3](#)では無限ストリームを用いる方法について学びました。私達の取り組み方は全ての可能なペアを生成し、これらから和が素数になるペアを選択するという方法でした。実際にペアの列全体を最初に生成する[Chapter 2](#)や、生成とフィルタリングを相互配置する [Chapter 3](#)に係わらず演算がどのように体系化されるかの本質的なイメージに対しては重要ではありませんでした。

The nondeterministic approach evokes a different image. Imagine simply that we choose (in some way) a number from the first list and a number from the second list and require (using some mechanism) that their sum be prime. This is expressed by following procedure:

非決定性の取り組み方は異なるイメージを喚起します。単純に(何らかの方法で)最初のリストから数値を選択し、別の数値を2つ目のリストから選択し、(何らかの仕組みで)それらの和が素数であることを要求とすると想像してみてください。これは以下の手続により表現されます。

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))
```

It might seem as if this procedure merely restates the problem, rather than specifying a way to solve it. Nevertheless, this is a legitimate nondeterministic program.<sup>79</sup>

この手続はただ単に問題を言い直したに過ぎなく、解法を指定したように見えないかもしれません。それにもかかわらず、これは正規の非決定性プログラムです。<sup>80</sup>

The key idea here is that expressions in a nondeterministic language can have more than one possible value. For instance, `an-element-of` might return any element of the given list. Our nondeterministic program evaluator will work by automatically choosing a possible value and keeping track of the choice. If a subsequent requirement is not met, the evaluator will try a different choice, and it will keep trying new choices until the evaluation succeeds, or until we run out of choices. Just as the lazy evaluator freed the programmer from the details

---

<sup>79</sup>We assume that we have previously defined a procedure `prime?` that tests whether numbers are prime. Even with `prime?` defined, the `prime-sum-pair` procedure may look suspiciously like the unhelpful “pseudo-Lisp” attempt to define the square-root function, which we described at the beginning of Section 1.1.7. In fact, a square-root procedure along those lines can actually be formulated as a nondeterministic program. By incorporating a search mechanism into the evaluator, we are eroding the distinction between purely declarative descriptions and imperative specifications of how to compute answers. We’ll go even farther in this direction in

<sup>80</sup>以前に数値が素数であるかをテストする手続 `prime?` を定義しました。例え `prime?` が定義されていたとしても、`prime-sum-pair` 手続は、Section 1.1.7の最初で説明した助けにならない“擬似 Lisp”による平方根関数の定義の試みのように疑わしく見えるかもしれません。現実に、あれらの行に沿った平方根手続が実際に非決定性プログラムとして定式化することができます。評価機に探索の仕組みを合併することで、どのようにして回答を計算するかについての純粋な宣言型の記述と命令型の仕様の間の区別を侵食していくきます。私達はSection 4.4にてこの方向へとさらに進みます。

of how values are delayed and forced, the nondeterministic program evaluator will free the programmer from the details of how choices are made.

ここでの鍵となる考えは、非決定性言語における式は1つ以上の可能な値を持つことができるということです。例えば `an-element-of` は与えられたリストのどの要素でも返すことが有り得ます。私達の非決定性プログラム評価機は自動的に可能な値を選択しその選択を追跡することで働きます。もし続く要件に合わなければ、評価機は異なる選択を試します。そして評価が成功するまで、または選択肢が無くなるまで、新しい選択を試し続けます。遅延評価がプログラマを値がどのように遅延化され強制されるかの詳細から解放されたのと同様に、非決定性プログラムの評価機はプログラムを選択がどのように行われるかの詳細から解放します。

It is instructive to contrast the different images of time evoked by nondeterministic evaluation and stream processing. Stream processing uses lazy evaluation to decouple the time when the stream of possible answers is assembled from the time when the actual stream elements are produced. The evaluator supports the illusion that all the possible answers are laid out before us in a timeless sequence. With nondeterministic evaluation, an expression represents the exploration of a set of possible worlds, each determined by a set of choices. Some of the possible worlds lead to dead ends, while others have useful values. The nondeterministic program evaluator supports the illusion that time branches, and that our programs have different possible execution histories. When we reach a dead end, we can revisit a previous choice point and proceed along a different branch.

非決定性評価とストリーム処理により起こった時間の異なるイメージの対比は示唆的です。ストリーム処理は遅延評価を可能な答のストリームが組まれた時間を実際のストリーム要素が生成された時間から分離します。評価機は全ての可能な回答が私達の前に永遠の列の中に横たわっているというイリュージョンを支えます。非決定性評価機では式は可能な世界の集合の調査を表現します。それぞれは選択の集合により判断されます。可能なせかいのいくつかは行き止まりへと導き、一方、他は役立つ値へと導きます。非決定性プログラム評価機は時間の分岐と私達のプログラムが異なる可能な実行履歴を持つというイリュージョンを支えます。行き止まりに辿り着いた時には直前の選択地点に戻り、異なる分岐に従って進むことが可能です。

The nondeterministic program evaluator implemented below is called the `amb` evaluator because it is based on a new special form called `amb`. We can type the above definition of `prime-sum-pair` at the `amb` evaluator driver loop

(along with definitions of `prime?`, `an-element-of`, and `require`) and run the procedure as follows:

以下で実装される非決定性プログラム評価機は `amb` 評価機と呼ばれます。それが `amb` と呼ばれる新しい特殊形式に基づくためです。上記の `prime-sum-pair` の定義を `amb` 評価機ドライバループにて (`prime?`, `an-element-of`, `require` と一緒に) 型を付け、以下のように手続を実行することができます。

```
;;; Amb-Eval input:  
(prime-sum-pair '(1 3 5 8) '(20 35 110))  
;;; Starting a new problem  
;;; Amb-Eval value:  
(3 20)
```

The value returned was obtained after the evaluator repeatedly chose elements from each of the lists, until a successful choice was made.

返り値は評価機が条件に合う選択が行われるまで繰り返し各リストから要素を選択した後に取得されました。

Section 4.3.1 introduces `amb` and explains how it supports nondeterminism through the evaluator's automatic search mechanism. Section 4.3.2 presents examples of nondeterministic programs, and Section 4.3.3 gives the details of how to implement the `amb` evaluator by modifying the ordinary Scheme evaluator.

Section 4.3.1は `amb` を紹介しそれがどのように非決定性を評価機の自動探索の仕組みを通してサポートするかについて説明します。Section 4.3.2は非決定性プログラムの例を与え、Section 4.3.3は `amb` の実装方法の詳細を通常のScheme評価機を変更することで与えます。

### 4.3.1 Amb and Search

To extend Scheme to support nondeterminism, we introduce a new special form called `amb`.<sup>81</sup> The expression

Scheme に非決定性を対応するよう拡張するために、`amb` と呼ばれる新しい特殊形式を導入します。<sup>82</sup>

`(amb <e1> <e2> ... <en>)`

<sup>81</sup>The idea of `amb` for nondeterministic programming was first described in 1961 by John McCarthy (see McCarthy 1963).

<sup>82</sup>非決定性プログラミングのための `amb` の考えは 1961 年に最初に John McCarthy により説明されました。(McCarthy 1963 参照)。

returns the value of one of the  $n$  expressions  $\langle e_i \rangle$  “ambiguously.” For example, the expression

上の式は  $n$  個の式  $\langle e_i \rangle$  の内 1 つの値を “ambiguously”(曖昧に) 返します。例えば以下の式は

```
(list (amb 1 2 3) (amb 'a 'b))
```

can have six possible values:

以下の 6 つの値の可能性があります。

```
(1 a) (1 b) (2 a) (2 b) (3 a) (3 b)
```

Amb with a single choice produces an ordinary (single) value.

单一の選択を行う amb は通常の(单一の) 値を生成します。

Amb with no choices—the expression (amb)—is an expression with no acceptable values. Operationally, we can think of (amb) as an expression that when evaluated causes the computation to “fail”: The computation aborts and no value is produced. Using this idea, we can express the requirement that a particular predicate expression p must be true as follows:

選択を行わない amb—式 (amb)—は受け入れられる値の無い式です。操作上、(amb) を評価された時に演算に対し “fail”(失敗) を起こさせると考えることができます。演算は異常終了し、何の値も生成されません。この考え方を用いて、ある特定の述語式 p が真でなければならないことを以下のように表現可能です。

```
(define (require p) (if (not p) (amb)))
```

With amb and require, we can implement the an-element-of procedure used above:

amb と require を用いて上で使用された an-element-of を実装可能です。

```
(define (an-element-of items)
  (require (not (null? items)))
  (amb (car items) (an-element-of (cdr items))))
```

An-element-of fails if the list is empty. Otherwise it ambiguously returns either the first element of the list or an element chosen from the rest of the list.

an-element-of はリストが空である場合には失敗します。そうでなければリストの最初の要素か、リストの残りの要素から選択された要素を曖昧に返します。

We can also express infinite ranges of choices. The following procedure potentially returns any integer greater than or equal to some given  $n$ :

無限の範囲の選択も表現可能です。以下の手続は可能性としてある与えられた  $n$  に等しいか大きな任意の整数を返します。

```
(define (an-integer-starting-from n)
  (amb n (an-integer-starting-from (+ n 1))))
```

This is like the stream procedure `integers-starting-from` described in Section 3.5.2, but with an important difference: The stream procedure returns an object that represents the sequence of all integers beginning with  $n$ , whereas the `amb` procedure returns a single integer.<sup>83</sup>

これはまるでSection 3.5.2で説明されたストリーム手続 `integers-starting-from` の様です。しかし重要な違いがあります。ストリーム処理は  $n$  で始まる全ての整数の列を表すオブジェクトを返します。一方、`amb` 手続は単一の整数を返します。<sup>84</sup>

Abstractly, we can imagine that evaluating an `amb` expression causes time to split into branches, where the computation continues on each branch with one of the possible values of the expression. We say that `amb` represents a *nondeterministic choice point*. If we had a machine with a sufficient number of processors that could be dynamically allocated, we could implement the search in a straightforward way. Execution would proceed as in a sequential machine, until an `amb` expression is encountered. At this point, more processors would be allocated and initialized to continue all of the parallel executions implied by the choice. Each processor would proceed sequentially as if it were the only choice, until it either terminates by encountering a failure, or it further subdivides, or it finishes.<sup>85</sup>

---

<sup>83</sup>In actuality, the distinction between nondeterministically returning a single choice and returning all choices depends somewhat on our point of view. From the perspective of the code that uses the value, the nondeterministic choice returns a single value. From the perspective of the programmer designing the code, the nondeterministic choice potentially returns all possible values, and the computation branches so that each value is investigated separately.

<sup>84</sup>本当は非決定的に单一の選択を返すことと全ての選択を返すこととの違いは私達の視点に幾分、依存します。値を使用するコードの視点からは非決定性による選択は单一の値を返します。コードを設計するプログラマの視点からは非決定性による選択は潜在的に全ての可能な値を返します。そして各値が個別に調査されるように演算は分岐するのです。

<sup>85</sup>One might object that this is a hopelessly inefficient mechanism. It might require

抽象的に、`amb` 式を評価することが時間に対して分岐を起こさせ、演算は各分岐上にて式の可能な値の 1 つに上で続行するのだと想像することができます。`amb` が *nondeterministic choice point*(非決定性選択点) を表現すると言えます。もし私達が動的に獲得できる十分な数のプロセッサを持つ計算機を持っているならば、探索を簡単な方法で実装できるでしょう。実行は `amb` 式に遭遇するまでは逐次的に行われます。遭遇した時点では多くのプロセッサが獲得され選択により暗示された全ての並列実行を続けるために初期化されます。各プロセッサは選択がそれしか無かったかのように逐次的に続行します。その処理は失敗に遭遇して停止するか、さらなる分岐が起こるか、完了するまで続けられます。<sup>86</sup>

On the other hand, if we have a machine that can execute only one process (or a few concurrent processes), we must consider the alternatives sequentially. One could imagine modifying an evaluator to pick at random a branch to follow whenever it encounters a choice point. Random choice, however, can easily lead to failing values. We might try running the evaluator over and over, making random choices and hoping to find a non-failing value, but it is better to *systematically search* all possible execution paths. The `amb` evaluator that we will develop and work with in this section implements a systematic search as follows: When the evaluator encounters an application of `amb`, it initially selects the first alternative. This selection may itself lead to a further choice. The evaluator will always initially choose the first alternative at each choice point. If a choice results in a failure, then the evaluator automagically<sup>87</sup> backtracks

---

millions of processors to solve some easily stated problem this way, and most of the time most of those processors would be idle. This objection should be taken in the context of history. Memory used to be considered just such an expensive commodity. In 1964 a megabyte of RAM cost about \$400,000. Now every personal computer has many megabytes of RAM, and most of the time most of that RAM is unused. It is hard to underestimate the cost of mass-produced electronics.

<sup>86</sup> これは絶望的に非効率だと異議を唱える人がいるかもしれません。この方法では簡単に規定された問題を解くのに数百万のプロセッサを必要とするかもしれません。そしてそれらの多くのプロセッサはほとんどの時間をアイドル状態(遊休状態)になるでしょう。この異議は歴史の文脈で捕えられるべきです。メモリはとても高価な消費財だと考えられてきました。1964 年にはメガバイトの RAM は \$400,000 の費用が掛かりました。現在では全ての PC が多数のメガバイトの RAM を積んでいます。そしてほとんどの時間でほとんどのメモリは利用されていません。大量生産された電子製品のコストを過小評価することは難しいことです。

<sup>87</sup> Automagically: “Automatically, but in a way which, for some reason (typically because it is too complicated, or too ugly, or perhaps even too trivial), the speaker

to the most recent choice point and tries the next alternative. If it runs out of alternatives at any choice point, the evaluator will back up to the previous choice point and resume from there. This process leads to a search strategy known as *depth-first search* or *chronological backtracking*.<sup>88</sup>

一方で、もし私達が1つのプロセス(またはいくつかの並行プロセス)しか実行できない計算機を持っている場合には逐次的に動作する代替法を考えねばなりません。1つの方法としては評価機を選択点に辿り着いた時に無作為に分岐を選択するよう変更することが考えられるでしょう。しかし無作為な選択は簡単に失敗する値へと導きます。評価機を何度も何度も実行し無作為な選択を行い失敗しない値を見つけることを期待するかもしれません。しかし全ての可能な実行パス(実行経路)を *systematically search*(体系的探索)をしたほうがよ

---

doesn't feel like explaining." (Steele et al. 1983, Raymond 1993)

<sup>88</sup>The integration of automatic search strategies into programming languages has had a long and checkered history. The first suggestions that nondeterministic algorithms might be elegantly encoded in a programming language with search and automatic backtracking came from Robert Floyd (1967). Carl Hewitt (1969) invented a programming language called Planner that explicitly supported automatic chronological backtracking, providing for a built-in depth-first search strategy. Sussman et al. (1971) implemented a subset of this language, called MicroPlanner, which was used to support work in problem solving and robot planning. Similar ideas, arising from logic and theorem proving, led to the genesis in Edinburgh and Marseille of the elegant language Prolog (which we will discuss in Section 4.4). After sufficient frustration with automatic search, McDermott and Sussman (1972) developed a language called Conniver, which included mechanisms for placing the search strategy under programmer control. This proved unwieldy, however, and Sussman and Stallman 1975 found a more tractable approach while investigating methods of symbolic analysis for electrical circuits. They developed a non-chronological backtracking scheme that was based on tracing out the logical dependencies connecting facts, a technique that has come to be known as *dependency-directed backtracking*. Although their method was complex, it produced reasonably efficient programs because it did little redundant search. Doyle (1979) and McAllester (1978; 1980) generalized and clarified the methods of Stallman and Sussman, developing a new paradigm for formulating search that is now called *truth maintenance*. Modern problem-solving systems all use some form of truth-maintenance system as a substrate. See Forbus and deKleer 1993 for a discussion of elegant ways to build truth-maintenance systems and applications using truth maintenance. Zabih et al. 1987 describes a nondeterministic extension to Scheme that is based on `amb`; it is similar to the interpreter described in this section, but more sophisticated, because it uses dependency-directed backtracking rather than chronological backtracking. Winston 1992 gives an introduction to both kinds of backtracking.

り良いです。私達がこの節で開発し働きかける `amb` 評価機は体系的探索を次のように実装します。評価機が `amb` の適用に遭遇した場合に初期値として最初の選択肢を選択します。この選択それ自身がさらなる選択へと導きます。評価機は常に初期値として最初の選択肢を各選択点にて選択します。もし選択の結果が失敗となれば評価機は `automagically`<sup>89</sup> に最も最近の選択に *backtracks*(バックトラック、引き返す) し、次の選択肢を試行します。もしどこかの選択点において、全ての選択肢を使用してしまえば、評価機は以前の選択点へと戻りそこから再開します。この処理は *depth-first search*(深さ優先探索) または *chronological backtracking*(クロノロジカルバックトラック、年代順バックトラック) として知られる探索戦略へと導きます。<sup>90</sup>

<sup>89</sup>automagically: “自動的に、しかし幾つかの理由で(典型的には複雑過ぎて、または酷すぎて、または恐らくさらにどうでも良すぎて)話者が説明する気にならない” (Steele et al. 1983, Raymond 1993)

<sup>90</sup>自動的な探索戦略のプログラミング言語への統合は長く功罪相半ばする歴史があります。非決定性アルゴリズムが美しくプログラミング言語へと探索と自動的バックトラックと共に組込まれただろう最初の提案は Robert Floyd (1967) により行われました。Carl Hewitt (1969) は Planner と呼ばれるプログラミング言語を発明しましたが、これは明示的に自動的なクロノロジカルバックトラックをサポートし、組込の深さ優先探索戦略を提供していました。Sussman et al. (1971) はこの言語の部分集合である MicroPlanner を実装しました。これは問題解決とロボット計画の仕事の支持に使用されました。同様なアイデアが論理と定理証明から提起され、エディンバラとマルセイユにて洗練された言語 Prolog の起源へと導きました。(Section 4.4 にて議論します)。自動探索に対する多くの不満の後に McDermott and Sussman (1972) は Conniver と呼ばれる言語を開発しました。これはプログラマのコントロール下に探索戦略を置くための仕組みを含みました。しかしこれは扱い難く、Sussman and Stallman 1975 はもっと御しやすい取り組み方を電子回路向け記号分析の手法の研究の間に発見しました。彼等は事実を繋げる論理的依存性の追跡を基にした非クロノロジカルなバックトラック計画を開発しました。これは *dependency-directed backtracking*(依存型バックトラック) として知られるようになった技術です。彼等の手法は複雑でしたが、合理的な効率の良いプログラムを生成しました。冗長な探索をほとんどしなかったためです。Doyle (1979) と McAllester (1978; 1980) は Stallman と Sussman の手法を一般化し、明確にしました。そして探索を定式化するための新しいパラダイムを開発しました。これは今では *truth maintenance*(真理維持) と呼ばれています。現代の問題解決システムは全て真理維持システムの何らかの形式を素地として使用しています。真理維持システムと真理維持を用いたアプリケーションを構築するための洗練された方法の議論については Forbus and deKleer 1993 を参照して下さい。Zabih et al. 1987 は Scheme に対する `amb` を基にした非決定性拡張を説明しています。これはこの節で説明されるインタプリタと同様です。しかしそれより高度な物です。それがクロノロジカルバックトラックではなく依存型バックトラックを使用しているためです。Winston 1992 は両方の種類のバックトラックに対する入門を提供しています。

## Driver loop

The driver loop for the `amb` evaluator has some unusual properties. It reads an expression and prints the value of the first non-failing execution, as in the `prime-sum-pair` example shown above. If we want to see the value of the next successful execution, we can ask the interpreter to backtrack and attempt to generate a second non-failing execution. This is signaled by typing the symbol `try-again`. If any expression except `try-again` is given, the interpreter will start a new problem, discarding the unexplored alternatives in the previous problem. Here is a sample interaction:

`amb` 評価機のドライバーループはいくつかの普通ではない性質を持っています。これは式を読み最初の失敗ではない実行の値を上で示された `prime-sum-pair` の例のように表示します。もし次の成功する実行の値を見たいのであれば、インタプリタにバックトラックして二番目の失敗ではない実行を生成する試行を命令します。これはシンボル `try-again` を入力することで伝えられます。もし `try-again` ではない任意の式が与えられたなら、インタプリタは新しい問題を開発し、直前の問題の調査されていない選択肢を捨てます。以下にサンプルの応答を示します。

```
;;; Amb-Eval input:  
(prime-sum-pair '(1 3 5 8) '(20 35 110))  
;;; Starting a new problem  
;;; Amb-Eval value:  
(3 20)  
  
;;; Amb-Eval input:  
try-again  
;;; Amb-Eval value:  
(3 110)  
  
;;; Amb-Eval input:  
try-again  
;;; Amb-Eval value:  
(8 35)  
  
;;; Amb-Eval input:  
try-again
```

```

;;; There are no more values of
(prime-sum-pair (quote (1 3 5 8)) (quote (20 35 110)))

;;; Amb-Eval input:
(prime-sum-pair '(19 27 30) '(11 36 58))
;;; Starting a new problem
;;; Amb-Eval value:
(30 11)

```

**Exercise 4.35:** Write a procedure `an-integer-between` that returns an integer between two given bounds. This can be used to implement a procedure that finds Pythagorean triples, i.e., triples of integers  $(i, j, k)$  between the given bounds such that  $i \leq j$  and  $i^2 + j^2 = k^2$ , as follows:

2つの与えられた境界値の間の整数を返す手続 `an-integer-between` を書け。これはピタゴラスの3つ組を求める手続を実装するのに使用できる。例えば与えられた範囲の間の整数の三つ組  $(i, j, k)$  で  $i \leq j$  and  $i^2 + j^2 = k^2$  の場合は以下の様になる。

```

(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (let ((j (an-integer-between i high)))
      (let ((k (an-integer-between j high)))
        (require (= (+ (* i i) (* j j)) (* k k)))
        (list i j k))))

```

**Exercise 4.36:** Exercise 3.69 discussed how to generate the stream of *all* Pythagorean triples, with no upper bound on the size of the integers to be searched. Explain why simply replacing `an-integer-between` by `an-integer-starting-from` in the procedure in Exercise 4.35 is not an adequate way to generate arbitrary Pythagorean triples. Write a procedure that actually will accomplish this. (That is, write a procedure for which repeatedly typing `try-again` would in principle eventually generate all Pythagorean triples.)

Exercise 3.69ではどのように“全ての”ピタゴラスの三つ組のストリームを探索対象の整数のサイズに上限無しで生成するかについて

て議論した。なぜ単純に `an-inte-ger-between` を Exercise 4.35 の手続内の `an-integer-starting-from` で置き換えることは、自由裁量なピタゴラスの三つ組を生成するのに適切でないのか、説明せよ。これを実際に達成する手續を書け。(すなわち、原理上は `try-again` を繰り返し入力することで全てピタゴラスの三つ組を生成する手續を書け)。

**Exercise 4.37:** Ben Bitdiddle claims that the following method for generating Pythagorean triples is more efficient than the one in Exercise 4.35. Is he correct? (Hint: Consider the number of possibilities that must be explored.)

Ben Bitdiddle は以下のピタゴラスの 3 つ組を生成する手法は Exercise 4.35 の手法に比べより効率的であると主張した。彼は正しいだろうか? (ヒント: 探索しなければならない可能性の数を考えよ)

```
(define (a-pythagorean-triple-between low high)
  (let ((i (an-integer-between low high)))
    (hsq (* high high)))
  (let ((j (an-integer-between i high)))
    (let ((ksq (+ (* i i) (* j j))))
      (require (>= hsq ksq))
      (let ((k (sqrt ksq)))
        (require (integer? k))
        (list i j k))))))
```

### 4.3.2 Examples of Nondeterministic Programs

Section 4.3.3 describes the implementation of the `amb` evaluator. First, however, we give some examples of how it can be used. The advantage of nondeterministic programming is that we can suppress the details of how search is carried out, thereby expressing our programs at a higher level of abstraction.

Section 4.3.3 は `amb` 評価機の実装を説明します。しかし最初にそれがどのように使用できるかについて、いくつかの例を与えます。非決定性プログラミングの利点は探索がどのように実行されるのかについての詳細を隠すことができることです。従って抽象の高いレベルにてプログラムを表現できます。

## Logic Puzzles

The following puzzle (taken from Dinesman 1968) is typical of a large class of simple logic puzzles:

以下のパズル (Dinesman 1968から拝借しました) は典型的な大きなクラスの簡単な論理パズルです。

Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's. Where does everyone live?

Baker, Cooper, Fletcher, Miller, それに Smith は同じ 5 階しかないアパートの異なる階に住んでいます。Baker は最上階には住んでいません。Cooper は最下階には住んでいません。Fletcher は最上階にも最下階にも住んでいません。Miller は Cooper よりも高い階に住んでいます。Smith は Fletcher のすぐ隣の階には住んでいません。Fletcher は Cooper のすぐ隣の階には住んでいません。皆はどの階に住んでいるでしょうか？

We can determine who lives on each floor in a straightforward way by enumerating all the possibilities and imposing the given restrictions:

誰がどの階に住んでいるかを、全ての可能性を列举し与えられた制約を与える簡単な方法で決定することができます。<sup>91</sup>

---

<sup>91</sup>Our program uses the following procedure to determine if the elements of a list are distinct:

私達のプログラムは以下の手続をリストの要素が識別可能であるか判断するために使用しています。

```
(define (distinct? items)
  (cond ((null? items) true)
        ((null? (cdr items)) true)
        ((member (car items) (cdr items)) false)
        (else (distinct? (cdr items)))))
```

Member is like memq except that it uses equal? instead of eq? to test for equality.

member は memq と同様ですがこれは eq? の代わりに equal? を等価性のテストに用いています。

```

(define (multiple-dwelling)
  (let ((baker      (amb 1 2 3 4 5)) (cooper (amb 1 2 3 4 5))
        (fletcher   (amb 1 2 3 4 5)) (miller  (amb 1 2 3 4 5))
        (smith      (amb 1 2 3 4 5)))
    (require
      (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)          (list 'cooper cooper)
          (list 'fletcher fletcher) (list 'miller miller)
          (list 'smith smith))))
```

Evaluating the expression (`multiple-dwelling`) produces the result  
式 (`multiple-dwelling`) を評価すると結果を生成します。

```
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
```

Although this simple procedure works, it is very slow. [Exercise 4.39](#) and [Exercise 4.40](#) discuss some possible improvements.

この簡単な手続はうまく行きますが、とても遅いです。[Exercise 4.39](#)と[Exercise 4.40](#)はいくらかの可能な改善法について議論します。

**Exercise 4.38:** Modify the multiple-dwelling procedure to omit the requirement that Smith and Fletcher do not live on adjacent floors. How many solutions are there to this modified puzzle?

`multiple-dwelling` 手続を変更し、Smith と Fletcher が隣接する階に住んでいないという要件を取り除く。この変更したパズルにはいくつの解が存在するか？

**Exercise 4.39:** Does the order of the restrictions in the multiple-dwelling procedure affect the answer? Does it affect the time to find an answer? If you think it matters, demonstrate a faster program obtained from the given one by reordering the restrictions. If you think it does not matter, argue your case.

multiple-dwelling 手続内の制約の順は解に影響するだろうか? 回答を見つけるのにかかる時間には影響を与えるだろうか? もしそれが重要であると考えるのなら、制約の順を変えることで与えられる物から得られるより速いプログラムを実演せよ。もしそれが問題ではないと考えるのなら、あなたの考えを論ぜよ。

**Exercise 4.40:** In the multiple dwelling problem, how many sets of assignments are there of people to floors, both before and after the requirement that floor assignments be distinct? It is very inefficient to generate all possible assignments of people to floors and then leave it to backtracking to eliminate them. For example, most of the restrictions depend on only one or two of the person-floor variables, and can thus be imposed before floors have been selected for all the people. Write and demonstrate a much more efficient nondeterministic procedure that solves this problem based upon generating only those possibilities that are not already ruled out by previous restrictions. (Hint: This will require a nest of `let` expressions.)

複数の住居の問題において、人を階へと割り当てる方法は、requirement の前と後で階の割り当てが区別可能な方法でいくつあるか? 全ての人から階への可能な割り当てを生成してからその次にそれらを排除するために backtrack にまかせることは非常に非効率である。例えば制約のほとんどは一つか二つの人と階の変数を持ち、従って全ての人に対して階が選択される前に制約を与えることができる。この問題を先の制約により既に排除されたもの以外の可能性のみを生成することに基づいて解くずっと効率の良い非決定性手続を書き、実演せよ。

**Exercise 4.41:** Write an ordinary Scheme program to solve the multiple dwelling puzzle.

複数住居パズルを解く通常の Scheme プログラムを書け。

**Exercise 4.42:** Solve the following “Liars” puzzle (from Phillips 1934):

次の“嘘つき”パズル (Phillips 1934から) を解け

Five schoolgirls sat for an examination. Their parents—so they thought—showed an undue degree of interest in the result. They

therefore agreed that, in writing home about the examination, each girl should make one true statement and one untrue one. The following are the relevant passages from their letters:

5人の女生徒が試験のために座っている。彼女達は、彼女等の両親が結果に過大な興味を見せていると考えている。従って彼女達は次のことを合意した。それぞれの少女が1つの正しい文と嘘の文を作り、各家庭に試験についての手紙を書く。以下は彼女らの手紙の該当する一節である。

- Betty: “Kitty was second in the examination. I was only third.”

Betty: “Kitty が試験では二位だった。私だけが 3 位だった。”

- Ethel: “You’ll be glad to hear that I was on top. Joan was 2nd.”

Ethel: “喜んで、私がトップ。Joan が 2 位だった。”

- Joan: “I was third, and poor old Ethel was bottom.”

Joan: “私が 3 番。可哀想な Ethel は最下位だった。”

- Kitty: “I came out second. Mary was only fourth.”

Kitty: “私が 2 番。Mary が単独で 4 位。”

- Mary: “I was fourth. Top place was taken by Betty.”

Mary: “私が 4 位。トップは Betty が取ったわ。”

What in fact was the order in which the five girls were placed?  
実際にはどの順に 5人の女の子は並べられるか?

**Ex. 4.43:** Use the `amb` evaluator to solve the following puzzle:<sup>92</sup>

`amb` 評価機を用いて以下のパズルを解け<sup>93</sup>

@quotation Mary Ann Moore's father has a yacht and so has each of his four friends: Colonel Downing, Mr. Hall, Sir Barnacle Hood, and Dr. Parker. Each of the five also has one daughter and each has

<sup>92</sup>This is taken from a booklet called “Problematical Recreations,” published in the 1960s by Litton Industries, where it is attributed to the *Kansas State Engineer*.

<sup>93</sup>これは 1960 年代に Litton Industries により出版された“問題の多いレクリエーション”と呼ばれる小冊子から引用しました。 *Kansas State Engineer* 著。

named his yacht after a daughter of one of the others. Sir Barnacle's yacht is the Gabrielle, Mr. Moore owns the Lorna; Mr. Hall the Rosalind. The Melissa, owned by Colonel Downing, is named after Sir Barnacle's daughter. Gabrielle's father owns the yacht that is named after Dr. Parker's daughter. Who is Lorna's father?

Mary Ann Moore の父はヨットを持っており、彼の 4 人の友達、Downing 大佐, Hall さん, Barnacle Hood 卿, Dr. Parker もそれぞれが持っていました。5 人のそれぞれに一人の娘がおり、それぞれが各自のヨットに他人の娘の名を取って付けていました。Barnacle 卿のヨットは Gabrielle で、Moore さんは Lorna です。Hall さんは Rosalind です。Downing 大佐の Melissa は Barnacle 卿の娘の名を取って付けました。Gabrielle の父のヨットは Dr. Parker の娘からです。Lorna の父は誰でしょう？

Try to write the program so that it runs efficiently (see [Exercise 4.40](#)). Also determine how many solutions there are if we are not told that Mary Ann's last name is Moore.

効率良く実行されるプログラムを書くように努めること ([Exercise 4.40](#) 参照)。また、もし Mary Ann の家族名が Moore であることを伝えなければ、いくつの解が存在するだろうか? @end quotation

**Exercise 4.44:** [Exercise 2.42](#) described the “eight-queens puzzle” of placing queens on a chessboard so that no two attack each other. Write a nondeterministic program to solve this puzzle.

[Exercise 2.42](#) は チェス盤に 8 つのクイーンをどの 2 つもお互いに攻撃することが無いように置く “8 クイーンパズル” について説明した。このパズルを解く非決定性プログラムを書け。

## Parsing natural language

Programs designed to accept natural language as input usually start by attempting to *parse* the input, that is, to match the input against some grammatical structure. For example, we might try to recognize simple sentences consisting of an article followed by a noun followed by a verb, such as “The cat eats.” To accomplish such an analysis, we must be able to identify the parts of speech of individual words. We could start with some lists that classify various

words:<sup>94</sup>

自然言語を入力として受け入れるよう設計されたプログラムは通常、その入力を *parse*(パース、構文解析)することから始めます。つまり入力をある文法構造に対して合わせることです。例えば冠詞と続く名詞、続く動詞から成る簡単な文、"The cat eats."(猫は食べる)のような物を認識しようとしているします。そのような分析を達成するためには個別の単語の品詞を判別できなければなりません。多種の単語を判別できるいくつかのリストから始めることができるでしょう。<sup>95</sup>

```
(define nouns '(noun student professor cat class))
(define verbs '(verb studies lectures eats sleeps))
(define articles '(article the a))
```

We also need a *grammar*, that is, a set of rules describing how grammatical elements are composed from simpler elements. A very simple grammar might stipulate that a sentence always consists of two pieces—a noun phrase followed by a verb—and that a noun phrase consists of an article followed by a noun. With this grammar, the sentence "The cat eats" is parsed as follows:

またgrammar(文法)も必要です。つまり、文法上の要素がどのようにより簡単な要素から組み立てられるのかを説明するルールの集合です。とても簡単な文法は文は常に2つの要素—名詞句とそれに続く動詞—により成ると規定することができるかもしれません。そして名詞句は定冠詞とそれに続く名詞から成り立ちます。この文法を用いて、文 "The cat eats." は以下のようにパースされます。

```
(sentence (noun-phrase (article the) (noun cat))
          (verb eats))
```

We can generate such a parse with a simple program that has separate procedures for each of the grammatical rules. To parse a sentence, we identify its two constituent pieces and return a list of these two elements, tagged with the symbol **sentence**:

そのようなパースを、手続を各文法上のルールに分割する簡単なプログラムを用いて生成することができます。文をパースするためには、それを構成す

---

<sup>94</sup>Here we use the convention that the first element of each list designates the part of speech for the rest of the words in the list.

<sup>95</sup>ここでは各リストの最初の要素はリストの残りの単語の品詞を示すという仕様を用いています。

る 2 つの要素を判別し、これらの 2 つの要素のリストをシンボル `sentence` のタグを付けて返します。

```
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-word verbs)))
```

A noun phrase, similarly, is parsed by finding an article followed by a noun:  
名詞句も同様に定冠詞とそれに続く名詞を見つけることでパースされます。

```
(define (parse-noun-phrase)
  (list 'noun-phrase
        (parse-word articles)
        (parse-word nouns)))
```

At the lowest level, parsing boils down to repeatedly checking that the next unparsed word is a member of the list of words for the required part of speech. To implement this, we maintain a global variable `*unparsed*`, which is the input that has not yet been parsed. Each time we check a word, we require that `*unparsed*` must be non-empty and that it should begin with a word from the designated list. If so, we remove that word from `*unparsed*` and return the word together with its part of speech (which is found at the head of the list).<sup>96</sup>

最も低いレベルでは、パースとは繰り返し次のパースされていない単語が必要とされる品詞のための単語のリストのメンバであるかをチェックすることだとまとめられます。これを実装するために、私達はグローバル変数 `*unparsed*`を持ちます。これはまだパースされていない入力です。単語をチェックする各時点で `*unparsed*` が空ではないことを要求し、また指定されたリストの単語で始まるることを要求します。もしそうであるならその単語を `*unparsed*` から削除し、その単語をその品詞（これはリストの先頭に見つかります）と共に返します。<sup>97</sup>

```
(define (parse-word word-list)
```

---

<sup>96</sup>Notice that `parse-word` uses `set!` to modify the unparsed input list. For this to work, our `amb` evaluator must undo the effects of `set!` operations when it backtracks.

<sup>97</sup>`parse-word` が `parse` されていない入力リストの変更するのに `set!` を用いているおおに注意して下さい。これをうまく行うためには、`amb` 評価機は `set!` 命令の効果をバックトラックする時に取り消しきれなければいけません。

```
(require (not (null? *unparsed*)))
(require (memq (car *unparsed*) (cdr word-list)))
(let ((found-word (car *unparsed*)))
  (set! *unparsed* (cdr *unparsed*))
  (list (car word-list) found-word)))
```

To start the parsing, all we need to do is set `*unparsed*` to be the entire input, try to parse a sentence, and check that nothing is left over:

ペースを始めるために行わなければいけないこと全ては `*unparsed*` に入力全体を設定し、文のペースを試み、何も残っていないことをチェックすることです。

```
(define *unparsed* '())
(define (parse input)
  (set! *unparsed* input)
  (let ((sent (parse-sentence)))
    (require (null? *unparsed*)) sent)))
```

We can now try the parser and verify that it works for our simple test sentence:

これでパーザ (parser、ペースを行うプログラム) を試し、簡単なテスト文に対してうまく働くことを確認することができます。

```
;;; Amb-Eval input:
(parse '(the cat eats))
;;; Starting a new problem
;;; Amb-Eval value:

(sentence (noun-phrase (article the) (noun cat)) (verb eats))
```

The `amb` evaluator is useful here because it is convenient to express the parsing constraints with the aid of `require`. Automatic search and backtracking really pay off, however, when we consider more complex grammars where there are choices for how the units can be decomposed.

`amb` 評価機はここでとても役立ちます。`require` の助けを用いてペースする上での制約を表現するのにとっても便利なためです。しかし、自動的な探索とバックトラックが本当に効果を生むのはより複雑な文法について考えた時に、1つの単位の分解方法に多数の選択肢が存在する場合です。

Let's add to our grammar a list of prepositions:  
私達の文法に前置詞を追加してみましょう。

```
(define prepositions '(prep for to in by with))
```

and define a prepositional phrase (e.g., “for the cat”) to be a preposition followed by a noun phrase:

そして前置詞句(例えば“for the cat”(猫のために))を名詞句の前の前置詞として定義します。

```
(define (parse-prepositional-phrase)
  (list 'prep-phrase
    (parse-word prepositions)
    (parse-noun-phrase)))
```

Now we can define a sentence to be a noun phrase followed by a verb phrase, where a verb phrase can be either a verb or a verb phrase extended by a prepositional phrase:<sup>98</sup>

これで文は名詞句に動詞句が続くと定義でき、動詞句は動詞か、または前置詞句で拡張された動詞句となります。<sup>99</sup>

```
(define (parse-sentence)
  (list 'sentence (parse-noun-phrase) (parse-verb-phrase)))
(define (parse-verb-phrase)
  (define (maybe-extend verb-phrase)
    (amb verb-phrase
      (maybe-extend
        (list 'verb-phrase
          verb-phrase
          (parse-prepositional-phrase))))))
(maybe-extend (parse-word verbs)))
```

While we’re at it, we can also elaborate the definition of noun phrases to permit such things as “a cat in the class.” What we used to call a noun phrase, we’ll now call a simple noun phrase, and a noun phrase will now be either a simple noun phrase or a noun phrase extended by a prepositional phrase:

ここまでを行っている間に、名詞句の定義に“a cat in the class”(クラスの猫)のような物を認めるよう詳細を詰めることができます。今まで名詞句と呼

---

<sup>98</sup>Observe that this definition is recursive—a verb may be followed by any number of prepositional phrases.

<sup>99</sup>この定義が再帰的であることに注意して下さい。動詞には任意の数の前置詞句が続けられます。

んできた物はこれからはシンプルな名詞句と呼びます。そして名詞句はこれからシンプルな名詞句か前置詞句で拡張した名詞句となります。

```
(define (parse-simple-noun-phrase)
  (list 'simple-noun-phrase
        (parse-word articles)
        (parse-word nouns)))
(define (parse-noun-phrase)
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
         (maybe-extend
          (list 'noun-phrase
                noun-phrase
                (parse-prepositional-phrase))))))
  (maybe-extend (parse-simple-noun-phrase)))
```

Our new grammar lets us parse more complex sentences. For example  
私達の新しい文法はより複雑な文をパースできます。例えは、

```
(parse '(the student with the cat sleeps in the class))
```

produces

(猫と一緒にその生徒はクラスで寝ている) は以下を生成します。

```
(sentence
  (noun-phrase
    (simple-noun-phrase (article the) (noun student)))
  (prep-phrase
    (prep with)
    (simple-noun-phrase (article the) (noun cat))))
  (verb-phrase
    (verb sleeps)
  (prep-phrase
    (prep in)
    (simple-noun-phrase (article the) (noun class)))))
```

Observe that a given input may have more than one legal parse. In the sentence “The professor lectures to the student with the cat,” it may be that the professor is lecturing with the cat, or that the student has the cat. Our nondeterministic program finds both possibilities:

与えられた入力が二つ以上の有効な分析結果を持つかもしれないことを確認して下さい。文 “The professor lectures to the student with the cat” は professor(教授) が猫と一緒に講義をしている場合と、学生が猫を持っている場合が有り得ます。私達の非決定性プログラムは両方の可能性を見つけます。

```
(parse '(the professor lectures to the student with the cat))
```

produces

は以下を生成します。

```
(sentence
  (simple-noun-phrase (article the) (noun professor))
  (verb-phrase
    (verb-phrase
      (verb lectures)
      (prep-phrase
        (prep to)
        (simple-noun-phrase (article the) (noun student))))
    (prep-phrase
      (prep with)
      (simple-noun-phrase (article the) (noun cat)))))
```

Asking the evaluator to try again yields

評価機にもう一度試行を命ずると以下を生じます。

```
(sentence
  (simple-noun-phrase (article the) (noun professor))
  (verb-phrase
    (verb lectures)
    (prep-phrase
      (prep to)
      (noun-phrase
        (simple-noun-phrase (article the) (noun student))
        (prep-phrase
          (prep with)
          (simple-noun-phrase (article the) (noun cat))))))
```

**Exercise 4.45:** With the grammar given above, the following sentence can be parsed in five different ways: “The professor lectures

to the student in the class with the cat.” Give the five parses and explain the differences in shades of meaning among them.

上で与えられた文法を用いて次の文は 5 通りにパースできる。“The professor lectures to the student in the class with the cat”。5 通りの結果を与えてそれらの間の様々な意味の違いを説明せよ。

**Exercise 4.46:** The evaluators in Section 4.1 and Section 4.2 do not determine what order operands are evaluated in. We will see that the `amb` evaluator evaluates them from left to right. Explain why our parsing program wouldn’t work if the operands were evaluated in some other order.

Section 4.1 と Section 4.2 は評価機はどの順でオペランドが評価されるかを決定しない。我々は `amb` 評価機がそれらを左から右へと評価するのを見るだろう。なぜ我々のバーザはオペランドが何らかの他の順で評価されたならうまく働くのか説明せよ。

**Exercise 4.47:** Louis Reasoner suggests that, since a verb phrase is either a verb or a verb phrase followed by a prepositional phrase, it would be much more straightforward to define the procedure `parse-verb-phrase` as follows (and similarly for noun phrases):

Louis Reasoner は動詞句は動詞か前置詞句が続く動詞句であるのだから、手続 `parse-verb-phrase` を以下のように (そして同様に名詞句に対しても) 定義すればずっと簡単になるのではないかと提案した。

```
(define (parse-verb-phrase)
  (amb (parse-word verbs)
        (list 'verb-phrase
              (parse-verb-phrase)
              (parse-prepositional-phrase))))
```

Does this work? Does the program’s behavior change if we interchange the order of expressions in the `amb`?

これはうまく行くだろうか? このプログラムの振舞はもし `amb` 内の式の順を置き換えたら変わるだろうか?

**Exercise 4.48:** Extend the grammar given above to handle more complex sentences. For example, you could extend noun phrases

and verb phrases to include adjectives and adverbs, or you could handle compound sentences.<sup>100</sup>

上で与えられた文法をより複雑な文を取り扱うように拡張せよ。例えば、名詞句と動詞句を拡張し形容詞と福祉を含める、または複合文を取り扱えるようできるだろう。<sup>101</sup>

**Exercise 4.49:** Alyssa P. Hacker is more interested in generating interesting sentences than in parsing them. She reasons that by simply changing the procedure `parse-word` so that it ignores the “input sentence” and instead always succeeds and generates an appropriate word, we can use the programs we had built for parsing to do generation instead. Implement Alyssa’s idea, and show the first half-dozen or so sentences generated.<sup>102</sup>

Alyssa P. Hacker はパースするよりも面白い文を生成することにより興味を持っている。彼女は手続 `parse-word` を簡単に変更してそれが“入力文”を無視し、その代わりに常に成功して適切な単語を生成するようにすれば、このパースのために構築したプログラムを用いて代わりに生成を行えるのではないかと推測した。Alyssa の考えを実装せよ。そして生成された文の最初の半ダースかそこらを示せ。<sup>103</sup>

---

<sup>100</sup>This kind of grammar can become arbitrarily complex, but it is only a toy as far as real language understanding is concerned. Real natural-language understanding by computer requires an elaborate mixture of syntactic analysis and interpretation of meaning. On the other hand, even toy parsers can be useful in supporting flexible command languages for programs such as information-retrieval systems. Winston 1992 discusses computational approaches to real language understanding and also the applications of simple grammars to command languages.

<sup>101</sup>この種の文法は任意の複雑さに成り得ます。しかし現実の言語の理解を考える限りはただの玩具に過ぎません。現実の自然言語のコンピュータによる理解は構文解析と意味解釈の入り混じりが要求されます。一方で、例えおもちゃのパーザでもプログラムのための柔軟な命令言語、例えば情報取得システム等をサポートするには実用的です。Winston 1992は計算機による自然言語理解への取り組みと共に簡単な文法の命令言語のアプリケーションについても議論しています。

<sup>102</sup>Although Alyssa’s idea works just fine (and is surprisingly simple), the sentences that it generates are a bit boring—they don’t sample the possible sentences of this language in a very interesting way. In fact, the grammar is highly recursive in many places, and Alyssa’s technique “falls into” one of these recursions and gets stuck. See Exercise 4.50 for a way to deal with this.

<sup>103</sup>Alyssa のアイデアはちゃんとうまく働きますが(そして驚くほど簡単ですが)、それ

### 4.3.3 Implementing the Amb Evaluator

The evaluation of an ordinary Scheme expression may return a value, may never terminate, or may signal an error. In nondeterministic Scheme the evaluation of an expression may in addition result in the discovery of a dead end, in which case evaluation must backtrack to a previous choice point. The interpretation of nondeterministic Scheme is complicated by this extra case.

通常の Scheme 式の評価機は値を返すか、永遠に停止しないか、またはエラーを発します。非決定性 Scheme では式の評価はそれに加えて探索が行き止まりに帰結します。その場合には評価機は依然の選択点へバックトラックしなければなりません。非決定性 Scheme の解釈はこの特別な場合により複雑になります。

We will construct the `amb` evaluator for nondeterministic Scheme by modifying the analyzing evaluator of Section 4.1.7.<sup>104</sup> As in the analyzing evaluator, evaluation of an expression is accomplished by calling an execution procedure produced by analysis of that expression. The difference between the interpretation of ordinary Scheme and the interpretation of nondeterministic Scheme will be entirely in the execution procedures.

私達は非決定性 Scheme のための `amb` 評価機を、Section 4.1.7 の分析評価機を変更することで構築します。<sup>105</sup> 分析評価機のように、式の評価は式の分析により生成される実行手続を呼ぶことにより達成されます。通常の Scheme の解釈と非決定性 Scheme の解釈との違いは完全に実行手続の中に存在します。

---

が生成する文は少しつまらないです。それらはこの言語の可能な文からとても面白い様には抽出はしません。実際に文法は多くの場所で高度に再帰し、Alyssa の技術はこれらの再帰の 1 つに “falls into”(陥り)、抜け出せなくなります。これに対処する方法は Exercise 4.50 を参照して下さい。

<sup>104</sup>We chose to implement the lazy evaluator in Section 4.2 as a modification of the ordinary metacircular evaluator of Section 4.1.1. In contrast, we will base the `amb` evaluator on the analyzing evaluator of Section 4.1.7, because the execution procedures in that evaluator provide a convenient framework for implementing backtracking.

<sup>105</sup>私達はSection 4.2の遅延評価機をSection 4.1.1の通常のメタ循環評価機に対する変更として実装することを選択しました。対照的に、Section 4.1.7の分析評価機を `amb` 評価機の基にします。その評価機内の実行手続がバックトラックを実装するのに便利なフレームワークを提供するためです。

## Execution procedures and continuations

Recall that the execution procedures for the ordinary evaluator take one argument: the environment of execution. In contrast, the execution procedures in the `amb` evaluator take three arguments: the environment, and two procedures called *continuation procedures*. The evaluation of an expression will finish by calling one of these two continuations: If the evaluation results in a value, the *success continuation* is called with that value; if the evaluation results in the discovery of a dead end, the *failure continuation* is called. Constructing and calling appropriate continuations is the mechanism by which the nondeterministic evaluator implements backtracking.

通常の評価機の実行手続が1つの引数、実行の環境を取ることを思い出して下さい。対照的に、`amb` 評価機の実行手続は3つの引数を取ります。環境と*continuation procedures*(継続手続)と呼ばれる2つの手続です。式の評価はこれらの2つの継続の1つを呼ぶことで完了します。もし評価の結果が値に帰結するならば、*success continuation*(成功継続)がその値と共に呼ばれます。もし評価が行き止まりの発見に帰結したのであれば、*failure continuation*(失敗継続)が呼ばれます。適切な継続の構築と呼出が非決定性評価機のバックトラックが実装される仕組みです。

It is the job of the success continuation to receive a value and proceed with the computation. Along with that value, the success continuation is passed another failure continuation, which is to be called subsequently if the use of that value leads to a dead end.

値を受け取り計算を続行することが成功継続の仕事です。その値と共に、成功継続は別の失敗継続も渡されます。これはその後にもしその値の使用が行き止まりに導いたなら呼び出されます。

It is the job of the failure continuation to try another branch of the nondeterministic process. The essence of the nondeterministic language is in the fact that expressions may represent choices among alternatives. The evaluation of such an expression must proceed with one of the indicated alternative choices, even though it is not known in advance which choices will lead to acceptable results. To deal with this, the evaluator picks one of the alternatives and passes this value to the success continuation. Together with this value, the evaluator constructs and passes along a failure continuation that can be called later to choose a different alternative.

非決定性処理の他の分岐を試すのは失敗継続の仕事です。非決定性言語の本質は式が選択肢の間の選択を表現するだろうという事実の中に存在します。

そのような式の評価は例え前もってどの選択肢受け入れ可能な結果に導くか知らなくても指示された代替となる選択の一つを用いて続行しなければなりません。これを処理するためには、評価機は選択肢から1つを取り出しこの値を成功継続に渡します。この値と共に、評価機は後で異なる選択肢を選択するために呼び出し可能な失敗継続を構築し、一緒に渡します。

A failure is triggered during evaluation (that is, a failure continuation is called) when a user program explicitly rejects the current line of attack (for example, a call to `require` may result in execution of `(amb)`, an expression that always fails—see [Section 4.3.1](#)). The failure continuation in hand at that point will cause the most recent choice point to choose another alternative. If there are no more alternatives to be considered at that choice point, a failure at an earlier choice point is triggered, and so on. Failure continuations are also invoked by the driver loop in response to a `try-again` request, to find another value of the expression.

失敗は評価の間に引き起こされます。(言い換えれば失敗継続が呼ばれます)。それはユーザプログラムが明示的に現在の一連の取り組みを拒絶した場合に起こります。(例えば、`require`の呼出は結果として `(amb)` が実行される場合があります。これは常に失敗する式です。—[Section 4.3.1](#)参照)。その時点で手中にある失敗継続が最も最近の選択点に他の選択肢を選択させます。もしもうその選択点にて考えられる他の選択肢が無い場合には、直前の選択点の失敗が引き起こされます。以下その繰り返しです。失敗継続はまた式の別の値を見つけるためのドライバループによる `try-again` 要求への応答としても起動されます。

In addition, if a side-effect operation (such as assignment to a variable) occurs on a branch of the process resulting from a choice, it may be necessary, when the process finds a dead end, to undo the side effect before making a new choice. This is accomplished by having the side-effect operation produce a failure continuation that undoes the side effect and propagates the failure.

加えて、もし副作用命令(変数への代入等)がある選択の結果としての分岐処理上で起こったならば、処理が行き止まりを見つけた時に、新しい選択を行う前にその副作用を取り消しする必要があるかもしれません。これは副作用命令に副作用を取り消し失敗を伝播させる失敗継続を生成させることで達成されます。

In summary, failure continuations are constructed by  
まとめとして、失敗継続は以下により構築されます。

- `amb` expressions—to provide a mechanism to make alternative choices if

the current choice made by the `amb` expression leads to a dead end;  
`amb` 式 —`amb` 式により行われた現在の選択が行き止まりに導いた場合に  
別の選択を行う仕組みを提供します

- the top-level driver—to provide a mechanism to report failure when the choices are exhausted;  
トップレベルドライバ—選択肢が枯渇した時に失敗を報告する仕組みを  
提供します
- assignments—to intercept failures and undo assignments during back-tracking.  
代入—失敗に割り込み、バックトラックの間に代入を取り消します

Failures are initiated only when a dead end is encountered. This occurs  
失敗は行き止まりに遭遇したその時のみ起動されます。これは以下の場合  
に起こります。

- if the user program executes `(amb)`;  
ユーザプログラムが `(amb)` を実行した
- if the user types `try-again` at the top-level driver.  
ユーザがトップレベルドライバにて `try-again` を入力した

Failure continuations are also called during processing of a failure:  
失敗継続はまた失敗処理の間にも呼ばれます。

- When the failure continuation created by an assignment finishes undoing a side effect, it calls the failure continuation it intercepted, in order to propagate the failure back to the choice point that led to this assignment or to the top level.  
代入が副作用の取消を完了させることで失敗継続が作成された時に、そ  
れは割り込んだ失敗継続を、失敗を伝播させこの代入に導いた選択点に  
戻すために、またはトップレベルに戻すために呼びます。
- When the failure continuation for an `amb` runs out of choices, it calls the failure continuation that was originally given to the `amb`, in order to propagate the failure back to the previous choice point or to the top level.  
`amb` に対する失敗継続が選択肢を使い切った時、失敗を直前の選択点か  
トップレベルに伝播させるために、`amb` に対して元々与えられた失敗継  
続を呼び出します。

## Structure of the evaluator

The syntax- and data-representation procedures for the `amb` evaluator, and also the basic `analyze` procedure, are identical to those in the evaluator of Section 4.1.7, except for the fact that we need additional syntax procedures to recognize the `amb` special form:<sup>106</sup>

`amb` 評価機に対する文法とデータの表現手続、また基本的な `analyze` 手續は、Section 4.1.7 の評価機のそれらに等しい物です。しかし私達が追加の構文手続を `amb` の特殊形式を認識するために必要とすることが異なります。<sup>107</sup>

```
(define (amb? exp)
  (tagged-list? exp 'amb))
(define (amb-choices exp) (cdr exp))
```

We must also add to the dispatch in `analyze` a clause that will recognize this special form and generate an appropriate execution procedure:

`analyze` 内にこの特殊形式を認識し、適切な実行手続を生成する呼出を追加しなければなりません。

```
((amb? exp) (analyze-amb exp))
```

The top-level procedure `ambeval` (similar to the version of `eval` given in Section 4.1.7) analyzes the given expression and applies the resulting execution procedure to the given environment, together with two given continuations:

トップレベル手続 `ambeval`(Section 4.1.7で与えられた `eval` の版に似た物)は与えられた式を分析し、実行手続を与えられた環境に対し、2つの与えられた継続と一緒に適用しなければなりません。

```
(define (ambeval exp env succeed fail)
  ((analyze exp) env succeed fail))
```

A success continuation is a procedure of two arguments: the value just obtained and another failure continuation to be used if that value leads to a subsequent failure. A failure continuation is a procedure of no arguments. So the general form of an execution procedure is

---

<sup>106</sup>We assume that the evaluator supports `let` (see Exercise 4.22), which we have used in our nondeterministic programs.

<sup>107</sup>評価機は `let` をサポートすると想定しています。(Exercise 4.22 参照)。私達はこれを非決定性プログラム内にて利用してきました。

成功継続は2引数の手続です。2引数は得られたばかりの値とその値がその後に失敗へと導いたなら使用される別の失敗継続です。失敗継続は引数無しの手続です。そのため実行手続の一般的な形は以下のようになります。

```
(lambda (env succeed fail)
  ;; succeed is (lambda (value fail) ...)
  ;; fail is (lambda () ...)
  ...)
```

For example, executing

例えば、以下を実行すると、

```
(amb eval <exp>
      the-global-environment
      (lambda (value fail) value)
      (lambda () 'failed))
```

will attempt to evaluate the given expression and will return either the expression's value (if the evaluation succeeds) or the symbol `failed` (if the evaluation fails). The call to `amb eval` in the driver loop shown below uses much more complicated continuation procedures, which continue the loop and support the `try-again` request.

与えられた式を評価し、式の値(評価が成功した場合)かシンボル `failed`(評価が失敗した場合)を返します。以下で示されるドライバーループ内での `amb eval` の呼出はより多くの複雑な継続手続を使用します。これらはループを継続し `try-again` 要求をサポートします。

Most of the complexity of the `amb` evaluator results from the mechanics of passing the continuations around as the execution procedures call each other. In going through the following code, you should compare each of the execution procedures with the corresponding procedure for the ordinary evaluator given in [Section 4.1.7](#).

`amb` の複雑さの多くは実行手続がお互いを呼ぶに従い、継続をたらい回しにすることから来ています。以下のコードを通して読むに当って、それぞれの実行手続を [Section 4.1.7](#) で与えられた通常の評価機のための対応する手続と比べて下さい。

## Simple expressions

The execution procedures for the simplest kinds of expressions are essentially the same as those for the ordinary evaluator, except for the need to manage the continuations. The execution procedures simply succeed with the value of the expression, passing along the failure continuation that was passed to them.

最も単純な種類の式に対する実行手続は本質的に通常の評価機に対するものと同じです。ただし、継続を管理する必要があることが異なります。これらの実行手続は式の値と共に単純に成功し、渡された失敗継続をそのまま手渡します。

```
(define (analyze-self-evaluating exp)
  (lambda (env succeed fail)
    (succeed exp fail)))
(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env succeed fail)
      (succeed qval fail))))
(define (analyze-variable exp)
  (lambda (env succeed fail)
    (succeed (lookup-variable-value exp env)
             fail)))
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env succeed fail)
      (succeed (make-procedure vars bproc env)
               fail))))
```

Notice that looking up a variable always ‘succeeds.’ If `lookup-variable-value` fails to find the variable, it signals an error, as usual. Such a “failure” indicates a program bug—a reference to an unbound variable; it is not an indication that we should try another nondeterministic choice instead of the one that is currently being tried.

変数の検索が常に‘成功’することに注意して下さい。もし `lookup-variable-value` が変数を見つけるのに失敗した場合、それはいつも通りにエラーを発します。そのような“失敗”はプログラムのバグ—未束縛な変数への参照—を示

します。これは現在試行中の物の代わりに別の非決定性選択を試すことを示してはいません。

## Conditionals and sequences

Conditionals are also handled in a similar way as in the ordinary evaluator. The execution procedure generated by `analyze-if` invokes the predicate execution procedure `pproc` with a success continuation that checks whether the predicate value is true and goes on to execute either the consequent or the alternative. If the execution of `pproc` fails, the original failure continuation for the `if` expression is called.

条件文もまた通常の評価機と同様に取り扱われます。`analyze-if`により生成される実行手続は述語実行手続 `pproc` を成功継続と共に起動します。成功継続は述語の値が真であるかチェックし、結果部 (consequent) か代替部 (alternative) を実行します。もし `pproc` の実行が失敗したなら `if` 式に対する元の失敗継続が呼ばれます。

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp))))
    (aprocs (analyze (if-alternative exp))))
  (lambda (env succeed fail)
    (pproc env
           ;;; success continuation for evaluating the predicate
           ;;; to obtain pred-value
           ;;;述語を評価するための成功継続を置く
           ;;; pred-valueを得るため
           (lambda (pred-value fail2)
             (if (true? pred-value)
                 (cproc env succeed fail2)
                 (aprocs env succeed fail2))))
    ;;; failure continuation for evaluating the predicate
    ;;;述語を評価するための失敗継続
    fail))))
```

Sequences are also handled in the same way as in the previous evaluator, except for the machinations in the subprocedure `sequentially` that are required for

passing the continuations. Namely, to sequentially execute **a** and then **b**, we call **a** with a success continuation that calls **b**.

例もまた以前の評価機と同様に取り扱われます。ただし継続を渡すために必要とされる内部手続 **sequentially** 内の企みが異なります。具体的には **a** を実行し、次に **b** と順に行うために、**a** を成功継続と共に呼び、成功継続が **b** を呼びます。

```
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail)
      (a env
          ;; success continuation for calling a
          ;; a を呼ぶための成功継続
          (lambda (a-value fail2)
            (b env succeed fail2)))
          ;; failure continuation for calling a
          ;; a を呼ぶための失敗継続
          fail)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc
                            (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence: ANALYZE"))
    (loop (car procs) (cdr procs))))
```

## Definitions and assignments

Definitions are another case where we must go to some trouble to manage the continuations, because it is necessary to evaluate the definition-value expression before actually defining the new variable. To accomplish this, the definition-value execution procedure **vproc** is called with the environment, a success continuation, and the failure continuation. If the execution of **vproc**

succeeds, obtaining a value `val` for the defined variable, the variable is defined and the success is propagated:

定義は継続を管理するために手間をかけなければいけない一例です。定義の値(definition-value)の式を実際に新しい値を定義する前に評価する必要があります。これを達成するためには定義値実行手続 `vproc` が環境、成功継続、失敗継続と共に呼ばれます。もし `vproc` の実行が成功したなら定義値のための値 `val` を取得し、変数が定義され成功が伝播されます。

```
(define (analyze-definition exp)
  (let ((var (definition-variable exp)))
    (vproc (analyze (definition-value exp))))
  (lambda (env succeed fail)
    (vproc env
      (lambda (val fail2)
        (define-variable! var val env)
        (succeed 'ok fail2))
      fail))))
```

Assignments are more interesting. This is the first place where we really use the continuations, rather than just passing them around. The execution procedure for assignments starts out like the one for definitions. It first attempts to obtain the new value to be assigned to the variable. If this evaluation of `vproc` fails, the assignment fails.

代入はもっと面白いです。これは継続をたらい回しにするのではなく、本当に継続を使用する最初の場所です。代入のための実行手続は定義のためのものと同様に開始します。最初に変数に代入される新しい値を取得しようと試みます。もしこの `vproc` の評価が失敗したら代入は失敗します。

If `vproc` succeeds, however, and we go on to make the assignment, we must consider the possibility that this branch of the computation might later fail, which will require us to backtrack out of the assignment. Thus, we must arrange to undo the assignment as part of the backtracking process.<sup>108</sup>

しかし `vproc` が成功し代入を行なおうとした場合には、この計算の分岐が後に失敗する可能性について考えねばなりません。この場合には代入から外れてバックトラックする必要があります。従ってこの代入をバックトラック処理

---

<sup>108</sup>We didn't worry about undoing definitions, since we can assume that internal definitions are scanned out (Section 4.1.6).

の一部分として取消する準備をしなければなりません。<sup>109</sup>

This is accomplished by giving `vproc` a success continuation (marked with the comment “\*1\*” below) that saves the old value of the variable before assigning the new value to the variable and proceeding from the assignment. The failure continuation that is passed along with the value of the assignment (marked with the comment “\*2\*” below) restores the old value of the variable before continuing the failure. That is, a successful assignment provides a failure continuation that will intercept a subsequent failure; whatever failure would otherwise have called `fail2` calls this procedure instead, to undo the assignment before actually calling `fail2`.

これは `vproc` に (下でコメント “\*1\*” が記された) 成功継続を与えることで達成されます。この成功継続は新しい値を代入し、結果として存在する前に、変数の古い値を保存します。代入値と共に渡された (下でコメント “\*2\*” が記された) 失敗継続は失敗を続ける前に変数の古い値を再格納します。つまり、代入の成功は後の失敗に割り込む失敗継続を提供します。そうでなければ `fail2` を呼んでいたはずのどんな失敗もこの手続を代わりに呼ぶことで、実際に `fail2` を呼ぶ前に代入を取り消します。

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp)))
    (vproc (analyze (assignment-value exp)))))

  (lambda (env succeed fail)
    (vproc env
      (lambda (val fail2) ; *1*
        (let ((old-value
              (lookup-variable-value var env)))
          (set-variable-value! var val env)
          (succeed 'ok
            (lambda () ; *2*
              (set-variable-value!
                var old-value env)
              (fail2))))))
      fail))))
```

---

<sup>109</sup> 定義の取消については心配しません。内部定義は走査されたことが想定可能なためです。(Section 4.1.6)

## Procedure applications

The execution procedure for applications contains no new ideas except for the technical complexity of managing the continuations. This complexity arises in `analyze-application`, due to the need to keep track of the success and failure continuations as we evaluate the operands. We use a procedure `get-args` to evaluate the list of operands, rather than a simple `map` as in the ordinary evaluator.

適用のための実行手続は新しいアイデアを含んではいません。ただし継続を管理する技術的な複雑さが異なります。この複雑さは `analyze-application` の中でオペランドを評価するに従い成功と失敗の継続を追跡する必要があるために、浮上します。私達は通常の評価機の中の様に単純に `map` を使うのではなく、手続 `get-args` を用いてオペランドのリストを評価しています。

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env succeed fail)
      (fproc env
        (lambda (proc fail2)
          (get-args aprocs
            env
            (lambda (args fail3)
              (execute-application
                proc args succeed fail3)
              fail3))
          fail2))))
```

In `get-args`, notice how `cdr-ing` down the list of `aproc` execution procedures and `consing` up the resulting list of `args` is accomplished by calling each `aproc` in the list with a success continuation that recursively calls `get-args`. Each of these recursive calls to `get-args` has a success continuation whose value is the `cons` of the newly obtained argument onto the list of accumulated arguments:

`get-args` の中では、どのようにして `aproc` 実行手続のリストを `cdr` で下り、そして結果の `args` のリストを全て `cons` するかについて注意して下さい。これはリスト中の全ての `aproc` を、再帰的に `get-args` を呼ぶ成功継続と共に呼ぶことにより達成されます。これらの `get-args` に対する全ての再帰的呼出は、蓄積された引数のリストの上に新しく取得された引数を `cons` した値を返

す成功継続を持っています。

```
(define (get-args aprocs env succeed fail)
  (if (null? aprocs)
      (succeed '() fail)
      ((car aprocs)
       env
       ; ;success continuation for this aproc
       ; ;この aproc のための成功継続
      (lambda (arg fail2)
        (get-args
         (cdr aprocs)
         env
         ; ;success continuation for
         ; ;recursive call to get-args
         ; ;get-args の再帰呼出のための
         ; ;成功継続
        (lambda (args fail3)
          (succeed (cons arg args) fail3))
        fail2)))
      fail)))
```

The actual procedure application, which is performed by `exe@-cute-application`, is accomplished in the same way as for the ordinary evaluator, except for the need to manage the continuations.

`exe@-cute-application` により実行される実際の手続適用は通常の評価機に対する物と同じ方法にて達成されます。ただし継続の管理の必要が異なります。

```
(define (execute-application proc args succeed fail)
  (cond ((primitive-procedure? proc)
         (succeed (apply-primitive-procedure proc args)
                  fail))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment
           (procedure-parameters proc)
           args)
```

```

(procedure-environment proc))
succeed
fail))
(else (error "Unknown procedure type:
EXECUTE-APPLICATION"
proc))))
```

## Evaluating amb expressions

The `amb` special form is the key element in the nondeterministic language. Here we see the essence of the interpretation process and the reason for keeping track of the continuations. The execution procedure for `amb` defines a loop `try-next` that cycles through the execution procedures for all the possible values of the `amb` expression. Each execution procedure is called with a failure continuation that will try the next one. When there are no more alternatives to try, the entire `amb` expression fails.

`amb` の特殊形式は非決定性言語の鍵となる要素です。ここでは逐次翻訳処理の本質と継続を追跡する理由について学びます。`amb` に対する実行手続はループ `try-next` を定義します。これは全ての `amb` 式の可能な値のために、全実行手続を通して実行します。各実行手続は次の実行手続を試す失敗継続と共に呼ばれます。試行する選択肢が無くなった時には、`amb` 式全体が失敗します。

```

(define (analyze-amb exp)
  (let ((cprocs (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
            ((car choices)
             env
             succeed
             (lambda () (try-next (cdr choices)))))))
    (try-next cprocs))))
```

## Driver loop

The driver loop for the `amb` evaluator is complex, due to the mechanism that permits the user to try again in evaluating an expression. The driver uses a procedure called `internal-loop`, which takes as argument a procedure `try-again`. The intent is that calling `try-again` should go on to the next untried alternative in the nondeterministic evaluation. `Internal-loop` either calls `try-again` in response to the user typing `try-again` at the driver loop, or else starts a new evaluation by calling `ambeval`.

`amb` 評価機のドライバループは複雑です。ユーザに式の評価を再試行 (`try-again`) することを可能にする仕組みのためです。ドライバは `internal-loop` と呼ばれる手続を使用します。これは引数として手続 `try-again` を取ります。これの意図は `try-again` の呼出は次のまだ試行されていない非決定性評価における選択肢へ続けなければならぬことです。`internal-loop` はユーザのドライバループでの `try-again` の入力への応答として `try-again` を呼ぶか、またはそうでなければ `ambeval` を呼ぶことにより新しい評価を開始します。

The failure continuation for this call to `ambeval` informs the user that there are no more values and re-invokes the driver loop.

`ambeval` へのこの呼出のための失敗継続はユーザにもう値は残っていないと伝え、ドライバループを再起動する。

The success continuation for the call to `ambeval` is more subtle. We print the obtained value and then invoke the internal loop again with a `try-again` procedure that will be able to try the next alternative. This `next-alternative` procedure is the second argument that was passed to the success continuation. Ordinarily, we think of this second argument as a failure continuation to be used if the current evaluation branch later fails. In this case, however, we have completed a successful evaluation, so we can invoke the “failure” alternative branch in order to search for additional successful evaluations.

`ambeval` への呼出のための成功継続はより微妙です。獲得した値を表示し、次に内部ループを再び起動します。起動には次の選択肢を試行可能な `try-again` 手続を伴います。この `next-alternative` 手続は成功継続に二番目の引数として渡されます。通常はこの二番目の引数はもし現在の評価分岐が後に失敗場合に利用される失敗継続として考えます。今回の場合はしかし、評価を成功裏に完了しました。そのため“失敗”的代替分岐を追加の成功する評価を探すために起動することができます。

```
(define input-prompt ";; Amb-Eval input:")
```

```

(define output-prompt ";; Amb-Eval value:")
(define (driver-loop)
  (define (internal-loop try-again)
    (prompt-for-input input-prompt)
    (let ((input (read)))
      (if (eq? input 'try-again)
          (try-again)
          (begin
            (newline)
            (display ";; Starting a new problem ")
            (ambeval
              input
              the-global-environment
              ;; ambeval success
              (lambda (val next-alternative)
                (announce-output output-prompt)
                (user-print val)
                (internal-loop next-alternative)))
            ;; ambeval failure
            (lambda ()
              (announce-output
                ";; There are no more values of")
              (user-print input)
              (driver-loop))))))
  (internal-loop
    (lambda ()
      (newline)
      (display ";; There is no current problem")
      (driver-loop)))
)

```

The initial call to `internal-loop` uses a `try-again` procedure that complains that there is no current problem and restarts the driver loop. This is the behavior that will happen if the user types `try-again` when there is no evaluation in progress.

`internal-loop` の最初の呼出では現在、問題が無いと不服を述べ、ドライバループを再開します。これはユーザが `try-again` を入力し評価に進展が無い場合に起こる振舞です。

**Exercise 4.50:** Implement a new special form `ramb` that is like `amb` except that it searches alternatives in a random order, rather than from left to right. Show how this can help with Alyssa's problem in [Exercise 4.49](#).

`amb` と似ているが、次の選択肢を左から右へでなくランダムな順で探索する新しい特殊形式 `ramb` を実装せよ。[Exercise 4.49](#)における Alyssa の問題をどのように助けるか示せ。

**Exercise 4.51:** Implement a new kind of assignment called `permanent-set!` that is not undone upon failure. For example, we can choose two distinct elements from a list and count the number of trials required to make a successful choice as follows:

失敗時に取り消されない `permanent-set!` と呼ばれる新しい種類の代入を実装せよ。例えば、以下の様に 2 つの区別可能な要素をリストから選択し、成功した選択に必要とした試行の数をカウントする。

```
(define count 0)
(let ((x (an-element-of '(a b c)))
      (y (an-element-of '(a b c))))
  (permanent-set! count (+ count 1))
  (require (not (eq? x y)))
  (list x y count))
;; Starting a new problem
;; Amb-Eval value:
(a b 2)
;; Amb-Eval input:
try-again
;; Amb-Eval value:
(a c 3)
```

What values would have been displayed if we had used `set!` here rather than `permanent-set!`?

ここで `permanent-set!` の代わりに `set!` を使ったらどんな値が表示されるだろうか？

**Exercise 4.52:** Implement a new construct called `if-fail` that permits the user to catch the failure of an expression. `If-fail` takes

two expressions. It evaluates the first expression as usual and returns as usual if the evaluation succeeds. If the evaluation fails, however, the value of the second expression is returned, as in the following example:

ユーザに式の失敗を捕獲させることを可能にする新しいコンストラクタ `if-fail` を実装せよ。`if-fail` は 2 つの式を見る。最初の式を通常通りに評価し評価が成功したら普通に戻る。しかし、もし評価が失敗したら 2 つ目の式の値が以下の例の様に返される。

```
;; Amb-Eval input:  
(if-fail (let ((x (an-element-of '(1 3 5)))  
              (require (even? x))  
              x)  
         'all-odd)  
;; Starting a new problem  
;; Amb-Eval value:  
all-odd  
  
;; Amb-Eval input:  
(if-fail (let ((x (an-element-of '(1 3 5 8)))  
              (require (even? x))  
              x)  
         'all-odd)  
;; Starting a new problem  
;; Amb-Eval value:  
8
```

**Exercise 4.53:** With `permanent-set!` as described in Exercise 4.51 and `if-fail` as in Exercise 4.52, what will be the result of evaluating

Exercise 4.51で説明した `permanent-set!` と Exercise 4.52 の `if-fail` を用いて以下を評価した時どのような結果になるか。

```
(let ((pairs '()))  
  (if-fail  
    (let ((p (prime-sum-pair '(1 3 5 8)  
                             '(20 35 110))))
```

```
(permanent-set! pairs (cons p pairs))
(amb)
pairs))
```

**Exercise 4.54:** If we had not realized that `require` could be implemented as an ordinary procedure that uses `amb`, to be defined by the user as part of a nondeterministic program, we would have had to implement it as a special form. This would require syntax procedures

もし `require` が `amb` を用いた通常の手続として実装できることに気付かなかった場合、ユーザにより非決定性プログラムの一部として定義されるために、それを特殊形式として実装する必要があったはずである。これは以下の構文手続を必要としたであろう。

```
(define (require? exp)
  (tagged-list? exp 'require))
(define (require-predicate exp)
  (cadr exp))
```

and a new clause in the dispatch in `analyze`

そして `analyze` 内の呼出に新しい節が必要となった。

```
((require? exp) (analyze-require exp))
```

as well the procedure `analyze-require` that handles `require` expressions. Complete the following definition of `analyze-require`. また `require` 式を取り扱う手続 `analyze-require` も必要となつた。以下の `analyze-require` の定義を完成させよ。

```
(define (analyze-require exp)
  (let ((pproc (analyze (require-predicate exp))))
    (lambda (env succeed fail)
      (pproc env
        (lambda (pred-value fail2)
          (if (??)
              (??)
              (succeed 'ok fail2)))
        fail))))
```

## 4.4 Logic Programming

In [Chapter 1](#) we stressed that computer science deals with imperative (how to) knowledge, whereas mathematics deals with declarative (what is) knowledge. Indeed, programming languages require that the programmer express knowledge in a form that indicates the step-by-step methods for solving particular problems. On the other hand, high-level languages provide, as part of the language implementation, a substantial amount of methodological knowledge that frees the user from concern with numerous details of how a specified computation will progress.

[Chapter 1](#)では計算機科学は命令型（どうするか）の知識を扱い、一方、数学は宣言型（何であるか）の知識を扱うと強調しました。実際に、プログラミング言語はプログラマが特定の問題を解くために、段階的な手法を示す形式により、知識を表現することを要求します。一方、高級言語は言語の実装の一部としてユーザを、指定された演算がどのように進められるかについての詳細から解放する、十分な量の方法論的知識を提供します。

Most programming languages, including Lisp, are organized around computing the values of mathematical functions. Expression-oriented languages (such as Lisp, Fortran, and Algol) capitalize on the “pun” that an expression that describes the value of a function may also be interpreted as a means of computing that value. Because of this, most programming languages are strongly biased toward unidirectional computations (computations with well-defined inputs and outputs). There are, however, radically different programming languages that relax this bias. We saw one such example in [Section 3.3.5](#), where the objects of computation were arithmetic constraints. In a constraint system the direction and the order of computation are not so well specified; in carrying out a computation the system must therefore provide more detailed “how to” knowledge than would be the case with an ordinary arithmetic computation. This does not mean, however, that the user is released altogether from the responsibility of providing imperative knowledge. There are many constraint networks that implement the same set of constraints, and the user must choose from the set of mathematically equivalent networks a suitable network to specify a particular computation.

Lisp を含む多くのプログラミング言語は数学上の関数の値の演算の周りに体系化されています。式指向の言語（例えば Lisp, Fortran, Algol）は関数の値を記述する式がその値を求める手段としても解釈されるという“多義性”を十

分に活用しています。このため、多くのプログラミング言語は単向性演算(明確な入力と出力を持つ演算)に向けて強く偏っています。しかし、この偏りを緩和する完全に異なるプログラミング言語も存在します。そのような言語の例をSection 3.3.5で見ました。そこでは計算オブジェクトは数値的な制約でした。制約システムでは演算の向きと順はあまり明らかには指定されません。従って、演算の実行においてシステムはより詳細な“行い方”的知識を、通常の数値演算による場合よりも多く提供しなければなりません。しかし、これはユーザが命令型の知識を提供する責任から完全に解放されることは意味しません。同じ制約の集合を実装する制約ネットワークは数多く存在し、ユーザは数学的に等価なネットワークの中から特定の演算を指定するのに適切なネットワークを選択せねばなりません。

The nondeterministic program evaluator of Section 4.3 also moves away from the view that programming is about constructing algorithms for computing unidirectional functions. In a nondeterministic language, expressions can have more than one value, and, as a result, the computation is dealing with relations rather than with single-valued functions. Logic programming extends this idea by combining a relational vision of programming with a powerful kind of symbolic pattern matching called *unification*.<sup>110</sup>

---

<sup>110</sup>Logic programming has grown out of a long history of research in automatic theorem proving. Early theorem-proving programs could accomplish very little, because they exhaustively searched the space of possible proofs. The major breakthrough that made such a search plausible was the discovery in the early 1960s of the *unification algorithm* and the *resolution principle* (Robinson 1965). Resolution was used, for example, by Green and Raphael (1968) (see also Green 1969) as the basis for a deductive question-answering system. During most of this period, researchers concentrated on algorithms that are guaranteed to find a proof if one exists. Such algorithms were difficult to control and to direct toward a proof. Hewitt (1969) recognized the possibility of merging the control structure of a programming language with the operations of a logic-manipulation system, leading to the work in automatic search mentioned in Section 4.3.1 (Footnote 4.47). At the same time that this was being done, Colmerauer, in Marseille, was developing rule-based systems for manipulating natural language (see Colmerauer et al. 1973). He invented a programming language called Prolog for representing those rules. Kowalski (1973; 1979), in Edinburgh, recognized that execution of a Prolog program could be interpreted as proving theorems (using a proof technique called linear Horn-clause resolution). The merging of the last two strands led to the logic-programming movement. Thus, in assigning credit for the development of logic programming, the French can point to Prolog's genesis at the University of Marseille, while the British can highlight the work at the University of Edinburgh. According

Section 4.3の非決定性プログラム評価機もまたプログラミングとは一方向性関数のためのアルゴリズムを構築することであるという視点から離れています。非決定性言語においては、式は 2つ以上の値を持つことができ、結果として演算とは単一の値の関数ではなく関係性を取り扱う物になります。論理プログラミングはプログラミングの関係性の視点と *unification*(ユニフィケーション、单一化) と呼ばれる強力な種類の記号パターンマッチングとを組み合わせることでこの考えを拡張します。<sup>111</sup>

This approach, when it works, can be a very powerful way to write programs. Part of the power comes from the fact that a single “what is” fact can be used to solve a number of different problems that would have different “how to” components. As an example, consider the `append` operation, which takes two lists as arguments and combines their elements to form a single list. In a procedural language such as Lisp, we could define `append` in terms of the basic list constructor `cons`, as we did in Section 2.2.1:

---

to people at MIT, logic programming was developed by these groups in an attempt to figure out what Hewitt was talking about in his brilliant but impenetrable Ph.D. thesis. For a history of logic programming, see Robinson 1983.

<sup>111</sup>論理プログラミングは自動定理証明の研究の長い歴史から成長しました。早期の定理証明プログラムはあまり目的を達成することができませんでした。可能な証明空間を網羅的に探索するためです。受け入れ可能な探索を行える打開策の主な物は 1960 年代早期の *unification algorithm*(ユニフィケーションアルゴリズム) と *resolution principle*(導出原理)(Robinson 1965) の発見でした。例えば導出は Green and Raphael (1968) により (Green 1969 も参照) 演繹的質問応答システムの基盤として使用されました。この時期の多くの間、研究者達は証明が存在するのであれば見つけられることが保証されたアルゴリズムについて集中していました。そのようなアルゴリズムはコントロールして証明に向かわせることが難しいものでした。Hewitt (1969) はプログラミング言語のコントロール構造と論理操作システムとの結合の可能性を認識し、Section 4.3.1(Footnote 4.47) で述べられた自動探索の成果へと導きました。同時期に、マルセイユの Colmerauer は自然言語を扱うルールベースシステム (Colmerauer et al. 1973) により同じ事を達成しました。彼は Prolog と呼ばれるプログラミング言語を開発しそれらのルールを表現しました。Kowalski (1973; 1979) はエディンバラにて、Prolog プログラムの実行は(線形ホーン節導出と呼ばれる証明のテクニックを用いて) 定理証明として解釈できることを認めました。最後の 2つの糸を縫り合わせることが論理プログラミング運動へと導きました。従って論理プログラミングの開発に対して功績を与えることにおいて、フランス人はマルセイユ大学での Prolog の起源を指摘することができ、一方、イギリス人はエディンバラ大学の成果を強調することができます。MIT の人々に言わせれば、論理プログラミングはこれらのグループにより、Hewitt がその才能ある、しかし頑迷な博士論文にて何を伝えていたかを解き明かす試みにより開発されました。論理プログラミングの歴史については Robinson 1983 を参照して下さい。

この取り組み方は、うまく行く場合には、プログラムを書くのにとても強力な方法となります。その力の一部は単一の“何であるか”という事実が、異なる“行い方”的要素を持つかもしれないいくつかの異なる問題を解決するのに使用できるという事実から来ています。例として、`append` 命令について考えましょう。これは 2 つのリストを引数として取り、それらの要素を結合して単一のリストを形成します。Lisp のような手続型言語では `append` を基本的なリスト構造 `cons` を用いて、Section 2.2.1 で行ったように定義することができます。

```
(define (append x y)
  (if (null? x) y (cons (car x) (append (cdr x) y))))
```

This procedure can be regarded as a translation into Lisp of the following two rules, the first of which covers the case where the first list is empty and the second of which handles the case of a nonempty list, which is a `cons` of two parts:

この手続は以下の 2 つのルールにより Lisp へと翻訳したと捕えることができます。最初のルールは 1 つ目のリストが空である場合を扱い、2 つ目のルールは空でないリスト、つまり 2 つの部分による `cons` の場合を扱います。

- For any list `y`, the empty list and `y` `append` to form `y`.  
任意のリスト `y` に対し、空リストと `y` の `append` は `y` を形成する。
- For any `u`, `v`, `y`, and `z`, `(cons u v)` and `y` `append` to form `(cons u z)` if `v` and `y` `append` to form `z`.<sup>112</sup>  
任意の `u`, `v`, `y`, `z` に対し、`(cons u v)` と `y` の `append` はもし `v` と `y` の `append` が `z` を形成するならば `(cons u z)` を形成する。<sup>113</sup>

Using the `append` procedure, we can answer questions such as

`append` 手続を用いることで、私達は次のような質問に答えることができます。

---

<sup>112</sup>To see the correspondence between the rules and the procedure, let `x` in the procedure (where `x` is nonempty) correspond to `(cons u v)` in the rule. Then `z` in the rule corresponds to the `append` of `(cdr x)` and `y`.

<sup>113</sup>ルールと手続の間の対応を見るためには、手続における `x` (`x` が空でない場合) をルールの `(cons u v)` に対応させます。次にルールの `z` は `(cdr x)` と `y` の `append` に対応します。

Find the `append` of `(a b)` and `(c d)`.

`(a b)` と `(c d)` の `append` を求めよ。

But the same two rules are also sufficient for answering the following sorts of questions, which the procedure can't answer:

しかし同じ 2 つのルールがまた以下のような種類の質問に答えるためにも十分です。これらは手続では答えられません。

Find a list `y` that `appends` with `(a b)` to produce `(a b c d)`.

`(a b)` と `append` すると `(a b c d)` を生成するリスト `y` を求めよ。

Find all `x` and `y` that `append` to form `(a b c d)`.

`append` すると `(a b c d)` を生成する全ての `x` と `y` を求めよ。

In a logic programming language, the programmer writes an `append` “procedure” by stating the two rules about `append` given above. “How to” knowledge is provided automatically by the interpreter to allow this single pair of rules to be used to answer all three types of questions about `append`.<sup>114</sup>

論理プログラミング言語ではプログラマは `append`“手続”を上で与えられた `append` に関する 2 つのルールを提示することにより記述します。“行い方”の知識は自動的にインタプリタにより提供されこの単一ペアのルールが 3 つ全てのタイプの `append` に関する質問に対して答えることを可能にします。<sup>115</sup>

Contemporary logic programming languages (including the one we implement here) have substantial deficiencies, in that their general “how to” methods can lead them into spurious infinite loops or other undesirable behavior. Logic

---

<sup>114</sup>This certainly does not relieve the user of the entire problem of how to compute the answer. There are many different mathematically equivalent sets of rules for formulating the `append` relation, only some of which can be turned into effective devices for computing in any direction. In addition, sometimes “what is” information gives no clue “how to” compute an answer. For example, consider the problem of computing the `y` such that  $y^2 = x$ .

<sup>115</sup>これは確かにユーザをどのように回答を求めるかという問題全体からは解放しません。`append` の関係を形式化するための数学的に等価なルールは数多く存在します。それらのいくつかのみが任意の方向の演算に対する効果的な手段と成ります。付け加えて、時々、“何であるか”という情報は“どのように”回答を求めるかについて何の手掛かりも与えない場合があります。例えば  $y^2 = x$  となる `y` を求める問題について考えてみて下さい。

programming is an active field of research in computer science.<sup>116</sup>

現代の論理プログラミング言語(ここで私達が実装しているものを含めて)にはかなりの量の不足がそれらの一般的な“行い方”的手法について存在します。このことが偽の無限ループを引き起したり、他の望ましくない振舞へと導いてしまします。論理プログラミングは計算機科学において活発な研究領域です。<sup>117</sup>

Earlier in this chapter we explored the technology of implementing interpreters and described the elements that are essential to an interpreter for a Lisp-like language (indeed, to an interpreter for any conventional language). Now we will apply these ideas to discuss an interpreter for a logic programming language. We call this language the *query language*, because it is very useful for retrieving information from data bases by formulating *queries*, or questions, expressed in the language. Even though the query language is very different from Lisp, we will find it convenient to describe the language in terms of the same general framework we have been using all along: as a collection of primitive elements, together with means of combination that enable us to combine simple elements to create more complex elements and means of ab-

---

<sup>116</sup>Interest in logic programming peaked during the early 80s when the Japanese government began an ambitious project aimed at building superfast computers optimized to run logic programming languages. The speed of such computers was to be measured in LIPS (Logical Inferences Per Second) rather than the usual FLOPS (Floating-point Operations Per Second). Although the project succeeded in developing hardware and software as originally planned, the international computer industry moved in a different direction. See Feigenbaum and Shrobe 1993 for an overview evaluation of the Japanese project. The logic programming community has also moved on to consider relational programming based on techniques other than simple pattern matching, such as the ability to deal with numerical constraints such as the ones illustrated in the constraint-propagation system of Section 3.3.5.

<sup>117</sup>論理プログラミングへの興味は80年代早期に日本政府が論理プログラミング言語を実行するのに最適化されたとても速い計算機を構築することを狙った大望あるプロジェクトを開始した時にピークを迎えました。そのような計算機のスピードは通常のFLOPS(Floating-point Operations Per Second)ではなくLIPS(Logical Inferences Per Second)で計られます。プロジェクトはハードウェアとソフトウェアの開発において元々の計画通りに成功しましたが、国際的なコンピュータ業界は異なる方向へと向かいました。日本のプロジェクトの評価の概観についてはFeigenbaum and Shrobe 1993を参照して下さい。論理プログラミングコミュニティもまた、Section 3.3.5の制約伝播システムで説明されたような数値値上の制約を取り扱う能力の様な単純なパターンマッチングではない技術を基盤にしたリレーションナルプログラミングへと移行しました。

straction that enable us to regard complex elements as single conceptual units. An interpreter for a logic programming language is considerably more complex than an interpreter for a language like Lisp. Nevertheless, we will see that our query-language interpreter contains many of the same elements found in the interpreter of [Section 4.1](#). In particular, there will be an “eval” part that classifies expressions according to type and an “apply” part that implements the language’s abstraction mechanism (procedures in the case of Lisp, and *rules* in the case of logic programming). Also, a central role is played in the implementation by a frame data structure, which determines the correspondence between symbols and their associated values. One additional interesting aspect of our query-language implementation is that we make substantial use of streams, which were introduced in [Chapter 3](#).

この章の最初では私達はインタプリタの実装技術を探求し Lisp の様な言語のためのインタプリタに対して(実際に、任意の従来の言語に対して)本質である要素を説明しました。今から私達はこれらの考えを応用し論理プログラミング言語のためのインタプリタについて議論します。この言語を *query language*(クエリ言語)と呼ぶことにします。言語で内で表現される、*queries*(クエリ)、つまり質問を定式化することによりデータベースから情報を取得することに対してとても便利なためです。クエリ言語は Lisp と全く違うであるにも係らず、私達がここまで利用してきた同じ一般的なフレームワークを用いてこの言語を説明することがとても都合が良いことを理解するでしょう。このフレームワークはプリミティブな要素の集合として、簡単な要素を組み合わせることでより複雑な要素を作ることを可能にする組み合わせの手段と、複雑な要素を単純な概念の単位として見做すことを可能にする抽象化の手段とと一緒に用いました。論理プログラミング言語向けインタプリタは Lisp のような言語のインタプリタよりも大幅に複雑です。それでも、私達のクエリ言語インタプリタが [Section 4.1](#) のインタプリタ内にて見つけた多くの同じ要素を含むことを学ぶでしょう。具体的には式を方に従って分類する“eval”的パートが存在し、そして言語の抽象化の仕組み(Lisp の場合では手続であり、論理プログラミングの場合ではルール)を実装する“apply”的パートが存在します。また、中心的な役割はフレームデータ構造により実装の中で演じられます。このフレームデータ構造はシンボルとそれらに関連する値の間の対応を決定します。クエリ言語の実装の追加の面白い側面の 1 つは、[Chapter 3](#) で紹介したストリームを大量に使用することです。

#### 4.4.1 Deductive Information Retrieval

Logic programming excels in providing interfaces to data bases for information retrieval. The query language we shall implement in this chapter is designed to be used in this way.

論理プログラミングは情報取得のためのデータベースに対するインターフェイスの提供において秀でています。私達がこの章で実装するクエリ言語はこのように使用されるよう設計されています。

In order to illustrate what the query system does, we will show how it can be used to manage the data base of personnel records for Microshaft, a thriving high-technology company in the Boston area. The language provides pattern-directed access to personnel information and can also take advantage of general rules in order to make logical deductions.

クエリシステムが何を行うかを説明するために、ボストン区域に存在する成長中のハイテク企業、Microshaft の社員情報のデータベースを管理するためには、クエリシステムがどのように利用できるかについて示します。この言語はパターンにより示される社員情報へのアクセスを提供し、また論理的演繹法を行なうための一般的なルールの利点を得ることができます。

#### A sample data base

The personnel data base for Microshaft contains *assertions* about company personnel. Here is the information about Ben Bitdiddle, the resident computer wizard:

Microshaft の社員情報データベースは会社の全社員に関する *assertions*(アサーション、表明) を保持します。以下に常駐のコンピュータウィザード、Ben Bitdiddle に関する情報を挙げます。

```
(address (Bitdiddle Ben) (Slumerville (Ridge Road) 10))  
(job (Bitdiddle Ben) (computer wizard))  
(salary (Bitdiddle Ben) 60000)
```

Each assertion is a list (in this case a triple) whose elements can themselves be lists.

各アサーションはリスト(この場合 3 つ組)で、その要素はそれ自体がリストに成り得ます。

As resident wizard, Ben is in charge of the company's computer division, and he supervises two programmers and one technician. Here is the information

about them:

常駐のウィザードとして、Ben は会社のコンピュータ部門を管理し、二人のプログラマと一人の技術者を監督します。以下に部下に関する情報を挙げます。

```
(address (Hacker Alyssa P) (Cambridge (Mass Ave) 78))
(job (Hacker Alyssa P) (computer programmer))
(salary (Hacker Alyssa P) 40000)
(supervisor (Hacker Alyssa P) (Bitdiddle Ben))
```

```
(address (Fect Cy D) (Cambridge (Ames Street) 3))
(job (Fect Cy D) (computer programmer))
(salary (Fect Cy D) 35000)
(supervisor (Fect Cy D) (Bitdiddle Ben))
```

```
(address (Tweakit Lem E) (Boston (Bay State Road) 22))
(job (Tweakit Lem E) (computer technician))
(salary (Tweakit Lem E) 25000)
(supervisor (Tweakit Lem E) (Bitdiddle Ben))
```

There is also a programmer trainee, who is supervised by Alyssa:  
Alyssa に監督されているプログラマ見習いもいます。

```
(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
(job (Reasoner Louis) (computer programmer trainee))
(salary (Reasoner Louis) 30000)
(supervisor (Reasoner Louis) (Hacker Alyssa P))
```

All of these people are in the computer division, as indicated by the word **computer** as the first item in their job descriptions.

これらの人々全てはコンピュータ部門に属し、彼等の職位 (job) 記述の最初の項目である単語 **computer** により示されています。

Ben is a high-level employee. His supervisor is the company's big wheel himself:

Ben は高位の従業員です。彼の監督者は会社の有力者である彼自身です。

```
(supervisor (Bitdiddle Ben) (Warbucks Oliver))
(address (Warbucks Oliver) (Swellesley (Top Heap Road)))
(job (Warbucks Oliver) (administration big wheel))
(salary (Warbucks Oliver) 150000)
```

Besides the computer division supervised by Ben, the company has an accounting division, consisting of a chief accountant and his assistant:

コンピュータ部門が Ben に監督されているのに加えて、会社には会計士長とそのアシスタントから成る経理部門があります。

```
(address (Scrooge Eben) (Weston (Shady Lane) 10))
(job (Scrooge Eben) (accounting chief accountant))
(salary (Scrooge Eben) 75000)
(supervisor (Scrooge Eben) (Warbucks Oliver))

(address (Cratchet Robert) (Allston (N Harvard Street) 16))
(job (Cratchet Robert) (accounting scrivener))
(salary (Cratchet Robert) 18000)
(supervisor (Cratchet Robert) (Scrooge Eben))
```

There is also a secretary for the big wheel:

また重役のための秘書もいます。

```
(address (Aull DeWitt) (Slumerville (Onion Square) 5))
(job (Aull DeWitt) (administration secretary))
(salary (Aull DeWitt) 25000)
(supervisor (Aull DeWitt) (Warbucks Oliver))
```

The data base also contains assertions about which kinds of jobs can be done by people holding other kinds of jobs. For instance, a computer wizard can do the jobs of both a computer programmer and a computer technician:

データベースはまたどの職種が他の職種を持つ人々により行われることができるかに関するアサーションも含みます。例えばコンピュータウィザードはコンピュータプログラマとコンピュータ技術者の両方の職を行うことができます。

```
(can-do-job (computer wizard) (computer programmer))
(can-do-job (computer wizard) (computer technician))
```

A computer programmer could fill in for a trainee:

コンピュータプログラマは見習いを埋めることができるでしょう。

```
(can-do-job (computer programmer)
            (computer programmer trainee))
```

Also, as is well known,

また良く知られているように以下も言えます。

```
(can-do-job (administration secretary)
             (administration big wheel))
```

## Simple queries

The query language allows users to retrieve information from the data base by posing queries in response to the system's prompt. For example, to find all computer programmers one can say

クエリ言語はユーザにシステムプロンプトに対する応答としてクエリを提示させることで、データベースから情報を取得することを許します。

```
; ; ; Query input:
(job ?x (computer programmer))
```

The system will respond with the following items:

システムは以下の項目を返します。

```
; ; ; Query results:
(job (Hacker Alyssa P) (computer programmer))
(job (Fect Cy D) (computer programmer))
```

The input query specifies that we are looking for entries in the data base that match a certain *pattern*. In this example, the pattern specifies entries consisting of three items, of which the first is the literal symbol `job`, the second can be anything, and the third is the literal list `(computer programmer)`. The “anything” that can be the second item in the matching list is specified by a *pattern variable*, `?x`. The general form of a pattern variable is a symbol, taken to be the name of the variable, preceded by a question mark. We will see below why it is useful to specify names for pattern variables rather than just putting `?`  into patterns to represent “anything.” The system responds to a simple query by showing all entries in the data base that match the specified pattern.

入力クエリはある種のパターンにマッチするデータベース内のエントリを探すことを指示します。この例では、パターンは 3 つの項目から成るエントリを指定しています。最初が文字シンボルの `job`、2 つ目は任意の値に成り得て、3 番目は文字のリスト `(computer programmer)` です。マッチングリスト内

の 2 つ目の項目に成り得る“任意項”は *pattern variable*(パターン変数)?*x* で指定されます。パターン変数の一般的な形式はクエスチョンマークを前に置いた、変数の名前として取られるシンボルです。以下では、なぜこのことが単に? を“任意”を表すパターンに置くのではなく、パターン変数のために名前を指定することが便利であるかを学びます。システムは簡単なクエリに指定されたパターンにマッチするデータベース内の全てのエントリを表示することで応答します。

A pattern can have more than one variable. For example, the query  
パターンは複数の変数を持つことができます。例えば、以下のクエリ

```
(address ?x ?y)
```

will list all the employees' addresses.

は全ての従業員の住所を並べます。

A pattern can have no variables, in which case the query simply determines whether that pattern is an entry in the data base. If so, there will be one match; if not, there will be no matches.

パターンはクエリが単純にパターンがデータベース内のエントリであるかどうかを決定する場合には変数を持つことができません。もしそうであれば 1 つの一致が存在します。そうでなければ 1 つも一致は存在しません。

The same pattern variable can appear more than once in a query, specifying that the same “anything” must appear in each position. This is why variables have names. For example,

同じパターン変数が 1 つのクエリ内に複数存在することができ、同じ“任意項”が各位置に現われなければいけないこと指定します。これがなぜ変数が名前を持つのかの理由です。例えば、

```
(supervisor ?x ?x)
```

finds all people who supervise themselves (though there are no such assertions in our sample data base).

上のクエリは自分自身を監督する全ての人々を見つけます。(しかし私達のサンプルデータベース内のアサーションにはそのようなエントリがありません。)

The query

以下のクエリは、

```
(job ?x (computer ?type))
```

matches all job entries whose third item is a two-element list whose first item is `computer`:

3つ目の項目が二要素リストでありその1つ目の要素が`computer`である全ての職種エントリに適合します。

```
(job (Bitdiddle Ben) (computer wizard))
(job (Hacker Alyssa P) (computer programmer))
(job (Fect Cy D) (computer programmer))
(job (Tweakit Lem E) (computer technician))
```

This same pattern does *not* match

この同じパターンが以下にはマッチ“しません”。

```
(job (Reasoner Louis) (computer programmer trainee))
```

because the third item in the entry is a list of three elements, and the pattern's third item specifies that there should be two elements. If we wanted to change the pattern so that the third item could be any list beginning with `computer`, we could specify<sup>118</sup>

なぜならエントリの3つ目の項目が3要素のリストであり、パターンの3つ目の項目がそこは2要素でなければならないと指定しているためです。もし私達がパターンを変更し3つ目の項目が`computer`で始まる任意のリストでも良いようにしたければ、以下のように指定可能です。<sup>119</sup>

```
(job ?x (computer . ?type))
```

For example,

例えば、以下のクエリは、

```
(computer . ?type)
```

matches the data

次のデータに適合します。

```
(computer programmer trainee)
```

with `?type` as the list `(programmer trainee)`. It also matches the data

この時`?type`はリスト `(programmer trainee)`になります。これはまた次のデータにも適合します。

---

<sup>118</sup>This uses the dotted-tail notation introduced in [Exercise 2.20](#).

<sup>119</sup>これは[Exercise 2.20](#)で紹介されたドット付き末尾記述を用いています。

(computer programmer)

with ?type as the list (programmer), and matches the data

この時?type はリスト (programmer) になります。さらに以下のデータにも適合します。

(computer)

with ?type as the empty list () .

この時?type は空リスト () です。

We can describe the query language's processing of simple queries as follows:  
クエリ言語の簡単なクエリの処理は以下のように説明できます。

- The system finds all assignments to variables in the query pattern that *satisfy* the pattern—that is, all sets of values for the variables such that if the pattern variables are *instantiated with* (replaced by) the values, the result is in the data base.

システムはクエリパターン内の変数に対する、パターンを満たす全ての割り当てを見つけます—つまり、パターン変数が値によりインスタンス化されるような(例示されるような)、つまり値により置き換えられるような変数に対する値の全ての集合です。結果はデータベース内に存在します。

- The system responds to the query by listing all instantiations of the query pattern with the variable assignments that satisfy it.

システムはクエリに対し、パターンを満たす変数割り当てと共に、クエリパターンの全てのインスタンス(事例)を列挙することで応答します。

Note that if the pattern has no variables, the query reduces to a determination of whether that pattern is in the data base. If so, the empty assignment, which assigns no values to variables, satisfies that pattern for that data base.

もしパターンに変数が無い場合、クエリはそのパターンがデータベース内に存在するかどうかの決定に簡約されることに注意して下さい。もしそうならば、変数に何の値も割り当てない空割り当てがデータベースに対するそのパターンを満たします。

**Exercise 4.55:** Give simple queries that retrieve the following information from the data base:

以下の情報をデータベースから取り出す簡単なクエリを与えよ。

1. all people supervised by Ben Bitdiddle;  
Ben Bitdiddle により監督される (supervisor) 全ての人
2. the names and jobs of all people in the accounting division;  
経理部門に属す全ての人の名前 (name) と職種 (job)
3. the names and addresses of all people who live in Slumerville.  
Slumerville に済む全ての人の名前と住所 (address)

## Compound queries

Simple queries form the primitive operations of the query language. In order to form compound operations, the query language provides means of combination. One thing that makes the query language a logic programming language is that the means of combination mirror the means of combination used in forming logical expressions: `and`, `or`, and `not`. (Here `and`, `or`, and `not` are not the Lisp primitives, but rather operations built into the query language.)

単純なクエリはクエリ言語のプリミティブな命令を形成します。複雑な命令を形成するためには、クエリ言語は組み合わせの手段を提供します。クエリ言語を論理プログラミング言語と成す物の 1 つに組み合わせの手段が論理式を形成するのに用いられる組み合わせの手段に酷似することができます。`and`, `or`, `not` です。(ここでは `and`, `or`, `not` は Lisp のプリミティブではありません。クエリ言語の組込命令です。)

We can use `and` as follows to find the addresses of all the computer programmers:

`and` を以下の様に用いて全てのコンピュータプログラマの住所を見つけることができます。

```
(and (job ?person (computer programmer))
      (address ?person ?where))
```

The resulting output is

結果の出力は以下の通りです。

```
(and (job (Hacker Alyssa P) (computer programmer))
      (address (Hacker Alyssa P) (Cambridge (Mass Ave) 78)))
(and (job (Fect Cy D) (computer programmer))
      (address (Fect Cy D) (Cambridge (Ames Street) 3)))
```

In general,  
一般的に、

(**and** ⟨query<sub>1</sub>⟩ ⟨query<sub>2</sub>⟩ ... ⟨query<sub>n</sub>⟩))

is satisfied by all sets of values for the pattern variables that simultaneously satisfy ⟨query<sub>1</sub>⟩ ... ⟨query<sub>n</sub>⟩.

上の式はパターン変数に対する全ての値の集合が同時に ⟨query<sub>1</sub>⟩ ... ⟨query<sub>n</sub>⟩ を満す時に満たされます。

As for simple queries, the system processes a compound query by finding all assignments to the pattern variables that satisfy the query, then displaying instantiations of the query with those values.

簡単にクエリに関しては、システムはクエリを満たすパターン変数への全ての割り当てを見つけることにより複合クエリを処理します。そしてそれらの値によるクエリのインスタンスを表示します。

Another means of constructing compound queries is through **or**. For example, 複合クエリを構築する別の手段として **or** を通す方法があります。例えば、

(**or** (*supervisor* ?x (Bitdiddle Ben))  
    (*supervisor* ?x (Hacker Alyssa P)))

will find all employees supervised by Ben Bitdiddle or Alyssa P. Hacker:

上の式は Ben Bitdiddle、または Alyssa P. Hacker に監督される従業員全てを見つけます。

(**or** (*supervisor* (Hacker Alyssa P) (Bitdiddle Ben))  
    (*supervisor* (Hacker Alyssa P) (Hacker Alyssa P)))  
(**or** (*supervisor* (Fect Cy D) (Bitdiddle Ben))  
    (*supervisor* (Fect Cy D) (Hacker Alyssa P)))  
(**or** (*supervisor* (Tweakit Lem E) (Bitdiddle Ben))  
    (*supervisor* (Tweakit Lem E) (Hacker Alyssa P)))  
(**or** (*supervisor* (Reasoner Louis) (Bitdiddle Ben))  
    (*supervisor* (Reasoner Louis) (Hacker Alyssa P)))

In general,  
一般的に、

(**or** ⟨query<sub>1</sub>⟩ ⟨query<sub>2</sub>⟩ ... ⟨query<sub>n</sub>⟩))

is satisfied by all sets of values for the pattern variables that satisfy at least one of ⟨query<sub>1</sub>⟩ ... ⟨query<sub>n</sub>⟩.

上の式はパターン変数に対する全ての値の集合が、 $\langle query_1 \rangle \dots \langle query_n \rangle$  の内、少くとも 1 つを満たす場合に満たされます。

Compound queries can also be formed with `not`. For example,

複合クエリはまた `not` を用いても形成できます。例えば、

```
(and (supervisor ?x (Bitdiddle Ben))
      (not (job ?x (computer programmer))))
```

finds all people supervised by Ben Bitdiddle who are not computer programmers. In general,

上の式は Ben Bitdiddle に監督されるが、コンピュータプログラマではない全ての人を見つけます。一般的に、

```
(not ⟨query1⟩)
```

is satisfied by all assignments to the pattern variables that do not satisfy  $\langle query_1 \rangle$ .<sup>120</sup>

上の式はパターン変数に対する全ての割り当てが  $\langle query_1 \rangle$  を満たさない場合に満たされます。<sup>121</sup>

The final combining form is called `lisp-value`. When `lisp-value` is the first element of a pattern, it specifies that the next element is a Lisp predicate to be applied to the rest of the (instantiated) elements as arguments. In general,

最後の組み合わせ形式は `lisp-value` と呼ばれます。`lisp-value` がパターンの最初の要素の時、次の要素は（インスタンス化された）残りの要素を引数として適用される Lisp の述語であることを意味します。一般的に、

```
(lisp-value ⟨predicate⟩ ⟨arg1⟩ ... ⟨argn⟩)
```

will be satisfied by assignments to the pattern variables for which the  $\langle predicate \rangle$  applied to the instantiated  $\langle arg_1 \rangle \dots \langle arg_n \rangle$  is true. For example, to find all people whose salary is greater than \$30,000 we could write<sup>122</sup>

<sup>120</sup>Actually, this description of `not` is valid only for simple cases. The real behavior of `not` is more complex. We will examine `not`'s peculiarities in sections [Section 4.4.2](#) and [Section 4.4.3](#).

<sup>121</sup>実際にはこの `not` の説明は簡単な場合に対してのみ有効です。本当の `not` の振舞はより複雑です。`not` の奇妙な点については節[Section 4.4.2](#)と[Section 4.4.3](#)にて調査します。

<sup>122</sup>Lisp-value should be used only to perform an operation not provided in the query language. In particular, it should not be used to test equality (since that is what the matching in the query language is designed to do) or inequality (since that can be done with the `same` rule shown below).

上の式は  $\langle predicate \rangle$  がパターン変数に対してインスタンス化された  $\langle arg_1 \rangle$  …  $\langle arg_n \rangle$  に適用された時の値が true になる場合の割り当てにより満たされます。

```
(and (salary ?person ?amount) (lisp-value > ?amount 30000))
```

**Exercise 4.56:** Formulate compound queries that retrieve the following information:

以下の情報を取得する複合クエリを定式化せよ。

- a the names of all people who are supervised by Ben Bitdiddle, together with their addresses;

Ben Bitdiddle に監督される全ての人の名前と住所と共に

- b all people whose salary is less than Ben Bitdiddle's, together with their salary and Ben Bitdiddle's salary;

Ben Bitdiddle よりも給料 (salary) が安い全ての人をその給料と Ben Bitdiddle の給料と共に

- c all people who are supervised by someone who is not in the computer division, together with the supervisor's name and job.

コンピュータ部門ではない人に監督されている全ての人をその上司の名前と職種と共に

## Rules

In addition to primitive queries and compound queries, the query language provides means for abstracting queries. These are given by *rules*. The rule

プリミティブなクエリと複合クエリに加えて、クエリ言語はクエリを抽象化する手段を提供します。これらは *rules*(ルール) により提供されます。以下のルールは、

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
            (address ?person-2 (?town . ?rest-2)))
      (not (same ?person-1 ?person-2))))
```

specifies that two people live near each other if they live in the same town. The final `not` clause prevents the rule from saying that all people live near themselves. The `same` relation is defined by a very simple rule:<sup>123</sup>

二人の人が同じ街に住んでいるのなら、お互いに近くに住んでいると指定しています。最後の `not` 節はこのルールが全ての人がその人自身の近くに住んでいると言うことを防ぎます。`same` リレーションはとても簡単なルールにより定義されます。<sup>124</sup>

```
(rule (same ?x ?x))
```

The following rule declares that a person is a “wheel” in an organization if he supervises someone who is in turn a supervisor:

以下のルールはある人が監督する人が同様に監督者である場合に組織内の“wheel”(重要人物)であると宣言します。

```
(rule (wheel ?person)
      (and (supervisor ?middle-manager ?person)
            (supervisor ?x ?middle-manager)))
```

The general form of a rule is

ルールの一般的な形式は以下となります。

```
(rule <conclusion> <body>)
```

---

<sup>123</sup>Notice that we do not need `same` in order to make two things be the same: We just use the same pattern variable for each—in effect, we have one thing instead of two things in the first place. For example, see `?town` in the `lives-near` rule and `?middle-manager` in the `wheel` rule below. `Same` is useful when we want to force two things to be different, such as `?person-1` and `?person-2` in the `lives-near` rule. Although using the same pattern variable in two parts of a query forces the same value to appear in both places, using different pattern variables does not force different values to appear. (The values assigned to different pattern variables may be the same or different.)

<sup>124</sup>2つの物が同じであるようにするために `same` は必要ではないことに注意して下さい。単に同じパターン変数をそれぞれに使用するだけです。実際に、最初から2つの物でなく1つの物しか持ちません。例として `lives-near` ルールの `?town` や下記の `wheel` ルールの `?middle-manager` を参照して下さい。`same` は2つの物が異なることを強制する場合に便利です。例えば `lives-near` ルールの `?person-1` と `?person-2` です。同じパターン変数をクエリの2つの部分に使うことは両方の場所に同じ値が現れることを強制しますが、異なるパターン変数を用いることは異なる値が現れることを強制しません。(異なるパターン変数に割り当てられた値は同じにも違う値にもなり得ます。)

where  $\langle \text{conclusion} \rangle$  is a pattern and  $\langle \text{body} \rangle$  is any query.<sup>125</sup> We can think of a rule as representing a large (even infinite) set of assertions, namely all instantiations of the rule conclusion with variable assignments that satisfy the rule body. When we described simple queries (patterns), we said that an assignment to variables satisfies a pattern if the instantiated pattern is in the data base. But the pattern needn't be explicitly in the data base as an assertion. It can be an implicit assertion implied by a rule. For example, the query

$\langle \text{conclusion} \rangle$  がパターンであり  $\langle \text{body} \rangle$  が任意のクエリです。<sup>126</sup> ルールは大きな(例え無限でも)アサーションの集合を表現するものとして考えることができます。即ち、ルールのボディを満たす変数の割り当てを用いたルールの結果の全てのインスタンスです。簡単なクエリ(パターン)を説明した時、変数への割り当ては、インスタンス化されたパターンがデータベース内に存在する場合にパターンが満たされると説明しました。しかし、パターンは明示的にアサーションとしてデータベース内に存在する必要はありません。ルールにより暗示される暗黙的なアサーションに成り得ます。例えば、以下のクエリは、

```
(lives-near ?x (Bitdiddle Ben))
```

results in

次の結果を生みます。

```
(lives-near (Reasoner Louis) (Bitdiddle Ben))
(lives-near (Aull DeWitt) (Bitdiddle Ben))
```

To find all computer programmers who live near Ben Bitdiddle, we can ask Ben Bitdiddle の近くに住む全てのコンピュータプログラマを見つけるためには、以下のように質問することができます。

```
(and (job ?x (computer programmer))
      (lives-near ?x (Bitdiddle Ben)))
```

As in the case of compound procedures, rules can be used as parts of other rules (as we saw with the `lives-near` rule above) or even be defined recursively. For instance, the rule

---

<sup>125</sup>We will also allow rules without bodies, as in `same`, and we will interpret such a rule to mean that the rule conclusion is satisfied by any values of the variables.

<sup>126</sup>私達は `same` の様にボディの無いルールも認めます。またそのようなルールは、ルールの結論 (conclusion) が任意の変数の値により満たされたことを意味すると解釈します。

複合手続の場合と同様に、ルールは他のルールの一部分として(上記の `lives-near` ルールで見たように) 使用可能です。または再帰的に定義することさえもできます。例として、以下のルールは、

```
(rule (outranked-by ?staff-person ?boss)
      (or (supervisor ?staff-person ?boss)
           (and (supervisor ?staff-person ?middle-manager)
                 (outranked-by ?middle-manager ?boss))))
```

says that a staff person is outranked by a boss in the organization if the boss is the person's supervisor or (recursively) if the person's supervisor is outranked by the boss.

もしボスがスタッフの上司であるか、(再帰的に) スタッフの上司よりボスが上役(outranked)であるならばボスはスタッフより地位が上であると言えます。

**Exercise 4.57:** Define a rule that says that person 1 can replace person 2 if either person 1 does the same job as person 2 or someone who does person 1's job can also do person 2's job, and if person 1 and person 2 are not the same person. Using your rule, give queries that find the following:

人(person) その1が人その2を置き換えるとは人その1が人その2と同じ仕事をしているか、または第三者(someone)が人その1と同じ仕事をしつつ、かつ人その2の仕事を行え、そして人その1と人その2が異なる人である場合であると述べるルールを定義せよ。そのルールを用いて以下の条件を見つけるクエリを与えよ。

a all people who can replace Cy D. Fect;

Cy D. Fect を置き換える全ての人

b all people who can replace someone who is being paid more than they are, together with the two salaries.

自分より給料の高い誰かを置き換える全ての人を二人の給料と一緒に。

**Exercise 4.58:** Define a rule that says that a person is a “big shot” in a division if the person works in the division but does not have a supervisor who works in the division.

ある人が自分が働いている同じ部署に上司(監督者)がいない場合にその人を“big shot”(有力者)であると述べるルールを定義せよ。

**Exercise 4.59:** Ben Bitdiddle has missed one meeting too many. Fearing that his habit of forgetting meetings could cost him his job, Ben decides to do something about it. He adds all the weekly meetings of the firm to the Microshaft data base by asserting the following:

Ben Bitdiddle はある会議を何度も欠席してしまった。彼の会議を忘れる癖は仕事を失う恐れがある。Ben は何かしなければならないと決心した。彼は会社の週次ミーティング全てを Microshaft データベースに以下のアサーションとして加えた。

```
(meeting accounting (Monday 9am))  
(meeting administration (Monday 10am))  
(meeting computer (Wednesday 3pm))  
(meeting administration (Friday 1pm))
```

Each of the above assertions is for a meeting of an entire division. Ben also adds an entry for the company-wide meeting that spans all the divisions. All of the company's employees attend this meeting. 各アサーションは部門の全体ミーティングのためのものだ。Ben はまた全ての部門に渡る全社会議のエントリを追加した。会社の全従業員がこの会議に参加する。

```
(meeting whole-company (Wednesday 4pm))
```

- a On Friday morning, Ben wants to query the data base for all the meetings that occur that day. What query should he use?  
金曜の朝に、Ben はその日の全ての会議をデータベースからクエリしたいと思った。彼のクエリはどのような物になるか?
- b Alyssa P. Hacker is unimpressed. She thinks it would be much more useful to be able to ask for her meetings by specifying her name. So she designs a rule that says that a person's meetings include all **whole-company** meetings plus all meetings of that person's division. Fill in the body of Alyssa's rule.

Alyssa P. Hacker は感心しなかった。彼女は自分の名前を指定することで彼女の会議を尋ねることができればより便利になるだろうと考えた。そのため彼女はある人の会議は全ての whole-company(全社) 会議に加えてその人の部門会議を全て含むと言うルールを設計した。Alyssa のルールのボディを埋めよ。

```
(rule (meeting-time ?person ?day-and-time)
      ⟨rule-body⟩)
```

- c Alyssa arrives at work on Wednesday morning and wonders what meetings she has to attend that day. Having defined the above rule, what query should she make to find this out?

Alyssa は水曜の朝に仕事場に到着し、その日に何の会議があるかについて考えた。上記のルールを定義した上で、彼女のがこのことを見つけるためにはどのようなクエリを行うべきか？

**Exercise 4.60:** By giving the query

以下のクエリを与えることにより、

```
(lives-near ?person (Hacker Alyssa P))
```

Alyssa P. Hacker is able to find people who live near her, with whom she can ride to work. On the other hand, when she tries to find all pairs of people who live near each other by querying

Alyssa P. Hacker は仕事場に相乗りできる、彼女の近所に住む人を見つけることができる。一方で、お互いが近所に住んでいる全ての人々のペアを見つける場合には以下のクエリを用いる。

```
(lives-near ?person-1 ?person-2)
```

she notices that each pair of people who live near each other is listed twice; for example,

彼女はお互いに近所に住んでいる人々の各ペアが二度づつ挙げられていることに気付いた。例えば、

```
(lives-near (Hacker Alyssa P) (Fect Cy D))
(lives-near (Fect Cy D) (Hacker Alyssa P))
```

Why does this happen? Is there a way to find a list of people who live near each other, in which each pair appears only once? Explain.

なぜこれが起こるのか? お互いに近くに住んでいる人々のリストを各ペアが一度しか現れないように見つける方法は存在するか? 説明せよ。

## Logic as programs

We can regard a rule as a kind of logical implication: *If* an assignment of values to pattern variables satisfies the body, *then* it satisfies the conclusion. Consequently, we can regard the query language as having the ability to perform *logical deductions* based upon the rules. As an example, consider the `append` operation described at the beginning of Section 4.4. As we said, `append` can be characterized by the following two rules:

ルールを論理的意味合いの一種であると見做すことができます。もしパターン変数に対する値の割り当てがボディを満たす場合、それならば結論を満たします。必然的に、クエリ言語はルールを基にした*logical deductions*(論理的推理)を実行する能力を有すると見做すことができます。例として、Section 4.4の始めに説明した `append` 命令について考えてみましょう。既に述べたように、`append` は以下の 2 つのルールにて特徴づけられます。

- For any list  $y$ , the empty list and  $y$  append to form  $y$ .  
任意のリスト  $y$  に対し、空リストと  $y$  の `append` は  $y$  を形成する。
- For any  $u$ ,  $v$ ,  $y$ , and  $z$ ,  $(\text{cons } u \ v)$  and  $y$  append to form  $(\text{cons } u \ z)$  if  $v$  and  $y$  append to form  $z$ .  
任意の  $u$ ,  $v$ ,  $y$ ,  $z$  に対し、 $(\text{cons } u \ v)$  と  $y$  の `append` はもし  $v$  と  $y$  `append` が  $z$  を形成する場合、 $(\text{cons } u \ z)$  を形成する。

To express this in our query language, we define two rules for a relation  
これを私達のクエリ言語で表現するために、以下の関係に対する 2 つのルールを定義します。

`(append-to-form x y z)`

which we can interpret to mean “ $x$  and  $y$  append to form  $z$ ”:

上の関係は“ $x$  と  $y$  の `append` は  $z$  を形成する”ことを意味すると解釈できます。

```
(rule (append-to-form () ?y ?y))  
(rule (append-to-form (?u . ?v) ?y (?u . ?z))  
      (append-to-form ?v ?y ?z))
```

The first rule has no body, which means that the conclusion holds for any value of `?y`. Note how the second rule makes use of dotted-tail notation to name the `car` and `cdr` of a list.

最初のルールにはボディがありません。これは結果部分が`?y`の任意の値を保持することを意味します。2つ目のルールがどのようにドット付き末尾記述をリストの `car` と `cdr` に名前を付けるために使用しているかについて注意して下さい。

Given these two rules, we can formulate queries that compute the `append` of two lists:

これら2つのルールを与えられることで、2つのリストに対する `append` を求めるクエリを定式化することができます。

```
; ; ; Query input:  
(append-to-form (a b) (c d) ?z)  
; ; ; Query results:  
(append-to-form (a b) (c d) (a b c d))
```

What is more striking, we can use the same rules to ask the question “Which list, when appended to `(a b)`, yields `(a b c d)`?” This is done as follows:

より印象的なのは、同じルールを“(a b)に対し `append` したら(a b c d)になるリストは何”という質問に使用できることです。これは以下のように行われます。

```
; ; ; Query input:  
(append-to-form (a b) ?y (a b c d))  
; ; ; Query results:  
(append-to-form (a b) (c d) (a b c d))
```

We can also ask for all pairs of lists that `append` to form `(a b c d)`:

`append` すると `(a b c d)` を形成する全てのリストのペアを尋ねることも可能です。

```
; ; ; Query input:  
(append-to-form ?x ?y (a b c d))  
; ; ; Query results:
```

```
(append-to-form () (a b c d) (a b c d))
(append-to-form (a) (b c d) (a b c d))
(append-to-form (a b) (c d) (a b c d))
(append-to-form (a b c) (d) (a b c d))
(append-to-form (a b c d) () (a b c d))
```

The query system may seem to exhibit quite a bit of intelligence in using the rules to deduce the answers to the queries above. Actually, as we will see in the next section, the system is following a well-determined algorithm in unraveling the rules. Unfortunately, although the system works impressively in the `append` case, the general methods may break down in more complex cases, as we will see in [Section 4.4.3](#).

上記のクエリに対する答を推論するルールを用いることにおいておいて、クエリシステムはかなりの知性を示すように見えるかもしれません。実際には次の節で学ぶように、システムはルールをときほぐす明確なアルゴリズムに従っているに過ぎません。残念ながら、システムが `append` の場合では見事な程うまく行きますが、一般的な手法はより複雑な場合に分解されるかもしれません。このことは[Section 4.4.3](#)で学びます。

**Exercise 4.61:** The following rules implement a `next-to` relation that finds adjacent elements of a list:

以下のルールはリストの直前の要素を見つける関係 `next-to` を実装する。

```
(rule (?x next-to ?y in (?x ?y . ?u)))
(rule (?x next-to ?y in (?v . ?z))
      (?x next-to ?y in ?z))
```

What will the response be to the following queries?

以下のクエリの結果を答えよ。

```
(?x next-to ?y in (1 (2 3) 4))
(?x next-to 1 in (2 1 3 1))
```

**Exercise 4.62:** Define rules to implement the `last-pair` operation of [Exercise 2.17](#), which returns a list containing the last element of a nonempty list. Check your rules on queries such as `(last-pair (3) ?x)`, `(last-pair (1 2 3) ?x)` and `(last-pair (2 ?x) (3))`.

Do your rules work correctly on queries such as (last-pair ?x (3)) ?

Exercise 2.17 の last-pair 命令を実装するルールを定義せよ。これは空ではないリストの最後の要素を含むリストを返す。あなたのルールを (last-pair (3) ?x), (last-pair (1 2 3) ?x), (last-pair (2 ?x) (3)) のようなクエリにて確認せよ。あなたのルールは (last-pair ?x (3)) の様なクエリに対し正しく動作するだろうか?

**Exercise 4.63:** The following data base (see Genesis 4) traces the genealogy of the descendants of Ada back to Adam, by way of Cain:

以下のデータベース(創世記第4章を参照せよ)は Ada の子孫の家系を Cain を経由して Adam まで戻りながら辿っている。

```
(son Adam Cain)
(son Cain Enoch)
(son Enoch Irad)
(son Irad Mehujael)
(son Mehujael Methushael)
(son Methushael Lamech)
(wife Lamech Ada)
(son Ada Jabal)
(son Ada Jubal)
```

Formulate rules such as “If  $S$  is the son of  $f$ , and  $f$  is the son of  $G$ , then  $S$  is the grandson of  $G$ ” and “If  $W$  is the wife of  $M$ , and  $S$  is the son of  $W$ , then  $S$  is the son of  $M$ ” (which was supposedly more true in biblical times than today) that will enable the query system to find the grandson of Cain; the sons of Lamech; the grandsons of Methushael. (See Exercise 4.69 for some rules to deduce more complicated relationships.)

“もし  $S$  が  $f$  の息子であり、かつ、 $f$  が  $G$  の息子ならば、 $S$  は  $G$  の孫である”と“もし  $W$  が  $M$  の妻であり、かつ、 $S$  が  $W$  の息子ならば、 $S$  は  $M$  の息子である”(これは恐らく今日より聖書の時代にはより正確であっただろう)のルールを定式化せよ。これらはクエリシステムに対し Cain の孫、Lamech の息子、Methushael の孫

を見つけることを可能にする。(より複雑な関係を推論するいくつかのルールについてはExercise 4.69を参照せよ。)

#### 4.4.2 How the Query System Works

In Section 4.4.4 we will present an implementation of the query interpreter as a collection of procedures. In this section we give an overview that explains the general structure of the system independent of low-level implementation details. After describing the implementation of the interpreter, we will be in a position to understand some of its limitations and some of the subtle ways in which the query language's logical operations differ from the operations of mathematical logic.

Section 4.4.4ではクエリインタプリタを手続の集合として紹介します。この節では低レベルの実装上の詳細からは独立したシステムの一般的な構造について説明する概観を与えます。インタプリタの実装を説明した後に、私達はインタプリタのいくつかの限界と記号論理学の演算とは異なるクエリ言語の論理演算のいくつかの微妙な行い方を理解できる位置に辿り着きます。

It should be apparent that the query evaluator must perform some kind of search in order to match queries against facts and rules in the data base. One way to do this would be to implement the query system as a nondeterministic program, using the `amb` evaluator of Section 4.3 (see Exercise 4.78). Another possibility is to manage the search with the aid of streams. Our implementation follows this second approach.

クエリ評価機がクエリをデータベース内の事実とルールに対してマッチさせるためにある種の探索を実行せねばならないことは明らかでしょう。これを行う1つの方法はクエリシステムをSection 4.3の`amb`評価機を用いて非決定性プログラムとして実装することになります(Exercise 4.78参照)。別の可能性にはストリームの助けを用いて探索を管理する方法があります。私達の実装はこの2つ目のアプローチに従います。

The query system is organized around two central operations called *pattern matching* and *unification*. We first describe pattern matching and explain how this operation, together with the organization of information in terms of streams of frames, enables us to implement both simple and compound queries. We next discuss unification, a generalization of pattern matching needed to implement rules. Finally, we show how the entire query interpreter fits together

through a procedure that classifies expressions in a manner analogous to the way `eval` classifies expressions for the interpreter described in Section 4.1.

クエリシステムは2つの中心となる演算、*pattern matching*(パターンマッチング)と*unification*(ユニフィケーション、单一化)の周りに体系化されます。最初にパターンマッチングについて記述し、この演算がフレームのストリームを用いた情報体系と共にどのように単純クエリと複合クエリの両方を実装可能にするのか説明します。次に私達はユニフィケーション、つまりルールを実装ために必要なパターンマッチングの一般化について議論します。最後に、Section 4.1で説明されたインタプリタのために`eval`が式を分類する方法と同様の方法で、式を分類する手続を通してクエリインタプリタ全体がどのように組み合わされるかについて示します。

## Pattern matching

A *pattern matcher* is a program that tests whether some datum fits a specified pattern. For example, the data list  $((a\ b)\ c\ (a\ b))$  matches the pattern  $(?x\ c\ ?x)$  with the pattern variable  $?x$  bound to  $(a\ b)$ . The same data list matches the pattern  $(?x\ ?y\ ?z)$  with  $?x$  and  $?z$  both bound to  $(a\ b)$  and  $?y$  bound to  $c$ . It also matches the pattern  $((?x\ ?y)\ c\ (?x\ ?y))$  with  $?x$  bound to  $a$  and  $?y$  bound to  $b$ . However, it does not match the pattern  $(?x\ a\ ?y)$ , since that pattern specifies a list whose second element is the symbol  $a$ .

*pattern matcher*(パターンマッチャ)はあるデータが指定されたパターンに適合するかどうかを試すプログラムです。例えばデータリスト  $((a\ b)\ c\ (a\ b))$  はパターン  $(?x\ c\ ?x)$  に対しパターン変数  $?x$  が  $(a\ b)$  に束縛されることで適合します。同じデータリストがパターン  $(?x\ ?y\ ?z)$  に対し  $?x$  と  $?z$  の両者が  $(a\ b)$  に束縛され、 $?y$  が  $c$  に束縛されることで適合します。これはまたパターン  $((?x\ ?y)\ c\ (?x\ ?y))$  に対しても  $?x$  が  $a$  に、 $?y$  が  $b$  に束縛されることで適合します。しかし、これはパターン  $(?x\ a\ ?y)$  には適合しません。このパターンが2つ目の要素がシンボル  $a$  であるリストを指定しているためです。

The pattern matcher used by the query system takes as inputs a pattern, a datum, and a *frame* that specifies bindings for various pattern variables. It checks whether the datum matches the pattern in a way that is consistent with the bindings already in the frame. If so, it returns the given frame augmented by any bindings that may have been determined by the match. Otherwise, it indicates that the match has failed.

パターンマッチャはクエリシステムにより使用されます。クエリシステム

は入力としてパターン、データ、*frame*(フレーム)を取ります。フレームはさまざまなパターン変数に対する束縛を指定します。パターンマッチャはデータがフレームに既に存在する束縛と一致する状態でパターンに適合するかどうかをチェックします。もしそうであれば、その適合により決定された任意の束縛を増やしたフレームを返します。そうでなければ、適合が失敗したことを示します。

For example, using the pattern  $(?x ?y ?x)$  to match  $(a b a)$  given an empty frame will return a frame specifying that  $?x$  is bound to  $a$  and  $?y$  is bound to  $b$ . Trying the match with the same pattern, the same datum, and a frame specifying that  $?y$  is bound to  $a$  will fail. Trying the match with the same pattern, the same datum, and a frame in which  $?y$  is bound to  $b$  and  $?x$  is unbound will return the given frame augmented by a binding of  $?x$  to  $a$ .

例えば、パターン  $(?x ?y ?x)$  を用いて  $(a b a)$  に空のフレームを与えた場合に適合を行うと  $?x$  が  $a$  に、 $?y$  が  $b$  に束縛されることを指定するフレームを返します。同じパターン、同じデータで  $?y$  が  $a$  に束縛されていると指定するフレームを用いて適合を行うと失敗します。同じパターン、同じデータで  $?y$  が  $b$  に束縛され  $?x$  が未束縛であるフレームを用いて適合を行えば与えられたフレームに  $?x$  の  $a$  への束縛を増やした物が返されます。

The pattern matcher is all the mechanism that is needed to process simple queries that don't involve rules. For instance, to process the query

パターンマッチャはルールを含まない単純なクエリを処理するのに必要な仕組みの全てです。例えば、以下のクエリを処理する場合、

```
(job ?x (computer programmer))
```

we scan through all assertions in the data base and select those that match the pattern with respect to an initially empty frame. For each match we find, we use the frame returned by the match to instantiate the pattern with a value for  $?x$ .

データベース内の全てのアサーションを探査し、最初は空のフレームを考慮してパターンに適合する物を選択します。探索を行った各適合に対して、適合により返されたフレームを用いてパターンを  $?x$  の値と共にインスタンス化します。

## Streams of frames

The testing of patterns against frames is organized through the use of streams. Given a single frame, the matching process runs through the data-

base entries one by one. For each data-base entry, the matcher generates either a special symbol indicating that the match has failed or an extension to the frame. The results for all the data-base entries are collected into a stream, which is passed through a filter to weed out the failures. The result is a stream of all the frames that extend the given frame via a match to some assertion in the data base.<sup>127</sup>

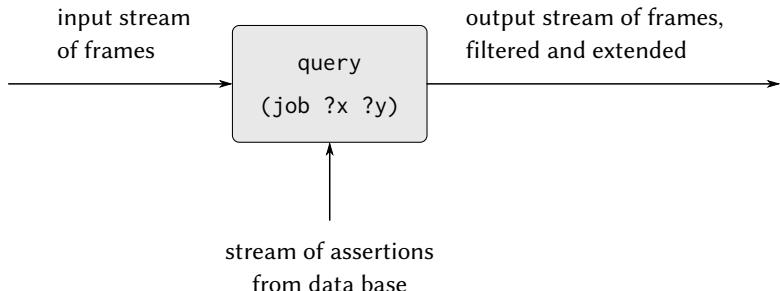
フレームに対してパターンのテストを行うことはストリームの使用を通して体系化されています。単一のフレームを与えられて、マッチング処理はデータベースのエントリを1つづつ通して実行します。各データベースエントリに対して、マッチャは適合が失敗したことを示す特別なシンボルか、フレームに対する拡張を生成します。全てのデータベースエントリに対する結果はストリーム内に集められ、フィルタを通すことで失敗が取り除かれます。結果は与えられたフレームを適合を通すことでデータベース内のあるアサーションに拡張した全てのフレームのストリームです。<sup>128</sup>

In our system, a query takes an input stream of frames and performs the above matching operation for every frame in the stream, as indicated in Figure 4.4. That is, for each frame in the input stream, the query generates a new stream consisting of all extensions to that frame by matches to assertions in the data base. All these streams are then combined to form one huge stream, which contains all possible extensions of every frame in the input stream. This stream is the output of the query.

---

<sup>127</sup>Because matching is generally very expensive, we would like to avoid applying the full matcher to every element of the data base. This is usually arranged by breaking up the process into a fast, coarse match and the final match. The coarse match filters the data base to produce a small set of candidates for the final match. With care, we can arrange our data base so that some of the work of coarse matching can be done when the data base is constructed rather than when we want to select the candidates. This is called *indexing* the data base. There is a vast technology built around database-indexing schemes. Our implementation, described in Section 4.4.4, contains a simple-minded form of such an optimization.

<sup>128</sup>マッチングは一般的にとても重いので、完全なマッチャをデータベースの全ての要素に対して適用することは防ぎたいと考えます。これは通常は高速で粗い適合と最終適合の部品に分解することで準備します。粗い適合はデータベースをフィルタし、最終適合のための候補の小さな集合を生成します。手間をかけて、粗い適合のいくつかの成果がデータベースが候補を選択したい時ではなく、構築された時に使えるようにデータベースを事前に準備することができます。データベースの索引の仕組みの周りには莫大な技術が構築されています。私達の実装はSection 4.4.4で説明されているように、そのような最適化のあまり賢くはない形態を含んでいます。



**Figure 4.4:** A query processes a stream of frames.

私達のシステムではFigure 4.4で示されるように、クエリはフレームの入力ストリームを取り、ストリーム内の各フレームに対して上記のマッチング処理を実行します。言い換えれば、入力ストリーム内の各フレームに対して、クエリはデータベース内のアサーションに対する適合による、全てのフレームの拡張から成る新しいストリームを生成します。これらのストリームの全ては次に組み合わされて1つの大きなストリームを形成します。これは入力ストリーム内の各フレームの全ての可能な拡張を含んでいます。このストリームがクエリの出力です。

To answer a simple query, we use the query with an input stream consisting of a single empty frame. The resulting output stream contains all extensions to the empty frame (that is, all answers to our query). This stream of frames is then used to generate a stream of copies of the original query pattern with the variables instantiated by the values in each frame, and this is the stream that is finally printed.

単純なクエリに答えるためにはクエリを单一の空フレームから成る入力ストリームと共に用います。結果としての出力ストリームは空にフレームに対する全ての拡張を含んでいます(言い換えれば、クエリに対する全ての答を含みます)。このフレームのストリームは次に、元々のクエリのパターンと各フレーム内の値でインスタンス化された変数のコピーのストリームを生成するのに利用されます。そしてこれが最終的に表示されるストリームです。

## Compound queries

The real elegance of the stream-of-frames implementation is evident when we deal with compound queries. The processing of compound queries makes use of the ability of our matcher to demand that a match be consistent with a specified frame. For example, to handle the `and` of two queries, such as

フレームのストリーム実装の真に優雅な点は複合クエリを扱う時に明白になります。複合クエリの処理は適合の結果が指定されたフレームに一致するという私達のマッチャが要求する能力を利用します。例えば、2つのクエリの `and` を取り扱う以下のようないくエリでは

```
(and (can-do-job ?x (computer programmer trainee))
       (job ?person ?x))
```

(informally, “Find all people who can do the job of a computer programmer trainee”), we first find all entries that match the pattern

(簡単に言えば、“コンピュータプログラマ見習いの職を行える全ての人を見つけろ”) まず以下のパターンに適合する全てのエントリを見つけます。

```
(can-do-job ?x (computer programmer trainee))
```

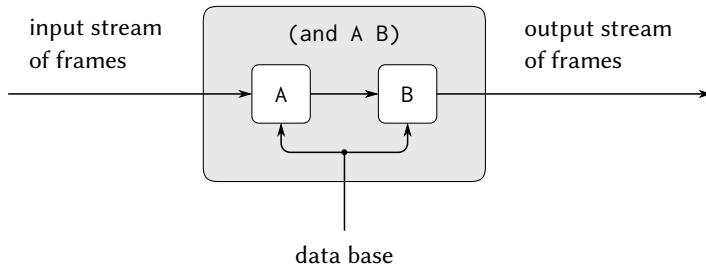
This produces a stream of frames, each of which contains a binding for `?x`. Then for each frame in the stream we find all entries that match

これはフレームのストリームを生成します。各フレームは`?x`に対する束縛を含んでいます。次にストリーム内の各フレームに対し、与えられた`?x`に対する束縛に一致する様に、以下のパターンに適合する全てのエントリを探します。

```
(job ?person ?x)
```

in a way that is consistent with the given binding for `?x`. Each such match will produce a frame containing bindings for `?x` and `?person`. The `and` of two queries can be viewed as a series combination of the two component queries, as shown in [Figure 4.5](#). The frames that pass through the first query filter are filtered and further extended by the second query.

そのような適合のそれぞれは`?x`と`?person`に対する束縛を含むフレームを生成します。2つのクエリの `and` は [Figure 4.5](#) に示されるように、一連の2つのクエリのコンポーネントの組み合わせであると見做すことができます。最初のクエリフィルタを通過するフレームはフィルタをかけられ、2つ目のクエリにてさらに拡張されます。



**Figure 4.5:** The **and** combination of two queries is produced by operating on the stream of frames in series.

Figure 4.6 shows the analogous method for computing the **or** of two queries as a parallel combination of the two component queries. The input stream of frames is extended separately by each query. The two resulting streams are then merged to produce the final output stream.

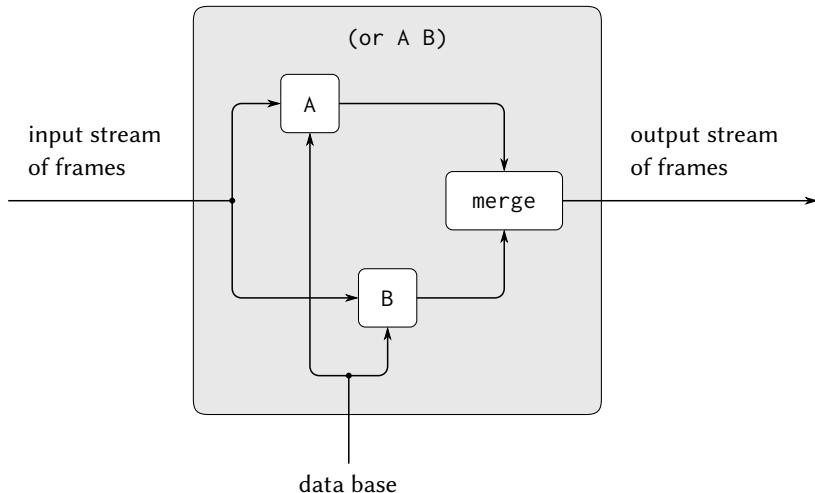
Figure 4.6は2つのクエリの**or**を2つのクエリコンポーネントの並列な組み合わせとして求めるための類似の手法を示しています。フレームの入力ストリームは各クエリにより別々に拡張されます。2つの結果ストリームは次にマージされ最終の出力ストリームを生成します。

Even from this high-level description, it is apparent that the processing of compound queries can be slow. For example, since a query may produce more than one output frame for each input frame, and each query in an **and** gets its input frames from the previous query, an **and** query could, in the worst case, have to perform a number of matches that is exponential in the number of queries (see Exercise 4.76).<sup>129</sup> Though systems for handling only simple queries are quite practical, dealing with complex queries is extremely difficult.<sup>130</sup>

この高いレベルの記述からでも複合クエリの処理が遅くなることがはつきりとわかります。例えば、クエリは各入力フレームに対して複数の出力ストリームを生成するかもしれません。そして各クエリも同様です。最悪の場合に

<sup>129</sup>But this kind of exponential explosion is not common in **and** queries because the added conditions tend to reduce rather than expand the number of frames produced.

<sup>130</sup>There is a large literature on data-base-management systems that is concerned with how to handle complex queries efficiently.



**Figure 4.6:** The **or** combination of two queries is produced by operating on the stream of frames in parallel and merging the results.

はクエリ数の指数関数となる多数のマッチングを実行しなければなりません (Exercise 4.76参照)。<sup>131</sup> 単純なクエリのみを扱うシステムのほうがとても実用的ではありますが、複合クエリを扱うことは極めて難しいのです。<sup>132</sup>

From the stream-of-frames viewpoint, the **not** of some query acts as a filter that removes all frames for which the query can be satisfied. For instance, given the pattern

フレームのストリームの視点から、あるクエリの **not** はクエリが満たされる全てのフレームを取り除くフィルタとして働きます。例えば、以下のパターンを与えらえると、

---

<sup>131</sup> しかし、この種の指数関数爆発は **and** クエリでは一般的ではありません。追加された条件が生成されるフレームの数を増やすのではなく、減らす傾向があるためです。

<sup>132</sup> 複合クエリをどのように効率的に扱うかに関連するデータベース管理システムの多数の文献が存在します。

```
(not (job ?x (computer programmer)))
```

we attempt, for each frame in the input stream, to produce extension frames that satisfy `(job ?x (computer programmer))`. We remove from the input stream all frames for which such extensions exist. The result is a stream consisting of only those frames in which the binding for `?x` does not satisfy `(job ?x (computer programmer))`. For example, in processing the query

入力ストリームの各フレームに対して `(job ?x (computer programmer))` を満たす拡張フレームの生成を試みます。入力ストリームからそのような拡張が存在する全てのフレームを削除します。結果はフレーム中の`?x` の束縛が `(job ?x (computer programmer))` を満たさないフレームのみから成るストリームとなります。例えば以下のクエリの処理においては、

```
(and (supervisor ?x ?y)
      (not (job ?x (computer programmer))))
```

the first clause will generate frames with bindings for `?x` and `?y`. The `not` clause will then filter these by removing all frames in which the binding for `?x` satisfies the restriction that `?x` is a computer programmer.<sup>133</sup>

最初の節は`?x` と`?y` に対する束縛を持つフレームを生成します。次に `not` 節はこれらから`?x` に対する束縛が`?x` がコンピュータプログラマであるという制約を満たす全てのフレームを削除することでフィルタリングします。<sup>134</sup>

The `lisp-value` special form is implemented as a similar filter on frame streams. We use each frame in the stream to instantiate any variables in the pattern, then apply the Lisp predicate. We remove from the input stream all frames for which the predicate fails.

`lisp-value` 特殊形式はフレームのストリーム上の同様なフィルタとして実装されます。ストリーム内の各フレームをパターン内の任意の変数をインスタンス化するために用い、そして Lisp 手続を適用します。入力ストリームから述語が失敗する全てのフレームを削除します。

---

<sup>133</sup>There is a subtle difference between this filter implementation of `not` and the usual meaning of `not` in mathematical logic. See [Section 4.4.3](#).

<sup>134</sup>この `not` のフィルタ実装と、記号論理学における通常の意味での `not` の間には微妙な違いが存在します。[Section 4.4.3](#)を参照して下さい。

## Unification

In order to handle rules in the query language, we must be able to find the rules whose conclusions match a given query pattern. Rule conclusions are like assertions except that they can contain variables, so we will need a generalization of pattern matching—called *unification*—in which both the “pattern” and the “datum” may contain variables.

クエリ言語内のルールを扱うために、ルールの結果が与えられたクエリパターンに適合するルールを見付けられねばなりません。ルールの結果はアサーションに似ていますが、変数を含められる所が異なります。そのためパターンマッチングの一般化—*unification*(ユニファイケーション)と呼ばれます—を必要とし、その中で“パターン”と“データ”的両方が変数を持ち得ます。

A unifier takes two patterns, each containing constants and variables, and determines whether it is possible to assign values to the variables that will make the two patterns equal. If so, it returns a frame containing these bindings. For example, unifying  $(?x \ a \ ?y)$  and  $(?y \ ?z \ a)$  will specify a frame in which  $?x$ ,  $?y$ , and  $?z$  must all be bound to  $a$ . On the other hand, unifying  $(?x \ ?y \ a)$  and  $(?x \ b \ ?y)$  will fail, because there is no value for  $?y$  that can make the two patterns equal. (For the second elements of the patterns to be equal,  $?y$  would have to be  $b$ ; however, for the third elements to be equal,  $?y$  would have to be  $a$ .) The unifier used in the query system, like the pattern matcher, takes a frame as input and performs unifications that are consistent with this frame.

ユニファイアは2つの定数と変数を含むパターンを取り、2つのパターンを等しくする変数への値の割り当てが可能であるかどうかを決定します。もし そうであれば、これらの束縛を含むフレームを返します。例えば  $(?x \ a \ ?y)$  と  $(?y \ ?z \ a)$  のユニファイケーションは  $?x$ ,  $?y$ ,  $?z$  が全て  $a$  に束縛されなければならないフレームを指示します。一方で、 $(?x \ ?y \ a)$  と  $(?x \ b \ ?y)$  のユニファイケーションは失敗します。2つのパターンを等しくできる  $?y$  の値が存在しないためです。(両方のパターンの2つ目の要素が等しくなるためには  $?y$  は  $b$  にならなければなりません。しかし、3番目の要素が等しくなるためには  $?y$  が  $a$  になるしかありません)。クエリシステムで用いられるユニファイアはパターンマッチャの様に、フレームを入力として取りこのフレームと一致するユニファイケーションを実行します。

The unification algorithm is the most technically difficult part of the query system. With complex patterns, performing unification may seem to require deduction. To unify  $(?x \ ?x)$  and  $((a \ ?y \ c) \ (a \ b \ ?z))$ , for example, the algorithm must infer that  $?x$  should be  $(a \ b \ c)$ ,  $?y$  should be  $b$ , and  $?z$  should

be  $c$ . We may think of this process as solving a set of equations among the pattern components. In general, these are simultaneous equations, which may require substantial manipulation to solve.<sup>135</sup> For example, unifying  $(?x ?x)$  and  $((a ?y c) (a b ?z))$  may be thought of as specifying the simultaneous equations

ユニファイケーションアルゴリズムはクエリシステムで最も技術的に難しい部分です。複雑なパターンを共なうため、ユニファイケーションの実行は演繹を必要とするように見えるかもしれません。例えば、 $(?x ?x)$  と  $((a ?y c) (a b ?z))$  をユニファイケーションするためにはアルゴリズムは $?x$  は  $(a b c)$  に、 $?y$  は  $b$  に、 $?z$  は  $c$  にならなければいけないことを推論しなければなりません。この処理はパターンコンポーネント間の等式の集合を解くこととして考えることができます。一般的には、これらは連立方程式であり、これを解くためには大量の操作が必要となるでしょう。<sup>136</sup> 例えば、 $(?x ?x)$  と  $((a ?y c) (a b ?z))$  のユニファイケーションは以下の連立方程式を指定することだと考えられるでしょう。

$$\begin{aligned} ?x &= (a ?y c) \\ ?x &= (a b ?z) \end{aligned}$$

These equations imply that

これらの方程式は以下を暗示します。

$$(a ?y c) = (a b ?z)$$

which in turn implies that

これは順に次を暗示します。

$$\begin{aligned} a &= a, \\ ?y &= b, \\ c &= ?z, \end{aligned}$$

and hence that

従って以下の通りです。

$$?x = (a b c)$$

<sup>135</sup>In one-sided pattern matching, all the equations that contain pattern variables are explicit and already solved for the unknown (the pattern variable).

<sup>136</sup>一方向のパターンマッチングでは、全てのパターン変数を含む等式は明白で未知数(パターン変数)について既に解かれています。

In a successful pattern match, all pattern variables become bound, and the values to which they are bound contain only constants. This is also true of all the examples of unification we have seen so far. In general, however, a successful unification may not completely determine the variable values; some variables may remain unbound and others may be bound to values that contain variables.

パターンマッチが成功する場合、全てのパターン変数は束縛され、それらに束縛される値は定数のみを持ちます。これはまたここまで見てきた全てのユニフィケーションの例に対しても真です。しかし一般的に、ユニフィケーションが成功する場合には変数の値が完全には決定されるとは限りません。いくつかの変数は未束縛のままで、他は変数を含む値に束縛されます。

Consider the unification of  $(?x \ a)$  and  $((b \ ?y) \ ?z)$ . We can deduce that  $?x = (b \ ?y)$  and  $a = ?z$ , but we cannot further solve for  $?x$  or  $?y$ . The unification doesn't fail, since it is certainly possible to make the two patterns equal by assigning values to  $?x$  and  $?y$ . Since this match in no way restricts the values  $?y$  can take on, no binding for  $?y$  is put into the result frame. The match does, however, restrict the value of  $?x$ . Whatever value  $?y$  has,  $?x$  must be  $(b \ ?y)$ . A binding of  $?x$  to the pattern  $(b \ ?y)$  is thus put into the frame. If a value for  $?y$  is later determined and added to the frame (by a pattern match or unification that is required to be consistent with this frame), the previously bound  $?x$  will refer to this value.<sup>137</sup>

$(?x \ a)$  と  $((b \ ?y) \ ?z)$  のユニフィケーションについて考えます。 $?x = (b \ ?y)$  であり  $a = ?z$  であると推論できます。しかしそれ以上  $?x$  と  $?y$  について解くことはできません。このユニフィケーションは失敗はしません。確かに 2 つのパターンを  $?x$  と  $?y$  に値を割り当てることで等しくすることは可能なためです。この適合が  $?y$  の取り得る値を全く限定しないため、結果フレームに  $?y$  の束縛は全く入りません。しかしこの適合は  $?x$  の値は限定します。 $?y$  がどのような値を取っても、 $?x$  は必ず  $(b \ ?y)$  になります。従って  $?x$  の  $(b \ ?y)$  への束縛はフレームへ入れられます。もし  $?y$  の値が(パターンマッチ、またはこのフレームに一致する必要のあるユニフィケーションにより)後に決定されフレームに追加されたなら、その前に束縛された  $?x$  はこの値を参照することになります。<sup>138</sup>

---

<sup>137</sup> Another way to think of unification is that it generates the most general pattern that is a specialization of the two input patterns. That is, the unification of  $(?x \ a)$  and  $((b \ ?y) \ ?z)$  is  $((b \ ?y) \ a)$ , and the unification of  $(?x \ a \ ?y)$  and  $(?y \ ?z \ a)$ , discussed above, is  $(a \ a \ a)$ . For our implementation, it is more convenient to think of the result of unification as a frame rather than a pattern.

<sup>138</sup> ユニフィケーションについて考えるもう 1 つの方法は、二つの入力パターンの特殊化

## Applying rules

Unification is the key to the component of the query system that makes inferences from rules. To see how this is accomplished, consider processing a query that involves applying a rule, such as

ユニファイケーションはルールから推論を行わせるクエリシステムのコンポーネントに対する鍵です。これがどのように達成されるかについて学ぶためには、ルールの適用を含むクエリの処理について考えてみましょう。例えば、以下について考えます。

```
(lives-near ?x (Hacker Alyssa P))
```

To process this query, we first use the ordinary pattern-match procedure described above to see if there are any assertions in the data base that match this pattern. (There will not be any in this case, since our data base includes no direct assertions about who lives near whom.) The next step is to attempt to unify the query pattern with the conclusion of each rule. We find that the pattern unifies with the conclusion of the rule

このクエリを処理するためには、最初に通常の上で説明されたパターンマッチ手続を用いてこのパターンに適合するアサーションがデータベース内に存在するかどうかを見ます。(この場合には存在しません。私達のデータベースには誰が誰の近くに住んでいるかについての直接のアサーションが全く含まれていないためです)。次のステップはクエリパターンと各ルールの結果とのユニファイケーションを試みることです。このパターンは以下のルールの結果とユニファイケーションすると、

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
            (address ?person-2 (?town . ?rest-2))
            (not (same ?person-1 ?person-2))))
```

resulting in a frame specifying that `?person-2` is bound to `(Hacker Alyssa P)` and that `?x` should be bound to (have the same value as) `?person-1`. Now, relative to this frame, we evaluate the compound query given by the body of the rule. Successful matches will extend this frame by providing a binding for

---

である最も一般的なパターンを生成するということです。言い換えれば、`(?x a)` と `((b ?y) ?z)` のユニファイケーションは `((b ?y) a)` であり、上で議論した `(?x a ?y)` と `(?y ?z a)` のユニファイケーションは `(a a a)` です。私達の実装に対しては、ユニファイケーションの結果をパターンではなく、フレームとして考えたほうがより便利です。

?person-1, and consequently a value for ?x, which we can use to instantiate the original query pattern.

結果としてフレームに?person-2 が (Hacker Alyssa P) に束縛され、?x が (同じ値として)?person-1 に束縛されなければならないとの指定が入ることを発見します。これで、このフレームに関連して、このルールのボディにより与えられた複合クエリを評価します。適合が成功すればこのフレームは?person-1 に対する束縛を与えることで拡張され、その結果として?x の値も決定し、元々のクエリパターンをインスタンス化するのに利用することができます。

In general, the query evaluator uses the following method to apply a rule when trying to establish a query pattern in a frame that specifies bindings for some of the pattern variables:

一般的に、クエリ評価機は以下の手法を用いて、パターン変数に対する束縛を指定するフレーム内のクエリパターンを定めようとする時に、ルールを適用します。

- Unify the query with the conclusion of the rule to form, if successful, an extension of the original frame.

クエリをルールの結論とユニフィケーションすることで (成功すれば) 元のフレームの拡張を形成する

- Relative to the extended frame, evaluate the query formed by the body of the rule.

拡張されたフレームを参照しながら、ルールのボディにより形成されたクエリを評価する

Notice how similar this is to the method for applying a procedure in the eval/apply evaluator for Lisp:

これがどれほど Lisp の eval/apply 評価機内での手続適用のための手法に似ているかについて注意して下さい。

- Bind the procedure's parameters to its arguments to form a frame that extends the original procedure environment.

手続のパラメタをその引数に束縛することで元々の手続環境を拡張するフレームを形成する

- Relative to the extended environment, evaluate the expression formed by the body of the procedure.

拡張された環境を参照しながら、手続のボディにより形成された式を評価する

The similarity between the two evaluators should come as no surprise. Just as procedure definitions are the means of abstraction in Lisp, rule definitions are the means of abstraction in the query language. In each case, we unwind the abstraction by creating appropriate bindings and evaluating the rule or procedure body relative to these.

2つの評価機の間の類似度は驚くべきことではありません手続定義がLispにおける抽象化の手段であるように、ルール定義はクエリ言語の抽象化の手段です。それぞれの場合において、適切な束縛を作成し、ルール、または手続のボディをこれらの束縛を参照することで抽象化を巻き戻します。

## Simple queries

We saw earlier in this section how to evaluate simple queries in the absence of rules. Now that we have seen how to apply rules, we can describe how to evaluate simple queries by using both rules and assertions.

私達はこの節の始めにルールを欠いた単純なクエリをどのように評価するかについて学びました。今ではルールの適用の仕方も学んだため、単純なクエリをルールとアサーションの両方を用いてどのように評価するかについても説明することができます。

Given the query pattern and a stream of frames, we produce, for each frame in the input stream, two streams:

クエリパターンとフレームのストリームを与えられた時、入力ストリーム内の各フレームに対して2つのストリームを生成します。

- a stream of extended frames obtained by matching the pattern against all assertions in the data base (using the pattern matcher), and  
(パターンマッチャを用いて) データベース内の全てのアサーションに  
対してパターンの適合を行うことにより得られた拡張フレームのスト  
リーム
- a stream of extended frames obtained by applying all possible rules (using the unifier).<sup>139</sup>

<sup>139</sup>Since unification is a generalization of matching, we could simplify the system by using the unifier to produce both streams. Treating the easy case with the simple matcher, however, illustrates how matching (as opposed to full-blown unification) can be useful in its own right.

(ユニファイアを用いて) 全ての可能なルールを適用することにより得られた拡張フレームのストリーム<sup>140</sup>

Appending these two streams produces a stream that consists of all the ways that the given pattern can be satisfied consistent with the original frame. These streams (one for each frame in the input stream) are now all combined to form one large stream, which therefore consists of all the ways that any of the frames in the original input stream can be extended to produce a match with the given pattern.

これらの 2 つのストリームをアペンドすることにより、与えられたパターンを元のフレームに一致して満たすことができる全ての方法により成り立つストリームを生成します。これらのストリーム（入力ストリームの各フレームに対して 1 つ）はこれで全てが 1 つの巨大なストリームに接続されます。従ってこの巨大ストリームは元の入力ストリーム内の任意のフレームを与えられたパターンへの適合を生成するために拡張した全ての方法から成り立っています。

## The query evaluator and the driver loop

Despite the complexity of the underlying matching operations, the system is organized much like an evaluator for any language. The procedure that coordinates the matching operations is called `qeval`, and it plays a role analogous to that of the `eval` procedure for Lisp. `Qeval` takes as inputs a query and a stream of frames. Its output is a stream of frames, corresponding to successful matches to the query pattern, that extend some frame in the input stream, as indicated in Figure 4.4. Like `eval`, `qeval` classifies the different types of expressions (queries) and dispatches to an appropriate procedure for each. There is a procedure for each special form (`and`, `or`, `not`, and `lisp-value`) and one for simple queries.

潜在的なマッチング操作の複雑さに係らず、システムは任意の言語のための評価機とそっくりに体系化されます。マッチング操作を統合する手続は `qeval` と呼ばれ、Lisp の `eval` 手続の役割と同様な役割を演じます。`qeval` は入力としてクエリとフレームのストリームを取ります。その出力はフレームのストリームであり、クエリパターンへの成功したマッチングに相応します。

<sup>140</sup>ユニフィケーションはマッチングの一般化であるため、ユニファイアを用いて両方のストリームを生成することによりシステムを簡略化することができました。しかし、簡単な場合を単純なマッチャで取り扱うことはマッチング（適合）がどのように（本格的なユニフィケーションとは逆に）それ自身の正しさにおいて便利であることを説明します。

これはFigure 4.4で示されるよう入力ストリームのいくつかを拡張しています。`eval` と同様に、`qeval` は異なる型の式(クエリ)を分類し、それぞれに対する適切な手続を呼び出します。各特殊形式(`and`, `or`, `not`, `lisp-value`)に手続が存在し、また単純なクエリにも手続が存在します。

The driver loop, which is analogous to the `driver-loop` procedure for the other evaluators in this chapter, reads queries from the terminal. For each query, it calls `qeval` with the query and a stream that consists of a single empty frame. This will produce the stream of all possible matches (all possible extensions to the empty frame). For each frame in the resulting stream, it instantiates the original query using the values of the variables found in the frame. This stream of instantiated queries is then printed.<sup>141</sup>

この章の他の評価機のための `driver-loop` 手続と同様のドライバループが端末からクエリを読み出します。各クエリに対して、ドライバループは `qeval` をそのクエリと 1 つの空フレームと共に呼び出します。これにより全ての可能な適合(全ての可能な空フレームに対する拡張)のストリームが生成されます。結果としてのストリームの各フレームに対して、ドライバループは元のクエリをフレーム内で見つかった変数の値を用いてインスタンス化します。次にこのインスタンス化されたクエリのストリームは表示されます。<sup>142</sup>

The driver also checks for the special command `assert!`, which signals that the input is not a query but rather an assertion or rule to be added to the data base. For instance,

ドライバはまた特別なコマンド `assert!` をチェックします。これは入力がクエリではなくデータベースに追加するアサーション、またはルールでることを示します。例えば、

```
(assert! (job (Bitdiddle Ben)
                 (computer wizard)))
(assert! (rule (wheel ?person))
```

<sup>141</sup>The reason we use streams (rather than lists) of frames is that the recursive application of rules can generate infinite numbers of values that satisfy a query. The delayed evaluation embodied in streams is crucial here: The system will print responses one by one as they are generated, regardless of whether there are a finite or infinite number of responses.

<sup>142</sup>私達がフレームの(リストではなく)ストリームを使う理由は、ルールの再帰的適用はクエリを満たす無限の数の値を生成することができるからです。ストリームに組込まれた遅延化された評価がここでは重要です。システムは応答を1つづつそれらが生成された順に、有限か無限の数の応答があるかに係らずに表示します。

```
(and (supervisor ?middle-manager ?person)
      (supervisor ?x ?middle-manager))))
```

#### 4.4.3 Is Logic Programming Mathematical Logic?

The means of combination used in the query language may at first seem identical to the operations `and`, `or`, and `not` of mathematical logic, and the application of query-language rules is in fact accomplished through a legitimate method of inference.<sup>143</sup> This identification of the query language with mathematical logic is not really valid, though, because the query language provides a *control structure* that interprets the logical statements procedurally. We can often take advantage of this control structure. For example, to find all of the supervisors of programmers we could formulate a query in either of two logically equivalent forms:

クエリ言語内で使用される組み合わせの手段は最初は記号論理学の `and`, `or`, `not` 命令と同じに見えるかもしれません。実際にクエリ言語のルールの適用は、推論という、まともな手段を通して達成されます。<sup>144</sup> しかし、このクエリ言語の記号論理学を用いた同定は実際には有効ではありません。クエリ言語が論理的な命題を手続的に解釈する *control structure*(制御構造) を提供するためです。私達は頻繁にこの制御構造を活用することができます。例えばプログラマの監督者全てを見るけるためには以下の 2 つの論理的に等価な形式のどちらかをクエリとして策定することができます。

```
(and (job ?x (computer programmer)) (supervisor ?x ?y))
```

or

または

---

<sup>143</sup>That a particular method of inference is legitimate is not a trivial assertion. One must prove that if one starts with true premises, only true conclusions can be derived. The method of inference represented by rule applications is *modus ponens*, the familiar method of inference that says that if *A* is true and *A implies B* is true, then we may conclude that *B* is true.

<sup>144</sup>推論の特定の手段がまともであるということは自明な主張ではありません。もし真となる前提で開始したのであれば、真となる結論のみが導き出されることを証明しなければなりません。ルール適用で表現された推論の手法は *modus ponens*(肯定式) という親しみのある推論の手法であり、もし *A* が真でありかつ *A implies B*(*A* ならば *B*) が真であるならば、*B* は真であると結論づけることができます。

```
(and (supervisor ?x ?y) (job ?x (computer programmer)))
```

If a company has many more supervisors than programmers (the usual case), it is better to use the first form rather than the second because the data base must be scanned for each intermediate result (frame) produced by the first clause of the `and`.

もし会社に(通常の場合として)プログラマより多くの監督者が存在するのであれば、2つ目よりも最初の形式を用いたほうが良いです。なぜならデータベースは `and` の最初の節により生成された中間結果(フレーム)全てに対して探索されねばならないためです。

The aim of logic programming is to provide the programmer with techniques for decomposing a computational problem into two separate problems: “what” is to be computed, and “how” this should be computed. This is accomplished by selecting a subset of the statements of mathematical logic that is powerful enough to be able to describe anything one might want to compute, yet weak enough to have a controllable procedural interpretation. The intention here is that, on the one hand, a program specified in a logic programming language should be an effective program that can be carried out by a computer. Control (“how” to compute) is effected by using the order of evaluation of the language. We should be able to arrange the order of clauses and the order of subgoals within each clause so that the computation is done in an order deemed to be effective and efficient. At the same time, we should be able to view the result of the computation (“what” to compute) as a simple consequence of the laws of logic.

論理プログラミングの目的はプログラマに演算問題を2つの分離された問題、“何”が求めらるべきかと“どのように”これが求められるべきかに分解する技術を与えることです。これは記号論理学の命題の部分集合を選択することで達成されます。これは人が演算したい対象全てを記述するのに十分に強く、けれども制御可能な手続的解釈を行うに十分に弱い物です。一方で、ここで意図は論理プログラミング言語で指示されたプログラムは計算機により実行され得る実効的なプログラムでなければなりません。制御 (“どのように”演算するか) は言語の評価順の使用に影響を受けます。私達は節の順と各節の中の下位目標の順とを操作し、演算が実効的、かつ効率的であると考えられる順で行われるようにせねばなりません。

Our query language can be regarded as just such a procedurally interpretable subset of mathematical logic. An assertion represents a simple fact (an atomic proposition). A rule represents the implication that the rule con-

clusion holds for those cases where the rule body holds. A rule has a natural procedural interpretation: To establish the conclusion of the rule, establish the body of the rule. Rules, therefore, specify computations. However, because rules can also be regarded as statements of mathematical logic, we can justify any “inference” accomplished by a logic program by asserting that the same result could be obtained by working entirely within mathematical logic.<sup>145</sup>

私達のクエリ言語は単なるそのような手続的に解釈可能な記号論理学の部分集合であると見做すことができます。アサーションは単純な事実（アトミックな命題）を表現します。ルールはルールのボディが持つ複数の場合に対する、ルールの結論が持つ推測の結果を表現します。ルールは自然な手続的解釈を持ちます。ルールの結論を成立させるためには、ルールのボディを定めます。従って、ルールは演算を提示しています。しかし、ルールはまた記号論理学の命題であるとも見做すことができるため、同じ結果が全体的に記号論理学の中で働くことにより得られることを主張することで、論理プログラムにより遂行された任意の“推論”を正当化することができます。<sup>146</sup>

---

<sup>145</sup>We must qualify this statement by agreeing that, in speaking of the “inference” accomplished by a logic program, we assume that the computation terminates. Unfortunately, even this qualified statement is false for our implementation of the query language (and also false for programs in Prolog and most other current logic programming languages) because of our use of `not` and `lisp-value`. As we will describe below, the `not` implemented in the query language is not always consistent with the `not` of mathematical logic, and `lisp-value` introduces additional complications. We could implement a language consistent with mathematical logic by simply removing `not` and `lisp-value` from the language and agreeing to write programs using only simple queries, `and`, and `or`. However, this would greatly restrict the expressive power of the language. One of the major concerns of research in logic programming is to find ways to achieve more consistency with mathematical logic without unduly sacrificing expressive power.

<sup>146</sup>私達はこの命題を以下に同意することで制限しなければなりません。“推論”が論理プログラムにより正当化されるに言及するにおいて、私達は演算が停止することを前提としています。残念なことに、例えこの制限された命題もクエリ言語の私達の実装においては正しくありません。（そして同時に Prolog のプログラムにとどめても、そして他のほとんどの現在の論理プログラミング言語においてもこれは正しくありません）。原因は私達の `not` と `lisp-value` の使用のためです。この先で議論するように、クエリ言語で実装された `not` は常に記号論理学の `not` と一致しません。そして `lisp-value` は複雑さを増します。私達は単純に `not` と `lisp-value` を言語から削除し、プログラムを単純なクエリ、`and`, `or` のみを用いて書くことに同意することで、記号論理学と一致する言語を実装することができます。しかし、これは言語の表現力を大きく制限してしまいます。論理プログラミングにおける主要な研究課題の1つは過度に表現力を犠牲にすることなく、

## Infinite loops

A consequence of the procedural interpretation of logic programs is that it is possible to construct hopelessly inefficient programs for solving certain problems. An extreme case of inefficiency occurs when the system falls into infinite loops in making deductions. As a simple example, suppose we are setting up a data base of famous marriages, including

論理プログラムの手続的な解釈の結果は絶望的に非効率なプログラムを一部の問題に対して構築することが有り得ることです。極端に非効率な場合にはシステムは演繹を行う無限ループに落ち込んでしまいます。簡単な例として、縁組のデータベースを構築したと考えてみましょう。以下を含みます。

```
(assert! (married Minnie Mickey))
```

If we now ask

ここで以下を尋ねた場合、

```
(married Mickey ?who)
```

we will get no response, because the system doesn't know that if *A* is married to *B*, then *B* is married to *A*. So we assert the rule

応答は有りません。なぜならシステムはもし *A* が *B* に結婚した場合、*B* が *A* に結婚することになることを知らないためです。そのため以下のルールを宣言します。

```
(assert! (rule (married ?x ?y) (married ?y ?x)))
```

and again query

そして再び質問します。

```
(married Mickey ?who)
```

Unfortunately, this will drive the system into an infinite loop, as follows:

残念ながら、これはシステムを無限ループに追いやります。以下のとおりです。

- The system finds that the `married` rule is applicable; that is, the rule conclusion `(married ?x ?y)` successfully unifies with the query pattern `(married Mickey ?who)` to produce a frame in which `?x` is bound to `Mickey` and `?y` is bound to `?who`. So the interpreter proceeds to evaluate

---

記号論理学とより一致する方法を見つけることです。

the rule body (`(married ?y ?x)`) in this frame—in effect, to process the query (`(married ?who Mickey)`).

システムは `married` ルールが適用可能であることを見つけます。言い換えれば、ルールの結論 (`(married ?x ?y)`) は成功裏にクエリパターン (`(married Mickey ?who)`) と单一化し、`?x` が `Mickey` に、`?y` が `?who` に束縛されるフレームを生成します。

- One answer appears directly as an assertion in the data base: (`(married Minnie Mickey)`).  
1つの答は直接データベース内のアサーションとして現れます: (`(married Minnie Mickey)`)
- The `married` rule is also applicable, so the interpreter again evaluates the rule body, which this time is equivalent to (`(married Mickey ?who)`).  
`married` ルールもまた適用可能です。そのためインタプリタは再度ルールのボディを評価し、今回は (`(married Mickey ?who)`) に等しくなります。

The system is now in an infinite loop. Indeed, whether the system will find the simple answer (`(married Minnie Mickey)`) before it goes into the loop depends on implementation details concerning the order in which the system checks the items in the data base. This is a very simple example of the kinds of loops that can occur. Collections of interrelated rules can lead to loops that are much harder to anticipate, and the appearance of a loop can depend on the order of clauses in an `and` (see [Exercise 4.64](#)) or on low-level details concerning the order in which the system processes queries.<sup>147</sup>

これでシステムは無限ループの中です。実際に、システムが簡単な答、(`(married Minnie Mickey)`) をループに入る前に見つけるかどうかは、システム

---

<sup>147</sup>This is not a problem of the logic but one of the procedural interpretation of the logic provided by our interpreter. We could write an interpreter that would not fall into a loop here. For example, we could enumerate all the proofs derivable from our assertions and our rules in a breadth-first rather than a depth-first order. However, such a system makes it more difficult to take advantage of the order of deductions in our programs. One attempt to build sophisticated control into such a program is described in [deKleer et al. 1977](#). Another technique, which does not lead to such serious control problems, is to put in special knowledge, such as detectors for particular kinds of loops ([Exercise 4.67](#)). However, there can be no general scheme for reliably preventing a system from going down infinite paths in performing deductions. Imagine a diabolical rule of the form “To show  $P(x)$  is true, show that  $P(f(x))$  is true,” for some suitably chosen function  $f$ .

がデータベース内のアイテムをチェックする順に関連する実装上の詳細に依存します。これは起り得るループのとても単純な種類の例です。相互に関連するルールの蓄積は予想することがより難しいループへと導きます。そしてループの出現は `and` 内の節の順 (Exercise 4.64 参照) か、またはシステムがクエリを処理する順に関連する低レベルの詳細に依存します。<sup>148</sup>

## Problems with `not`

Another quirk in the query system concerns `not`. Given the data base of Section 4.4.1, consider the following two queries:

もう 1 つのクエリシステムの予測できない出来事は `not` に関連します。Section 4.4.1 のデータベースを受け取った時、以下の 2 つのクエリについて考えてみます。

```
(and (supervisor ?x ?y)
      (not (job ?x (computer programmer))))
(and (not (job ?x (computer programmer)))
      (supervisor ?x ?y))
```

These two queries do not produce the same result. The first query begins by finding all entries in the data base that match `(supervisor ?x ?y)`, and then filters the resulting frames by removing the ones in which the value of `?x` satisfies `(job ?x (computer programmer))`. The second query begins by filtering the incoming frames to remove those that can satisfy `(job ?x (computer programmer))`. Since the only incoming frame is empty, it checks the data base to see if there are any patterns that satisfy `(job ?x (computer programmer))`. Since there generally are entries of this form, the `not` clause filters out the

<sup>148</sup> これは論理の問題ではなく、私達のインタプリタにより提供される手続的な解釈の問題です。ここでループに陥らないインタプリタを書くこともできました。例えばアサーションとルールから導きだせる全ての証明を深さ優先探索でなく、幅優先探索で列挙することもできました。しかし、そのようなシステムは私達のプログラムの中における推論の順序を活用することがより難しくなります。そのようなプログラムの中に洗練された制御を構築する試みが de Kleer et al. 1977 に説明されています。そのような深刻な制御上の問題に導かない別のテクニックとして、特定の種類のループの検知器のような特別な知識を組むことがあります (Exercise 4.67)。しかし、推論の実行において無限の小道を下すことから確実にシステムを防ぐ一般的な理論体系は有りません。“ $P(x)$  が真であることを示すためには、 $P(f(x))$  が真であることを示せ” という様式の悪魔のルールをいくつかの適切に選択された関数  $f$  に対して想像してみて下さい。

empty frame and returns an empty stream of frames. Consequently, the entire compound query returns an empty stream.

これらの 2 つのクエリは同じ結果を生成しません。最初のクエリはデータベース中の (`supervisor ?x ?y`) に適合する全てのエントリを見つけ、次に結果のフレームから `?x` の値が (`job ?x (computer programmer)`) を満たす物を削除します。2 つ目のクエリは入力フレームから (`job ?x (computer programmer)`) を満たす物を消すフィルタから開始します。入力フレームだけでは空であるため、データベースから (`job ?x (computer programmer)`) を満たすパターンが存在するか確認します。通常はこの形式のエントリが存在するので、`not` 節は空のフレームを取り除き、空のフレームのストリームを返します。結果として、複合クエリ全体が空ストリームを返します。

The trouble is that our implementation of `not` really is meant to serve as a filter on values for the variables. If a `not` clause is processed with a frame in which some of the variables remain unbound (as does `?x` in the example above), the system will produce unexpected results. Similar problems occur with the use of `lisp-value`—the Lisp predicate can't work if some of its arguments are unbound. See [Exercise 4.77](#).

問題は `not` の私達の実装は本当に変数の値上のフィルタとしての役目を果たすことを意図しています。もし `not` 節がいくつかの束縛されていない変数を持つフレームと処理された場合(上記の例における `?x` が行うように)、システムは予想外の結果を生成します。同様の問題が `lisp-value` の使用でも起こります。Lisp の述語はその引数のいくつかが未束縛な場合働くことができません。[Exercise 4.77](#) を参照して下さい。

There is also a much more serious way in which the `not` of the query language differs from the `not` of mathematical logic. In logic, we interpret the statement “not  $P$ ” to mean that  $P$  is not true. In the query system, however, “not  $P$ ” means that  $P$  is not deducible from the knowledge in the data base. For example, given the personnel data base of [Section 4.4.1](#), the system would happily deduce all sorts of `not` statements, such as that Ben Bitdiddle is not a baseball fan, that it is not raining outside, and that  $2 + 2$  is not 4.<sup>149</sup> In other words, the `not` of logic programming languages reflects the so-called *closed*

---

<sup>149</sup> Consider the query (`(not (baseball-fan (Bitdiddle Ben)))`). The system finds that (`baseball-fan (Bitdiddle Ben)`) is not in the data base, so the empty frame does not satisfy the pattern and is not filtered out of the initial stream of frames. The result of the query is thus the empty frame, which is used to instantiate the input query to produce (`(not (baseball-fan (Bitdiddle Ben)))`).

*world assumption* that all relevant information has been included in the data base.<sup>150</sup>

クエリ言語の `not` が記号論理学の `not` と異なるずっと深刻な部分があります。論理学では命題 “`not P`” を `P` は真ではないことを意味すると解釈します。しかし、クエリシステムでは “`not P`” は `P` がデータベース内の知識から推論不可能であることを意味しています。例えば、Section 4.4.1 の社員情報データベースを与えられた場合、システムは幸いにも全ての種類の `not` 命令を推論することができるでしょう。例えば Ben Bitdiddle は野球のファンではない、外で雨は振っていない、 $2 + 2$  は 4 ではないなどです。<sup>151</sup>

**Exercise 4.64:** Louis Reasoner mistakenly deletes the `outranked-by` rule (Section 4.4.1) from the data base. When he realizes this, he quickly reinstalls it. Unfortunately, he makes a slight change in the rule, and types it in as

Louis Reasoner は誤って `outranked-by` ルール (Section 4.4.1) をデータベースから削除してしまった。彼はこのことに気付いた時、直ぐに再インストールした。残念なことに、彼はルールにわずかな変更を行い、以下のように入力した。

```
(rule (outranked-by ?staff-person ?boss)
      (or (supervisor ?staff-person ?boss)
           (and (outranked-by ?middle-manager ?boss)
                 (supervisor ?staff-person
                           ?middle-manager))))
```

Just after Louis types this information into the system, DeWitt Aull comes by to find out who outranks Ben Bitdiddle. He issues the query

Louis がこの情報をシステムに入力して直ぐに、DeWitt Aull がやってきて Ben Bitdiddle の上司は誰かを調べようとした。彼は以下のクエリを入力した。

---

<sup>150</sup> A discussion and justification of this treatment of `not` can be found in the article by Clark (1978).

<sup>151</sup> クエリ `(not (baseball-fan (Bitdiddle Ben)))` について考えてみましょう。システムはデータベースに `(baseball-fan (Bitdiddle Ben))` が無いことを知り、そのため空フレームはパターンを満たさず初期値のフレームのストリームから取り除かれません。クエリの結果は従って空フレームであり、これが入力クエリのインスタンス化に用いられ、`(not (baseball-fan (Bitdiddle Ben)))` が生成されます。

```
(outranked-by (Bitdiddle Ben) ?who)
```

After answering, the system goes into an infinite loop. Explain why.

回答を行った後、システムは無限ループへと陥った。何故であるか、説明せよ。

**Exercise 4.65:** Cy D. Fect, looking forward to the day when he will rise in the organization, gives a query to find all the wheels (using the `wheel` rule of [Section 4.4.1](#)):

組織内での昇進の日を待ち望んでいる Cy D. Fect は全ての重役を探すクエリを入力してみた ([Section 4.4.1](#)のルール `wheel` を用いた)。

```
(wheel ?who)
```

To his surprise, the system responds

驚いたことにシステムは以下の内容を応答した。

```
;;; Query results:  
(wheel (Warbucks Oliver))  
(wheel (Bitdiddle Ben))  
(wheel (Warbucks Oliver))  
(wheel (Warbucks Oliver))  
(wheel (Warbucks Oliver))
```

Why is Oliver Warbucks listed four times?

何故、Oliver Warbucks は 4 度表示されたのか？

**Exercise 4.66:** Ben has been generalizing the query system to provide statistics about the company. For example, to find the total salaries of all the computer programmers one will be able to say Ben はクエリシステムを一般化し会社に関する統計を提供する。例えば、全てのコンピュータプログラマの給料の合計を求めるためには、以下のように入力することができるだろう。

```
(sum ?amount (and (job ?x (computer programmer))  
                    (salary ?x ?amount)))
```

In general, Ben's new system allows expressions of the form  
全般に、Ben の新しいシステムは以下の形式の式を可能にする。

(*accumulation-function* *<variable>* *<query pattern>*)

where *accumulation-function* can be things like *sum*, *average*, or *maximum*. Ben reasons that it should be a cinch to implement this. He will simply feed the query pattern to *qeval*. This will produce a stream of frames. He will then pass this stream through a mapping function that extracts the value of the designated variable from each frame in the stream and feed the resulting stream of values to the accumulation function. Just as Ben completes the implementation and is about to try it out, Cy walks by, still puzzling over the *wheel* query result in [Exercise 4.65](#). When Cy shows Ben the system's response, Ben groans, "Oh, no, my simple accumulation scheme won't work!"

ここで *accumulation-function* は *sum*, *average*, または *maximum* のような物である。Ben はこれを実装するのは簡単なはずだと考えた。単純にクエリパターンを *qeval* に追加するだろう。これはフレームのストリームを生成するだろう。すると彼はこのストリームを *map* 関数を通してストリーム内の各フレームから指定した変数の値を抽出し、結果の値のストリームを *accumulation*(集積) 関数へと与えるだろう。Ben が実装を完成し、丁度試験を行おうとした時に Cy が依然として [Exercise 4.65](#) の *wheel* クエリの結果に悩みながら歩いてきた。Cy が Ben にシステムの応答を見せた時、Ben はうなってから “なんてこった。私の簡単な集積の仕組みは動かない!” と述べた。

What has Ben just realized? Outline a method he can use to salvage the situation.

Ben は何に気付いたのか? この状況を救い出すため用いられる手段の要点を述べよ。

**Exercise 4.67:** Devise a way to install a loop detector in the query system so as to avoid the kinds of simple loops illustrated in the text and in [Exercise 4.64](#). The general idea is that the system should maintain some sort of history of its current chain of deductions and should not begin processing a query that it is already working on. Describe what kind of information (patterns

and frames) is included in this history, and how the check should be made. (After you study the details of the query-system implementation in [Section 4.4.4](#), you may want to modify the system to include your loop detector.)

クエリシステムにループ検知器をインストールし、テキストと [Exercise 4.64](#)で説明されたような単純なループを防ぐための手段を工夫せよ。一般的なアイデアは、システムに現在の推論の連鎖のある種の履歴を管理させ、既に取り組んでいるクエリの処理を始めないようにすることである。どのような種類の情報(パターンとフレーム)がこの履歴に含まれるか、そしてどのように検査が行われるべきかについて説明せよ。[\(Section 4.4.4\)](#)におけるクエリシステムの実装の詳細を学んだ後に、あなたはシステムを変更してループ検知器を入れたいと思うだろう)。

**Exercise 4.68:** Define rules to implement the `reverse` operation of [Exercise 2.18](#), which returns a list containing the same elements as a given list in reverse order. (Hint: Use `append-to-form`.) Can your rules answer both `(reverse (1 2 3) ?x)` and `(reverse ?x (1 2 3))`?

[Exercise 2.18](#)の `reverse` 命令を実装するルールを定義せよ。これは与えられたリストの逆順で同じ要素を含むリストを返す。(ヒント：`append-to-form` を使用せよ)。あなたのルールは `(reverse (1 2 3) ?x)` と `(reverse ?x (1 2 3))` の両方に回答することができるだろうか?

**Exercise 4.69:** Beginning with the data base and the rules you formulated in [Exercise 4.63](#), devise a rule for adding “greats” to a grandson relationship. This should enable the system to deduce that Irad is the great-grandson of Adam, or that Jabal and Jubal are the great-great-great-great-grandsons of Adam. (Hint: Represent the fact about Irad, for example, as `((great grandson) Adam Irad)`. Write rules that determine if a list ends in the word `grandson`. Use this to express a rule that allows one to derive the relationship `((great . ?rel) ?x ?y)`, where `?rel` is a list ending in `grandson`) Check your rules on queries such as `((great grandson) ?g ?ggs)` and `(?relationship Adam Irad)`.

[Exercise 4.63](#)で策定したデータベースとルールから始めて、孫の

関係に “great” を追加するためのルールを工夫せよ。これはシステムに対し Irad が Adam の great-grandson(ひ孫)であること、また Jabal と Jubal が great-great-great-great-grandsons(ひひひ孫)であることを推論することを可能にしなければならない。(ヒント：例えば Irad に関する事実を ((great grandson) Adam Irad) として表現する。リストの終端が单語 `grandson` であるかを決定するルールを書け。これを用いて?rel が `grandson` で終わるリストである場合に、関係 ((great . ?rel) ?x ?y) を導き出すことが可能なルールを表現せよ)。あなたのルールを ((great grandson) ?g ?ggs) と (?relationship Adam Irad) のようなクエリを用いて確認せよ。

#### 4.4.4 Implementing the Query System

Section 4.4.2 described how the query system works. Now we fill in the details by presenting a complete implementation of the system.

Section 4.4.2はどのようにクエリシステムが働くかについて説明した。ここでは完全なシステムの実装を公開することにより詳細を知らせる。

##### 4.4.4.1 The Driver Loop and Instantiation

The driver loop for the query system repeatedly reads input expressions. If the expression is a rule or assertion to be added to the data base, then the information is added. Otherwise the expression is assumed to be a query. The driver passes this query to the evaluator `qeval` together with an initial frame stream consisting of a single empty frame. The result of the evaluation is a stream of frames generated by satisfying the query with variable values found in the data base. These frames are used to form a new stream consisting of copies of the original query in which the variables are instantiated with values supplied by the stream of frames, and this final stream is printed at the terminal:

クエリシステムのためのドライバープロセスは繰り返し入力式を読み込みます。もし式が追加されるべきルールかアサーションであるのならばその情報が追加されます。そうでなければ式はクエリであると見做されます。ドライバはこのクエリを評価機 `qeval` に单一の空のフレームから成る初期フレームストリームと共に渡されます。評価の結果はクエリをデータベース内で見つかった変数の

値で満たすことにより生成されたフレームのストリームです。これらのフレームは、フレームのストリームにより提供された値を用いて変数がインスタンス化された元のクエリのコピーから成る新しいストリームを形成するのに用いられます。そしてこの最終的なストリームが端末に表示されます。

```
(define input-prompt ";;; Query input:")
(define output-prompt ";;; Query results:")

(define (query-driver-loop)
  (prompt-for-input input-prompt)
  (let ((q (query-syntax-process (read))))
    (cond ((assertion-to-be-added? q)
           (add-rule-or-assertion! (add-assertion-body q))
           (newline)
           (display "Assertion added to data base.")
           (query-driver-loop))
          (else
           (newline)
           (display output-prompt)
           (display-stream
             (stream-map
               (lambda (frame)
                 (instantiate
                   q
                   frame
                   (lambda (v f)
                     (contract-question-mark v))))
               (qevel q (singleton-stream '())))))
           (query-driver-loop)))))
```

Here, as in the other evaluators in this chapter, we use an abstract syntax for the expressions of the query language. The implementation of the expression syntax, including the predicate `assertion-to-be-added?` and the selector `add-assertion-body`, is given in [Section 4.4.4.7. Add-rule-or-assertion!](#) is defined in [Section 4.4.4.5.](#)

ここで、この章の他の評価機と同様に、クエリ言語の式に対して抽象構文を用います。式の構文の実装は述語 `assertion-to-be-added?` とセレクタ `add-`

`assertion-body` を含めて、Section 4.4.4.7にて与えられます。`add-rule-or-assertion!` はSection 4.4.4.5で定義されます。

Before doing any processing on an input expression, the driver loop transforms it syntactically into a form that makes the processing more efficient. This involves changing the representation of pattern variables. When the query is instantiated, any variables that remain unbound are transformed back to the input representation before being printed. These transformations are performed by the two procedures `query-syntax-process` and `contract-question-mark` (Section 4.4.4.7).

入力式のどんな処理を行う前にも、ドライバループは処理をより効率的にする形式へと構文的に変換します。これはパターン変数の表現の変更を含みます。クエリが初期化される時、未束縛である任意の変数は表示される前に入力時の表現に戻されます。これらの変換は 2 つの手続、`query-syntax-process` と `contract-question-mark` により実行されます (Section 4.4.4.7)。

To instantiate an expression, we copy it, replacing any variables in the expression by their values in a given frame. The values are themselves instantiated, since they could contain variables (for example, if `?x` in `exp` is bound to `?y` as the result of unification and `?y` is in turn bound to 5). The action to take if a variable cannot be instantiated is given by a procedural argument to `instantiate`.

式をインスタンス化するためにはまずコピーを行い、式中の全ての変数を与えられたフレーム内のそれらの値にて置き換えます。値はそれ自身がインスタンス化されます。それらが変数を含む可能性があるためです(例えば、式の中の`?x` がユニフィケーションの結果として`?y` に束縛され、`?y` が同様に 5 に束縛されている場合)。変数がインスタンス化できない場合に取るべき行動は手続 `instantiate` の引数に渡されます。

```
(define (instantiate exp frame unbound-var-handler)
  (define (copy exp)
    (cond ((var? exp)
            (let ((binding (binding-in-frame exp frame)))
              (if binding
                  (copy (binding-value binding))
                  (unbound-var-handler exp frame))))
          ((pair? exp)
           (cons (copy (car exp)) (copy (cdr exp))))
          (else exp)))
```

```
(copy exp))
```

The procedures that manipulate bindings are defined in [Section 4.4.4.8](#).

束縛を操作する手続は[Section 4.4.4.8](#)で定義されます。

#### 4.4.4.2 The Evaluator

The `qeval` procedure, called by the `query-driver-loop`, is the basic evaluator of the query system. It takes as inputs a query and a stream of frames, and it returns a stream of extended frames. It identifies special forms by a data-directed dispatch using `get` and `put`, just as we did in implementing generic operations in [Chapter 2](#). Any query that is not identified as a special form is assumed to be a simple query, to be processed by `simple-query`.

`query-driver-loop` により呼ばれる `qeval` 手続はクエリシステムの基本的な評価機です。入力としてクエリとフレームのストリームを取り、拡張されたフレームのストリームを返します。[Chapter 2](#)で総称的な命令を実装したのと同様に、`get` と `put` を用いたデータ適従による呼出により特殊形式を判別します。特殊形式とは判別されない任意のクエリは単純なクエリと見做され `simple-query` により処理されます。

```
(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)))
    (if qproc
        (qproc (contents query) frame-stream)
        (simple-query query frame-stream))))
```

`Type` and `contents`, defined in [Section 4.4.4.7](#), implement the abstract syntax of the special forms.

`type` と `contents` は[Section 4.4.4.7](#)で定義され、特殊形式の抽象構文を実装します。

### Simple queries

The `simple-query` procedure handles simple queries. It takes as arguments a simple query (a pattern) together with a stream of frames, and it returns the stream formed by extending each frame by all data-base matches of the query.

`simple-query` 手続は単純なクエリを扱います。引数として単純なクエリ(パターン)をフレームのストリームと共に取り、クエリのデータベースへの適

合全てにより各フレームを拡張することにより形成されたストリームを返します。

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
    (lambda (frame)
      (stream-append-delayed
        (find-assertions query-pattern frame)
        (delay (apply-rules query-pattern frame))))))
  frame-stream))
```

For each frame in the input stream, we use `find-assertions` (Section 4.4.4.3) to match the pattern against all assertions in the data base, producing a stream of extended frames, and we use `apply-rules` (Section 4.4.4.4) to apply all possible rules, producing another stream of extended frames. These two streams are combined (using `stream-append-delayed`, Section 4.4.4.6) to make a stream of all the ways that the given pattern can be satisfied consistent with the original frame (see Exercise 4.71). The streams for the individual input frames are combined using `stream-flatmap` (Section 4.4.4.6) to form one large stream of all the ways that any of the frames in the original input stream can be extended to produce a match with the given pattern.

入力ストリーム中の各フレームに対し、`find-assertions`(Section 4.4.4.3)を用いてデータベース内の全てのアサーションに対してパターンを適合し、拡張フレームのストリームを生成します。そして `apply-rules`(Section 4.4.4.4)を用いて全ての可能なルールを適用し、拡張フレームのもう 1 つのストリームを生成します。これらの 2 つのストリームは (`stream-append-delayed`(Section 4.4.4.6) を用いて) 接続され、与えられたパターンが元のフレームに一致して満たされることが可能な全ての手段でストリームを作ります (Exercise 4.71 参照)。個別の入力フレームに対するストリームは `stream-flatmap` (Section 4.4.4.6) を用いて接続され、元の入力ストリーム内の任意のフレームが与えられたパターンを用いて適合を生成するために拡張される能够な全ての手段により、1 つの巨大なストリームが形成されます。

## Compound queries

And queries are handled as illustrated in Figure 4.5 by the `conjoin` procedure. `Conjoin` takes as inputs the conjuncts and the frame stream and returns

the stream of extended frames. First, `conjoin` processes the stream of frames to find the stream of all possible frame extensions that satisfy the first query in the conjunction. Then, using this as the new frame stream, it recursively applies `conjoin` to the rest of the queries.

`and` クエリはFigure 4.5にて説明されているように `conjoin` 手続により扱われます。`conjoin` は入力として結合 (`conjuncts`) とフレームのストリームを取り、拡張されたフレームのストリームを返します。最初に `conjoin` はフレームのストリームを処理し、結合内の最初のクエリを満たす全ての可能なフレームの拡張のストリームを探します。次に、これを新しいフレームのストリームとして用いて、再帰的にクエリの残りに対して `conjoin` を適用します。

```
(define (conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
      frame-stream
      (conjoin (rest-conjuncts conjuncts)
               (qeval (first-conjunct conjuncts)
                      frame-stream))))
```

The expression

以下の式は

```
(put 'and 'qeval conjoin)
```

sets up `qeval` to dispatch to `conjoin` when an `and` form is encountered.

`qeval` に対し、`and` の型に遭遇した場合に `conjoin` を呼び出すように設定します。

`Or` queries are handled similarly, as shown in Figure 4.6. The output streams for the various disjuncts of the `or` are computed separately and merged using the `interleave-delayed` procedure from Section 4.4.4.6. (See Exercise 4.71 and Exercise 4.72.)

`or` クエリも同様に、Figure 4.6に示されるように扱われます。`or` の多様な選言肢に対する出力ストリームは別々に求められ、Section 4.4.4.6の `interleave-delayed` 手続を用いて結合されます。(Exercise 4.71とExercise 4.72を参照)

```
(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave-delayed
       (qeval (first-disjunct disjuncts)
```

```

        frame-stream)
(delay (disjoin (rest-disjuncts disjuncts
                                         frame-stream))))))
(put 'or 'qeval disjoin)

```

The predicates and selectors for the syntax of conjuncts and disjuncts are given in Section 4.4.4.7.

論理積 (conjuncts) と論理和 (disjuncts) の構文のための述語とセレクタはSection 4.4.4.7で提供されます。

## Filters

`Not` is handled by the method outlined in Section 4.4.2. We attempt to extend each frame in the input stream to satisfy the query being negated, and we include a given frame in the output stream only if it cannot be extended.

`not` はSection 4.4.2にて概説された手法により扱われます。入力ストリーム内の各フレームを否定されたクエリを満たすように拡張することを試みます。そして拡張できない場合にのみ出力ストリームに与えられたフレームを含めます。

```

(define (negate operands frame-stream)
  (stream-flatmap
    (lambda (frame)
      (if (stream-null?
            (qeval (negated-query operands)
                  (singleton-stream frame)))
          (singleton-stream frame)
          the-empty-stream))
    frame-stream))
(put 'not 'qeval negate)

```

`Lisp-value` is a filter similar to `not`. Each frame in the stream is used to instantiate the variables in the pattern, the indicated predicate is applied, and the frames for which the predicate returns false are filtered out of the input stream. An error results if there are unbound pattern variables.

`lisp-value` は `not` に似たフィルタです。ストリーム内の各フレームはパターン内の変数をインスタンス化するために用いられ、指定された述語が適用さ

れ、述語が偽を返したフレームは入力ストリームから取り除かれます。未束縛なパターン変数が存在する場合には結果はエラーとなります。

```
(define (lisp-value call frame-stream)
  (stream-flatmap
    (lambda (frame)
      (if (execute
            (instantiate
              call
              frame
              (lambda (v f)
                (error "Unknown pat var: LISP-VALUE"
                      v))))
        (singleton-stream frame)
        the-empty-stream))
    frame-stream))
(put 'lisp-value 'qeval lisp-value)
```

`Execute`, which applies the predicate to the arguments, must `eval` the predicate expression to get the procedure to apply. However, it must not evaluate the arguments, since they are already the actual arguments, not expressions whose evaluation (in Lisp) will produce the arguments. Note that `execute` is implemented using `eval` and `apply` from the underlying Lisp system.

`execute` は述語を引数に適用しますが、述語式を評価し適用する手続を得なければなりません。しかし引数は評価してはいけません。なぜならそれらは既に実際の引数であり、その (Lisp における) 評価が引数を生成する式ではないためです。`execute` が基礎を成す Lisp システムの `eval` と `apply` を使用して実装されていることに注意して下さい。

```
(define (execute exp)
  (apply (eval (predicate exp)
               user-initial-environment)
         (args exp)))
```

The `always-true` special form provides for a query that is always satisfied. It ignores its contents (normally empty) and simply passes through all the frames in the input stream. `Always-true` is used by the `rule-body` selector (Section 4.4.4.7) to provide bodies for rules that were defined without bodies (that is, rules whose conclusions are always satisfied).

特殊形式 `always-true` はクエリに対し常に満たされた状態を与えます。これはその中身(通常は空)を無視し、単純に入力ストリームの全てのフレームを通します。`always-true` は `rule-body` セレクタ (Section 4.4.4.7) により利用され、ボディ成しで定義されたルールに対しボディを提供します。(言い換えれば、その結果部分が常に満たされます。)

```
(define (always-true ignore frame-stream) frame-stream)
(put 'always-true 'qeval always-true)
```

The selectors that define the syntax of `not` and `lisp-value` are given in Section 4.4.4.7.

`not` と `lisp-value` の構文を定義するセレクタはSection 4.4.4.7で提供されます。

#### 4.4.4.3 Finding Assertions by Pattern Matching

`Find-assertions`, called by `simple-query` (Section 4.4.4.2), takes as input a pattern and a frame. It returns a stream of frames, each extending the given one by a data-base match of the given pattern. It uses `fetch-assertions` (Section 4.4.4.5) to get a stream of all the assertions in the data base that should be checked for a match against the pattern and the frame. The reason for `fetch-assertions` here is that we can often apply simple tests that will eliminate many of the entries in the data base from the pool of candidates for a successful match. The system would still work if we eliminated `fetch-assertions` and simply checked a stream of all assertions in the data base, but the computation would be less efficient because we would need to make many more calls to the matcher.

`find-assertions` は `simple-query` (Section 4.4.4.2) により呼ばれ、入力としてパターンとフレームを取ります。フレームのストリームを返し、各フレームは与えられた物を与えたパターンへのデータベースの適合により拡張されています。`fetch-assertions` (Section 4.4.4.5) を用いてデータベース内の全てのアサーションのストリームを得ます。これはパターンとフレームに対して適合するか確認されなければなりません。ここで `fetch-assertions` する理由は、私達は良く簡単なテストをここで適用するためです。このテストは適合を成功する候補のプールからデータベース内のエントリを数多く削減することができます。システムは例え `fetch-assertions` を削除して単純にデータベース内の全てのアサーションのストリームを確認するだけでも動くでしょう。しか

し演算は効率的ではなくなります。より多くのマッチャに対する呼出を行わねばならなくなるためです。

```
(define (find-assertions pattern frame)
  (stream-flatmap
    (lambda (datum)
      (check-an-assertion datum pattern frame)))
  (fetch-assertions pattern frame)))
```

`Check-an-assertion` takes as arguments a pattern, a data object (assertion), and a frame and returns either a one-element stream containing the extended frame or `the-empty-stream` if the match fails.

`check-an-assertion` は引数としてパターン、データオブジェクト(アサーション)、フレームを取り、拡張されたフレームを含む 1 要素のストリームか、適合を失敗した場合に `the-empty-stream` を返します。

```
(define (check-an-assertion
  assertion query-pat query-frame)
  (let ((match-result
        (pattern-match query-pat assertion query-frame)))
    (if (eq? match-result 'failed)
        the-empty-stream
        (singleton-stream match-result))))
```

The basic pattern matcher returns either the symbol `failed` or an extension of the given frame. The basic idea of the matcher is to check the pattern against the data, element by element, accumulating bindings for the pattern variables. If the pattern and the data object are the same, the match succeeds and we return the frame of bindings accumulated so far. Otherwise, if the pattern is a variable we extend the current frame by binding the variable to the data, so long as this is consistent with the bindings already in the frame. If the pattern and the data are both pairs, we (recursively) match the `car` of the pattern against the `car` of the data to produce a frame; in this frame we then match the `cdr` of the pattern against the `cdr` of the data. If none of these cases are applicable, the match fails and we return the symbol `failed`.

基本的なパターンマッチャはシンボル `failed` か、与えられたフレームの拡張を返します。マッチャの基本的な考えはパターンをデータに対して要素毎に確認し、パターン変数に対する束縛を集積します。もしパターンとデータオブ

ジェクトが同じであるなら、適合は成功しそこまで集積された束縛のフレームを返します。そうでなければ、もしパターンが変数ならば、変数をデータに対して束縛することで現在のフレームを拡張することをフレーム内に既に存在する束縛に一致するまで行います。もしパターンとデータの両方がペアであるなら、(再帰的に) パターンの `car` をデータの `car` に対して適合を行いフレームを生成します。次にこのフレームの中でパターンの `cdr` をデータのに対して適合を行います。もしこれらの場合全てが当て嵌らない場合、適合は失敗し、シンボル `failed` を返します。

```
(define (pattern-match pat dat frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? pat dat) frame)
        ((var? pat) (extend-if-consistent pat dat frame))
        ((and (pair? pat) (pair? dat))
         (pattern-match
          (cdr pat)
          (cdr dat)
          (pattern-match (car pat) (car dat) frame)))
        (else 'failed)))
```

Here is the procedure that extends a frame by adding a new binding, if this is consistent with the bindings already in the frame:

次が、フレーム内に既に存在している束縛に一致するなら、新しい束縛を追加することによりフレームを拡張する手続です。

```
(define (extend-if-consistent var dat frame)
  (let ((binding (binding-in-frame var frame)))
    (if binding
        (pattern-match
         (binding-value binding) dat frame)
        (extend var dat frame))))
```

If there is no binding for the variable in the frame, we simply add the binding of the variable to the data. Otherwise we match, in the frame, the data against the value of the variable in the frame. If the stored value contains only constants, as it must if it was stored during pattern matching by `extend-if-consistent`, then the match simply tests whether the stored and new values are the same. If so, it returns the unmodified frame; if not, it returns a failure indication. The

stored value may, however, contain pattern variables if it was stored during unification (see [Section 4.4.4.4](#)). The recursive match of the stored pattern against the new data will add or check bindings for the variables in this pattern. For example, suppose we have a frame in which  $?x$  is bound to  $(f ?y)$  and  $?y$  is unbound, and we wish to augment this frame by a binding of  $?x$  to  $(f b)$ . We look up  $?x$  and find that it is bound to  $(f ?y)$ . This leads us to match  $(f ?y)$  against the proposed new value  $(f b)$  in the same frame. Eventually this match extends the frame by adding a binding of  $?y$  to  $b$ .  $?X$  remains bound to  $(f ?y)$ . We never modify a stored binding and we never store more than one binding for a given variable.

もしフレーム内の変数に対する束縛が無い場合、単純に変数のデータに対する束縛を追加します。そうでなければこのフレーム内で、データをフレーム内の変数の値に対して適合を行います。もし格納されていた値が定数のみを持つならば、つまり `extend-if-consistent` によりパターンマッチングの間に格納されたのであれば、適合は単純に格納されていた値と新しい値が同じであるかどうかを確認します。もしそうならば、フレームを変更せずに返します。そうでないならば、失敗を示す印を返します。しかし格納されたいた値は、それがユニフィケーションの間に格納されたのであればパターン変数を含む場合があります ([Section 4.4.4.4](#) 参照)。格納されたパターンの新しいデータに対する再帰的な適合はこのパターン内の変数に対する束縛の追加、または確認を行います。例えば、 $?x$  が  $(f ?y)$  に束縛され  $?y$  が未束縛であるフレームを持っているとしましょう。そしてこのフレームを  $?x$  の  $(f b)$  への束縛で拡大させたいとします。私達は  $?x$  を探し、それが  $(f ?y)$  に束縛されているのを見つけてます。このことがこの同じフレームの中で提案された新しい値  $(f b)$  に対して  $(f ?y)$  を適合させることへと導きます。最終的に、この適合は  $?y$  から  $b$  への束縛を追加することによりこのフレームを拡張します。 $?x$  は  $(f ?y)$  への束縛を維持します。格納されていた束縛を変更することはありません。また与えられた変数に対して複数の束縛を格納することもありません。

The procedures used by `extend-if-consistent` to manipulate bindings are defined in [Section 4.4.4.8](#).

`extend-if-consistent` により使用される束縛を操作するための複数の手続は [Section 4.4.4.8](#) で定義されます。

## Patterns with dotted tails

If a pattern contains a dot followed by a pattern variable, the pattern variable matches the rest of the data list (rather than the next element of the data list), just as one would expect with the dotted-tail notation described in [Exercise 2.20](#). Although the pattern matcher we have just implemented doesn't look for dots, it does behave as we want. This is because the Lisp `read` primitive, which is used by `query-driver-loop` to read the query and represent it as a list structure, treats dots in a special way.

パターンがドットとそれに続くパターン変数を含む場合、そのパターン変数はデータリストの（次の要素ではなく）残りに適合します。誰かが予想するように[Exercise 2.20](#)にて説明されたドット付き末尾記述と同様です。私達が実装したばかりのパターンマッチャはドットを探しませんが、私達が望むとおりに振舞います。これは `query-driver-loop` で用いられる Lisp の `read` プリミティブがクエリを読み込みリスト構造として表現する時にドットを特別な方法で扱うためです。

When `read` sees a dot, instead of making the next item be the next element of a list (the `car` of a `cons` whose `cdr` will be the rest of the list) it makes the next item be the `cdr` of the list structure. For example, the list structure produced by `read` for the pattern `(computer ?type)` could be constructed by evaluating the expression `(cons 'computer (cons '?type '()))`, and that for `(computer . ?type)` could be constructed by evaluating the expression `(cons 'computer '?type)`.

`read` がドットを見た時、次の項目をリストの次の要素にするのではなく (`cons` の `car` のこと、`cdr` はリストの残り)、リスト構造の `cdr` を次の項目にします。例えば、パターン `(computer ?type)` に対する `read` により生成されるリスト構造は式 `(cons 'computer (cons '?type '()))` を評価することにより構築されます。またパターン `(computer . ?type)` に対する場合は式 `(cons 'computer '?type)` を評価することにより構築されます。

Thus, as `pattern-match` recursively compares `cars` and `cdrs` of a data list and a pattern that had a dot, it eventually matches the variable after the dot (which is a `cdr` of the pattern) against a sublist of the data list, binding the variable to that list. For example, matching the pattern `(computer . ?type)` against `(computer programmer trainee)` will match `?type` against the list `(programmer trainee)`.

従って `pattern-match` が再帰的にデータリストとドットを持つパターンの `car` と `cdr` を比較するにつれ、最終的にはドットの後ろの変数（パターンの

`cdr`) がデータリストの部分リストに対して適合され、そのリストに対してその変数が束縛されます。例えば、パターン (`computer . ?type`) を (`programmer trainee`) に適合することは `?type` をリスト (`programmer trainee`) に適合させます。

#### 4.4.4.4 Rules and Unification

`Apply-rules` is the rule analog of `find-assertions` (Section 4.4.4.3). It takes as input a pattern and a frame, and it forms a stream of extension frames by applying rules from the data base. `Stream-flatmap` maps `apply-a-rule` down the stream of possibly applicable rules (selected by `fetch-rules`, Section 4.4.4.5) and combines the resulting streams of frames.

`apply-rules` は `find-assertions` の類似のルールです (Section 4.4.4.3)。入力としてパターンとフレームを取り、データベースからルールを適用することにより拡張フレームのストリームを形成します。`stream-flatmap` は `apply-a-rule` を (`fetch-rules` により選択された (Section 4.4.4.5)) 恐らく適用可能なルールのストリームに対し `map` し、結果のフレームのストリーム群を結合します。

```
(define (apply-rules pattern frame)
  (stream-flatmap (lambda (rule)
    (apply-a-rule rule pattern frame))
    (fetch-rules pattern frame)))
```

`Apply-a-rule` applies rules using the method outlined in Section 4.4.2. It first augments its argument frame by unifying the rule conclusion with the pattern in the given frame. If this succeeds, it evaluates the rule body in this new frame.

`apply-a-rule` は Section 4.4.2 で概説された手法を用いてルールを適用します。最初にルールの結論を与えたされたフレーム内のパターンとユニフィケーションを行うことで引数フレームを増大させます。これが成功したならこの新しいフレーム内でルールのボディを評価します。

Before any of this happens, however, the program renames all the variables in the rule with unique new names. The reason for this is to prevent the variables for different rule applications from becoming confused with each other. For instance, if two rules both use a variable named `?x`, then each one may add a binding for `?x` to the frame when it is applied. These two `?x`'s have nothing to do with each other, and we should not be fooled into thinking that the two

bindings must be consistent. Rather than rename variables, we could devise a more clever environment structure; however, the renaming approach we have chosen here is the most straightforward, even if not the most efficient. (See [Exercise 4.79](#).) Here is the `apply-a-rule` procedure:

しかしこの全てが起こる前に、プログラムはルール内の全ての変数を個別の新しい名前に変更します。この理由は異なるルールの適用に対する変数が御互いに混同されることを防ぐためです。例えば、もし 2 つのルールの両方が`?x`と名付けられた変数を用いる場合、それぞれが適用された時に`?x`に対する束縛をフレームに追加するかもしれません。これら 2 つの`?x`は御互いに関係がありません。そして私達は 2 つの束縛が一致するはずだと考るよう惑わされではいけません。変数名を変えるのでなく、より賢い環境構造を工夫することもできるでしょう。しかし、私達がここで選択した改名による取り組み方は最も効率的ではないとしても、最も簡単です ([Exercise 4.79](#)参照)。以下が `apply-a-rule` 手続です。

```
(define (apply-a-rule rule query-pattern query-frame)
  (let ((clean-rule (rename-variables-in rule)))
    (let ((unify-result
          (unify-match query-pattern
                        (conclusion clean-rule)
                        query-frame)))
      (if (eq? unify-result 'failed)
          the-empty-stream
          (qeval (rule-body clean-rule)
                 (singleton-stream unify-result)))))))
```

The selectors `rule-body` and `conclusion` that extract parts of a rule are defined in [Section 4.4.4.7](#).

セレクタ `rule-body` と `conclusion` はルールの部分を抜き出します。これは [Section 4.4.4.7](#) で定義されます。

We generate unique variable names by associating a unique identifier (such as a number) with each rule application and combining this identifier with the original variable names. For example, if the rule-application identifier is 7, we might change each `?x` in the rule to `?x-7` and each `?y` in the rule to `?y-7`. (`Make-new-variable` and `new-rule-application-id` are included with the syntax procedures in [Section 4.4.4.7](#).)

私達はユニークな (unique、独自の) 識別子 (例えば番号) を各ルールの適用

に関連付けし、この識別子を元の変数名に接続することで、ユニークな変数名を生成します。例えば、もしルール適用識別子が 7 なら、ルール内の各?x を?x-7 に、各?y を?y-7 に変更するでしょう。(make-new-variable と new-rule-application-id はSection 4.4.4.7 の構文手続に含まれます。)

```
(define (rename-variables-in rule)
  (let ((rule-application-id (new-rule-application-id)))
    (define (tree-walk exp)
      (cond ((var? exp)
             (make-new-variable
              exp rule-application-id))
            ((pair? exp)
             (cons (tree-walk (car exp))
                   (tree-walk (cdr exp))))
            (else exp)))
    (tree-walk rule)))
```

The unification algorithm is implemented as a procedure that takes as inputs two patterns and a frame and returns either the extended frame or the symbol `failed`. The unifier is like the pattern matcher except that it is symmetrical—variables are allowed on both sides of the match. `Unify-match` is basically the same as `pattern-match`, except that there is extra code (marked “\*\*\*” below) to handle the case where the object on the right side of the match is a variable.

ユニファイケーションアルゴリズムは手続として実装され、入力として 2 つのパターンとフレームを取り、拡張されたフレームかシンボル `failed` を返します。ユニファイアはパターンマッチャに似ていますが、対照的であることが異なります—つまり、変数が適合の両サイドに存在することが許されます。`unify-match` は基本的には `pattern-match` と同じですが、(以下で “\*\*\*” のマークを付けた) 拡張コードの存在が異なります。これは適合の右側のオブジェクトが変数である場合を扱います。

```
(define (unify-match p1 p2 frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? p1 p2) frame)
        ((var? p1) (extend-if-possible p1 p2 frame))
        ((var? p2) (extend-if-possible p2 p1 frame)) ; ***
        ((and (pair? p1) (pair? p2))
         (unify-match (cdr p1)
```

```

  (cdr p2)
  (unify-match (car p1)
    (car p2)
    frame)))
(else 'failed)))

```

In unification, as in one-sided pattern matching, we want to accept a proposed extension of the frame only if it is consistent with existing bindings. The procedure `extend-if-possible` used in unification is the same as the `extend-if-consistent` used in pattern matching except for two special checks, marked “\*\*\*” in the program below. In the first case, if the variable we are trying to match is not bound, but the value we are trying to match it with is itself a (different) variable, it is necessary to check to see if the value is bound, and if so, to match its value. If both parties to the match are unbound, we may bind either to the other.

ユニファイケーションにおいては一方向マッチングのように、既存の束縛に一致する場合のみ提案されたフレームの拡張を受け入れたいです。手続 `extend-if-possible` はユニファイケーションにおいて使用され、パターンマッチにて利用される `extend-if-consistent` と同じですが、下記のプログラムで “\*\*\*” がマークされている、2つの特別なチェックが異なります。最初のケースでは、もし適合を試す変数が未束縛であり、かつそれに対して適合させようとしている値それ自体が（異なる）変数である場合に、その値が束縛されているかを確認する必要があります。そしてもしそうであれば、その値を適合する必要があります。もし適合の両側が共に未束縛である場合、それぞれを御互いに束縛します。

The second check deals with attempts to bind a variable to a pattern that includes that variable. Such a situation can occur whenever a variable is repeated in both patterns. Consider, for example, unifying the two patterns `(?x ?x)` and `(?y <expression involving ?y>)` in a frame where both `?x` and `?y` are unbound. First `?x` is matched against `?y`, making a binding of `?x` to `?y`. Next, the same `?x` is matched against the given expression involving `?y`. Since `?x` is already bound to `?y`, this results in matching `?y` against the expression. If we think of the unifier as finding a set of values for the pattern variables that make the patterns the same, then these patterns imply instructions to find a `?y` such that `?y` is equal to the expression involving `?y`. There is no general method for solving such equations, so we reject such bindings; these cases are recognized by the predicate `depends-on?`.

2つ目のチェックは変数を、変数を含むパターンに対して束縛する試みを

取り扱います。そのような状況は変数が両方のパターン内で繰り返される場合に常に起こります。例えば2つのパターン、( $?x ?x$ )と( $?y <?y を含む式>$ )を、 $?x$ と $?y$ の両方が未束縛である場合のフレーム内にてユニフィケーションを行う場合について考えてみて下さい。最初の $?x$ は $?y$ に対して適合し、 $?x$ から $?y$ への束縛を作成します。次に同じ $?x$ が与えられた $?y$ を含む式に対して適合されます。 $?x$ は既に $?y$ に対して束縛されているため、これは結果として $?y$ をその式に対して適合することになります。もし私達がユニファイアを2つのパターンと同じにするパターン変数に対する値の集合を見つけるものとして考えているなら、これらのパターンは $?y$ が $?y$ を含む式に等しいような $?y$ を見付ける命令を暗示します。そのような方程式を解く一般的な手法は存在しませんので、私達はそのような束縛を却下します。このような場合が述語 depends-

on? により認識されます。<sup>152</sup>

<sup>152</sup>In general, unifying  $?y$  with an expression involving  $?y$  would require our being able to find a fixed point of the equation  $?y = <\text{expression involving } ?y>$ . It is sometimes possible to syntactically form an expression that appears to be the solution. For example,  $?y = (f ?y)$  seems to have the fixed point  $(f (f (f \dots)))$ , which we can produce by beginning with the expression  $(f ?y)$  and repeatedly substituting  $(f ?y)$  for  $?y$ . Unfortunately, not every such equation has a meaningful fixed point. The issues that arise here are similar to the issues of manipulating infinite series in mathematics. For example, we know that 2 is the solution to the equation  $y = 1 + y/2$ . Beginning with the expression  $1 + y/2$  and repeatedly substituting  $1 + y/2$  for  $y$  gives

一般的に、 $?y$  を  $?y$  を含む式にユニフィケーションを行う場合には、方程式  $?y = <\text{expression involving } ?y>$  の不動点を見つけられなければなりません。偶に解が存在する式を構文的に形成することができます。例えば、 $?y = (f ?y)$  は不動点  $(f (f (f \dots)))$  を持つように見えます。これは式  $(f ?y)$  で始め、繰り返し  $?y$  を  $(f ?y)$  で置き換えることで生成できます。残念ながら全てのそのような方程式が意味のある不動点を持つわけではありません。ここで浮かび上がる問題は数学で無限級数を扱う場合の問題と似ています。例えば、私達は 2 が方程式  $y = 1 + y/2$  の解であることを知っています。式  $1 + y/2$  で始めて、繰り返し  $y$  を  $1 + y/2$  で置き換えていくと以下の様になります。

$$2 = y = 1 + \frac{y}{2} = 1 + \frac{1}{2} \left( 1 + \frac{y}{2} \right) = 1 + \frac{1}{2} + \frac{y}{4} = \dots,$$

which leads to

これは以下の式へと導きます。

$$2 = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

However, if we try the same manipulation beginning with the observation that -1 is the solution to the equation  $y = 1 + 2y$ , we obtain

しかし、もし同じ操作を式  $y = 1 + 2y$  の解が -1 であるという結果から始めると、

$$-1 = y = 1 + 2y = 1 + 2(1 + 2y) = 1 + 2 + 4y = \dots,$$

which leads to

これは以下の式へと導きます。

$$-1 = 1 + 2 + 4 + 8 + \dots$$

Although the formal manipulations used in deriving these two equations are identical, the first result is a valid assertion about infinite series but the second is not. Similarly, for our unification results, reasoning with an arbitrary syntactically constructed expression may lead to errors.

これらの 2 つの等式を導き出した形式的な操作は同一であるにも係らず、最初の結果は無限級数に関して有効な正しい主張となります。しかし、2 つ目はそうではありません。同様に、私達のユニフィケーションの結果に対して無計画に構文に従い構築された式はエラーへと繋るでしょう。

On the other hand, we do not want to reject attempts to bind a variable to itself. For example, consider unifying ( $?x$   $?x$ ) and ( $?y$   $?y$ ). The second attempt to bind  $?x$  to  $?y$  matches  $?y$  (the stored value of  $?x$ ) against  $?y$  (the new value of  $?x$ ). This is taken care of by the `equal?` clause of `unify-match`.

一方で、変数をそれ自体へ束縛する試みを拒否したくはありません。例として、( $?x$   $?x$ ) と ( $?y$   $?y$ ) のユニフィケーションについて考えてみましょう。二度目の $?x$ を $?y$ へ束縛する試行は $?y$ ( $?x$ の新しい値)に対する $?y$ ( $?x$ に格納された値)に適合します。これは `unify-match` の `equal?` 節により担当されます。

```
(define (extend-if-possible var val frame)
  (let ((binding (binding-in-frame var frame)))
    (cond (binding
            (unify-match
              (binding-value binding) val frame))
          ((var? val) ; ***
           (let ((binding (binding-in-frame val frame)))
             (if binding
                 (unify-match
                   var (binding-value binding) frame)
                 (extend var val frame))))
          ((depends-on? val var frame) ; ***
           'failed)
          (else (extend var val frame)))))
```

`Depends-on?` is a predicate that tests whether an expression proposed to be the value of a pattern variable depends on the variable. This must be done relative to the current frame because the expression may contain occurrences of a variable that already has a value that depends on our test variable. The structure of `depends-on?` is a simple recursive tree walk in which we substitute for the values of variables whenever necessary.

`depends-on?` はパターン変数の値であると提案された式がその変数に依存するかを確認します。これは現在のフレームと比較して行われなければなりません。式がテスト変数に依存する値を既に持つ変数の存在を含むかもしれません。`depends-on?` の構造は簡単な再帰木の探索であり、この中で必要な場合いつでも変数の値を置き換えます。

```
(define (depends-on? exp var frame)
  (define (tree-walk e)
```

```

(cond ((var? e)
       (if (equal? var e)
           true
           (let ((b (binding-in-frame e frame)))
             (if b
                 (tree-walk (binding-value b))
                 false))))
      ((pair? e)
       (or (tree-walk (car e))
           (tree-walk (cdr e))))
      (else false)))
(tree-walk exp))

```

#### 4.4.4.5 Maintaining the Data Base

One important problem in designing logic programming languages is that of arranging things so that as few irrelevant data-base entries as possible will be examined in checking a given pattern. In our system, in addition to storing all assertions in one big stream, we store all assertions whose `cars` are constant symbols in separate streams, in a table indexed by the symbol. To fetch an assertion that may match a pattern, we first check to see if the `car` of the pattern is a constant symbol. If so, we return (to be tested using the matcher) all the stored assertions that have the same `car`. If the pattern's `car` is not a constant symbol, we return all the stored assertions. Cleverer methods could also take advantage of information in the frame, or try also to optimize the case where the `car` of the pattern is not a constant symbol. We avoid building our criteria for indexing (using the `car`, handling only the case of constant symbols) into the program; instead we call on predicates and selectors that embody our criteria.

論理プログラミング言語の設計における重要な問題の1つは、与えられたパターンの確認においてできる限り少ないデータベースのエントリが検査されるように物事を準備することです。私達のシステムでは、全てのアサーションを1つの大きなストリームに格納することに加えて、`car`が静的なシンボルである全てのアサーションをそのシンボルで索引付けられたテーブル内の分離されたストリームに格納します。パターンに適合するかもしれないアサーションを取り出すためには、最初にパターンの `car` が静的なシンボルであるかを確認

します。もしそうならば、(マッチャを用いて確認するため) 同じ `car` を持つ全ての格納されたアサーションを返します。もしパターンの `car` が静的なシンボルでない場合には、格納されたアサーションを全て返します。より賢い方法ではフレーム内の情報も活用するか、パターンの `car` が静的なシンボルでない場合にも最適化を行うことに挑むことができるでしょう。私達は検索作成の基準(`car` を用いる、静的シンボルの場合のみを扱う)をこのプログラムの中に構築することを避けました。その代わりに私達の基準を具現する述語とセレクタを呼び出します。

```
(define THE-ASSERTIONS the-empty-stream)
(define (fetch-assertions pattern frame)
  (if (use-index? pattern)
      (get-indexed-assertions pattern)
      (get-all-assertions)))
(define (get-all-assertions) THE-ASSERTIONS)
(define (get-indexed-assertions pattern)
  (get-stream (index-key-of pattern) 'assertion-stream))
```

`Get-stream` looks up a stream in the table and returns an empty stream if nothing is stored there.

`get-stream` はテーブル内のストリームを探し、そこに何にも格納されていない場合には空ストリームを返します。

```
(define (get-stream key1 key2)
  (let ((s (get key1 key2)))
    (if s s the-empty-stream)))
```

Rules are stored similarly, using the `car` of the rule conclusion. Rule conclusions are arbitrary patterns, however, so they differ from assertions in that they can contain variables. A pattern whose `car` is a constant symbol can match rules whose conclusions start with a variable as well as rules whose conclusions have the same `car`. Thus, when fetching rules that might match a pattern whose `car` is a constant symbol we fetch all rules whose conclusions start with a variable as well as those whose conclusions have the same `car` as the pattern. For this purpose we store all rules whose conclusions start with a variable in a separate stream in our table, indexed by the symbol `?`.

ルールも同様にルールの結論の `car` を用いて格納されます。しかしルールの結論は任意のパターンであるため、変数を含められることがアサーションと

は異なります。`car` が静的なシンボルであるパターンは結論が変数で始まるルールと、結論が同じ `car` を持つルールにも適合できます。従って、`car` が静的なシンボルであるパターンに適合するかもしれないルールを取り出す場合、結論が変数で始まる全てのルールと、結論がそのパターンと同じ `car` を持つルールを取り出します。この目的のために、結論が変数で始まる全てのルールをテーブル内の分離されたストリームに、シンボル? で索引付けして格納します。

```
(define THE-RULES the-empty-stream)
(define (fetch-rules pattern frame)
  (if (use-index? pattern)
      (get-indexed-rules pattern)
      (get-all-rules)))
(define (get-all-rules) THE-RULES)
(define (get-indexed-rules pattern)
  (stream-append
   (get-stream (index-key-of pattern) 'rule-stream)
   (get-stream '? 'rule-stream)))
```

`Add-rule-or-assertion!` is used by `query-driver-loop` to add assertions and rules to the data base. Each item is stored in the index, if appropriate, and in a stream of all assertions or rules in the data base.

`add-rule-or-assertion!` は `query-driver-loop` により使用されアサーションとルールとデータベースに追加します。各アイテムは適切であればインデックスに格納され、データベース内の全てのアサーション、またはルールのストリームに格納されます。

```
(define (add-rule-or-assertion! assertion)
  (if (rule? assertion)
      (add-rule! assertion)
      (add-assertion! assertion)))
(define (add-assertion! assertion)
  (store-assertion-in-index assertion)
  (let ((old-assertions THE-ASSERTIONS))
    (set! THE-ASSERTIONS
          (cons-stream assertion old-assertions)))
  'ok))
(define (add-rule! rule)
  (store-rule-in-index rule))
```

```
(let ((old-rules THE-RULES))
  (set! THE-RULES (cons-stream rule old-rules))
  'ok))
```

To actually store an assertion or a rule, we check to see if it can be indexed. If so, we store it in the appropriate stream.

実際にアサーション、またはルールを格納するためには、索引を付けられるかを確認します。もしそうであれば、適切なストリームに格納します。

```
(define (store-assertion-in-index assertion)
  (if (indexable? assertion)
      (let ((key (index-key-of assertion)))
        (let ((current-assertion-stream
              (get-stream key 'assertion-stream)))
          (put key
                'assertion-stream
                (cons-stream
                  assertion
                  current-assertion-stream))))))
(define (store-rule-in-index rule)
  (let ((pattern (conclusion rule)))
    (if (indexable? pattern)
        (let ((key (index-key-of pattern)))
          (let ((current-rule-stream
                (get-stream key 'rule-stream)))
            (put key
                  'rule-stream
                  (cons-stream rule
                                current-rule-stream)))))))
```

The following procedures define how the data-base index is used. A pattern (an assertion or a rule conclusion) will be stored in the table if it starts with a variable or a constant symbol.

以下の手続はデータベースのインデックス(索引)がどのように使用されかについて定義します。パターン(アサーション、またはルールの結論)が変数、または静的なシンボルで始まる場合にテーブルに格納されます。

```
(define (indexable? pat)
```

```
(or (constant-symbol? (car pat))
     (var? (car pat))))
```

The key under which a pattern is stored in the table is either ? (if it starts with a variable) or the constant symbol with which it starts.

パターンがその下に格納されるテーブル内のキーは(変数で始まる場合は)?、またはパターンの始めの静的なシンボルです。

```
(define (index-key-of pat)
  (let ((key (car pat)))
    (if (var? key) '? key)))
```

The index will be used to retrieve items that might match a pattern if the pattern starts with a constant symbol.

インデックスはパターンが静的なシンボルで始まる場合、パターンにマッチするかもしれないアイテムを取得するために利用されます。

```
(define (use-index? pat) (constant-symbol? (car pat)))
```

**Exercise 4.70:** What is the purpose of the let bindings in the procedures `add-assertion!` and `add-rule!`? What would be wrong with the following implementation of `add-assertion!`? Hint: Recall the definition of the infinite stream of ones in Section 3.5.2:  
`(define ones (cons-stream 1 ones))`.

手続 `add-assertion!` と `add-rule!` 内の `let` の束縛の目的は何か? 以下の `add-assertion!` の実装の誤りは何か? ヒント: Section 3.5.2における1の無限ストリームの定義を思い出せ: `(define ones (cons-stream 1 ones))`

```
(define (add-assertion! assertion)
  (store-assertion-in-index assertion)
  (set! THE-ASSERTIONS
        (cons-stream assertion THE-ASSERTIONS))
  'ok)
```

#### 4.4.4.6 Stream Operations

The query system uses a few stream operations that were not presented in Chapter 3.

クエリシステムはChapter 3には存在しなかつたいくつかのストリーム命令を用います。

`stream-append-delayed` and `interleave-delayed` are just like `stream-append` and `interleave` (Section 3.5.3), except that they take a delayed argument (like the `integral` procedure in Section 3.5.4). This postpones looping in some cases (see Exercise 4.71).

`stream-append-delayed` と `interleave-delayed` は `stream-append` と `interleave`(Section 3.5.3) と同じですが、それらが(Section 3.5.4の `integral` の様に)遅延化された引数を取ることが異なります。これはいくつかの場合においてループを先送りします。(Exercise 4.71参照)

```
(define (stream-append-delayed s1 delayed-s2)
  (if (stream-null? s1)
    (force delayed-s2)
    (cons-stream
      (stream-car s1)
      (stream-append-delayed
        (stream-cdr s1)
        delayed-s2))))
(define (interleave-delayed s1 delayed-s2)
  (if (stream-null? s1)
    (force delayed-s2)
    (cons-stream
      (stream-car s1)
      (interleave-delayed
        (force delayed-s2)
        (delay (stream-cdr s1))))))
```

`Stream-flatmap`, which is used throughout the query evaluator to map a procedure over a stream of frames and combine the resulting streams of frames, is the stream analog of the `flatmap` procedure introduced for ordinary lists in Section 2.2.3. Unlike ordinary `flatmap`, however, we accumulate the streams with an interleaving process, rather than simply appending them (see Exercise 4.72 and Exercise 4.73).

`stream-flatmap` はクエリ評価機を通して使用され手続をフレームのストリーム上に対して map し、結果としての複数のフレームのストリームを接続します。`stream-flatmap` はSection 2.2.3にて通常のリストのために導入された

`flatmap` 手続のストリーム向け類似品です。しかし通常の `flatmap` と異なり、単純にストリームを `append` していくのではなく、相互配置処理により蓄積します。(Exercise 4.72とExercise 4.73参照)

```
(define (stream-flatmap proc s)
  (flatten-stream (stream-map proc s)))

(define (flatten-stream stream)
  (if (stream-null? stream)
      the-empty-stream
      (interleave-delayed
       (stream-car stream)
       (delay (flatten-stream (stream-cdr stream))))))
```

The evaluator also uses the following simple procedure to generate a stream consisting of a single element:

評価機はまた以下の単純な手続を用いて单一要素から成るストリームを生成します。

```
(define (singleton-stream x)
  (cons-stream x the-empty-stream))
```

#### 4.4.4.7 Query Syntax Procedures

Type and `contents`, used by `qeval` (Section 4.4.4.2), specify that a special form is identified by the symbol in its `car`. They are the same as the `type-tag` and `contents` procedures in Section 2.4.2, except for the error message.

`qeval`(Section 4.4.4.2) により用いられる `type` と `contents` は、特殊形式がその `car` に存在するシンボルにより判別されることを指示します。これらはSection 2.4.2の `type-tag` と `contents` 手続と同じですが、エラーメッセージが異なります。

```
(define (type exp)
  (if (pair? exp)
      (car exp)
      (error "Unknown expression TYPE" exp)))
(define (contents exp)
  (if (pair? exp)
```

```
(cdr exp)
(error "Unknown expression CONTENTS" exp)))
```

The following procedures, used by `query-driver-loop` (in [Section 4.4.4.1](#)), specify that rules and assertions are added to the data base by expressions of the form (`assert! <rule-or-assertion>`):

以下の手続は[Section 4.4.4.1](#)の `query-driver-loop` にて使用されます。これはルールとアサーションがデータベースに (`assert! <rule-or-assertion>`) の形式の式により追加されることを指示します。

```
(define (assertion-to-be-added? exp)
  (eq? (type exp) 'assert!))
(define (add-assertion-body exp) (car (contents exp)))
```

Here are the syntax definitions for the `and`, `or`, `not`, and `lisp-value` special forms ([Section 4.4.4.2](#)):

以下は特殊形式 `and`, `or`, `not`, `lisp-value` のための構文定義です。([Section 4.4.4.2](#))

```
(define (empty-conjunction? exps) (null? exps))
(define (first-conjunct exps) (car exps))
(define (rest-conjuncts exps) (cdr exps))
(define (empty-disjunction? exps) (null? exps))
(define (first-disjunct exps) (car exps))
(define (rest-disjuncts exps) (cdr exps))
(define (negated-query exps) (car exps))
(define (predicate exps) (car exps))
(define (args exps) (cdr exps))
```

The following three procedures define the syntax of rules:

以下の 3 つの手続はルールの構文を定義します。

```
(define (rule? statement)
  (tagged-list? statement 'rule))
(define (conclusion rule) (cadr rule))
(define (rule-body rule)
  (if (null? (cddr rule)) '(always-true) (caddr rule)))
```

`Query-driver-loop` ([Section 4.4.4.1](#)) calls `query-syntax-process` to transform pattern variables in the expression, which have the form `?symbol`, into the

internal format (`? symbol`). That is to say, a pattern such as `(job ?x ?y)` is actually represented internally by the system as `(job (? x) (? y))`. This increases the efficiency of query processing, since it means that the system can check to see if an expression is a pattern variable by checking whether the `car` of the expression is the symbol `?`, rather than having to extract characters from the symbol. The syntax transformation is accomplished by the following procedure:

`query-driver-loop`(Section 4.4.4.1) は `query-syntax-process` を呼び、`?symbol` の形態を持つ式のパターン変数を内部形式 (`? symbol`) に変形します。これは言ってみれば、`(job ?x ?y)` のようなパターンが実際には内部的にシステムにより `(job (? x) (? y))` と表現されているということです。これによりクエリ処理の効率が良くなります。システムが式がパターン変数であるかを確認するのにシンボルから文字を抽出する必要が無しに、式の `car` がシンボル? であるかどうかを確認することにより確認できることを意味するためです。構文変形は以下の手続により達成されます。<sup>153</sup>

```
(define (query-syntax-process exp)
  (map-over-symbols expand-question-mark exp))
(define (map-over-symbols proc exp)
  (cond ((pair? exp)
         (cons (map-over-symbols proc (car exp))
               (map-over-symbols proc (cdr exp)))))
```

---

<sup>153</sup>Most Lisp systems give the user the ability to modify the ordinary `read` procedure to perform such transformations by defining *reader macro characters*. Quoted expressions are already handled in this way: The reader automatically translates '`expression`' into `(quote expression)` before the evaluator sees it. We could arrange for `?expression` to be transformed into `(? expression)` in the same way; however, for the sake of clarity we have included the transformation procedure here explicitly.

多くの Lisp システムは通常の `read` 手続を *reader macro characters*(リーダマクロキャラクタ)を定義することにより変更し、そのような変形を実行させる能力をユーザに与えています。クオートされた式は既にこのような取り扱われています。リーダーは評価機が式を見る前に自動的に'`expression`'を `(quote expression)` に変形します。私達は同様に`?expression`が `(? expression)` に変形されるように準備することも可能でした。しかし、明快さのために、私達はここに明示的に変形手続を含めました。

`Expand-question-mark` and `contract-question-mark` use several procedures with `string` in their names. These are Scheme primitives.

`expand-question-mark` と `contract-question-mark` は名前に `string` を持ついくつかの手続を使用します。これらは Scheme のプリミティブです。

```

((symbol? exp) (proc exp))
(else exp)))
(define (expand-question-mark symbol)
  (let ((chars (symbol->string symbol)))
    (if (string=? (substring chars 0 1) "?")
        (list '? 
              (string->symbol
                (substring chars 1 (string-length chars))))
        symbol)))

```

Once the variables are transformed in this way, the variables in a pattern are lists starting with ?, and the constant symbols (which need to be recognized for data-base indexing, Section 4.4.4.5) are just the symbols.

一旦、変数がこのように変形されれば、パターン内の変数は?で始まるリストであり、静的なシンボル(データベースの索引付けのために必要、Section 4.4.4.5)はただのシンボルです。

```

(define (var? exp) (tagged-list? exp '?))
(define (constant-symbol? exp) (symbol? exp))

```

Unique variables are constructed during rule application (in Section 4.4.4.4) by means of the following procedures. The unique identifier for a rule application is a number, which is incremented each time a rule is applied.

他とは異なる変数がルールの適用の間に以下の手続を用いて構築されます(Section 4.4.4.4)。ルール適用のための独自識別子は数値であり、ルールが適用される度にインクリメントされます。

```

(define rule-counter 0)
(define (new-rule-application-id)
  (set! rule-counter (+ 1 rule-counter))
  rule-counter)
(define (make-new-variable var rule-application-id)
  (cons '?' (cons rule-application-id (cdr var))))

```

When `query-driver-loop` instantiates the query to print the answer, it converts any unbound pattern variables back to the right form for printing, using `query-driver-loop`が回答を表示するためにクエリをインスタンス化する時、全ての未束縛のパターン変数を表示に適した形式に以下を用いて戻します。

```
(define (contract-question-mark variable)
  (string->symbol
    (string-append "?"
      (if (number? (cadr variable))
        (string-append (symbol->string (caddr variable))
                      " - "
                      (number->string (cadr variable)))
        (symbol->string (cadr variable)))))))
```

#### 4.4.4.8 Frames and Bindings

Frames are represented as lists of bindings, which are variable-value pairs:  
 フレームは変数と値のペアである束縛のリストとして表現されます。

```
(define (make-binding variable value)
  (cons variable value))
(define (binding-variable binding) (car binding))
(define (binding-value binding) (cdr binding))
(define (binding-in-frame variable frame)
  (assoc variable frame))
(define (extend variable value frame)
  (cons (make-binding variable value) frame))
```

**Exercise 4.71:** Louis Reasoner wonders why the `simple-query` and `disjoin` procedures (Section 4.4.4.2) are implemented using explicit `delay` operations, rather than being defined as follows:

Louis Reasoner はなぜ `simple-query` と `disjoin` の手続 (Section 4.4.4.2) は以下のようない定義ではなく、明示的な `delay` 命令を用いて実装されたのか不思議だった。

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
    (lambda (frame)
      (stream-append
        (find-assertions query-pattern frame)
        (apply-rules query-pattern frame))))
  frame-stream))
```

```
(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave
       (qevel (first-disjunct disjuncts)
              frame-stream)
       (disjoin (rest-disjuncts disjuncts)
              frame-stream))))
```

Can you give examples of queries where these simpler definitions would lead to undesirable behavior?

あなたはこれらのより簡単な定義を望まない振舞いへと導くクエリの例を与えることができるか?

**Exercise 4.72:** Why do `disjoin` and `stream-flatmap` interleave the streams rather than simply append them? Give examples that illustrate why interleaving works better. (Hint: Why did we use `interleave` in Section 3.5.3?)

なぜ `disjoin` と `stream-flatmap` は単純にそれらを `append` せずに相互配置するのか? なぜ相互配置のほうがより良く働くのかを説明する例を与えよ。 (ヒント : なぜ私達は Section 3.5.3において `interleave` を使用したのか?)

**Exercise 4.73:** Why does `flatten-stream` use `delay` explicitly? What would be wrong with defining it as follows:

なぜ `flatten-stream` は明示的に `delay` を用いるのか? 以下のように定義した場合に何が間違っているのか?

```
(define (flatten-stream stream)
  (if (stream-null? stream)
      the-empty-stream
      (interleave
       (stream-car stream)
       (flatten-stream (stream-cdr stream)))))
```

**Exercise 4.74:** Alyssa P. Hacker proposes to use a simpler version of `stream-flatmap` in `negate`, `lisp-value`, and `find-assertions`. She observes that the procedure that is mapped over the frame

stream in these cases always produces either the empty stream or a singleton stream, so no interleaving is needed when combining these streams.

Alyssa P. Hacker はより簡単な版の `stream-flatmap` を `negate`, `lisp-value`, `find-assertions` の中で使用することを提案した。彼女はフレームのストリーム上に `map` される手続はこれらの場合において常に空ストリームか、單一要素のストリームを生成する。そのためこれらのストリームを接続する場合、相互配置する必要が無いと気付いた。

- a Fill in the missing expressions in Alyssa's program.

Alyssa のプログラムに欠けている式を埋めよ。

```
(define (simple-stream-flatmap proc s)
  (simple-flatten (stream-map proc s)))
(define (simple-flatten stream)
  (stream-map ???
    (stream-filter ??? stream)))
```

- b Does the query system's behavior change if we change it in this way?

クエリシステムの振舞はこのように変更した場合に変化するだろうか？

**Exercise 4.75:** Implement for the query language a new special form called `unique`. `Unique` should succeed if there is precisely one item in the data base satisfying a specified query. For example,

クエリ言語に対して新しい特殊形式 `unique` を実装せよ。`unique` は正確にデータベース内の 1 つの項目が指定されたクエリを満たす場合に成功しなければならない。例えば、

```
(unique (job ?x (computer wizard)))
```

should print the one-item stream

上の式は 1 つの項目のストリームを表示しなければならない。

```
(unique (job (Bitdiddle Ben) (computer wizard)))
```

since Ben is the only computer wizard, and  
Ben はただ 1 人のコンピュータ ウィザードであるためである。次に、  
(unique (job ?x (computer programmer)))

should print the empty stream, since there is more than one computer programmer. Moreover,

上は空ストリームを表示しなければならない。複数のコンピュータ プログラマが存在するためである。さらに、

(and (job ?x ?j) (unique (job ?anyone ?j)))

should list all the jobs that are filled by only one person, and the people who fill them.

上はただ 1 人により埋められた役職とその人達を全て表示しなければならない。

There are two parts to implementing `unique`. The first is to write a procedure that handles this special form, and the second is to make `qeval` dispatch to that procedure. The second part is trivial, since `qeval` does its dispatching in a data-directed way. If your procedure is called `uniquely-asserted`, all you need to do is

`unique` を実装するには 2 つの部分が存在する。1 つ目はこの特殊形式を扱う手続を書くことであり、2 つ目は `qeval` にその手続を呼出させることである。2 つ目の部分は自明だ。`qeval` はその呼出をデータ適従の方法に従うためである。もしあなたの手続が `uniquely-asserted` という名前であるなら、やらなければいけないことは以下である。

(put 'unique 'qeval uniquely-asserted)

and `qeval` will dispatch to this procedure for every query whose type (`car`) is the symbol `unique`.

これで `qeval` は型 (`car`) がシンボル `unique` である全てのクエリに対してこの手続を呼び出す。

The real problem is to write the procedure `uniquely-asserted`. This should take as input the `contents` (`cdr`) of the `unique` query, together with a stream of frames. For each frame in the stream, it should use `qeval` to find the stream of all extensions to the frame

that satisfy the given query. Any stream that does not have exactly one item in it should be eliminated. The remaining streams should be passed back to be accumulated into one big stream that is the result of the `unique` query. This is similar to the implementation of the `not` special form.

真の問題は手続 `uniquely-asserted` を書くことである。これは入力として `unique` クエリの `contents(cdr)` をフレームのストリームと共に受け取る。ストリームの各フレームに対し、`qeval` を用いて与えられたクエリを満たすフレームの全ての拡張のストリームを見つけなければならない。正確に 1 つのアイテムのみを持たないストリームは全て取り除かなければならない。残ったストリームは `unique` クエリの結果である 1 つの巨大なストリームに蓄積するために戻されなければならない。これは特殊形式 `not` の実装に似ている。

Test your implementation by forming a query that lists all people who supervise precisely one person.

あなたの実装を正確に 1 人だけを監督する全ての人々を並べるクエリを形成することによりテストせよ。

**Exercise 4.76:** Our implementation of `and` as a series combination of queries (Figure 4.5) is elegant, but it is inefficient because in processing the second query of the `and` we must scan the data base for each frame produced by the first query. If the data base has  $n$  elements, and a typical query produces a number of output frames proportional to  $n$  (say  $n/k$ ), then scanning the data base for each frame produced by the first query will require  $n^2/k$  calls to the pattern matcher. Another approach would be to process the two clauses of the `and` separately, then look for all pairs of output frames that are compatible. If each query produces  $n/k$  output frames, then this means that we must perform  $n^2/k^2$  compatibility checks—a factor of  $k$  fewer than the number of matches required in our current method.

一連のクエリの結合としての `and` の実装 (Figure 4.5) は洗練されているが非効率だ。`and` の 2 つ目のクエリの処理において最初のクエリにより生成された各フレームに対してデータベースを走査しなければならないためである。もしデータベースが  $n$  個の要素を

持ち、典型的なクエリが  $n$  に比例した数 (仮に  $n/k$  個) の出力フレームを生成する場合、最初のクエリにより生成された各フレームに対するデータベースの走査は  $n^2/k$  のパターンマッチャの呼出を必要とする。別の取り組み方としては `and` の 2 つの節を分離して処理し、矛盾のない出力フレームの全てのペアを探すことになるだろう。もし各クエリが  $n/k$  個の出力フレームを生成するなら、これは  $n^2/k^2$  回の無矛盾テストを実行しなければならないことを意味する。 $k$  の係数が現在の手法で必要な適合数よりもより少ない。

Devise an implementation of `and` that uses this strategy. You must implement a procedure that takes two frames as inputs, checks whether the bindings in the frames are compatible, and, if so, produces a frame that merges the two sets of bindings. This operation is similar to unification.

この戦略を用いる `and` の実装を工夫せよ。入力として 2 つのフレームを取り、両フレームの中の束縛が無矛盾であることを確認しなければならない。もしそうであるなら束縛の 2 つの集合をマージするフレームを生成する。この操作はユニフィケーションに似ている。

**Exercise 4.77:** In Section 4.4.3 we saw that `not` and `lisp-value` can cause the query language to give “wrong” answers if these filtering operations are applied to frames in which variables are unbound. Devise a way to fix this shortcoming. One idea is to perform the filtering in a “delayed” manner by appending to the frame a “promise” to filter that is fulfilled only when enough variables have been bound to make the operation possible. We could wait to perform filtering until all other operations have been performed. However, for efficiency’s sake, we would like to perform filtering as soon as possible so as to cut down on the number of intermediate frames generated.

Section 4.4.3において `not` と `lisp-value` がクエリ言語に対しもしこれらのフィルタリング命令が変数が束縛されていないフレームに適用された場合に“間違った”回答を与えることがあることを学んだ。この欠陥を直す方法を工夫せよ。1 つの考えはフィルタリングを“遅延”の様式で実行することだ。フレームに“プロミス”を追加することで十分な変数がその操作を可能にする場合にのみそれを果たすようにする。フィルタリングの実行は全ての他の命令

が実行を終えるまで待つことができる。しかし、効率のために生成される中間フレームの数を削減できるようフィルタリングができるだけ早く実行したい。

**Exercise 4.78:** Redesign the query language as a nondeterministic program to be implemented using the evaluator of Section 4.3, rather than as a stream process. In this approach, each query will produce a single answer (rather than the stream of all answers) and the user can type `try-again` to see more answers. You should find that much of the mechanism we built in this section is subsumed by nondeterministic search and backtracking. You will probably also find, however, that your new query language has subtle differences in behavior from the one implemented here. Can you find examples that illustrate this difference?

クエリ言語をストリーム処理ではなく非決定性プログラムとして、Section 4.3の評価機を用いて実装されるように再設計せよ。この取り組み方においては、各クエリは(全ての回答のストリームではなく)単一の回答を生成し、ユーザは`try-again`を入力することでより多くの回答を見ることができる。この節で構築した仕組みの多くは非決定性探索とバックトラックにより組込まれていることに気付かなければならない。しかし、新しいクエリ言語の振舞にここで実装されたものからわずかな違うことにも気付くだろう。この違いを説明する例を見つけることができるだろうか?

**Exercise 4.79:** When we implemented the Lisp evaluator in Section 4.1, we saw how to use local environments to avoid name conflicts between the parameters of procedures. For example, in evaluating Section 4.1で Lisp 評価機を実装した時に、どのようにローカル環境を使用して手続のパラメタ間の名前衝突を防ぐかについて学んだ。例えば以下を評価する場合において、

```
(define (square x) (* x x))
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(sum-of-squares 3 4)
```

there is no confusion between the `x` in `square` and the `x` in `sum-of-squares`, because we evaluate the body of each procedure in an

environment that is specially constructed to contain bindings for the local variables. In the query system, we used a different strategy to avoid name conflicts in applying rules. Each time we apply a rule we rename the variables with new names that are guaranteed to be unique. The analogous strategy for the Lisp evaluator would be to do away with local environments and simply rename the variables in the body of a procedure each time we apply the procedure.

`square` の `x` と `sum-of-squares` の `x` の間に混乱は無い。なぜなら各手続のボディをローカル変数のための束縛を含めるために特別に構築した環境の中で評価するからである。クエリシステムではルール適用における名前衝突を避けるために異なる戦略を用いた。ルールを適用する度に変数名を唯一であることを保証された新しい名前に変えている。Lisp 評価機に対する同様の戦略はローカルな環境を排除し、手続を適用する度に手続のボディの中の変数を改名することになるだろう。

Implement for the query language a rule-application method that uses environments rather than renaming. See if you can build on your environment structure to create constructs in the query language for dealing with large systems, such as the rule analog of block-structured procedures. Can you relate any of this to the problem of making deductions in a context (e.g., “If I supposed that  $P$  were true, then I would be able to deduce  $A$  and  $B$ .”) as a method of problem solving? (This problem is open-ended. A good answer is probably worth a Ph.D.)

クエリ言語に対して改名でなく、環境を用いるルール適用手法を実装せよ。あなたの環境構造上に巨大システムを取り扱うために、クエリ言語内にブロック構造化された手続に同等なルールのような構成概念を作るために構築できるか確かめよ。これの何かを文脈中での推論を行う問題に、問題解決の手段として関連付けることはできるだろうか？(例えば“もし  $P$  が真であるとするならば、 $A$  と  $B$  を推論することができます”。)

(この問題には明確な回答やルールは存在しない。良い回答は恐らく博士号の価値があるだろう。)

# 5

## Computing with Register Machines

My aim is to show that the heavenly machine is not a kind of divine, live being, but a kind of clockwork (and he who believes that a clock has soul attributes the maker's glory to the work), insofar as nearly all the manifold motions are caused by a most simple and material force, just as all motions of the clock are caused by a single weight.

私の目的は天の機械は神からの授かり物や生き物ではなく、時計仕掛けのような物であることを示すことです。(そして時計が魂を持つと信ずる人はその理由をその仕事に対するメーカーの栄光に帰するでしょう。) それはほとんど全ての多様な運動が最も単純な物質の力により引き起こされるとする限りにおいて、時計の全ての動作が1つの錘により引き起こされるのと全く同じように。

—Johannes Kepler (Herwart von Hohenburgへの手紙, 1605)

We began this book by studying processes and by describing processes in terms of procedures written in Lisp. To explain the meanings of these procedures, we used a succession of models of evaluation: the substitution model of [Chapter 1](#), the environment model of [Chapter 3](#), and the metacircular evaluator of [Chapter 4](#). Our examination of the metacircular evaluator, in particular, dispelled much of the mystery of how Lisp-like languages are interpreted. But even the

metacircular evaluator leaves important questions unanswered, because it fails to elucidate the mechanisms of control in a Lisp system. For instance, the evaluator does not explain how the evaluation of a subexpression manages to return a value to the expression that uses this value, nor does the evaluator explain how some recursive procedures generate iterative processes (that is, are evaluated using constant space) whereas other recursive procedures generate recursive processes. These questions remain unanswered because the metacircular evaluator is itself a Lisp program and hence inherits the control structure of the underlying Lisp system. In order to provide a more complete description of the control structure of the Lisp evaluator, we must work at a more primitive level than Lisp itself.

私達はこの本をプロセスを学ぶことと、プロセスを Lisp で書かれた手続を用いて説明することにより始めました。これらの手続の意味を説明するために、いくつかの評価モデルを用いました。[Chapter 1](#)の置換モデル、[Chapter 3](#)の環境モデル、[Chapter 4](#)のメタ循環評価機です。私達のメタ循環評価機の調査は特に Lisp のような言語がどのように解釈されるのかについての謎の大部分を氷解させました。しかしメタ循環評価機ですらも重要な疑問を未知の状態に残します。Lisp システム中の制御の仕組みは明らかにしないためです。例えば、この評価機は部分式の評価がこの式の値を用いる式にどのようにその値を返すのかについて説明しません。またこの評価機は、ある再帰関数が反復プロセス(つまり、定量的な記憶域で評価されるもの)を生成するのに對し、一方で他の再帰関数が再帰プロセスを生成することもまた説明しません。これらの疑問は未解決のままであります。なぜならメタ循環評価機はそれ自身が Lisp プログラムであり、それ故に根底に存在する Lisp システムの制御構造を引き継ぐためです。より完全な Lisp 評価機の制御構造の説明を与えるためには、Lisp それ自身よりもよりプリミティブなレベルについて取り組まねばなりません。

In this chapter we will describe processes in terms of the step-by-step operation of a traditional computer. Such a computer, or *register machine*, sequentially executes *instructions* that manipulate the contents of a fixed set of storage elements called *registers*. A typical register-machine instruction applies a primitive operation to the contents of some registers and assigns the result to another register. Our descriptions of processes executed by register machines will look very much like “machine-language” programs for traditional computers. However, instead of focusing on the machine language of any particular computer, we will examine several Lisp procedures and design a specific register machine to execute each procedure. Thus, we will approach our task from

the perspective of a hardware architect rather than that of a machine-language computer programmer. In designing register machines, we will develop mechanisms for implementing important programming constructs such as recursion. We will also present a language for describing designs for register machines. In Section 5.2 we will implement a Lisp program that uses these descriptions to simulate the machines we design.

この章ではプロセスを旧来の計算機の個々の操作を用いて説明します。そのような計算機、つまり *register machine*(レジスタマシン) は *registers*(レジスタ) と呼ばれる固定長の記憶要素の集合の中身を操作する *instructions*(命令) を順に実行します。典型的なレジスタマシンの命令はプリミティブな操作をいくつかのレジスタの中身に対して適用し、その結果を他のレジスタに割り当てます。レジスタマシンにより実行されるプロセスの私達の説明は伝統的な計算機向けの“機械語”にとても良く似ているでしょう。しかし、何らかの特定の計算機の機械語に注力する代わりに、私達はいくつかの Lisp 手続を調査し、各手続を実行するための特定のレジスタマシンを設計します。従って私達はこの目的に機械語のコンピュータプログラマではなく、ハードウェアーアーキテクトの視点から取り組みます。レジスタマシンの設計において、私達は再帰のような重要なプログラミング構造を実装するための仕組みを開発します。またレジスタマシンの設計を記述するための言語も与えます。Section 5.2 ではこれらの記述を用いて設計したマシンをシミュレートする Lisp プログラムを実装します。

Most of the primitive operations of our register machines are very simple. For example, an operation might add the numbers fetched from two registers, producing a result to be stored into a third register. Such an operation can be performed by easily described hardware. In order to deal with list structure, however, we will also use the memory operations `car`, `cdr`, and `cons`, which require an elaborate storage-allocation mechanism. In Section 5.3 we study their implementation in terms of more elementary operations.

私達のレジスタマシンのプリミティブな命令の多くはとても簡単です。例えばある命令は 2 つのレジスタから取得した数値を足し、結果を生成して 3 つ目のレジスタに格納します。そのような命令は簡単に記述されたハードウェアにより実行することができます。しかし、リスト構造を取り扱うためにはメモリ操作命令 `car`, `cdr`, `cons` もまた使用します。これは複雑なストレージ(記憶領域)獲得の仕組みを必要とします。Section 5.3 でより初歩的な命令を用いてのそれらの実装について学びます。

In Section 5.4, after we have accumulated experience formulating simple procedures as register machines, we will design a machine that carries out the

algorithm described by the metacircular evaluator of [Section 4.1](#). This will fill in the gap in our understanding of how Scheme expressions are interpreted, by providing an explicit model for the mechanisms of control in the evaluator. In [Section 5.5](#) we will study a simple compiler that translates Scheme programs into sequences of instructions that can be executed directly with the registers and operations of the evaluator register machine.

[Section 5.4](#)ではレジスタマシンによる簡単な手続の形式化についての経験を貯めた後に、[Section 4.1](#)のメタ循環評価機により説明されたアルゴリズムを実行するマシンを設計します。これがSchemeがどのように解釈されるのかについての私達の理解のギャップを、評価機の制御の仕組みに対する明確なモデルを与えることにより、埋めることでしょう。[Section 5.5](#)ではSchemeプログラムを評価機のレジスタマシンのレジスタと命令を用いて直接実行可能な一連の命令に変換する簡単なコンパイラについて学びます。

## 5.1 Designing Register Machines

To design a register machine, we must design its *data paths* (registers and operations) and the *controller* that sequences these operations. To illustrate the design of a simple register machine, let us examine Euclid's Algorithm, which is used to compute the greatest common divisor (GCD) of two integers. As we saw in [Section 1.2.5](#), Euclid's Algorithm can be carried out by an iterative process, as specified by the following procedure:

レジスタマシンを設計するためには、その*data paths*(データパス)(レジスタと命令)とこれらの命令を順序付ける*controller*(コントローラ)を設計する必要があります。簡単なレジスタマシンの設計を説明するために、2つの整数の最大公約数(GCD)を求めるために使用したユークリッドのアルゴリズムを検討しましょう。[Section 1.2.5](#)で学んだように、ユークリッドのアルゴリズムは反復プロセスにて以下の手続にて指定されるように実行することができます。

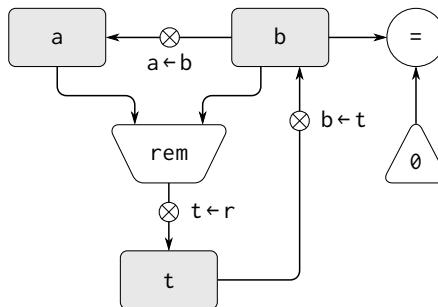
```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

A machine to carry out this algorithm must keep track of two numbers,  $a$  and  $b$ , so let us assume that these numbers are stored in two registers with those

names. The basic operations required are testing whether the contents of register  $b$  is zero and computing the remainder of the contents of register  $a$  divided by the contents of register  $b$ . The remainder operation is a complex process, but assume for the moment that we have a primitive device that computes remainders. On each cycle of the GCD algorithm, the contents of register  $a$  must be replaced by the contents of register  $b$ , and the contents of  $b$  must be replaced by the remainder of the old contents of  $a$  divided by the old contents of  $b$ . It would be convenient if these replacements could be done simultaneously, but in our model of register machines we will assume that only one register can be assigned a new value at each step. To accomplish the replacements, our machine will use a third “temporary” register, which we call  $t$ . (First the remainder will be placed in  $t$ , then the contents of  $b$  will be placed in  $a$ , and finally the remainder stored in  $t$  will be placed in  $b$ .)

このアルゴリズムを実行する機械は2つの数値、 $a$ と $b$ を追跡しなければいけません。そうすることできれらの数値がそれらの名前と共に2つのレジスタに格納されることが推測できます。必要とされる基本的な命令はレジスタ  $b$  の値が0であるかどうかを確認し、レジスタ  $a$  の中身をレジスタ  $b$  の中身で割った余りを求めます。剩余の命令は複雑な処理ですが、当座は剩余を求めるプリミティブな手法が存在すると仮定します。GCDアルゴリズムの各サイクルにおいて、レジスタ  $a$  の中身はレジスタ  $b$  の中身で置き換えられ、レジスタ  $b$  の中身は  $a$  の古い中身を  $b$  の古い中身で割った場合の余りで置き換えられなければなりません。もしこれらの置換が同時に行われれば便利でしょう。しかし私達のレジスタマシンのモデルではただ1つのレジスタのみが各ステップで新しい値を割り当てることができます。置換を達成するためには、私達の機械は3つ目の“temporary”(一時的な) レジスタを使用します。これを  $t$  と呼びます。(最初に剩余は  $t$  に置かれます。次に  $b$  の中身が  $a$  に置かれます。最後に  $t$  に格納されている剩余が  $b$  に置かれます。)

We can illustrate the registers and operations required for this machine by using the data-path diagram shown in [Figure 5.1](#). In this diagram, the registers ( $a$ ,  $b$ , and  $t$ ) are represented by rectangles. Each way to assign a value to a register is indicated by an arrow with an  $X$  behind the head, pointing from the source of data to the register. We can think of the  $X$  as a button that, when pushed, allows the value at the source to “flow” into the designated register. The label next to each button is the name we will use to refer to the button. The names are arbitrary, and can be chosen to have mnemonic value (for example,  $a \leftarrow b$  denotes pushing the button that assigns the contents of register  $b$  to



**Figure 5.1:** Data paths for a GCD machine.

register **a**). The source of data for a register can be another register (as in the  $a \leftarrow b$  assignment), an operation result (as in the  $t \leftarrow r$  assignment), or a constant (a built-in value that cannot be changed, represented in a data-path diagram by a triangle containing the constant).

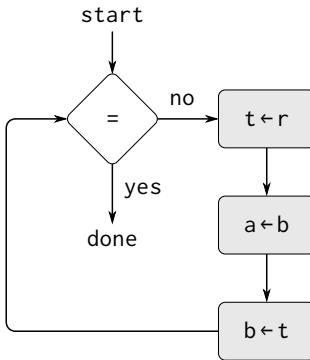
この機械のレジスタと命令をFigure 5.1に示されるデータパス図を用いて説明することができます。この図では、レジスタ (**a**, **b**, **t**) は長方形で表現されます。値をレジスタに割り当てる方向は  $x$  が頭の後ろにあり、データの元からレジスタを指す矢印により示されます。 $x$  は押された時に元の値が指定されたレジスタに“flow”する(流れる)ボタンだと考えることができます。各ボタンの横にあるラベルはそのボタンを参照するのに使われる名前です。この名前は自由で、かつ記憶を助ける値を持つことを選択することができます。(例えば、 $a \leftarrow b$  はボタンを押すとレジスタ **b** の中身を **a** に割り当てるることを意味します)。レジスタに対するデータ元は別のレジスタであることも可能で ( $a \leftarrow b$  の代入のように)、また ( $t \leftarrow r$  の代入のように) 命令の結果や、定数(変更できない組込の値、データパス図では定数を持つ三角形で表現される)にもなり得ます。

An operation that computes a value from constants and the contents of registers is represented in a data-path diagram by a trapezoid containing a name for the operation. For example, the box marked **rem** in Figure 5.1 represents an operation that computes the remainder of the contents of the registers **a** and **b** to which it is attached. Arrows (without buttons) point from the input registers and constants to the box, and arrows connect the operation's output value to registers. A test is represented by a circle containing a name for the test. For

example, our GCD machine has an operation that tests whether the contents of register  $b$  is zero. A test also has arrows from its input registers and constants, but it has no output arrows; its value is used by the controller rather than by the data paths. Overall, the data-path diagram shows the registers and operations that are required for the machine and how they must be connected. If we view the arrows as wires and the X buttons as switches, the data-path diagram is very like the wiring diagram for a machine that could be constructed from electrical components.

定数とレジスタの中身から値を求める命令はデータパス図では命令に対する名前を持つ台形により表現されます。例えばFigure 5.1で `rem` と印された箱はそれに取り付けられたレジスタ  $a$  と  $b$  の中身の剰余を求める命令を表します。ボタンの無い矢印は入力レジスタと定数から箱へと指し、別の矢印は命令の出力値からレジスタへと接続しています。テストはそのテストを表す名前を持つ円で表現されます。例えば、私達の GCD マシンはレジスタ  $b$  の中身がゼロであるかをテストする命令を持ちます。テストはまたその入力レジスタと定数からの矢印を持ちます。しかし出力の矢印を持ちません。その値はデータパスでなくコントローラにより使用されます。全体としては、データパス図は機械にとって必要とされるレジスタと命令と、それらがどのように接続されるべきかを示しています。もし私達が矢印を配線に、X ボタンをスイッチだと見れば、データパス図は電子部品から構築することができる機械の配線図にとても似ています。

In order for the data paths to actually compute GCDs, the buttons must be pushed in the correct sequence. We will describe this sequence in terms of a controller diagram, as illustrated in Figure 5.2. The elements of the controller diagram indicate how the data-path components should be operated. The rectangular boxes in the controller diagram identify data-path buttons to be pushed, and the arrows describe the sequencing from one step to the next. The diamond in the diagram represents a decision. One of the two sequencing arrows will be followed, depending on the value of the data-path test identified in the diamond. We can interpret the controller in terms of a physical analogy: Think of the diagram as a maze in which a marble is rolling. When the marble rolls into a box, it pushes the data-path button that is named by the box. When the marble rolls into a decision node (such as the test for  $b = 0$ ), it leaves the node on the path determined by the result of the indicated test. Taken together, the data paths and the controller completely describe a machine for computing GCDs. We start the controller (the rolling marble) at



**Figure 5.2:** Controller for a GCD machine.

the place marked `start`, after placing numbers in registers `a` and `b`. When the controller reaches `done`, we will find the value of the GCD in register `a`.

データパスに対し実際に GCD を求めるためには、複数のボタンが正しい順序で押される必要があります。私達はこの順序を Figure 5.2 で図示されるコントローラ図を用いて説明します。コントローラ図の要素はデータパスのコンポーネントがどのように操作されるべきかであるかを示します。コントローラ図の長方形の箱は押されるべきデータパスのボタンを判別します。そして矢印はあるステップから次への順を示します。図の中のひし形は選択を表現します。ひし形内で確認されたデータパスのテストの値に依存し、2つの順路矢印の1つに従います。私達はコントローラを物質的なアナロジーを用いて解釈することができます。この図をビー玉が転がっている迷路だと考えることができます。ビー玉が箱に転がり込んだ時に、箱により名付けられたデータパスボタンを押します。ビー玉が ( $b = 0$  のテストのような) 決断点に転がり込んだ時には、示されたテストの結果により決定された道に乗りその点を去ります。これらをもとに、データパスとコントローラは完全に GCD を求めるための機械を説明します。私達はコントローラ (転がるビー玉) を `start` と印された地点から、レジスタ `a` と `b` に数値を置いてから開始します。コントローラが `done` に辿り着いた時、GCD の値はレジスタ `a` の中に見つかります。

**Exercise 5.1:** Design a register machine to compute factorials using the iterative algorithm specified by the following procedure. Draw

data-path and controller diagrams for this machine.

以下の手続で指定される反復アルゴリズムを用いて階乗を求めるレジスタマシンを設計せよ。このマシンに対するデータパスとコントローラの図を描け。

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

### 5.1.1 A Language for Describing Register Machines

Data-path and controller diagrams are adequate for representing simple machines such as GCD, but they are unwieldy for describing large machines such as a Lisp interpreter. To make it possible to deal with complex machines, we will create a language that presents, in textual form, all the information given by the data-path and controller diagrams. We will start with a notation that directly mirrors the diagrams.

データパスとコントローラの図は GCD の様な簡単な機械を表現するには適切です。しかしそれらは Lisp インタプリタのような大きな機械を記述するには扱いにくい物です。複雑な機械を扱うことを可能にするために、私達はテキスト形式でデータパスとコントローラの図により与えられる全ての情報を表現する言語を作成することにします。まずは直接図を写し取る表記法から始めます。

We define the data paths of a machine by describing the registers and the operations. To describe a register, we give it a name and specify the buttons that control assignment to it. We give each of these buttons a name and specify the source of the data that enters the register under the button's control. (The source is a register, a constant, or an operation.) To describe an operation, we give it a name and specify its inputs (registers or constants).

機械のデータパスをレジスタと命令を記述することにより定義します。レジスタを記述するために、それに名前を与え、それに対する代入をコントロールするボタンを指定します。これらのボタン全てに名前を与え、ボタンのコントロールの下にレジスタに入れられるデータの代入元を指定します。(代入元

はレジスタ、定数、または命令です)。命令を記述するために、それに名前を与え、その入力(レジスタ、または定数)を指定します。

We define the controller of a machine as a sequence of *instructions* together with *labels* that identify *entry points* in the sequence. An instruction is one of the following:

機械のコントローラを *instructions*(命令) の列として、その列の *entry points*(エントリポイント、入口) を特定する *labels*(ラベル) と共に定義します。

- The name of a data-path button to push to assign a value to a register. (This corresponds to a box in the controller diagram.)

レジスタに値を割り当てるために押すデータパスボタンの名前。(これはコントローラ図の箱に対応する)

- A **test** instruction, that performs a specified test.

**test**(テスト) 命令、特定のテストを実行する。

- A conditional branch (**branch** instruction) to a location indicated by a controller label, based on the result of the previous test. (The test and branch together correspond to a diamond in the controller diagram.) If the test is false, the controller should continue with the next instruction in the sequence. Otherwise, the controller should continue with the instruction after the label.

直前のテストの結果に基づくコントローララベルにより示された地点への条件分岐(**branch** 命令)。(テストと分岐は共にコントローラ図のひし形に対応する)。もしテストが偽であれば、コントローラは命令列の次の命令へと続ける。そうでなければ、コントローラはラベルの次の命令から続ける。

- An unconditional branch (**goto** instruction) naming a controller label at which to continue execution.

無条件分岐(**goto** 命令)は実行を続ける地点にコントローララベルを名付ける

The machine starts at the beginning of the controller instruction sequence and stops when execution reaches the end of the sequence. Except when a branch changes the flow of control, instructions are executed in the order in which they are listed.

機械はコントローラの命令列の初めから開始し、列の終わりに辿り付いた時に実行を停止する。ただし分岐が制御の流れを変更した場合、命令はそれが並べられた順に向かい実行される。

**Figure 5.3:** ↓ A specification of the GCD machine.

```
(data-paths
  (registers
    ((name a)
      (buttons ((name a<-b) (source (register b))))))
    ((name b)
      (buttons ((name b<-t) (source (register t))))))
    ((name t)
      (buttons ((name t<-r) (source (operation rem)))))))
  (operations
    ((name rem) (inputs (register a) (register b)))
    ((name =) (inputs (register b) (constant 0))))
  (controller
    test-b ; label
    (test =) ; test
    (branch (label gcd-done)) ; conditional branch
    (t<-r) ; button push
    (a<-b) ; button push
    (b<-t) ; button push
    (goto (label test-b)) ; unconditional branch
    gcd-done ; label
```

Figure 5.3 shows the GCD machine described in this way. This example only hints at the generality of these descriptions, since the GCD machine is a very simple case: Each register has only one button, and each button and test is used only once in the controller.

Figure 5.3はこの方法で記述された GCD マシンを示します。この例はこれらの記述の一般性を暗示しているに過ぎません。GCD マシンはとても単純な場合であるからです。各レジスタはたった 1 つのボタンしか持たず、各ボタンとテストはコントローラによりただ 1 度しか利用されていません。

Unfortunately, it is difficult to read such a description. In order to understand the controller instructions we must constantly refer back to the definitions of the button names and the operation names, and to understand what the buttons do we may have to refer to the definitions of the operation names. We will thus transform our notation to combine the information from the data-path and controller descriptions so that we see it all together.

残念なことに、このような記述を読むことは難しいことです。コントローラの命令を理解するためには、常にボタンの名前と命令の名前の定義に戻らねばならず、またボタンが何をするのか理解するためには命令の名前の定義を参照する必要があるでしょう。従って私達はこの表記法を変形し、データパスとコントローラの記述からの情報を組み合わせることで全てと一緒に見られるようになります。

To obtain this form of description, we will replace the arbitrary button and operation names by the definitions of their behavior. That is, instead of saying (in the controller) “Push button  $t \leftarrow r$ ” and separately saying (in the data paths) “Button  $t \leftarrow r$  assigns the value of the `rem` operation to register  $t$ ” and “The `rem` operation’s inputs are the contents of registers  $a$  and  $b$ ,” we will say (in the controller) “Push the button that assigns to register  $t$  the value of the `rem` operation on the contents of registers  $a$  and  $b$ .” Similarly, instead of saying (in the controller) “Perform the  $=$  test” and separately saying (in the data paths) “The  $=$  test operates on the contents of register  $b$  and the constant 0,” we will say “Perform the  $=$  test on the contents of register  $b$  and the constant 0.” We will omit the data-path description, leaving only the controller sequence. Thus, the GCD machine is described as follows:

記述のこの形式を得るために、自由裁量なボタンと命令の名前をそれらの振舞の定義により置き換えます。つまり、(コントローラの中で)“ボタン  $t \leftarrow r$  を押せ”と言い、別に(データバスの中で)“ボタン  $t \leftarrow r$  は `rem` 命令の値をレジスタ  $t$  に代入”と“`rem` 命令の入力はレジスタ  $a$  と  $b$  の中身”と言う代わりに、これからは(コントローラの中で)“レジスタ  $a$  と  $b$  の中身上での `rem` 命令の値をレジスタ  $t$  に代入するボタンを押せ”と言うことにします。同様に、(コントローラの中で) $=$  テストを実行せよ”と言い、別に(データバスの中で) $=$  テストはレジスタ  $b$  の中身と定数 0 の上で動作する”と言う代わりに、これからは“ $=$  テストをレジスタ  $b$  の中と定数 0 の上で実行せよ”と言います。データバスの記述は省略し、コントローラの命令列のみを残します。従って、GCD マシンは以下のように記述されます。

```
(controller
  test-b
  (test (op =) (reg b) (const 0))
  (branch (label gcd-done))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t)))
```

```
(goto (label test-b))  
gcd-done)
```

This form of description is easier to read than the kind illustrated in Figure 5.3, but it also has disadvantages:

この記述の形式はFigure 5.3で説明されたものよりも読み易いでしょう。しかし同時に欠点も持ちます。

- It is more verbose for large machines, because complete descriptions of the data-path elements are repeated whenever the elements are mentioned in the controller instruction sequence. (This is not a problem in the GCD example, because each operation and button is used only once.) Moreover, repeating the data-path descriptions obscures the actual data-path structure of the machine; it is not obvious for a large machine how many registers, operations, and buttons there are and how they are interconnected.

大きな機械に対してはより冗長である。データパス要素の複雑な記述がその要素がコントローラ命令列内で触れられる度に繰り返されるため。(これは GCD の例では問題にならない。命令とボタンのそれぞれがただ 1 度しか使用されないため)。さらに、データパス記述の繰り返しが実際の機械のデータパス構造を分かりにくくする。大きな機械にとっていくつのレジスタ、命令、ボタンが存在し、それらがどのように相互接続されているのかは自明では無い。

- Because the controller instructions in a machine definition look like Lisp expressions, it is easy to forget that they are not arbitrary Lisp expressions. They can notate only legal machine operations. For example, operations can operate directly only on constants and the contents of registers, not on the results of other operations.

機械の定義内のコントローラの命令は Lisp 式の様に見えるため、それらが自由裁量な Lisp 式ではないことを簡単に忘れてしまう。それらは正式な機械の命令のみを記述できる。例えば、命令は直接には定数とレジスタの中身のみに対して操作ができる。他の命令の結果に対してはできない。

In spite of these disadvantages, we will use this register-machine language throughout this chapter, because we will be more concerned with understanding controllers than with understanding the elements and connections in data

paths. We should keep in mind, however, that data-path design is crucial in designing real machines.

これらの欠点にも係らず、私達はこのレジスタマシンの言語をこの章を通して使用します。データパスの要素と接続を理解することよりもコントローラを理解することにより関係していくためです。しかし、私達はデータパスの設計は実際の機械の設計において、とても重要であることを肝に命じておかねばなりません。

**Exercise 5.2:** Use the register-machine language to describe the iterative factorial machine of [Exercise 5.1](#).

レジスタマシン言語を用いて[Exercise 5.1](#)の反復階乗機械を記述せよ。

## Actions

Let us modify the GCD machine so that we can type in the numbers whose GCD we want and get the answer printed at our terminal. We will not discuss how to make a machine that can read and print, but will assume (as we do when we use `read` and `display` in Scheme) that they are available as primitive operations.<sup>1</sup>

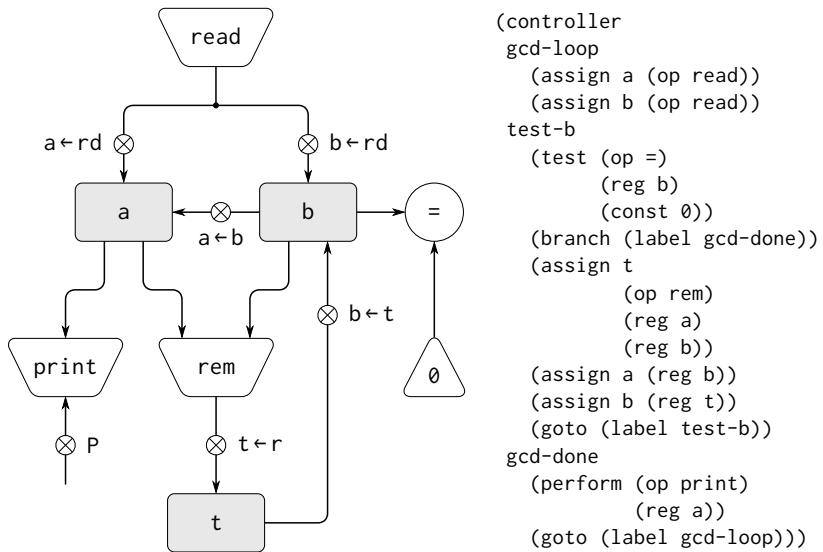
GCD マシンを変更して、GCD が欲しい数値を入力し、端末に答が表示されるようにしてみましょう。私達は読み込みや表示ができる機械をどのように作るかについては議論しません。しかし (私達が Scheme にて `read` と `display` を使う時に使うように) それらがプリミティブな命令として既に存在すると仮定します。<sup>2</sup>

`Read` is like the operations we have been using in that it produces a value that can be stored in a register. But `read` does not take inputs from any registers; its value depends on something that happens outside the parts of the machine we are designing. We will allow our machine's operations to have such behavior, and thus will draw and notate the use of `read` just as we do any other operation that computes a value.

---

<sup>1</sup>This assumption glosses over a great deal of complexity. Usually a large portion of the implementation of a Lisp system is dedicated to making reading and printing work.

<sup>2</sup>この仮定は多量の複雑さを言い繕っています。通常、Lisp システムの実装の大きな部分が読み込みと表示を可能にすることに関してささげられています。



**Figure 5.4:** A GCD machine that reads inputs and prints results.

`read` は私達が使用してきた、その中でレジスタに格納することができる値を生成する命令のような物です。しかし `read` は入力をどのレジスタからも取得しません。その値は私達が設計している機械の外側の部品で起こる何かに依存しています。私達は私達の機械の命令にそのようは振舞を持つことを許します。従って `read` の使用を描き、記述することを他の任意の値を求める命令と全く同様に行います。

`Print`, on the other hand, differs from the operations we have been using in a fundamental way: It does not produce an output value to be stored in a register. Though it has an effect, this effect is not on a part of the machine we are designing. We will refer to this kind of operation as an *action*. We will represent an action in a data-path diagram just as we represent an operation that computes a value—as a trapezoid that contains the name of the action. Arrows point to the action box from any inputs (registers or constants). We also associate a button with the action. Pushing the button makes the action

happen. To make a controller push an action button we use a new kind of instruction called `perform`. Thus, the action of printing the contents of register `a` is represented in a controller sequence by the instruction

一方で、`print` は私達が使用してきた命令とは基本的な意味において異なります。これはレジスタに格納できる出力の値を生成しません。この種の命令は *action*(アクション) として参照することにします。データパス図ではアクションは値を求める命令と同じように、アクションの名前を含む台形として表現します。矢印は任意の入力(レジスタ、または定数)からアクションの箱へと指します。またボタンをアクションと関連付けることもあります。ボタンを押すとアクションが起ります。コントローラにアクションボタンを押させるために、`perform`(パフォーム、実行) と呼ばれる新しい種類の命令を用います。従ってレジスタ `a` の中身を表示するアクションはコントローラの命令列の中でその命令により表現されます。

```
(perform (op print) (reg a))
```

Figure 5.4 shows the data paths and controller for the new GCD machine. Instead of having the machine stop after printing the answer, we have made it start over, so that it repeatedly reads a pair of numbers, computes their GCD, and prints the result. This structure is like the driver loops we used in the interpreters of Chapter 4.

Figure 5.4 は新しい GCD マシンのデータパスとコントローラを示しています。回答を表示した後にマシンをストップさせる代わりに、再開させています。そのため数値のペアを読み込み、それらの GCD を計算し、結果を表示することを繰り返します。この構造は Chapter 4 のインタプリタにて使用したドライバ ループに似ています。

### 5.1.2 Abstraction in Machine Design

We will often define a machine to include “primitive” operations that are actually very complex. For example, in Section 5.4 and Section 5.5 we will treat Scheme’s environment manipulations as primitive. Such abstraction is valuable because it allows us to ignore the details of parts of a machine so that we can concentrate on other aspects of the design. The fact that we have swept a lot of complexity under the rug, however, does not mean that a machine design is unrealistic. We can always replace the complex “primitives” by simpler primitive operations.

私達はこれから頻繁に、実際には複雑な“プリミティブな”命令を含む機械を定義します。例えば Section 5.4 と Section 5.5 では Scheme の環境の操作をプリミティブとして扱います。そのような抽象化はそれにより機械の部品の詳細を無視することを可能にし、設計の他の側面に集中することを可能にするため有益です。しかし、私達が数多くの複雑さを敷物の下に隠してしまった事実は機械設計が非現実的であることを意味しません。私達は常に複雑な“プリミティブ”をより簡単なプリミティブな命令で置き換えることができます。

Consider the GCD machine. The machine has an instruction that computes the remainder of the contents of registers  $a$  and  $b$  and assigns the result to register  $t$ . If we want to construct the GCD machine without using a primitive remainder operation, we must specify how to compute remainders in terms of simpler operations, such as subtraction. Indeed, we can write a Scheme procedure that finds remainders in this way:

GCD マシンについて考えます。マシンはレジスタ  $a$  と  $b$  の中身の剩余を求める、結果をレジスタ  $t$  に割り当てる命令を持ちます。もし GCD マシンをプリミティブな剩余命令を使用すること無しに構築したい場合、より単純な命令、例えば引き算を用いてどのように剩余を求めるのかを指定しなければなりません。実際に、この方法で剩余を見つける Scheme の手続を描くことができます。

```
(define (remainder n d)
  (if (< n d) n (remainder (- n d) d)))
```

We can thus replace the remainder operation in the GCD machine's data paths with a subtraction operation and a comparison test. Figure 5.5 shows the data paths and controller for the elaborated machine. The instruction

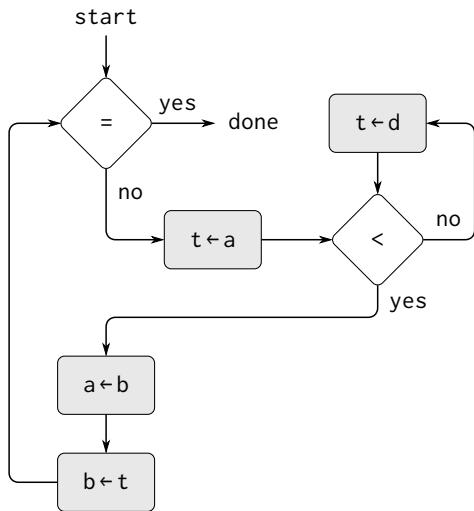
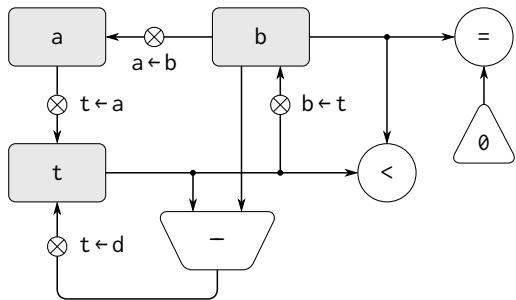
従って GCD マシンのデータパス内の剩余命令を引き算命令と比較テストで置き換えることができます。Figure 5.5 は緻密化されたマシンのデータパスとコントローラを示します。GCD コントローラ定義内の以下の命令は、

```
(assign t (op rem) (reg a) (reg b))
```

in the GCD controller definition is replaced by a sequence of instructions that contains a loop, as shown in Figure 5.6.

Figure 5.6 に示されるように、ループを含む一連の命令により置き換えることができます。

**Figure 5.6:** ↓ Controller instruction sequence for the GCD machine in Figure 5.5.



**Figure 5.5:** Data paths and controller for the elaborated GCD machine.

```

(controller test-b
    (test (op =) (reg b) (const 0))
    (branch (label gcd-done))
    (assign t (reg a))
  rem-loop
    (test (op <) (reg t) (reg b))
    (branch (label rem-done))
    (assign t (op -) (reg t) (reg b))
    (goto (label rem-loop))
  rem-done
    (assign a (reg b))
    (assign b (reg t))
    (goto (label test-b))
  gcd-done)

```

**Exercise 5.3:** Design a machine to compute square roots using Newton's method, as described in [Section 1.1.7](#):

平方根を求める機械を[Section 1.1.7](#)で説明されたようにニュートン法を用いて設計せよ。

```

(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))

```

Begin by assuming that `good-enough?` and `improve` operations are available as primitives. Then show how to expand these in terms of arithmetic operations. Describe each version of the `sqrt` machine design by drawing a data-path diagram and writing a controller definition in the register-machine language.

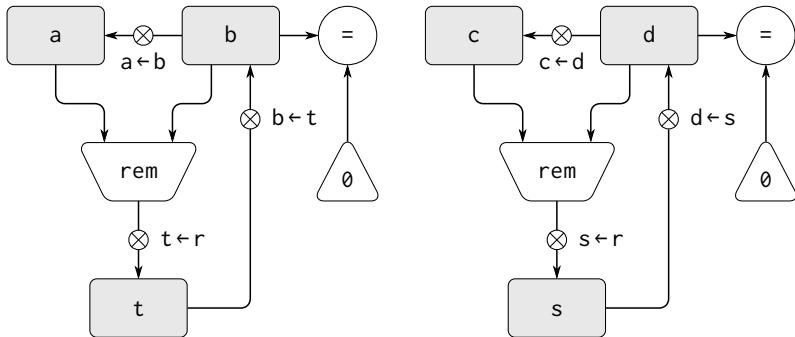
`good-enough?` と `improve` 命令はプリミティブとして存在するとして始めよ。次にこれらを算術演算子を用いてどのように展開するか示せ。`sqrt` マシン設計の各版をデータパス図を描き、レジスタマシンのコントローラ定義を記述することで説明せよ。

### 5.1.3 Subroutines

When designing a machine to perform a computation, we would often prefer to arrange for components to be shared by different parts of the computation rather than duplicate the components. Consider a machine that includes two GCD computations—one that finds the GCD of the contents of registers `a` and `b` and one that finds the GCD of the contents of registers `c` and `d`. We might start by assuming we have a primitive `gcd` operation, then expand the two instances of `gcd` in terms of more primitive operations. Figure 5.7 shows just the GCD portions of the resulting machine’s data paths, without showing how they connect to the rest of the machine. The figure also shows the corresponding portions of the machine’s controller sequence.

演算を実行する機械を設計する時、私達は良くコンポーネントを複製するのではなく、演算の異なる部品により共有されるコンポーネントを準備することを好みます。2つのGCD演算を含む機械について考えてみましょう。1つはレジスタ `a` と `b` の中身の GCD を求め、もう1つはレジスタ `c` と `d` の GCD を求めます。私達はまずプリミティブな `gcd` 命令を持つと仮定することから始め、次に2つの `gcd` のインスタンスをよりプリミティブな命令を用いて展開するでしょう。Figure 5.7は結果としての機械のデータパスの GCD の部分を、それらが機械の残りの部分にどのように接続されていかを除いて示しています。この図はまた機械のコントローラシーケンス(命令列)の対応する部分も示しています。

This machine has two remainder operation boxes and two boxes for testing equality. If the duplicated components are complicated, as is the remainder box, this will not be an economical way to build the machine. We can avoid duplicating the data-path components by using the same components for both GCD computations, provided that doing so will not affect the rest of the larger machine’s computation. If the values in registers `a` and `b` are not needed by the time the controller gets to `gcd-2` (or if these values can be moved to other registers for safekeeping), we can change the machine so that it uses registers



```

gcd-1
(test (op =) (reg b) (const 0))
(branch (label after-gcd-1))
(assign t (op rem) (reg a) (reg b))
(assign a (reg b))
(assign b (reg t))
(goto (Label gcd-1))
after-gcd-1

gcd-2
(test (op =) (reg d) (const 0))
(branch (label after-gcd-2))
(assign s (op rem) (reg c) (reg d))
(assign c (reg d))
(assign d (reg s))
(goto (Label gcd-2))
after-gcd-2

```

**Figure 5.7:** Portions of the data paths and controller sequence for a machine with two GCD computations.

a and b, rather than registers c and d, in computing the second GCD as well as the first. If we do this, we obtain the controller sequence shown in Figure 5.8.

この機械は 2 つの剰余命令の箱と 2 つの等値テストの箱を持っています。もし複製されたコンポーネントが剰余の箱のように複雑なら、これは機械を構築するのに経済的な方法ではありません。私達はより大きな機械の演算に影響を与えないように与えられた場合に、同じコンポーネントを両方の GCD 演算に用いることでデータパスコンポーネントの複製を防ぎます。もしレジスタ a と b の値がコントローラが gcd-2 に取り掛かっている時に必要無いのであれば（またはもしこれらの値が安全のために他のレジスタに移動しておくことができるのならば）、機械を変更し、レジスタ c と d ではなく、レジスタ a と b を 2 つ目の GCD を 1 つ目と同じに求めるおおができます。もしこれを行うなら、Figure 5.8 に示されるコントローラシーケンスを得ます。

**Figure 5.8:** ↓ Portions of the controller sequence for a machine

that uses the same data-path components for two different GCD computations.

```
gcd-1
  (test (op =) (reg b) (const 0))
  (branch (label after-gcd-1))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd-1))
after-gcd-1
  ...
gcd-2
  (test (op =) (reg b) (const 0))
  (branch (label after-gcd-2))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd-2))
after-gcd-2
```

We have removed the duplicate data-path components (so that the data paths are again as in Figure 5.1), but the controller now has two GCD sequences that differ only in their entry-point labels. It would be better to replace these two sequences by branches to a single sequence—a *gcd subroutine*—at the end of which we branch back to the correct place in the main instruction sequence. We can accomplish this as follows: Before branching to `gcd`, we place a distinguishing value (such as 0 or 1) into a special register, `continue`. At the end of the `gcd` subroutine we return either to `after-gcd-1` or to `after-gcd-2`, depending on the value of the `continue` register. Figure 5.9 shows the relevant portion of the resulting controller sequence, which includes only a single copy of the `gcd` instructions.

私達はデータパスコンポーネントの複製を削除しました。(そうすることでデータパスは Figure 5.1 の状態に戻りました)。しかしコントローラは今ではそれらのエントリポイントのラベルのみが異なる 2 つの GCD シーケンスを持ちます。これら 2 つのシーケンスを 1 つのシーケンス—*gcd subroutine*(サブルーチン)—への分岐により置き換えたほうが良くなるでしょう。サブルーチンの

終わりにメインの命令列の正しい場所へと戻ります。これを次のように達成することができます。`gcd` に分岐する前に、(0 か 1 のような) 識別するための値を特別なレジスタ、`continue` に置きます。[Figure 5.9](#)は結果としてのコントローラシーケンスの関連する部分を示しています。これはただ 1 つの `gcd` 命令列のコピーを含みます。

**Figure 5.9:** ↓ Using a `continue` register to avoid the duplicate controller sequence in [Figure 5.8](#).

```
gcd
  (test (op =) (reg b) (const 0))
  (branch (label gcd-done))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd))

gcd-done
  (test (op =) (reg continue) (const 0))
  (branch (label after-gcd-1))
  (goto (label after-gcd-2))
  ...
;; Before branching to gcd from the first place where
;; it is needed, we place 0 in the continue register
;; gcd を必要とする場所からその場所へと分岐する前に
;; レジスタ continue に 0 を置く
  (assign continue (const 0))
  (goto (label gcd))

after-gcd-1
  ...
;; Before the second use of gcd, we place 1
;; in the continue register
;; gcd の二度目の使用の前にはレジスタ continue に 1 を置く
  (assign continue (const 1))
  (goto (label gcd))

after-gcd-2
```

This is a reasonable approach for handling small problems, but it would be awkward if there were many instances of GCD computations in the controller sequence. To decide where to continue executing after the `gcd` subroutine, we would need tests in the data paths and branch instructions in the controller for all the places that use `gcd`. A more powerful method for implementing subroutines is to have the `continue` register hold the label of the entry point in the controller sequence at which execution should continue when the subroutine is finished. Implementing this strategy requires a new kind of connection between the data paths and the controller of a register machine: There must be a way to assign to a register a label in the controller sequence in such a way that this value can be fetched from the register and used to continue execution at the designated entry point.

これは小さな問題に対応するのに妥当な取り組み方です。しかもし数多くの GCD 演算がコントローラシーケンスの中にある場合には困ったことになります。GCD サブルーチンの後に実行をどこで続けるかを決定するために、データパス内のテストとコントローラ内に分岐命令が GCD を置く全ての場所に対して必要となるでしょう。サブルーチンを実装するためのより強力な手法は、`continue` レジスタにサブルーチンが終了した時に実行が続行しなければならない場所のコントローラシーケンス内のエントリポイントのラベルを持たせることです。この戦略の実装にはレジスタマシンのデータパスとコントローラの間に新しい種類のコネクションが必要です。ラベルの値をレジスタから取得し指定されたエントリポイントから実行を再開するのに使用できるような方法のため、レジスタにコントローラシーケンス内のラベルを代入するための方法が必要です。

To reflect this ability, we will extend the `assign` instruction of the register-machine language to allow a register to be assigned as value a label from the controller sequence (as a special kind of constant). We will also extend the `goto` instruction to allow execution to continue at the entry point described by the contents of a register rather than only at an entry point described by a constant label. Using these new constructs we can terminate the `gcd` subroutine with a branch to the location stored in the `continue` register. This leads to the controller sequence shown in Figure 5.10.

この能力を反映するために、レジスタマシン言語の `assign` 命令を拡張し、レジスタに値としてラベルをコントローラシーケンスから（特別な種類の中身として）代入することを許可する拡張を行います。また `goto` 命令にも静的ラベルにより記述されたエントリポイントのみでなく、レジスタの中により表

されたエントリポイントから実行を続行することを許可する拡張を行います。これらの新しい構造物を用いることで、`continue` レジスタ内に格納された場所に分岐することにより、`gcd` サブルーチンを停止することができます。これはFigure 5.10に示されたコントローラシーケンスへと導きます。

**Figure 5.10:** ↓ Assigning labels to the `continue` register simplifies and generalizes the strategy shown in Figure 5.9.

```
gcd
  (test (op =) (reg b) (const 0))
  (branch (label gcd-done))
  (assign t (op rem) (reg a) (reg b))
  (assign a (reg b))
  (assign b (reg t))
  (goto (label gcd))
gcd-done
  (goto (reg continue))
  ...
;; Before calling gcd, we assign to continue
;; the label to which gcd should return.
;; gcd を呼ぶ前に、continue に gcd が戻るべきラベルを代入します
  (assign continue (label after-gcd-1))
  (goto (label gcd))
after-gcd-1
  ...
;; Here is the second call to gcd,
;; with a different continuation.
;; 異なる継続を持つ 2 つ目の gcd 呼出
  (assign continue (label after-gcd-2))
  (goto (label gcd))
after-gcd-2
```

A machine with more than one subroutine could use multiple continuation registers (e.g., `gcd-continue`, `factorial-continue`) or we could have all subroutines share a single `continue` register. Sharing is more economical, but we must be careful if we have a subroutine (`sub1`) that calls another subroutine

(`sub2`). Unless `sub1` saves the contents of `continue` in some other register before setting up `continue` for the call to `sub2`, `sub1` will not know where to go when it is finished. The mechanism developed in the next section to handle recursion also provides a better solution to this problem of nested subroutine calls.

複数のサブルーチンを持つマシンは複数の継続レジスタ (例えば `gcd-continue`, `factorial-continue`) を用いるか、または全てのサブルーチンが单一の `continue` レジスタを共有することができるでしょう。共有はより経済的ですが、別のサブルーチン (`sub2`) を呼び出すサブルーチン (`sub1`) を持っていないか注意しなければなりません。`sub1` が `continue` の中身を何か他のレジスタに、`continue` を `sub2` の呼出のために設定する前に保存しなければ、`sub1` は完了した時点でどこに行けば良いのか知ることができません。次の節で開発される再帰を扱う仕組みはこの入れ子のサブルーチン呼出の問題にもより良い解法を提供します。

### 5.1.4 Using a Stack to Implement Recursion

With the ideas illustrated so far, we can implement any iterative process by specifying a register machine that has a register corresponding to each state variable of the process. The machine repeatedly executes a controller loop, changing the contents of the registers, until some termination condition is satisfied. At each point in the controller sequence, the state of the machine (representing the state of the iterative process) is completely determined by the contents of the registers (the values of the state variables).

ここまでに説明されたアイデアを用いて、そのプロセスの各状態変数に対応するレジスタを持つレジスタマシンを指定することにより、任意の反復プロセスを実装することができます。この機械はレジスタの中身を変更しながら、繰り返しコントローラのループを、ある停止条件が満たされるまで実行します。コントローラシーケンスの各地点において、(反復プロセスの状態を表現する) 機械の状態はレジスタの状態 (状態変数の値) により完全に決定されます。

Implementing recursive processes, however, requires an additional mechanism. Consider the following recursive method for computing factorials, which we first examined in [Section 1.2.1](#):

しかし、再帰プロセスを実装する場合には追加の仕組みを必要とします。以下の階乗を求めるための再帰手法について考えましょう。これは [Section 1.2.1](#) で最初に調査しました。

```
(define (factorial n)
  (if (= n 1) 1 (* (factorial (- n 1)) n)))
```

As we see from the procedure, computing  $n!$  requires computing  $(n - 1)!$ . Our GCD machine, modeled on the procedure

この手続から見てとれるように、 $n!$  の演算は  $(n - 1)!$  の演算を必要とします。私達の GCD は以下の手続からモデル化されていますが、

```
(define (gcd a b)
  (if (= b 0) a (gcd b (remainder a b))))
```

similarly had to compute another GCD. But there is an important difference between the `gcd` procedure, which reduces the original computation to a new GCD computation, and `factorial`, which requires computing another factorial as a subproblem. In GCD, the answer to the new GCD computation is the answer to the original problem. To compute the next GCD, we simply place the new arguments in the input registers of the GCD machine and reuse the machine's data paths by executing the same controller sequence. When the machine is finished solving the final GCD problem, it has completed the entire computation.

同様に別の GCD を求める必要があります。しかし、元の演算から新しい GCD 演算へと簡約する GCD 手續と、部分問題として別の階乗を求める必要がある `factorial` の間には重要な違いがあります。GCDにおいては新しい GCD 演算に対する答は元の問題の答です。次の GCD を求めるためには、単純に新しい引数を GCD マシンの入力レジスタに置き、機械のデータバスを同じコントローラーシーケンスを実行することにより再利用します。機械が最後の GCD 問題を解くことを完了した時には、演算全体を完了したことになります。

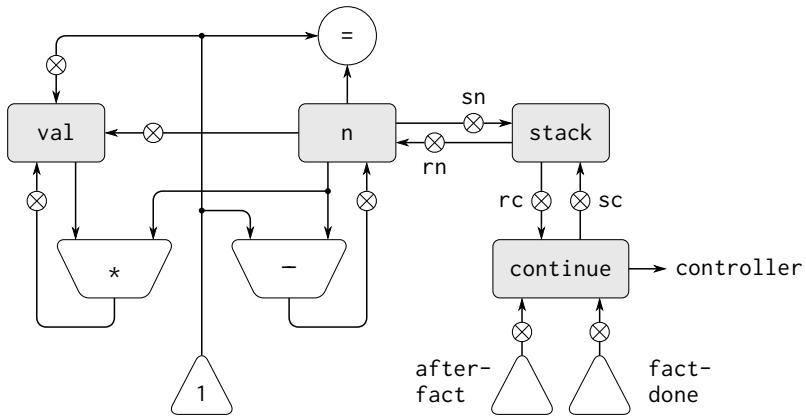
In the case of `factorial` (or any recursive process) the answer to the new factorial subproblem is not the answer to the original problem. The value obtained for  $(n - 1)!$  must be multiplied by  $n$  to get the final answer. If we try to imitate the GCD design, and solve the factorial subproblem by decrementing the `n` register and rerunning the factorial machine, we will no longer have available the old value of `n` by which to multiply the result. We thus need a second factorial machine to work on the subproblem. This second factorial computation itself has a factorial subproblem, which requires a third factorial machine, and so on. Since each factorial machine contains another factorial machine within it, the total machine contains an infinite nest of similar machines and hence cannot be constructed from a fixed, finite number of parts.

階乗の場合（または任意の再帰プロセス）においては新しい階乗の部分問題の回答は元の問題の回答ではありません。 $(n - 1)!$  に対して得られた値は最終回答を得るために  $n$  で乗算しなければなりません。もし GCD の設計を真似し、階乗の部分問題をレジスタ  $n$  をデクリメント（1 引く）し、階乗マシンに戻ることのより解決したいとしても、その結果に乗算を行う有効な古い  $n$  の値は既に存在しません。従って部分問題上で働くための 2 つ目の階乗マシンが必要です。この 2 つ目の階乗の演算はそれ自身が階乗の部分問題を持ち、それは 3 つ目の階乗マシンを必要とし、以下繰り返されます。各階乗マシンがその中に別の階乗マシンを持つため、総計の機械は同様な機械の無限の入れ子を含み、従って固定長の有限数な部品から構築することはできません。

Nevertheless, we can implement the factorial process as a register machine if we can arrange to use the same components for each nested instance of the machine. Specifically, the machine that computes  $n!$  should use the same components to work on the subproblem of computing  $(n - 1)!$ , on the subproblem for  $(n - 2)!$ , and so on. This is plausible because, although the factorial process dictates that an unbounded number of copies of the same machine are needed to perform a computation, only one of these copies needs to be active at any given time. When the machine encounters a recursive subproblem, it can suspend work on the main problem, reuse the same physical parts to work on the subproblem, then continue the suspended computation.

それでもかかわらず、もし機械の各入れ子のインスタンスが同じコンポーネントを使用するように準備ができれば階乗プロセスをレジスタマシンとして実装できます。具体的に言えば、 $n!$  を求める機械は  $(n - 1)!$  を求める部分問題、 $(n - 2)!$  の部分問題、以下繰り返しの仕事に同じコンポーネントを使用せねばなりません。これはもっともらしく見えます。例え階乗プロセスが同じ機械のコピーの未束縛の数値が演算を実行するのに必要だと指図したとしても、これらのコピーのただ 1 つが一度に有効になる必要があるためです。この機械が再帰の部分問題に遭遇した時に、メインの問題上の仕事を中断し、同じ物理部品を部分問題上の仕事に再利用し、そして中断した演算を続けることが可能です。

In the subproblem, the contents of the registers will be different than they were in the main problem. (In this case the  $n$  register is decremented.) In order to be able to continue the suspended computation, the machine must save the contents of any registers that will be needed after the subproblem is solved so that these can be restored to continue the suspended computation. In the case of factorial, we will save the old value of  $n$ , to be restored when we are finished



```

(controller
  (assign continue (label fact-done)) ;set up final return address
fact-loop
  (test (op =) (reg n) (const 1))
  (branch (label base-case))
  ;; Set up for the recursive call by saving n and continue.
  ;; Set up continue so that the computation will continue
  ;; at after-fact when the subroutine returns.
  (save continue)
  (save n)
  (assign n (op -) (reg n) (const 1))
  (assign continue (label after-fact))
  (goto (label fact-loop))
after-fact
  (restore n)
  (restore continue)
  (assign val (op *) (reg n) (reg val)) ;val now contains n(n - 1)!
  (goto (reg continue)) ;return to caller
base-case
  (assign val (const 1)) ;base case: 1! = 1
  (goto (reg continue)) ;return to caller
fact-done)

```

**Figure 5.11:** A recursive factorial machine.

computing the factorial of the decremented  $n$  register.<sup>3</sup>

部分問題の中では、レジスタの中身はメインの問題の中の物と異なります。(この場合にはレジスタ  $n$  はデクリメントされます)。中断された演算を続けることを可能にするために、機械は部分問題が解決した後に必要となる全てのレジスタの中身を保存しなければなりません。そうすることで、中断した演算を続ける時にこれらの値が再格納することができます。階乗の場合には、デクリメントされたレジスタ  $n$  の階乗の演算が完了した時に再格納されるように  $n$  の古い値を保存します。<sup>4</sup>

Since there is no *a priori* limit on the depth of nested recursive calls, we may need to save an arbitrary number of register values. These values must be restored in the reverse of the order in which they were saved, since in a nest of recursions the last subproblem to be entered is the first to be finished. This dictates the use of a *stack*, or “last in, first out” data structure, to save register values. We can extend the register-machine language to include a stack by adding two kinds of instructions: Values are placed on the stack using a **save** instruction and restored from the stack using a **restore** instruction. After a sequence of values has been **saved** on the stack, a sequence of **restores** will retrieve these values in reverse order.<sup>5</sup>

予測可能な限界が入れ子の再帰呼出の深さには存在しないため、任意の数のレジスタ値を保存する必要があるでしょう。これらの値は保存された順の逆順に再格納されねばなりません。入れ子の再帰では突入する最後の部分問題が最初に完了するためです。このことが *stack*(スタック)、つまり“last in, first out”(LIFO、後入れ先出し) データ構造をレジスタ値の保存への使用することを指示しています。レジスタマシン言語を拡張し、2つの種類の命令を追加することでスタックを含めることができます。値はスタックに **save** 命令を用いて置かれて、**restore** 命令を用いてスタックから再格納されます。スタック上に

---

<sup>3</sup>One might argue that we don't need to save the old  $n$ ; after we decrement it and solve the subproblem, we could simply increment it to recover the old value. Although this strategy works for factorial, it cannot work in general, since the old value of a register cannot always be computed from the new one.

<sup>4</sup>古い  $n$  を保存する必要はないと主張する人がいるかもしれません。デクリメントし、部分問題を解決した後に、単純に古い値を回復するためにインクリメントすることができると思われるでしょう。例えこの戦略が階乗に対しては働いたとしても、それは一般的にはうまく行きません。レジスタの古い値が常に新しい値から求められるとは限らないためです。

<sup>5</sup>In Section 5.3 we will see how to implement a stack in terms of more primitive operations.

一連の値が `save` された後に、連続した `restore` がこれらの値を逆順に取り出します。<sup>6</sup>

With the aid of the stack, we can reuse a single copy of the factorial machine's data paths for each factorial subproblem. There is a similar design issue in reusing the controller sequence that operates the data paths. To reexecute the factorial computation, the controller cannot simply loop back to the beginning, as with an iterative process, because after solving the  $(n - 1)!$  subproblem the machine must still multiply the result by  $n$ . The controller must suspend its computation of  $n!$ , solve the  $(n - 1)!$  subproblem, then continue its computation of  $n!$ . This view of the factorial computation suggests the use of the subroutine mechanism described in Section 5.1.3, which has the controller use a `continue` register to transfer to the part of the sequence that solves a subproblem and then continue where it left off on the main problem. We can thus make a factorial subroutine that returns to the entry point stored in the `continue` register. Around each subroutine call, we save and restore `continue` just as we do the `n` register, since each “level” of the factorial computation will use the same `continue` register. That is, the factorial subroutine must put a new value in `continue` when it calls itself for a subproblem, but it will need the old value in order to return to the place that called it to solve a subproblem.

スタックの助けを借りることで階乗マシンの各階乗部分問題のために、データパスの单一のコピーを再利用することができます。同様なデータパスを操作するコントローラシーケンスの再利用についても同様の設計上の問題が存在します。階乗演算を再実行するためには、コントローラは単純には最初に反復プロセスのようにループバックすることはできません。 $(n - 1)!$  を解いた後には機械は依然としてその結果と  $n$  を掛ける必要があるためです。コントローラは  $n!$  の演算を中断し、部分問題  $(n - 1)!$  を解き、そして  $n!$  の演算を続けなければなりません。階乗演算のこの見方はSection 5.1.3で説明されたサブルーチンの仕組みの使用を推奨しており、これはコントローラにレジスタ `continue` を使用させて部分問題を解く列の一部へと移動し、そしてメイン問題を中止した場所から続行します。このようにして `continue` レジスタに格納されたエントリポイントに帰る階乗のサブルーチンを作ることができます。各サブルーチン呼出の周りでは、`continue` を `n` レジスタに行うのと同じように保存し再格納します。階乗演算の各“レベル”が同じ `continue` レジスタを利用するためです。つまり、階乗サブルーチンはそれが自分自身を部分問題として呼び出す時

---

<sup>6</sup>Section 5.3において、よりプリミティブな命令を用いてどのようにスタックを実装するかについて学びます。

に、新しい値を `continue` に設定しなければいけません。しかし部分問題を解くために呼び出した場所に戻るために古い値が必要となるのです。

Figure 5.11 shows the data paths and controller for a machine that implements the recursive `factorial` procedure. The machine has a stack and three registers, called `n`, `val`, and `continue`. To simplify the data-path diagram, we have not named the register-assignment buttons, only the stack-operation buttons (`sc` and `sn` to save registers, `rc` and `rn` to restore registers). To operate the machine, we put in register `n` the number whose factorial we wish to compute and start the machine. When the machine reaches `fact-done`, the computation is finished and the answer will be found in the `val` register. In the controller sequence, `n` and `continue` are saved before each recursive call and restored upon return from the call. Returning from a call is accomplished by branching to the location stored in `continue`. `Continue` is initialized when the machine starts so that the last return will go to `fact-done`. The `val` register, which holds the result of the factorial computation, is not saved before the recursive call, because the old contents of `val` is not useful after the subroutine returns. Only the new value, which is the value produced by the subcomputation, is needed.

Figure 5.11は再帰 `factorial` 手続を実装する機械のためのデータパスとコントローラを示しています。この機械はスタックと 3 つのレジスタ、`n`, `val`, `continue`を持ちます。データパス図を単純化するために、レジスタ代入ボタンには名前を付けず、スタック命令ボタン（レジスタを保存する `sc` と `sn`、レジスタに戻す `rc` と `rn`）のみに付けています。機械を運用するには、レジスタ `n` に階乗を求める数を入れ、それから機械を開始します。機械が `fact-done` に辿り着いた時に演算は完了し、答はレジスタ `val` に見つかります。コントローラシーケンスでは `n` と `continue` が各再帰呼出の前に保存され、その呼出から戻る時に再格納されます。呼出からの復帰は `continue` に格納された場所に分岐することにより達成されます。`continue` は機械が開始した時に最後の復帰が `fact-done` に向かうように初期化されます。階乗演算の結果を持つ `val` レジスタは再帰呼出の前に保存されません。`val` の古い中身はサブルーチンから復帰後には役に立たないためです。部分問題により生成された新しい値のみが必要とされます。

Although in principle the factorial computation requires an infinite machine, the machine in Figure 5.11 is actually finite except for the stack, which is potentially unbounded. Any particular physical implementation of a stack, however, will be of finite size, and this will limit the depth of recursive calls that

can be handled by the machine. This implementation of factorial illustrates the general strategy for realizing recursive algorithms as ordinary register machines augmented by stacks. When a recursive subproblem is encountered, we save on the stack the registers whose current values will be required after the subproblem is solved, solve the recursive subproblem, then restore the saved registers and continue execution on the main problem. The `continue` register must always be saved. Whether there are other registers that need to be saved depends on the particular machine, since not all recursive computations need the original values of registers that are modified during solution of the subproblem (see [Exercise 5.4](#)).

例え原理上は階乗演算が無限の機械を必要とするとしても、[Figure 5.11](#)の機械は実際には限りが無いかもしれないスタックを除けば有限です。しかし、スタックのどんな特定の物理実装も有限のサイズを持ち、このことが機械により扱うことが可能な再帰呼出の深さを制限します。この階乗の実装は再帰アルゴリズムをスタックで容量が増加された通常のレジスタマシンとして実現するための一般的な戦略を説明します。再帰部分問題に遭遇した時にはその現在の値が部分問題が解決された後に必要とされるレジスタをスタック上に保存します。次に再帰部分問題を解決し、保存されたレジスタを戻してメイン問題の実行を続行します。`continue` レジスタは常に保存されなければなりません。保存する必要のあるレジスタが他に存在するかどうかは機械に依存します。全ての再帰演算が部分問題の解決の間に変更されるレジスタの元の値を必要とはしないためです。[\(Exercise 5.4 参照\)](#)。

## A double recursion

Let us examine a more complex recursive process, the tree-recursive computation of the Fibonacci numbers, which we introduced in [Section 1.2.2](#):

より複雑な再帰プロセス、[Section 1.2.2](#)で紹介したフィボナッチ数の木再帰演算について調査してみましょう。

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Just as with factorial, we can implement the recursive Fibonacci computation as a register machine with registers `n`, `val`, and `continue`. The machine is

more complex than the one for factorial, because there are two places in the controller sequence where we need to perform recursive calls—once to compute  $\text{Fib}(n - 1)$  and once to compute  $\text{Fib}(n - 2)$ . To set up for each of these calls, we save the registers whose values will be needed later, set the  $n$  register to the number whose Fib we need to compute recursively ( $n - 1$  or  $n - 2$ ), and assign to `continue` the entry point in the main sequence to which to return (`afterfib-n-1` or `afterfib-n-2`, respectively). We then go to `fib-loop`. When we return from the recursive call, the answer is in `val`. Figure 5.12 shows the controller sequence for this machine.

階乗と同じ様に、再帰フィボナッチ演算をレジスタマシンとしてレジスタ  $n$ , `val`, `continue` と用いて実装することができます。この機械は階乗のものよりも、より複雑です。コントローラシーケンスの中に二箇所の再帰呼出の実行が必要な箇所が存在するためです。一度目は  $\text{Fib}(n - 1)$  を求めるために、二度目は  $\text{Fib}(n - 2)$  を求めるためです。これらの各呼出に準備するために、後にその値が必要となるレジスタを保存し、レジスタ  $n$  に再帰的に求める ( $n - 1$  または  $n - 2$ ) 必要のあるフィボナッチ数を設定します。そして `continue` に戻り先のメインシーケンスのエントリポイント (それぞれ `afterfib-n-1` または `afterfib-n-2`) を割り当てます。そうしたら `fib-loop` へと飛びます。再帰呼出から帰る時には、回答は `val` の中にあります。Figure 5.12はこの機械のためのコントローラシーケンスを示しています。

**Figure 5.12:** Controller for a machine to compute Fibonacci numbers.

```
(controller
  (assign continue (label fib-done))
  fib-loop
    (test (op <) (reg n) (const 2))
    (branch (label immediate-answer))
    ;; Fib( $n - 1$ ) を求める準備
    (save continue)
    (assign continue (label afterfib-n-1))
    (save n) ; n の古い値を保存
    (assign n (op -) (reg n) (const 1)) ; n を  $n - 1$  で
    上書き
    (goto (label fib-loop)) ; 再帰呼出の実行
  afterfib-n-1 ; リターン時に, val が  $\text{Fib}(n - 1)$  を持つ
```

```

(restore n)
(restore continue)
;; Fib( $n - 2$ ) を求める準備
(assign n (op -) (reg n) (const 2))
(save continue)
(assign continue (label afterfib-n-2))
(save val) ; Fib( $n - 1$ ) を保存
(goto (label fib-loop))
afterfib-n-2 ; リターン時に, val が Fib( $n - 2$ ) を持つ
(assign n (reg val)) ; n がここで Fib( $n - 2$ ) を持つ
(
  (restore val) ; val がここで Fib( $n - 1$ ) を持つ
)
(restore continue)

(assign val ; Fib( $n - 1$ ) + Fib( $n - 2$ )
      (op +) (reg val) (reg n))
(goto (reg continue)) ; 呼び出しから戻る, 答は val
) 中にある

immediate-answer
(assign val (reg n)) ; 基底の場合: Fib( $n$ ) =  $n$ 
(goto (reg continue))
fib-done)

```

**Exercise 5.4:** Specify register machines that implement each of the following procedures. For each machine, write a controller instruction sequence and draw a diagram showing the data paths.

次の手続のそれぞれを実装するレジスタマシンを指定せよ。各マシンに対して、コントローラ命令列を書き、データパスを示す図を描け。

a Recursive exponentiation:

再帰指数計算

```

(define (expt b n)
  (if (= n 0)
    1

```

```
(* b (expt b (- n 1))))
```

b Iterative exponentiation:

反復指數計算

```
(define (expt b n)
  (define (expt-iter counter product)
    (if (= counter 0)
        product
        (expt-iter (- counter 1)
                   (* b product))))
  (expt-iter n 1))
```

**Exercise 5.5:** Hand-simulate the factorial and Fibonacci machines, using some nontrivial input (requiring execution of at least one recursive call). Show the contents of the stack at each significant point in the execution.

階乗とフィボナッチの機械をいくつかの非自明な入力を用いて手動でシミュレートせよ。(少なくとも 1 回の再帰呼出を必要とする)。実行中の各重要な地点におけるスタックの中身を示せ。

**Exercise 5.6:** Ben Bitdiddle observes that the Fibonacci machine's controller sequence has an extra `save` and an extra `restore`, which can be removed to make a faster machine. Where are these instructions?

Ben Bitdiddle はフィボナッチマシンのコントローラシーケンスが余分な `save` と `restore` を持ち、より速いマシンを作るために取り除くことができることに気付いた。これらの命令はどこにあるか?

### 5.1.5 Instruction Summary

A controller instruction in our register-machine language has one of the following forms, where each  $\langle \text{input}_i \rangle$  is either `(reg <register-name>)` or `(const <constant-value>)`. These instructions were introduced in [Section 5.1.1](#):

私達のレジスタマシン言語のコントローラ命令は以下の形式の内 1 つを持ち、各  $\langle \text{input}_i \rangle$  は `(reg <register-name>)` か `(const <constant-value>)` の何れかです。これらの命令は [Section 5.1.1](#) で導入されました。

```
(assign <register-name> (reg <register-name>))
(assign <register-name> (const <constant-value>))
(assign <register-name>
       (op <operation-name>)
       <input1> ... <inputn>)
(perform (op <operation-name>) <input1> ... <inputn>)
(test (op <operation-name>) <input1> ... <inputn>)
(branch (label <label-name>))
(goto (label <label-name>))
```

The use of registers to hold labels was introduced in [Section 5.1.3](#):

レジスタを用いてラベルを保存することは[Section 5.1.3](#)で導入されました。

```
(assign <register-name> (label <label-name>))
(goto (reg <register-name>))
```

Instructions to use the stack were introduced in [Section 5.1.4](#):

スタックを使用する命令は[Section 5.1.4](#)で導入されました。

```
(save <register-name>)
	restore <register-name>)
```

The only kind of <constant-value> we have seen so far is a number, but later we will use strings, symbols, and lists. For example,

ここまでで見た <constant-value> の種類は数値のみです。しかし後程、文字列、シンボル、それにリストを使用します。

(const "abc") は文字列 "abc" ,  
(const abc) はシンボル abc ,  
(const (a b c)) はリスト (a b c) ,  
and (const ()) は空リスト

## 5.2 A Register-Machine Simulator

In order to gain a good understanding of the design of register machines, we must test the machines we design to see if they perform as expected. One way to test a design is to hand-simulate the operation of the controller, as in [Exercise 5.5](#). But this is extremely tedious for all but the simplest machines. In this

section we construct a simulator for machines described in the register-machine language. The simulator is a Scheme program with four interface procedures. The first uses a description of a register machine to construct a model of the machine (a data structure whose parts correspond to the parts of the machine to be simulated), and the other three allow us to simulate the machine by manipulating the model:

レジスタマシンの設計を良く理解するために、私達は設計した機械を期待通りに実行されるか確認するためにテストをする必要があります。設計のテストを行う1つの方法として [Exercise 5.5](#)と同様にコントローラの命令を手動でシミュレートする方法があります。しかしこれは簡単な機械を除いてとんでもなく退屈な方法です。この節ではレジスタマシン言語で記述された機械のためのシミュレータを構築します。このシミュレータは4つのインターフェイス手続を持つSchemeのプログラムです。1つ目はレジスタマシンの記述をマシンのモデルを構築するために利用します(データ構造の部品がシミュレートされるマシンの部品に対応します)。残りの3つがモデルを操作することにより機械のシミュレーションを可能にします。

```
(make-machine <register-names> <operations> <controller>)
```

constructs and returns a model of the machine with the given registers, operations, and controller.

与えられたレジスタ、命令、コントローラを持つ機械のモデルを構築し、返します。

```
(set-register-contents! <machine-model>
                      <register-name>
                      <value>)
```

stores a value in a simulated register in the given machine.

与えられた機械でシミュレートされるレジスタに値を格納します。

```
(get-register-contents <machine-model> <register-name>)
```

returns the contents of a simulated register in the given machine.

与えられた機械のシミュレートされるレジスタの中身を返す。

```
(start <machine-model>)
```

simulates the execution of the given machine, starting from the beginning of the controller sequence and stopping when it reaches the end of the sequence.

与えられた機械の実行をシミュレートする。コントローラシーケンスの最初から開始し、シーケンスの最後に辿り着いた時に停止する。

As an example of how these procedures are used, we can define `gcd-machine` to be a model of the GCD machine of [Section 5.1.1](#) as follows:

これらの手続がどのように利用されるかの例として、[Section 5.1.1](#)の GCD マシンのモデルとなる `gcd-machine` を以下のように定義します。

```
(define gcd-machine
  (make-machine
    '(a b t)
    (list (list 'rem remainder) (list '= =))
    '(test-b (test (op =) (reg b) (const 0))
              (branch (label gcd-done))
              (assign t (op rem) (reg a) (reg b))
              (assign a (reg b))
              (assign b (reg t))
              (goto (label test-b))
              gcd-done)))
```

The first argument to `make-machine` is a list of register names. The next argument is a table (a list of two-element lists) that pairs each operation name with a Scheme procedure that implements the operation (that is, produces the same output value given the same input values). The last argument specifies the controller as a list of labels and machine instructions, as in [Section 5.1](#).

`make-machine` に対する最初の引数はレジスタ名のリストです。次の引数は各命令名とその命令を実装する Scheme 手続 (つまり、同じ入力値を与えて同じ出力値を生成します) をペアにするテーブル (2 要素リストのリスト) です。最後の引数は [Section 5.1](#) にあるようにラベルと機械の命令 (機械語) のリストとしてのコントローラを指定します。

To compute GCDS with this machine, we set the input registers, start the machine, and examine the result when the simulation terminates:

この機械を用いて GCD を求めるために、入力レジスタを設定し、機械を開始し、シミュレーションが停止した時に結果を検査します。

```
(set-register-contents! gcd-machine 'a 206)
done
(set-register-contents! gcd-machine 'b 40)
done
(start gcd-machine)
done
(get-register-contents gcd-machine 'a)
2
```

This computation will run much more slowly than a `gcd` procedure written in Scheme, because we will simulate low-level machine instructions, such as `assign`, by much more complex operations.

この演算は Scheme で書かれた `gcd` 手続よりもとても遅く実行します。なぜなら `assign` のような低レベルの機械語をより複雑な命令によりシミュレートするためです。

**Exercise 5.7:** Use the simulator to test the machines you designed in Exercise 5.4.

シミュレータを用いて Exercise 5.4 で自分で設計した機械をテストせよ。

### 5.2.1 The Machine Model

The machine model generated by `make-machine` is represented as a procedure with local state using the message-passing techniques developed in Chapter 3. To build this model, `make-machine` begins by calling the procedure `make-new-machine` to construct the parts of the machine model that are common to all register machines. This basic machine model constructed by `make-new-machine` is essentially a container for some registers and a stack, together with an execution mechanism that processes the controller instructions one by one.

`make-machine` にて生成された機械のモデルは Chapter 3 で開発されたメッセージパッシングの技術を用いた局所状態を持つ手続として表現されています。このモデルを構築するために、`make-machine` は手続 `make-new-machine` を呼び全てのレジスタマシンに対して共通なマシンモデルの部品を構築することから始めます。`make-new-machine` により構築されるこの基本的な機械のモデルは本質的にはいくつかのレジスタとスタックと、コントローラ命令を 1 つずつ処理する実行の仕組みと一緒にしたコンテナです。

**Make-machine** then extends this basic model (by sending it messages) to include the registers, operations, and controller of the particular machine being defined. First it allocates a register in the new machine for each of the supplied register names and installs the designated operations in the machine. Then it uses an *assembler* (described below in [Section 5.2.2](#)) to transform the controller list into instructions for the new machine and installs these as the machine's instruction sequence. **Make-machine** returns as its value the modified machine model.

**make-machine** は次にこの基本的なモデルを (それに対してメッセージを送ることで) 拡張し、レジスタ、命令、定義される特定の機械のコントローラを含めます。最初に新しい機械の中に与えられた各レジスタ名に対するレジスタを獲得し、指定された命令をその機械にインストール(導入)します。次に *assembler*(アセンブラー)(下記の [Section 5.2.2](#)で説明されます)を用いてコントローラリストを新しい機械に対する命令に変換し、これらを機械の命令列としてインストールします。**make-machine** はその値として変更された機械のモデルを返します。

```
(define (make-machine register-names ops controller-text)
  (let ((machine (make-new-machine)))
    (for-each
      (lambda (register-name)
        ((machine 'allocate-register) register-name))
      register-names)
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
      (assemble controller-text machine))
    machine))
```

## Registers

We will represent a register as a procedure with local state, as in [Chapter 3](#). The procedure **make-register** creates a register that holds a value that can be accessed or changed:

レジスタは [Chapter 3](#) の様に局所状態を持つ手続として表現されます。手続 **make-register** はアクセスと変更が可能な値を持つレジスタを作成します。

```
(define (make-register name)
```

```
(let ((contents '*unassigned*))
  (define (dispatch message)
    (cond ((eq? message 'get) contents)
          ((eq? message 'set)
           (lambda (value) (set! contents value)))
          (else
            (error "Unknown request: REGISTER" message))))
  dispatch))
```

The following procedures are used to access registers:

以下の手続はレジスタにアクセスするために使用されます。

```
(define (get-contents register) (register 'get))
(define (set-contents! register value)
  ((register 'set) value))
```

## The stack

We can also represent a stack as a procedure with local state. The procedure `make-stack` creates a stack whose local state consists of a list of the items on the stack. A stack accepts requests to `push` an item onto the stack, to `pop` the top item off the stack and return it, and to `initialize` the stack to empty.

スタックもまた局所状態を持つ手続として表現されます。手続 `make-stack` は局所状態がスタック上のアイテム（項目）のリストから成るスタックを作成します。スタックはスタック上にアイテムを `push` とスタックから最上位のアイテムを取り去りそれを返す `pop`、スタックを空に初期化する `initialize` のリクエストを受け付けます。

```
(define (make-stack)
  (let ((s '()))
    (define (push x) (set! s (cons x s)))
    (define (pop)
      (if (null? s)
          (error "Empty stack: POP")
          (let ((top (car s)))
            (set! s (cdr s))
            top))))
```

```

(define (initialize)
  (set! s '())
  'done)
(define (dispatch message)
  (cond ((eq? message 'push) push)
        ((eq? message 'pop) (pop))
        ((eq? message 'initialize) (initialize))
        (else (error "Unknown request: STACK"
                     message))))
  dispatch)

```

The following procedures are used to access stacks:

以下の手続はスタックへのアクセスに使用されます。

```

(define (pop stack) (stack 'pop))
(define (push stack value) ((stack 'push) value))

```

## The basic machine

The `make-new-machine` procedure, shown in Figure 5.13, constructs an object whose local state consists of a stack, an initially empty instruction sequence, a list of operations that initially contains an operation to initialize the stack, and a *register table* that initially contains two registers, named `flag` and `pc` (for “program counter”). The internal procedure `allocate-register` adds new entries to the register table, and the internal procedure `lookup-register` looks up registers in the table.

Figure 5.13に示す `make-new-machine` 手続は局所状態がスタック、初期値が空の命令列、初期値がスタックを初期化する命令を持つ命令のリスト、初期値として2つのレジスタ `flag`(フラグ)と `pc` (“program counter”、プログラムカウンタ)を持つ *register table*(レジスタテーブル)から成り立ちます。内部手続 `lookup-register` はテーブル内のレジスタを探します。

The `flag` register is used to control branching in the simulated machine. Test instructions set the contents of `flag` to the result of the test (true or false). Branch instructions decide whether or not to branch by examining the contents of `flag`.

`flag` レジスタはシミュレートされる機械にて分岐をコントロールするために使用されます。`test` 命令は `flag` の中身にテストの結果(真、または、偽)を

設定します。`branch` 命令は分岐するかしないかを `flag` の中身を調査して決定します。

The `pc` register determines the sequencing of instructions as the machine runs. This sequencing is implemented by the internal procedure `execute`. In the simulation model, each machine instruction is a data structure that includes a procedure of no arguments, called the *instruction execution procedure*, such that calling this procedure simulates executing the instruction. As the simulation runs, `pc` points to the place in the instruction sequence beginning with the next instruction to be executed. `Execute` gets that instruction, executes it by calling the instruction execution procedure, and repeats this cycle until there are no more instructions to execute (i.e., until `pc` points to the end of the instruction sequence).

`pc` レジスタは機械が実行する命令の順序付けを決定します。この順序付けは内部手続 `execute` により実装されています。シミュレーションモデルでは各機械命令は *instruction execution procedure*(命令実行手続) と呼ばれる引数無しの手続を含むデータ構造であり、この手続を呼ぶことにより命令の実行をシミュレートします。シミュレーションが実行されるにつれ、`pc` は次に実行される命令から始まる命令列の地点を指します。`execute` はその命令を得て、それを命令実行手続を呼ぶことにより実行し、このサイクルを実行する命令が無くなるまで(すなわち、`pc` が命令列の最後を指すまで)繰り返します。

**Figure 5.13:** ↓ The `make-new-machine` procedure, which implements the basic machine model.

↓ 基本の機械モデルを実装する `make-new-machine` 手続

```
(define (make-new-machine)
  (let ((pc (make-register 'pc))
        (flag (make-register 'flag))
        (stack (make-stack))
        (the-instruction-sequence '()))
    (let ((the-ops
          (list (list 'initialize-stack
                      (lambda () (stack 'initialize)))))
          (register-table
            (list (list 'pc pc) (list 'flag flag))))
      (define (allocate-register name)
        (if (assoc name register-table)
            (error "Multiply defined register: " name)
            (let ((new-register (make-register name)))
              (set! register-table (cons (list name new-register)
                                         register-table))
              new-register))))))
```

```

(set! register-table
      (cons (list name (make-register name))
            register-table)))
'register-allocated)
(define (lookup-register name)
  (let ((val (assoc name register-table)))
    (if val
        (cadr val)
        (error "Unknown register:" name))))
(define (execute)
  (let ((insts (get-contents pc)))
    (if (null? insts)
        'done
        (begin
          ((instruction-execution-proc (car insts))
           (execute))))))
(define (dispatch message)
  (cond ((eq? message 'start)
         (set-contents! pc the-instruction-sequence)
         (execute))
        ((eq? message 'install-instruction-sequence)
         (lambda (seq)
           (set! the-instruction-sequence seq)))
        ((eq? message 'allocate-register)
         allocate-register)
        ((eq? message 'get-register)
         lookup-register)
        ((eq? message 'install-operations)
         (lambda (ops)
           (set! the-ops (append the-ops ops))))
        ((eq? message 'stack) stack)
        ((eq? message 'operations) the-ops)
        (else (error "Unknown request: MACHINE"
                     message))))
  dispatch)))

```

As part of its operation, each instruction execution procedure modifies `pc` to

indicate the next instruction to be executed. `Branch` and `goto` instructions change `pc` to point to the new destination. All other instructions simply advance `pc`, making it point to the next instruction in the sequence. Observe that each call to `execute` calls `execute` again, but this does not produce an infinite loop because running the instruction execution procedure changes the contents of `pc`.

工程の一部として、各命令の実行手続は `pc` を変更し次に実行される命令を指すようにします。`branch` と `goto` 命令は `pc` を変更し新しい行き先を指すようにします。全ての他の命令は単純に `pc` を進めて列の次の命令を指すようにします。各 `execute` の呼出が `execute` を再び呼び出すことに中止して下さい。これはしかし無限ループにはなりません。命令実行手続の実行は `pc` の中身を変更するためです。

`Make-new-machine` returns a `dispatch` procedure that implements message-passing access to the internal state. Notice that starting the machine is accomplished by setting `pc` to the beginning of the instruction sequence and calling `execute`.

`make-new-machine` は `dispatch` 手続を返します。これは内部の状態にアクセスするメッセージパッキングを実装します。機械の開始は `pc` に命令列の最初を設定し、`execute` を呼ぶことにより達成されることに注意して下さい。

For convenience, we provide an alternate procedural interface to a machine's `start` operation, as well as procedures to set and examine register contents, as specified at the beginning of Section 5.2:

利便性のために、機械の `start` 命令の代替となる手続のインターフェイスを提供します。同様に、レジスタの中身の設定、試験の手続も Section 5.2 の最初にて指示されたように提供します。

```
(define (start machine) (machine 'start))
(define (get-register-contents machine register-name)
  (get-contents (get-register machine register-name)))
(define (set-register-contents! machine register-name value)
  (set-contents! (get-register machine register-name)
                value))
'done)
```

These procedures (and many procedures in Section 5.2.2 and Section 5.2.3) use the following to look up the register with a given name in a given machine:

これらの手続 (と Section 5.2.2 と Section 5.2.3 の多くの手続) は以下を用い

て与えられた機械とレジスタ名のレジスタを探します。

```
(define (get-register machine reg-name)
  ((machine 'get-register) reg-name))
```

### 5.2.2 The Assembler

The assembler transforms the sequence of controller expressions for a machine into a corresponding list of machine instructions, each with its execution procedure. Overall, the assembler is much like the evaluators we studied in Chapter 4—there is an input language (in this case, the register-machine language) and we must perform an appropriate action for each type of expression in the language.

アセンブラーはコントローラの機械のための式の列を対応する機械の命令のリストへと変形します。各命令はその実行手続を持ちます。概して、アセンブラーはChapter 4で学習した評価機にとても似ています。入力言語が存在し(この場合にはレジスタマシン言語)、言語の式の各型に対して適切なアクションを実行しなければなりません。

The technique of producing an execution procedure for each instruction is just what we used in Section 4.1.7 to speed up the evaluator by separating analysis from runtime execution. As we saw in Chapter 4, much useful analysis of Scheme expressions could be performed without knowing the actual values of variables. Here, analogously, much useful analysis of register-machine-language expressions can be performed without knowing the actual contents of machine registers. For example, we can replace references to registers by pointers to the register objects, and we can replace references to labels by pointers to the place in the instruction sequence that the label designates.

各命令のための実行手続を生成する技術はSection 4.1.7で実行時に実行から分析を分離することで高速化するために用いたのものと同じです。Chapter 4で学んだように、Schemeの式の多くの実用的な分析は変数の実際の値を知らなくとも実行することができます。ここでも同様に、レジスタマシン言語の式の多くの実用的な分析が実際の機械のレジスタの値を知ることなしに実行することができます。例えばレジスタへの参照をレジスタオブジェクトへのポインタにより置き換えたり、ラベルをラベルが指定する命令列内の地点へのポインタで置き換えることができます。

Before it can generate the instruction execution procedures, the assembler

must know what all the labels refer to, so it begins by scanning the controller text to separate the labels from the instructions. As it scans the text, it constructs both a list of instructions and a table that associates each label with a pointer into that list. Then the assembler augments the instruction list by inserting the execution procedure for each instruction.

アセンブラーが命令実行手続を生成する前に、全てのテーブルが何を参照するのか知つておく必要があります。そのためコントローラテキストを走査し命令からラベルを分離することから始めます。アセンブラーがテキストを走査するにつれ、命令のリストと各ラベルをそのリスト内部を指すポインタと関連付けるテーブルの両方を構築します。そうしたらアセンブラーは命令リストを各命令に対する実行手続を挿入することで増補します。

The `assemble` procedure is the main entry to the assembler. It takes the controller text and the machine model as arguments and returns the instruction sequence to be stored in the model. `Assemble` calls `extract-labels` to build the initial instruction list and label table from the supplied controller text. The second argument to `extract-labels` is a procedure to be called to process these results: This procedure uses `update-insts!` to generate the instruction execution procedures and insert them into the instruction list, and returns the modified list.

`assemble` 手続はアセンブラーに対する主な入口です。コントローラテキストとマシンモデルを引数として取り、モデルに格納するべき命令列を返します。`assemble` は `extract-labels` を呼び初期命令リストと与えられたコントローラテキストからラベルテーブルを構築します。`extract-labels` の 2 つの引数はこれらの結果を処理するために呼ばれるべきものです。この手続は `update-insts!` を用いて命令実行手続を再生し、それらを命令リストの中に挿入し、変更されたリストを返します。

```
(define (assemble controller-text machine)
  (extract-labels
    controller-text
    (lambda (insts labels)
      (update-insts! insts labels machine)
      insts)))
```

`Extract-labels` takes as arguments a list `text` (the sequence of controller instruction expressions) and a `receive` procedure. `Receive` will be called with two values: (1) a list `insts` of instruction data structures, each containing an

instruction from `text`; and (2) a table called `labels`, which associates each label from `text` with the position in the list `insts` that the label designates.

`extract-labels` は引数としてリスト `text`(コントローラ命令式の列) と `receive` 手続を取ります。`receive` は 2 つの値と共に呼び出されます。(1) 命令データ構造のリスト `insts` はそれぞれが `text` からの命令を含みます。(2) テーブル `labels` は `text` からの各ラベルとそのラベルが指定するリスト `insts` 内の位置とを関連付けします。

```
(define (extract-labels text receive)
  (if (null? text)
      (receive '() '())
      (extract-labels
        (cdr text)
        (lambda (insts labels)
          (let ((next-inst (car text)))
            (if (symbol? next-inst)
                (receive insts
                  (cons (make-label-entry next-inst
                                            insts)
                        labels))
                (receive (cons (make-instruction next-inst)
                               insts)
                      labels)))))))
```

`Extract-labels` works by sequentially scanning the elements of the `text` and accumulating the `insts` and the `labels`. If an element is a symbol (and thus a label) an appropriate entry is added to the `labels` table. Otherwise the element is accumulated onto the `insts` list.

`extract-labels` は連続して `text` の要素を走査し、`insts` と `labels` を集積することで働きます。もし要素がシンボル(従ってラベル)なら適切なエントリが `labels` テーブルに追加されます。そうでなければその要素は `insts` リスト上に集積されます。<sup>7</sup>

---

<sup>7</sup>Using the `receive` procedure here is a way to get `extract-labels` to effectively return two values—`labels` and `insts`—without explicitly making a compound data structure to hold them. An alternative implementation, which returns an explicit pair of values, is

`receive` 手続をここで使用するのは `extract-labels` を得て、効率的に 2 つの値、

`Update-insts!` modifies the instruction list, which initially contains only the text of the instructions, to include the corresponding execution procedures:

`update-insts!` は命令リストを変更します。これは初期値としては命令のテキストのみを含みますが、対応する実行手続を含むようになります。

```
(define (update-insts! insts labels machine)
  (let ((pc (get-register machine 'pc))
        (flag (get-register machine 'flag)))
```

---

`labels` と `insts` をそれを保持する複合データ構造を明示的に作ること無しに返すための方法です。代替となる、明示的に値のペアを返す実装は以下の通りです。

```
(define (extract-labels text)
  (if (null? text)
      '()
      (cons '() '())
      (let ((result (extract-labels (cdr text))))
        (let ((insts (car result)) (labels (cdr result)))
          (let ((next-inst (car text)))
            (if (symbol? next-inst)
                (cons insts
                      (cons (make-label-entry next-inst insts)
                            labels))
                (cons (cons (make-instruction next-inst) insts)
                      labels)))))))
```

which would be called by `assemble` as follows:

これは `assemble` により以下のように呼び出されます。

```
(define (assemble controller-text machine)
  (let ((result (extract-labels controller-text)))
    (let ((insts (car result)) (labels (cdr result)))
      (update-insts! insts labels machine)
      insts)))
```

You can consider our use of `receive` as demonstrating an elegant way to return multiple values, or simply an excuse to show off a programming trick. An argument like `receive` that is the next procedure to be invoked is called a “continuation.” Recall that we also used continuations to implement the backtracking control structure in the `amb` evaluator in Section 4.3.3.

`receive` の使用は複数の値を返す洗練された手法の実演、または単純にプログラミング上のトリックを見せ付けるための言い訳として考えることができます。`receive` のような次に実行されるべき手続引数は“継続”と呼ばれます。Section 4.3.3で私達が継続を `amb` 評価機のバックトラック制御構造の実装に用いたのを思い出して下さい。

```

(stack (machine 'stack))
(ops (machine 'operations)))
(for-each
(lambda (inst)
(set-instruction-execution-proc!
inst
(make-execution-procedure
(instruction-text inst)
labels machine pc flag stack ops)))
insts)))

```

The machine instruction data structure simply pairs the instruction text with the corresponding execution procedure. The execution procedure is not yet available when `extract-labels` constructs the instruction, and is inserted later by `update-insts!`.

マシン語データ構造は単純に命令テキストと対応する実行手続のペアを作ります。実行手続は `extract-labels` が命令を構築した時にはまだ存在せず、後に `update-insts!` により挿入されます。

```

(define (make-instruction text) (cons text '()))
(define (instruction-text inst) (car inst))
(define (instruction-execution-proc inst) (cdr inst))
(define (set-instruction-execution-proc! inst proc)
  (set-cdr! inst proc))

```

The instruction text is not used by our simulator, but it is handy to keep around for debugging (see [Exercise 5.16](#)).

命令テキストはシミュレータでは使用されません。しかし、デバッグのために手元に置いておくと便利です。[\(Exercise 5.16参照\)](#)

Elements of the label table are pairs:

ラベルテーブルの要素はペアです。

```

(define (make-label-entry label-name insts)
  (cons label-name insts))

```

Entries will be looked up in the table with

テーブル内の要素は以下により検索されます。

```
(define (lookup-label labels label-name))
```

```
(let ((val (assoc label-name labels)))
  (if val
      (cdr val)
      (error "Undefined label: ASSEMBLE"
             label-name))))
```

**Exercise 5.8:** The following register-machine code is ambiguous, because the label `here` is defined more than once:

以下のレジスタマシンのコードは曖昧である。ラベル `here` が複数回、定義されているためである。

```
start
  (goto (label here))
here
  (assign a (const 3))
  (goto (label there))
here
  (assign a (const 4))
  (goto (label there))
there
```

With the simulator as written, what will the contents of register `a` be when control reaches `there`? Modify the `extract-labels` procedure so that the assembler will signal an error if the same label name is used to indicate two different locations.

シミュレータが書かれているままの状態で、レジスタ `a` の中身はコントローラが `there` に辿り着いた時に何になるか? 手続 `extract-labels` を変更し、同じラベル名が 2 つの異なる地点を指し示すのに使用された場合にエラーを発するようにせよ。

### 5.2.3 Generating Execution Procedures for Instructions

The assembler calls `make-execution-procedure` to generate the execution procedure for an instruction. Like the `analyze` procedure in the evaluator of Section 4.1.7, this dispatches on the type of instruction to generate the appropriate execution procedure.

アセンブラーは命令の実行手続を生成するために `make-execution-procedure` を呼びます。Section 4.1.7 の評価機の `analyze` 手続と同様に、これは適切な実行手続を生成するために命令の型に従い呼出を行います。

```
(define (make-execution-procedure
           inst labels machine pc flag stack ops)
  (cond ((eq? (car inst) 'assign)
         (make-assign inst machine labels ops pc))
        ((eq? (car inst) 'test)
         (make-test inst machine labels ops flag pc))
        ((eq? (car inst) 'branch)
         (make-branch inst machine labels flag pc))
        ((eq? (car inst) 'goto)
         (make-goto inst machine labels pc))
        ((eq? (car inst) 'save)
         (make-save inst machine stack pc))
        ((eq? (car inst) 'restore)
         (make-restore inst machine stack pc))
        ((eq? (car inst) 'perform)
         (make-perform inst machine labels ops pc))
        (else
         (error "Unknown instruction type: ASSEMBLE"
                inst))))
```

For each type of instruction in the register-machine language, there is a generator that builds an appropriate execution procedure. The details of these procedures determine both the syntax and meaning of the individual instructions in the register-machine language. We use data abstraction to isolate the detailed syntax of register-machine expressions from the general execution mechanism, as we did for evaluators in Section 4.1.2, by using syntax procedures to extract and classify the parts of an instruction.

レジスタマシンの言語の命令の各型に対し、適切な実行手続を構築する生成器が存在します。これらの手続の詳細がレジスタマシン言語の構文と個別の命令の意味の両方を決定します。データ抽象化を用いることで全体的な実行の仕組みからレジスタマシンの式の詳細な構文を分離しています。これはSection 4.1.2で評価機に対して行ったのと同様で、構文手続を用いて命令の部分を抽出し、分類することによります。

## Assign instructions

The `make-assign` procedure handles `assign` instructions:

`make-assign` 手続は `assign` 命令を扱います。

```
(define (make-assign inst machine labels operations pc)
  (let ((target
         (get-register machine (assign-reg-name inst)))
        (value-exp (assign-value-exp inst)))
    (let ((value-proc
           (if (operation-exp? value-exp)
               (make-operation-exp
                 value-exp machine labels operations)
               (make-primitive-exp
                 (car value-exp) machine labels))))
      (lambda () ; assign に対する実行手続
        (set-contents! target (value-proc))
        (advance-pc pc))))))
```

`Make-assign` extracts the target register name (the second element of the instruction) and the value expression (the rest of the list that forms the instruction) from the `assign` instruction using the selectors

`make-assign` はターゲットとなるレジスタ名(命令の2つ目の要素)と値の式(命令を構成するリストの残りの部分)を `assign` 命令からセレクタを用いて抽出します。

```
(define (assign-reg-name assign-instruction)
  (cadr assign-instruction))
(define (assign-value-exp assign-instruction)
  (cddr assign-instruction))
```

The register name is looked up with `get-register` to produce the target register object. The value expression is passed to `make-operation-exp` if the value is the result of an operation, and to `make-primitive-exp` otherwise. These procedures (shown below) parse the value expression and produce an execution procedure for the value. This is a procedure of no arguments, called `value-proc`, which will be evaluated during the simulation to produce the actual value to be assigned to the register. Notice that the work of looking up the register name and parsing the value expression is performed just once, at assembly time, not

every time the instruction is simulated. This saving of work is the reason we use execution procedures, and corresponds directly to the saving in work we obtained by separating program analysis from execution in the evaluator of Section 4.1.7.

レジスタ名が `get-register` を用いて検索され目的のレジスタオブジェクトを生成します。値の式はもし値が命令の結果であるのなら `make-operation-exp` に渡され、そうでなければ `make-primitive-exp` に渡されます。これらの手続(以下に示されます)は値の式を構文解析しその値に対する実行手続を生成します。これは引数無しの手続で `value-proc` と呼ばれ、シミュレーションの間にレジスタに代入される実際の値を生成するために評価されます。レジスタ名の検索と値の式の構文解析の仕事はただ一度、アセンブリ時(アセンブル実行時)に実行されることに注意して下さい。その命令がシミュレートされる度に毎回ではありません。この仕事量の削減こそが私達が実行手続を使用する理由です。そしてこれが直接Section 4.1.7の評価機において、実行からプログラム分析を分離することにより仕事量の削減を得たことに対応します。

The result returned by `make-assign` is the execution procedure for the `assign` instruction. When this procedure is called (by the machine model's `execute` procedure), it sets the contents of the target register to the result obtained by executing `value-proc`. Then it advances the `pc` to the next instruction by running the procedure

`make-assign` により返される結果は `assign` 命令のための実行手続です。この手続が(マシンモデルの `execute` 手続により)呼ばれた時に、`value-proc` 手続を実行することにより得られた結果を目的のレジスタの中身に設定します。その後に `pc` を以下の手続を実行することにより次の命令へと進めます。

```
(define (advance-pc pc)
  (set-contents! pc (cdr (get-contents pc))))
```

`Advance-pc` is the normal termination for all instructions except `branch` and `goto`.

`advance-pc` は `branch` と `goto` を除く全ての命令に対する通常の終わりです。

### Test, branch, and goto instructions

`Make-test` handles `test` instructions in a similar way. It extracts the expression that specifies the condition to be tested and generates an execution

procedure for it. At simulation time, the procedure for the condition is called, the result is assigned to the `flag` register, and the `pc` is advanced:

`make-test` は `test` 命令を同様な方法で扱います。これはテストされる条件を指定する式を抽出し、それに対する実行手続を生成します。シミュレーション時に、条件のための手続が呼ばれ、その結果が `flag` レジスタに割り当てられ、`pc` が進められます。

```
(define (make-test inst machine labels operations flag pc)
  (let ((condition (test-condition inst)))
    (if (operation-exp? condition)
        (let ((condition-proc
              (make-operation-exp
               condition machine labels operations)))
          (lambda ()
            (set-contents! flag (condition-proc))
            (advance-pc pc)))
        (error "Bad TEST instruction: ASSEMBLE" inst)))
(define (test-condition test-instruction)
  (cdr test-instruction))
```

The execution procedure for a `branch` instruction checks the contents of the `flag` register and either sets the contents of the `pc` to the branch destination (if the branch is taken) or else just advances the `pc` (if the branch is not taken). Notice that the indicated destination in a `branch` instruction must be a label, and the `make-branch` procedure enforces this. Notice also that the label is looked up at assembly time, not each time the `branch` instruction is simulated.

`branch` 命令のための実行手続は `flag` レジスタの中身をチェックし、`pc` の中身に分岐の目的地を設定するか（分岐が選択された場合）、または単に `pc` を進めます（分岐が選択されなかった場合）。`branch` 命令内で指定された目的値はラベルでなければならず、`make-branch` 手続がこのことを強制することに注意して下さい。またラベルはアセンブリ時に検索され、`branch` 命令がシミュレートされる時に毎回検索される訳ではないことにも注意して下さい。

```
(define (make-branch inst machine labels flag pc)
  (let ((dest (branch-dest inst)))
    (if (label-exp? dest)
        (let ((insts
```

```

    (lookup-label
     labels
     (label-exp-label dest))))
  (lambda ()
    (if (get-contents flag)
        (set-contents! pc insts)
        (advance-pc pc))))
  (error "Bad BRANCH instruction: ASSEMBLE" inst)))
(define (branch-dest branch-instruction)
  (cadr branch-instruction))

```

A goto instruction is similar to a branch, except that the destination may be specified either as a label or as a register, and there is no condition to check—the pc is always set to the new destination.

goto 命令は branch に似ていますが、目的地がラベルか、またはレジスタにより指定されることが異なります。また条件分岐ではありません。pc は常に新しい目的地に設定されます。

```

(define (make-goto inst machine labels pc)
  (let ((dest (goto-dest inst)))
    (cond ((label-exp? dest)
           (let ((insts (lookup-label
                        labels
                        (label-exp-label dest))))
             (lambda () (set-contents! pc insts))))
          ((register-exp? dest)
           (let ((reg (get-register
                      machine
                      (register-exp-reg dest))))
             (lambda ()
               (set-contents! pc (get-contents reg)))))
          (else (error "Bad GOTO instruction: ASSEMBLE"
                       inst)))))

(define (goto-dest goto-instruction)
  (cadr goto-instruction))

```

## Other instructions

The stack instructions `save` and `restore` simply use the stack with the designated register and advance the `pc`:

スタック命令の `save` と `restore` は単純にスタックを指定したレジスタと共に用いて、`pc` を進めます。

```
(define (make-save inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (push stack (get-contents reg))
      (advance-pc pc))))
(define (make-restore inst machine stack pc)
  (let ((reg (get-register machine
                           (stack-inst-reg-name inst))))
    (lambda ()
      (set-contents! reg (pop stack))
      (advance-pc pc))))
(define (stack-inst-reg-name stack-instruction)
  (cadr stack-instruction))
```

The final instruction type, handled by `make-perform`, generates an execution procedure for the action to be performed. At simulation time, the action procedure is executed and the `pc` advanced.

`make-perform` で扱われる最後の命令型は実行されるべきアクションのための実行手続を生成します。シミュレーション時にこのアクション手続が実行され `pc` は進められます。

```
(define (make-perform inst machine labels operations pc)
  (let ((action (perform-action inst)))
    (if (operation-exp? action)
        (let ((action-proc
              (make-operation-exp
                action machine labels operations)))
          (lambda () (action-proc) (advance-pc pc)))
        (error "Bad PERFORM instruction: ASSEMBLE" inst)))
(define (perform-action inst) (cdr inst))
```

## Execution procedures for subexpressions

The value of a `reg`, `label`, or `const` expression may be needed for assignment to a register (`make-assign`) or for input to an operation (`make-operation-exp`, below). The following procedure generates execution procedures to produce values for these expressions during the simulation:

`reg`, `label`, または `const` 式の値はレジスタへの代入 (`make-assign`) のため、または演算命令の入力 (下記の `make-operation-exp`) のために必要になるかもしれません。以下の手続はこれらの式のための値をシミュレーションの間に生成するための実行手続を生成します。

```
(define (make-primitive-exp exp machine labels)
  (cond ((constant-exp? exp)
         (let ((c (constant-exp-value exp)))
           (lambda () c)))
        ((label-exp? exp)
         (let ((insts (lookup-label
                      labels
                      (label-exp-label exp))))
           (lambda () insts)))
        ((register-exp? exp)
         (let ((r (get-register machine
                                (register-exp-reg exp))))
           (lambda () (get-contents r))))
        (else
         (error "Unknown expression type: ASSEMBLE" exp))))
```

The syntax of `reg`, `label`, and `const` expressions is determined by

`reg`, `label`, `const` 式の構文は以下により決定されます。

```
(define (register-exp? exp) (tagged-list? exp 'reg))
(define (register-exp-reg exp) (cadr exp))
(define (constant-exp? exp) (tagged-list? exp 'const))
(define (constant-exp-value exp) (cadr exp))
(define (label-exp? exp) (tagged-list? exp 'label))
(define (label-exp-label exp) (cadr exp))
```

`Assign`, `perform`, and `test` instructions may include the application of a machine operation (specified by an `op` expression) to some operands (specified by

`reg` and `const` expressions). The following procedure produces an execution procedure for an “operation expression”—a list containing the operation and operand expressions from the instruction:

`assign`, `perform`, `test` 命令は (`op` 式により指定される) 機械の演算命令の (`reg` と `const` 式により指定される) いくつかのオペランドへの適用を含むかもしれません。以下の手続は“演算命令式”—命令からの演算命令とオペランドの式を含むリスト—に対する実行手続を生成します。

```
(define (make-operation-exp exp machine labels operations)
  (let ((op (lookup-prim (operation-exp-op exp)
                           operations)))
    (aprocs
      (map (lambda (e)
              (make-primitive-exp e machine labels))
           (operation-exp-operands exp))))
  (lambda ()
    (apply op (map (lambda (p) (p)) aprocs)))))
```

The syntax of operation expressions is determined by

演算命令式の構文は以下により決定されます。

```
(define (operation-exp? exp)
  (and (pair? exp) (tagged-list? (car exp) 'op)))
(define (operation-exp-op operation-exp)
  (cadr (car operation-exp)))
(define (operation-exp-operands operation-exp)
  (cdr operation-exp))
```

Observe that the treatment of operation expressions is very much like the treatment of procedure applications by the `analyze-application` procedure in the evaluator of [Section 4.1.7](#) in that we generate an execution procedure for each operand. At simulation time, we call the operand procedures and apply the Scheme procedure that simulates the operation to the resulting values. The simulation procedure is found by looking up the operation name in the operation table for the machine:

演算命令式の処理が [Section 4.1.7](#) の評価機において各オペランドに対して実行手続を生成したことにおいて `analyze-application` 手続による手続の適用の処理にとても似ていることに注意して下さい。シミュレーション時に、オ

ペランド手続を呼び、結果となる値に対して演算をシミュレートする Scheme 手続を適用します。シミュレーション手続は演算命令の名前を機械の演算命令テーブルから検索することで見つかります。

```
(define (lookup-prim symbol operations)
  (let ((val (assoc symbol operations)))
    (if val
        (cadr val)
        (error "Unknown operation: ASSEMBLE"
              symbol))))
```

**Exercise 5.9:** The treatment of machine operations above permits them to operate on labels as well as on constants and the contents of registers. Modify the expression-processing procedures to enforce the condition that operations can be used only with registers and constants.

上記の機械の演算命令の取扱はそれらにラベル、定数、レジスタの中身上での演算を可能にする。式を処理する手続を変更し、演算命令がレジスタと定数のみに対して使用できるような条件を強制するようにせよ。

**Exercise 5.10:** Design a new syntax for register-machine instructions and modify the simulator to use your new syntax. Can you implement your new syntax without changing any part of the simulator except the syntax procedures in this section?

レジスタマシンの命令に新しい構文を設計し、シミュレータを変更してその新しい構文を使用せよ。シミュレータの内、この節の構文手続以外を変更せずにあなたの新しい構文を実装することができるだろうか？

**Exercise 5.11:** When we introduced `save` and `restore` in Section 5.1.4, we didn't specify what would happen if you tried to restore a register that was not the last one saved, as in the sequence

Section 5.1.4で `save` と `restore` を導入した時、以下の順の様に最後に保存した物ではないレジスタに戻した場合に何が起こるのかは指定しなかった。

```
(save y)  (save x)  (restore y)
```

There are several reasonable possibilities for the meaning of `restore`: `restore` の意味に対してはいくつかの妥当な可能性が存在する。

a (`restore y`) puts into `y` the last value saved on the stack, regardless of what register that value came from. This is the way our simulator behaves. Show how to take advantage of this behavior to eliminate one instruction from the Fibonacci machine of Section 5.1.4 (Figure 5.12).

(`restore y`) はスタック上に最後に保存された値を、どのレジスタからその値が来たのか関係無しに `y` に入れる。これが私達のシミュレータの振舞である。この振舞の利点の活用法を示すため、Section 5.1.4 のフィボナッチマシンから 1 つ命令を削減して見せよ。 (Figure 5.12)

b (`restore y`) puts into `y` the last value saved on the stack, but only if that value was saved from `y`; otherwise, it signals an error. Modify the simulator to behave this way. You will have to change `save` to put the register name on the stack along with the value.

(`restore y`) はスタック上に最後に保存された値を `y` に入れる。しかしその値が `y` から保存された場合のみである。そうでなければエラーを発する。シミュレータを変更してこのように振る舞うようにせよ。`save` を変更してスタック上に値と共にレジスタ名を保存しなければならない。

c (`restore y`) puts into `y` the last value saved from `y` regardless of what other registers were saved after `y` and not restored. Modify the simulator to behave this way. You will have to associate a separate stack with each register. You should make the `initialize-stack` operation initialize all the register stacks.

(`restore y`) は `y` の後に他のどのレジスタが保存され、取り出されていなくても最後に `y` から保存した値を `y` に入れる。シミュレータをこのように振る舞うように変更せよ。分離されたスタックを各レジスタに関連付けする必要がある。また `initialize-stack` 命令に全てのレジスタのスタックを初期化させなければならない。

**Exercise 5.12:** The simulator can be used to help determine the data paths required for implementing a machine with a given controller. Extend the assembler to store the following information in the machine model:

シミュレータは与えられたコントローラと共に機械を実装するために必要とされるデータパスを決定することを手助けするために利用することが可能である。アセンブラーを拡張し以下の情報をマシンモデルに格納せよ。

- a list of all instructions, with duplicates removed, sorted by instruction type (`assign`, `goto`, and so on);  
全ての命令のリストを重複を削除し、命令の型でソートする(`assign`, `goto` 等)
- a list (without duplicates) of the registers used to hold entry points (these are the registers referenced by `goto` instructions);  
エントリポイントを持つのに使用されたレジスタの(重複の無い)リスト。(これらは `goto` 命令で参照されたレジスタである)
- a list (without duplicates) of the registers that are `saved` or `restored`;  
`save` または `restore` されたレジスタの(重複の無い)リスト
- for each register, a list (without duplicates) of the sources from which it is assigned (for example, the sources for register `val` in the factorial machine of Figure 5.11 are (`const 1`) and ((`op *`) (`reg n`) (`reg val`))).  
各レジスタに対し、代入元の(重複の無い)リスト。(例えばFigure 5.11の階乗マシンのレジスタ `val` の入力元は (`const 1`) と ((`op *`) (`reg n`) (`reg val`))).

Extend the message-passing interface to the machine to provide access to this new information. To test your analyzer, define the Fibonacci machine from Figure 5.12 and examine the lists you constructed.

メッセージパッシングの機械へのインターフェイスを拡張し、この新しい情報へのアクセスを提供せよ。あなたの分析器をテスト

するためにFigure 5.12のフィボナッチマシンを定義し、構築されたリストを試験せよ。

**Exercise 5.13:** Modify the simulator so that it uses the controller sequence to determine what registers the machine has rather than requiring a list of registers as an argument to `make-machine`. Instead of pre-allocating the registers in `make-machine`, you can allocate them one at a time when they are first seen during assembly of the instructions.

シミュレータを変更することで、`make-machine`に対する引数としてレジスタのリストを要求するのではなく、コントローラシーケンスを使用して機械がどんなレジスタを持つのか決定するようにせよ。`make-machine`の中でレジスタを事前に獲得しておく代わりに、命令のアセンブリ時の間に初めて現れた時に1つづレジスタを獲得するようにせよ。

#### 5.2.4 Monitoring Machine Performance

Simulation is useful not only for verifying the correctness of a proposed machine design but also for measuring the machine’s performance. For example, we can install in our simulation program a “meter” that measures the number of stack operations used in a computation. To do this, we modify our simulated stack to keep track of the number of times registers are saved on the stack and the maximum depth reached by the stack, and add a message to the stack’s interface that prints the statistics, as shown below. We also add an operation to the basic machine model to print the stack statistics, by initializing `the-ops` in `make-new-machine` to

シミュレーションは提案された機械設計の正しさを確認するためだけではなく、機械のパフォーマンスを計るためにも便利です。例えば、私達のシミュレータに演算中に使用されるスタック命令の数を計る“メーター”を導入することができます。これを行うためには、シミュレーションを行うスタックを変更しスタック上にレジスタが保存された回数とスタックが到達した最大の深さを追跡するにし、スタックのインターフェイスにメッセージを追加し以下のように統計を表示するようにします。また `make-new-machine` 内の `the-ops` を以下の様に初期化することで、基本的なマシンモデルにスタックの統計を表示する命令を追加します。

```
(list (list 'initialize-stack
            (lambda () (stack 'initialize)))
      (list 'print-stack-statistics
            (lambda () (stack 'print-statistics))))
```

Here is the new version of `make-stack`:

以下が新しい版の `make-stack` です。

```
(define (make-stack)
  (let ((s '())
        (number-pushes 0)
        (max-depth 0)
        (current-depth 0))
    (define (push x)
      (set! s (cons x s))
      (set! number-pushes (+ 1 number-pushes))
      (set! current-depth (+ 1 current-depth))
      (set! max-depth (max current-depth max-depth)))
    (define (pop)
      (if (null? s)
          (error "Empty stack: POP")
          (let ((top (car s)))
            (set! s (cdr s))
            (set! current-depth (- current-depth 1))
            top)))
    (define (initialize)
      (set! s '())
      (set! number-pushes 0)
      (set! max-depth 0)
      (set! current-depth 0)
      'done)
    (define (print-statistics)
      (newline)
      (display (list 'total-pushes '= number-pushes
                     'maximum-depth '= max-depth)))
    (define (dispatch message)
      (cond ((eq? message 'push) push)
```

```

((eq? message 'pop) (pop))
((eq? message 'initialize) (initialize))
((eq? message 'print-statistics)
 (print-statistics))
(else
  (error "Unknown request: STACK" message)))
dispatch))

```

**Exercise 5.15** through **Exercise 5.19** describe other useful monitoring and debugging features that can be added to the register-machine simulator.

**Exercise 5.15**から**Exercise 5.19**はレジスタマシンシミュレータに追加できる他の便利な監視とデバッグの機能を説明します。

**Exercise 5.14:** Measure the number of pushes and the maximum stack depth required to compute  $n!$  for various small values of  $n$  using the factorial machine shown in [Figure 5.11](#). From your data determine formulas in terms of  $n$  for the total number of push operations and the maximum stack depth used in computing  $n!$  for any  $n > 1$ . Note that each of these is a linear function of  $n$  and is thus determined by two constants. In order to get the statistics printed, you will have to augment the factorial machine with instructions to initialize the stack and print the statistics. You may want to also modify the machine so that it repeatedly reads a value for  $n$ , computes the factorial, and prints the result (as we did for the GCD machine in [Figure 5.4](#)), so that you will not have to repeatedly invoke `get-register-contents`, `set-register-contents!`, and `start`.

[Figure 5.11](#)で示された階乗マシンを用いて様々な小さな値  $n$  に対する  $n!$  の演算に必要とされる push の数とスタックの最大深さを計れ。データから任意  $n > 1$  に対する  $n!$  を求めるのに使用された push 命令の総数とスタックの最大深度に対する  $n$  を用いた方程式を決定せよ。それぞれが  $n$  の線形関数であり、従って 2 つの定数により決定されることに注意せよ。統計が表示されるために、階乗マシンをスタックを初期化する命令と統計を表示する命令を拡張しなければならない。また機械を変更することで  $n$  に対する値を繰り返し読み込み、階乗を求め、結果を表示できるようにしたいと思うかもしれないだろう（我々が [Figure 5.4](#) で丁度 GCD マシンに対して行ったように）。そうすることで繰り返し `get-register-`

`contents, set-register-contents!, start` を起動する必要が無くなる。

**Exercise 5.15:** Add *instruction counting* to the register machine simulation. That is, have the machine model keep track of the number of instructions executed. Extend the machine model's interface to accept a new message that prints the value of the instruction count and resets the count to zero.

レジスタマシンのシミュレータに*instruction counting*(命令数カウンタ)を追加せよ。これはマシンモデルに対して実行された命令数を追跡させる。マシンモデルのインターフェイスを拡張し、命令カウンタの値を表示する物とカウンタをゼロにリセットする新しいメッセージを受け入れるようにせよ。

**Exercise 5.16:** Augment the simulator to provide for *instruction tracing*. That is, before each instruction is executed, the simulator should print the text of the instruction. Make the machine model accept `trace-on` and `trace-off` messages to turn tracing on and off.

シミュレータを拡張し*instruction tracing*(命令トレーサ)を追加せよ。これは各命令が実行される前に、シミュレータが命令のテキストを表示する。マシンモデルに対しトレーサを on/off する `trace-on` と `trace-off` メッセージを受け入れるようにせよ。

**Exercise 5.17:** Extend the instruction tracing of Exercise 5.16 so that before printing an instruction, the simulator prints any labels that immediately precede that instruction in the controller sequence. Be careful to do this in a way that does not interfere with instruction counting (Exercise 5.15). You will have to make the simulator retain the necessary label information.

Exercise 5.16の命令トレーサを拡張し命令を表示する前にシミュレータがコントローラシーケンス内でその命令の直前のラベル表示するにせよ。命令数カウンタ(Exercise 5.15)に干渉しない方法で行うように注意すること。シミュレータに必要なラベル情報を維持するようにすることが必要だろう。

**Exercise 5.18:** Modify the `make-register` procedure of Section 5.2.1 so that registers can be traced. Registers should accept mes-

sages that turn tracing on and off. When a register is traced, assigning a value to the register should print the name of the register, the old contents of the register, and the new contents being assigned. Extend the interface to the machine model to permit you to turn tracing on and off for designated machine registers.

Section 5.2.1 の `make-register` 手続を変更し、レジスタをトレース可能にせよ。レジスタがト雷斯の on, off を行うメッセージを受け入れなければならない。レジスタがト雷斯されている時、そのレジスタに対する代入はレジスタの名前、レジスタの古い値、代入される新しい値が表示されること。マシンモデルへのインターフェイスを拡張し指定された機械のレジスタに対するトレーサの on, off を可能にせよ。

**Exercise 5.19:** Alyssa P. Hacker wants a *breakpoint* feature in the simulator to help her debug her machine designs. You have been hired to install this feature for her. She wants to be able to specify a place in the controller sequence where the simulator will stop and allow her to examine the state of the machine. You are to implement a procedure

Alyssa P. Hacker はシミュレータ内に *breakpoint*(ブレイクポイント) の機能を欲しいと思った。それにより彼女の機械設計を手助けするためである。あなたが彼女のためにこの機能を導入するために雇用された。彼女はコントローラシーケンス内でシミュレータが停止する場所を指定し、機械の状態を調査することができるようにして欲しかった。あなたは以下の手続を実装しようとしている。

```
(set-breakpoint <machine> <label> <n>)
```

that sets a breakpoint just before the  $n^{\text{th}}$  instruction after the given label. For example,

これは与えられたラベルの後ろの  $n$  番目の命令の直前にブレイクポイントを設定する。例えば、

```
(set-breakpoint gcd-machine 'test-b 4)
```

installs a breakpoint in `gcd-machine` just before the assignment to register `a`. When the simulator reaches the breakpoint it should

print the label and the offset of the breakpoint and stop executing instructions. Alyssa can then use `get-register-contents` and `set-register-contents!` to manipulate the state of the simulated machine. She should then be able to continue execution by saying 上の式はブレイクポイントを `gcd-machine` のレジスタ `a` への代入の直前にブレイクポイントを導入する。シミュレータがブレイクポイントに到達する時、ラベルとブレークポイントのオフセットを表示し、命令の実行を停止しなければなりません。すると Alyssa は `get-register-contents` と `set-register-contents!` を用いてシミュレートされている機械の状態を操作することが可能になる。次に彼女は以下を入力することで実行を続行できなければならぬ。

```
(proceed-machine <machine>)
```

She should also be able to remove a specific breakpoint by means of

また特定のブレイクポイントを以下を用いて削除できなければならぬ。

```
(cancel-breakpoint <machine> <label> <n>)
```

or to remove all breakpoints by means of

または全てのブレイクポイントを削除するためには以下を用いる。

```
(cancel-all-breakpoints <machine>)
```

## 5.3 Storage Allocation and Garbage Collection

In Section 5.4, we will show how to implement a Scheme evaluator as a register machine. In order to simplify the discussion, we will assume that our register machines can be equipped with a *list-structured memory*, in which the basic operations for manipulating list-structured data are primitive. Postulating the existence of such a memory is a useful abstraction when one is focusing on the mechanisms of control in a Scheme interpreter, but this does not reflect a realistic view of the actual primitive data operations of contemporary computers. To obtain a more complete picture of how a Lisp system operates, we must

investigate how list structure can be represented in a way that is compatible with conventional computer memories.

Section 5.4ではレジスタマシンとしての Scheme 評価機をどのように実装するかを示します。議論を簡易化するために、私達のレジスタマシンは *list-structured memory*(リスト構造メモリ) を供えていると仮定します。この機械ではリスト構造のデータを操作する命令はプリミティブです。そのようなメモリが存在するという仮定は Scheme インタプリタの制御の仕組みに集中する場合には有用な抽象化です。しかしこれは現在のコンピュータの実際のプリミティブなデータ操作の現実の光景を反映してはいません。Lisp システムがどのように動作するかのより完全な理解を得るために、リスト構造がどのように旧来のコンピュータのメモリに互換性のある方法で表現されるかについて調査しなければなりません。

There are two considerations in implementing list structure. The first is purely an issue of representation: how to represent the “box-and-pointer” structure of Lisp pairs, using only the storage and addressing capabilities of typical computer memories. The second issue concerns the management of memory as a computation proceeds. The operation of a Lisp system depends crucially on the ability to continually create new data objects. These include objects that are explicitly created by the Lisp procedures being interpreted as well as structures created by the interpreter itself, such as environments and argument lists. Although the constant creation of new data objects would pose no problem on a computer with an infinite amount of rapidly addressable memory, computer memories are available only in finite sizes (more's the pity). Lisp systems thus provide an *automatic storage allocation* facility to support the illusion of an infinite memory. When a data object is no longer needed, the memory allocated to it is automatically recycled and used to construct new data objects. There are various techniques for providing such automatic storage allocation. The method we shall discuss in this section is called *garbage collection*.

リスト構造の実装には 2 つの考慮点が存在します。1 つは純粹に表現上の問題です。Lisp のペアによる“箱とポインタ”構造をストレージと典型的なコンピュータのメモリのアドレス指定能力を用いてどのように表現するか。2 つの問題は演算が進行するにつれてのメモリ管理に関係します。Lisp システムの動作は決定的に、継続して新しいデータオブジェクトを作る能力に依存しています。これらは逐次実行される Lisp 手続により明示的に作成されるオブジェクトと同様に、インタプリタ自身により作成される環境や引数リストのような構造も含みます。持続的な新しいデータオブジェクトの作成は無限の容量でかつ、

高速にアドレス指定できるメモリを持つコンピュータ上では問題を起こさないでしようが、コンピュータのメモリは有限な量しかありません(残念なことに)。Lisp システムは従って無限のメモリという空想をサポートする *automatic storage allocation*(自動記憶域割当) の設備を提供します。データオブジェクトが既に必要でなくなった時に、それに割り当てられたメモリは自動的にリサイクルされ新しく構築されるデータオブジェクトに利用されます。そのような自動的な記憶域割当を提供する多様な技術が存在します。この節で私達が議論する手法は *garbage collection*(ガベージコレクション、ゴミ拾い) と呼ばれます。

### 5.3.1 Memory as Vectors

A conventional computer memory can be thought of as an array of cubbyholes, each of which can contain a piece of information. Each cubbyhole has a unique name, called its *address* or *location*. Typical memory systems provide two primitive operations: one that fetches the data stored in a specified location and one that assigns new data to a specified location. Memory addresses can be incremented to support sequential access to some set of the cubbyholes. More generally, many important data operations require that memory addresses be treated as data, which can be stored in memory locations and manipulated in machine registers. The representation of list structure is one application of such *address arithmetic*.

伝統的なコンピュータのメモリは小さな部屋の配列だと考えることができます。各部屋は情報の一片を入れることができます。各部屋は *address*(アドレス(住所)) または *location*(位置) と呼ばれる個有の名前を持ちます。典型的なメモリシステムは 2 つのプリミティブな命令を提供します。1 つは指定された位置に格納されたデータを取り出し、もう 1 つは指定された位置に新しいデータを割り当てます。メモリアドレスはある部屋の集合にシーケンシャル(順)なアクセスをサポートするためにインクリメントすることができます。より一般的には、多くの重要なデータの操作はメモリアドレスをデータとして扱うことを要求します。このデータはメモリ上の位置に格納でき、機械のレジスタ上で操作できなければいけません。リスト構造の表現はそのような *address arithmetic*(アドレス演算) の一つの応用です。

To model computer memory, we use a new kind of data structure called a *vector*. Abstractly, a vector is a compound data object whose individual elements can be accessed by means of an integer index in an amount of time

that is independent of the index.<sup>8</sup> In order to describe memory operations, we use two primitive Scheme procedures for manipulating vectors:

コンピュータメモリをモデル化するためには、*vector*(ベクタ) と呼ばれる新しい種類のデータ構造を用います。抽象的には、ベクタは複合データオブジェクトであり、その個別の要素が整数の索引を用いて、索引から独立した時間量でアクセスすることができます。<sup>9</sup> メモリ操作を説明するために、ベクタを扱うための 2 つのプリミティブな Scheme 手続を使用します。

- (`(vector-ref <vector> <n>)`) returns the  $n^{\text{th}}$  element of the vector.  
(`(vector-ref <vector> <n>)`) はベクタの  $n$  番目の要素を返す。
- (`(vector-set! <vector> <n> <value>)`) sets the  $n^{\text{th}}$  element of the vector to the designated value.  
(`(vector-set! <vector> <n> <value>)`) はベクタの  $n$  番目の要素に指定された値を設定する。

For example, if `v` is a vector, then (`(vector-ref v 5)`) gets the fifth entry in the vector `v` and (`(vector-set! v 5 7)`) changes the value of the fifth entry of the vector `v` to 7.<sup>10</sup> For computer memory, this access can be implemented through the use of address arithmetic to combine a *base address* that specifies the beginning location of a vector in memory with an *index* that specifies the offset of a particular element of the vector.

例えば、`v` がベクタであるならば、(`(vector-ref v 5)`) はベクタ `v` の 5 番目の項目を取得し、(`(vector-set! v 5 7)`) はベクタ `v` の 5 番目の項目の値を 7 に変更します。<sup>11</sup> コンピュータメモリに対して、このアクセスはアドレス演算を用いて、メモリ内のベクタの開始位置を指定する *base address*(ベース(基底))

---

<sup>8</sup>We could represent memory as lists of items. However, the access time would then not be independent of the index, since accessing the  $n^{\text{th}}$  element of a list requires  $n - 1$  `cdr` operations.

<sup>9</sup>メモリを項目のリストとして表現することはできます。しかし、アクセス時間はその場合、索引から独立しません。リストの  $n$  番目の要素へのアクセスが  $n - 1$  回の `cdr` 命令を必要とするためです。

<sup>10</sup>For completeness, we should specify a `make-vector` operation that constructs vectors. However, in the present application we will use vectors only to model fixed divisions of the computer memory.

<sup>11</sup>完全にするには、ベクタを構築する `make-vector` 命令を指定するべきです。しかし、現在のアプリケーションではベクタをコンピュータメモリの固定区域をモデル化するためにのみ使用します。

アドレス) とベクタの特定の項目のオフセットを指定する *index*(インデックス、索引) を組み合わせることで実装することができます。

## Representing Lisp data

We can use vectors to implement the basic pair structures required for a list-structured memory. Let us imagine that computer memory is divided into two vectors: **the-cars** and **the-cdrs**. We will represent list structure as follows: A pointer to a pair is an index into the two vectors. The **car** of the pair is the entry in **the-cars** with the designated index, and the **cdr** of the pair is the entry in **the-cdrs** with the designated index. We also need a representation for objects other than pairs (such as numbers and symbols) and a way to distinguish one kind of data from another. There are many methods of accomplishing this, but they all reduce to using *typed pointers*, that is, to extending the notion of “pointer” to include information on data type.<sup>12</sup> The data type enables the system to distinguish a pointer to a pair (which consists of the “pair” data type and an index into the memory vectors) from pointers to other kinds of data (which consist of some other data type and whatever is being used to represent data of that type). Two data objects are considered to be the same (*eq?*) if their pointers are identical.<sup>13</sup> Figure 5.14 illustrates the use of this method to represent the list ((1 2) 3 4), whose box-and-pointer diagram is also shown. We use letter prefixes to denote the data-type information. Thus, a pointer to the pair with index 5 is denoted **p5**, the empty list is denoted by the pointer **e0**, and a pointer to the number 4 is denoted **n4**. In the box-and-pointer diagram,

---

<sup>12</sup>This is precisely the same “tagged data” idea we introduced in Chapter 2 for dealing with generic operations. Here, however, the data types are included at the primitive machine level rather than constructed through the use of lists.

<sup>13</sup>Type information may be encoded in a variety of ways, depending on the details of the machine on which the Lisp system is to be implemented. The execution efficiency of Lisp programs will be strongly dependent on how cleverly this choice is made, but it is difficult to formulate general design rules for good choices. The most straightforward way to implement typed pointers is to allocate a fixed set of bits in each pointer to be a *type field* that encodes the data type. Important questions to be addressed in designing such a representation include the following: How many type bits are required? How large must the vector indices be? How efficiently can the primitive machine instructions be used to manipulate the type fields of pointers? Machines that include special hardware for the efficient handling of type fields are said to have *tagged architectures*.

we have indicated at the lower left of each pair the vector index that specifies where the `car` and `cdr` of the pair are stored. The blank locations in `the-cars` and `the-cdrs` may contain parts of other list structures (not of interest here).

ベクタを用いてリスト構造メモリに対する基本的なペア構造を実装することができます。コンピュータメモリが2つのベクタに分割されている所を想像してみましょう。`the-cars` と `the-cdrs` です。私達は次のようにリスト構造を表現します。ペアに対するポイントは2つのベクタへの索引です。ペアの `car` は `the-cars` に指定した索引を用いた項目です。そしてペアの `cdr` は指定された索引を用いた `the-cdrs` の項目です。またペア以外のオブジェクト(例えば数値やシンボル)に対する表現とデータの種類をお互いに見分けるための手法も必要になります。これを達成する方法は多数存在しますが、しかしそれらは全て *typed pointers*(型付きポインタ)の使用へと帰します。これはつまり、“ポインタ”の概念を拡張しデータの型の情報を含めることです。<sup>14</sup>データの型はシステムにペアのポインタ (“ペア”データ型とメモリベクタを指す索引から成り立つ) を他の種類のデータへのポインタ(何らかの他のデータ型とその型を表現するために利用された何かにより成り立つ)を見分けることを可能にします。2つのデータオブジェクトはそれらのポインタが全く同じである場合に同じ(`eq?`)だと判断されます。<sup>15</sup>Figure 5.14はこの手法を用いてリスト((1 2) 3 4)を表現する場合を図示しています。その箱とポインタ図もまた示されています。私達は文字接頭辞をデータ型情報を示すために使用しています。従って、ペアに対する索引5を伴なうポインタは `p5` と示されます。空リストはポインタ `e0` で示されます。そして数値4へのポインタは `n4` として示されます。箱とポインタ図において各ペアの左隅にペアの `car` と `cdr` がどこに格納されるかを指定するベクタの索引を表示しました。

<sup>14</sup>これは正確に Chapter 2で紹介したジェネリック(総称)な命令を扱うための“タグ付きデータ”と同じ考え方です。ここではしかし、データの型はリストの使用を通して構築されるのではなく、プリミティブな機械レベルにて含まれます。

<sup>15</sup>型情報は Lisp システムが実装される機械の詳細に依存して多様な方法でエンコード(`encode`、符号化)されるでしょう。Lisp プログラムの実行効率はこの選択がどれだけ明確に行われたかに強く依存します。しかし良い選択のための一般的な設計ルールを形式化することは難しいことです。型付きポインタを実装する最も簡単な方法は固定長のビット集合を各ポインタの中でデータ型をエンコードする *type field*(型フィールド)とする様に割り当てておくことです。そのような表現を設計するにおいて解決すべき重要な問題は次を含みます。いくつの型ビットが必要とされるか? ベクタの索引の長さはどれだけ必要か? どれだけ効率良くプリミティブな機械語命令がポインタの型フィールドの操作に使用できるか? 型フィールドを効率良く扱うための特別なハードウェアを含む機械は *tagged architectures*(タグアーキテクチャ)を持つと言われます。

A pointer to a number, such as `n4`, might consist of a type indicating numeric data together with the actual representation of the number 4.<sup>16</sup> To deal with numbers that are too large to be represented in the fixed amount of space allocated for a single pointer, we could use a distinct *bignum* data type, for which the pointer designates a list in which the parts of the number are stored.<sup>17</sup>

数値へのポインタ、例えば `n4` は数値データを示す型と実際の数値 4 の表現から成り立つでしょう。<sup>18</sup> 単一のポインタのために獲得された固定長のメモリの中で表現されるには大き過ぎる数値を扱うためには、独特な *bignum*(ビッグナンバー) データ型を使うことができるでしょう。このためのポインタは格納される数値の部分が格納されるリストを指定します。<sup>19</sup>

A symbol might be represented as a typed pointer that designates a sequence of the characters that form the symbol's printed representation. This sequence is constructed by the Lisp reader when the character string is initially encountered in input. Since we want two instances of a symbol to be recognized as the "same" symbol by `eq?` and we want `eq?` to be a simple test for equality of pointers, we must ensure that if the reader sees the same character string twice, it will use the same pointer (to the same sequence of characters) to represent both occurrences. To accomplish this, the reader maintains a table, traditionally called the *obarray*, of all the symbols it has ever encountered. When the reader encounters a character string and is about to construct a symbol, it checks the obarray to see if it has ever before seen the same character string. If it has not, it uses the characters to construct a new symbol (a typed pointer to a new

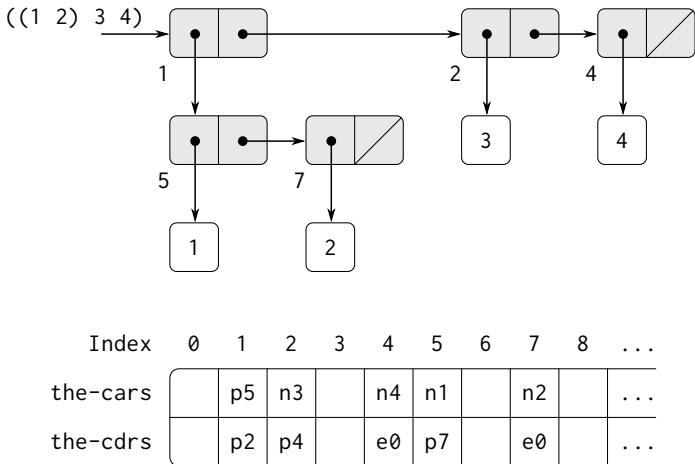
---

<sup>16</sup>This decision on the representation of numbers determines whether `eq?`, which tests equality of pointers, can be used to test for equality of numbers. If the pointer contains the number itself, then equal numbers will have the same pointer. But if the pointer contains the index of a location where the number is stored, equal numbers will be guaranteed to have equal pointers only if we are careful never to store the same number in more than one location.

<sup>17</sup>This is just like writing a number as a sequence of digits, except that each "digit" is a number between 0 and the largest number that can be stored in a single pointer.

<sup>18</sup>この数値の表現上の決断はポインタの等価性をテストする `eq?` が数値の等価性の試験に使用できるかどうかを決定します。もしポインタが数値それ自身を含む場合、等しい数値は同じポインタを持ちます。しかもしもポインタが数値が格納される位置の索引を持つ場合、私達が同じ数を複数の位置に格納しないことに注意しない場合に限り等しい数値が同じポインタになることが保証されます。

<sup>19</sup>これは丁度数値を数字の列として書くのに似ています。ただし各 "桁" が 0 から単一のポインタに格納できる最大の数の間になることが異なります。



**Figure 5.14:** Box-and-pointer and memory-vector representations of the list  $((1\ 2)\ 3\ 4)$ .

character sequence) and enters this pointer in the obarray. If the reader has seen the string before, it returns the symbol pointer stored in the obarray. This process of replacing character strings by unique pointers is called *interning* symbols.

シンボルはその表示内容を形成する文字の列を指定する型付きポインタとして表現されることができるでしょう。この列は Lisp の reader により、最初に入力の中の文字列に出くわした時に構築されます。2つのシンボルのインスタンスが eq? により “同じ” シンボルであると認識されて欲しいことと、eq? にポインタの等価性のための簡単なテストになって欲しいことから、もし reader が同じ文字列を 2 回見た場合、(同じ文字列に対する) 同じポインタを両方の出現に対して表現するために利用することを保証しなければなりません。これを達成するためには、reader は伝統的に *obarray*(オブジェクト配列) と呼ばれる出会った全てのシンボルの表を管理します。reader が文字列に遭遇しシンボルを構築しようとする時、obarray をチェックし同じ文字列を以前に見ていないか確認します。もし初見であれば、文字列を用いて新しいシンボル(新しい文字列に対する型付きポインタ)を構築し、このポインタを obarray に挿入しま

す。もし reader が既にその文字列を見ていれば、obarray に格納されているシンボルのポインタを返します。この文字列を一意なポインタで置き換える処理はシンボルの *interning*(抑留) と呼ばれます。

## Implementing the primitive list operations

Given the above representation scheme, we can replace each “primitive” list operation of a register machine with one or more primitive vector operations. We will use two registers, `the-cars` and `the-cdrs`, to identify the memory vectors, and will assume that `vector-ref` and `vector-set!` are available as primitive operations. We also assume that numeric operations on pointers (such as incrementing a pointer, using a pair pointer to index a vector, or adding two numbers) use only the index portion of the typed pointer.

上記の表現の構想を与えられた時に、レジスタマシンの各“プリミティブ”なリスト命令を複数のプリミティブなベクタ命令で置き換えることができます。2つのレジスタ `the-cars` と `the-cdrs` を用いてメモリベクタを特定し、`vector-ref` と `vector-set!` がプリミティブな命令として有効であると仮定します。またポインタ上の演算命令(例えばポインタをインクリメントする、ペアのポインタを用いてベクタを索引付けする、または2つの数値を足す)は型付きポインタの索引部分しか利用しません。

For example, we can make a register machine support the instructions

例えば、次の命令をサポートするレジスタマシンをその下の条件の下で作成することができます。

```
(assign <reg1> (op car) (reg <reg2>))
(assign <reg1> (op cdr) (reg <reg2>))
```

if we implement these, respectively, as

上の命令のそれぞれに対しこれらが実装されているとします。

```
(assign <reg1> (op vector-ref) (reg the-cars) (reg <reg2>))
(assign <reg1> (op vector-ref) (reg the-cdrs) (reg <reg2>))
```

The instructions

以下の命令は、

```
(perform (op set-car!) (reg <reg1>) (reg <reg2>))
(perform (op set-cdr!) (reg <reg1>) (reg <reg2>))
```

are implemented as

次のように実装されます。

```
(perform
  (op vector-set!) (reg the-cars) (reg <reg1>) (reg <reg2>))
(perform
  (op vector-set!) (reg the-cdrs) (reg <reg1>) (reg <reg2>))
```

`Cons` is performed by allocating an unused index and storing the arguments to `cons` in `the-cars` and `the-cdrs` at that indexed vector position. We presume that there is a special register, `free`, that always holds a pair pointer containing the next available index, and that we can increment the index part of that pointer to find the next free location.<sup>20</sup> For example, the instruction

`cons` は未使用の索引を割り当て、`cons` の引数を `the-cars` と `the-cdrs` の中で索引付けられたベクタの位置に格納します。私達は特別なレジスタ、`free` が存在し、常に次に使用可能な索引を持つペアポインタを保つと仮定します。そしてそのポインタの索引部分をインクリメントすることで次の空き位置を探すことができます。<sup>21</sup> 例えれば、以下の命令は

```
(assign <reg1> (op cons) (reg <reg2>) (reg <reg3>))
```

is implemented as the following sequence of vector operations:<sup>22</sup>

次の一連のベクタ命令として実装されます。<sup>23</sup>

```
(perform
  (op vector-set!) (reg the-cars) (reg free) (reg <reg2>))
```

---

<sup>20</sup>There are other ways of finding free storage. For example, we could link together all the unused pairs into a *free list*. Our free locations are consecutive (and hence can be accessed by incrementing a pointer) because we are using a compacting garbage collector, as we will see in Section 5.3.2.

<sup>21</sup>空きの記憶域を探す他の方法も存在します。例えば、全ての未使用的ペアをリンクして *free list*(空きリスト) にすることもできたでしょう。私達の空き位置は連続的(従ってポインタをインクリメントすることでアクセス可能であるため)です。なぜなら私達が圧縮 GC を用いているためです。またSection 5.3.2も参照して下さい。

<sup>22</sup>This is essentially the implementation of `cons` in terms of `set-car!` and `set-cdr!`, as described in Section 3.3.1. The operation `get-new-pair` used in that implementation is realized here by the `free` pointer.

<sup>23</sup>これは本質的にSection 3.3.1で説明した `set-car!` と `set-cdr!` を用いた `cons` の実装です。その実装内で使用された命令 `get-new-pair` はここでは `free` ポインタにより実現されています。

```
(perform
  (op vector-set!) (reg the-cdrs) (reg free) (reg <reg3>))
  (assign <reg1> (reg free))
  (assign free (op +) (reg free) (const 1))
```

The `eq?` operation

以下の `eq?` 命令は

```
(op eq?) (reg <reg1>) (reg <reg2>)
```

simply tests the equality of all fields in the registers, and predicates such as `pair?`, `null?`, `symbol?`, and `number?` need only check the type field.

単純にレジスタ内の全ての項目の等価性をテストします。そして `pair?`, `null?`, `symbol?`, `number?` 等のような述語は型フィールドのみを確認する必要があります。

## Implementing stacks

Although our register machines use stacks, we need do nothing special here, since stacks can be modeled in terms of lists. The stack can be a list of the saved values, pointed to by a special register `the-stack`. Thus, `(save <reg>)` can be implemented as

私達のレジスタマシンはスタックを用いますが、ここでは特に特別なことを行う必要がありません。スタックはリストを用いてモデル化することができます。スタックは保存した値のリストとすることができます、特別なレジスタ `the-stack` により指示されます。従って `(save <reg>)` は以下のように実装することができます。

```
(assign the-stack (op cons) (reg <reg>) (reg the-stack))
```

Similarly, `(restore <reg>)` can be implemented as

同様に、`(restore <reg>)` は次のように実装することができます。

```
(assign <reg> (op car) (reg the-stack))
(assign the-stack (op cdr) (reg the-stack))
```

and `(perform (op initialize-stack))` can be implemented as

そして `(perform (op initialize-stack))` は以下のように実装することができます。

```
(assign the-stack (const ()))
```

These operations can be further expanded in terms of the vector operations given above. In conventional computer architectures, however, it is usually advantageous to allocate the stack as a separate vector. Then pushing and popping the stack can be accomplished by incrementing or decrementing an index into that vector.

これらの命令は上で与えられたベクタ命令を用いてさらに伸展されます。しかし、伝統的な計算機アーキテクチャにおいてはスタックを別のベクタとして割り当てるることは通常は好都合です。そうすれば、スタックに push や pop を行うことはベクタに対する索引をインクリメント、デクリメントすることにより達成することができます。

**Exercise 5.20:** Draw the box-and-pointer representation and the memory-vector representation (as in [Figure 5.14](#)) of the list structure produced by

以下の式から生成されるリスト構造の表現と ([Figure 5.14](#)にあるような) メモリ-ベクタ表現の箱とポインタ図を描け。

```
(define x (cons 1 2))
(define y (list x x))
```

with the `free` pointer initially `p1`. What is the final value of `free`? What pointers represent the values of `x` and `y`?

ただし、`free` ポインタの初期値は `p1` とする。`free` の最終的な値は何か? どんなポインタが `x` と `y` の値を表現するか?

**Exercise 5.21:** Implement register machines for the following procedures. Assume that the list-structure memory operations are available as machine primitives.

以下の手続のためのレジスタマシンを実装せよ。リスト構造のメモリ命令は機械のプリミティブとして使用可能だと仮定せよ。

a Recursive `count-leaves`:

再帰 `count-leaves`:

```
(define (count-leaves tree)
  (cond ((null? tree) 0)
        ((not (pair? tree)) 1)
        (else (+ (count-leaves (car tree))
                  (count-leaves (cdr tree)))))))
```

b Recursive `count-leaves` with explicit counter:

明示的なカウンタを用いた再帰 `count-leaves`

```
(define (count-leaves tree)
  (define (count-iter tree n)
    (cond ((null? tree) n)
          ((not (pair? tree)) (+ n 1))
          (else
            (count-iter (cdr tree)
                        (count-iter (car tree)
                                    n)))))))
(count-iter tree 0))
```

**Exercise 5.22:** Exercise 3.12 of Section 3.3.1 presented an `append` procedure that appends two lists to form a new list and an `append!` procedure that splices two lists together. Design a register machine to implement each of these procedures. Assume that the list-structure memory operations are available as primitive operations.

Section 3.3.1のExercise 3.12は2つのリストを接続し1つの新しいリストを形成する `append` 手続と、2つのリストと一緒に繋ぎ合わせる `append!` 手続を紹介した。これらの手続それぞれを実装するレジスタマシンを設計せよ。リスト構造のメモリ命令はプリミティブな命令として使用可能と前提せよ。

### 5.3.2 Maintaining the Illusion of Infinite Memory

The representation method outlined in Section 5.3.1 solves the problem of implementing list structure, provided that we have an infinite amount of memory. With a real computer we will eventually run out of free space in which to construct new pairs.<sup>24</sup> However, most of the pairs generated in a

---

<sup>24</sup>This may not be true eventually, because memories may get large enough so that it would be impossible to run out of free memory in the lifetime of the computer. For example, there are about  $3 \cdot 10^{13}$  microseconds in a year, so if we were to `cons` once per microsecond we would need about  $10^{15}$  cells of memory to build a machine that could operate for 30 years without running out of memory. That much memory seems absurdly large by today's standards, but it is not physically impossible. On the other

typical computation are used only to hold intermediate results. After these results are accessed, the pairs are no longer needed—they are *garbage*. For instance, the computation

Section 5.3.1で概観した表現手法はリスト構造の実装上の問題を解決しましたが、無限の容量のメモリを持っている場合という条件付きでした。実際のコンピュータではいつかは新しいペアを構築するための空き容量を使い切ってしまいます。<sup>25</sup>しかし、典型的な演算により生成されるペアの多くは中間結果を保つためだけに使用されます。これらの結果がアクセスされた後には、それらのペアはもう必要ありません。それらは*garbage*(ゴミ)です。例えば、以下の演算は

```
(accumulate + 0 (filter odd? (enumerate-interval 0 n)))
```

constructs two lists: the enumeration and the result of filtering the enumeration. When the accumulation is complete, these lists are no longer needed, and the allocated memory can be reclaimed. If we can arrange to collect all the garbage periodically, and if this turns out to recycle memory at about the same rate at which we construct new pairs, we will have preserved the illusion that there is an infinite amount of memory.

2つのリストを構築します。enumeration(列挙)と列挙をフィルタリングした結果です。accumulation(集積)が完了した時に、これらのリストはもう必要ありません。そして割り当てられたメモリは返還要求できます。もし全てのゴミを定期的に回収する準備を行えるのであれば、そもそもしこれが新しいペアを構築するのと大体同じ比率でメモリをリサイクルすることになれば、無限の容量のメモリが存在するという錯覚を維持することができます。

In order to recycle pairs, we must have a way to determine which allocated

---

hand, processors are getting faster and a future computer may have large numbers of processors operating in parallel on a single memory, so it may be possible to use up memory much faster than we have postulated.

<sup>25</sup>これはいつかは正しくはなくなるかもしれません。なぜならメモリが十分に大きくなればコンピュータの生存時間の間には空きメモリを使い切ることは不可能になるかもしれませんからです。例えば一年は  $3 \cdot 10^{13}$  マイクロ秒ですから、もし 1 マイクロ秒に 1 回 cons を行うのであれば、30 年間はメモリを使い切ることのないコンピュータを構築するには約  $10^{15}$  セルのメモリを必要とします。それだけのメモリは今日の標準では話にならない程大きく見えますが、しかし物理的に不可能ではありません。一方で、プロセッサはより速くなりつつあり未来のコンピュータは数多くのプロセッサを並列に单一のメモリ上で作動するかもしれません。従って私達の前提よりもよい早くメモリを使い切ることが可能かもしれません。

pairs are not needed (in the sense that their contents can no longer influence the future of the computation). The method we shall examine for accomplishing this is known as *garbage collection*. Garbage collection is based on the observation that, at any moment in a Lisp interpretation, the only objects that can affect the future of the computation are those that can be reached by some succession of `car` and `cdr` operations starting from the pointers that are currently in the machine registers.<sup>26</sup> Any memory cell that is not so accessible may be recycled.

ペアをリサイクルするためには、どの割り当てられたペアが必要でないか(それらの中身がその後将来の演算に影響しないという意味で)決定する方法を持たねばなりません。これを達成するために調査する手法は*garbage collection*(ガベージコレクション、GC)として知られています。ガベージコレクションは Lisp の逐次実行における任意の時点で、将来の演算に影響を与えることができるオブジェクトは現状で機械のレジスタ内に存在するポインタにより辿り着くことができるオブジェクトのみであるという観察結果に基いています。<sup>27</sup> そのようにアクセスできないどのメモリセルもリサイクルして良いでしょう。

There are many ways to perform garbage collection. The method we shall examine here is called *stop-and-copy*. The basic idea is to divide memory into two halves: “working memory” and “free memory.” When `cons` constructs pairs, it allocates these in working memory. When working memory is full, we perform garbage collection by locating all the useful pairs in working memory and copying these into consecutive locations in free memory. (The useful pairs are located by tracing all the `car` and `cdr` pointers, starting with the machine registers.) Since we do not copy the garbage, there will presumably be additional free memory that we can use to allocate new pairs. In addition, nothing in the working memory is needed, since all the useful pairs in it have been copied. Thus, if we interchange the roles of working memory and free memory, we can continue processing; new pairs will be allocated in the new working memory (which was the old free memory). When this is full, we can copy the useful pairs into the new free memory (which was the old working memory).

ガベージコレクションを実行する方法は数多く存在します。ここで調査す

---

<sup>26</sup>We assume here that the stack is represented as a list as described in Section 5.3.1, so that items on the stack are accessible via the pointer in the stack register.

<sup>27</sup>ここではスタックはSection 5.3.1で説明されたリストとして表現されていると仮定しています。そのためスタック上の項目はスタックレジスタ内のポインタを通してアクセスすることができます。

る手法は *stop-and-copy* と呼ばれます。基本的な考えはメモリを 2 つに割ります。“ワーキングメモリ”と“空きメモリ”です。`cons` がペアを構築する時、ワーキングメモリに割り当てます。ワーキングメモリに空きが無い時、ワーキングメモリ内の使い道のある全てのペアを探し出し、これらをフリーメモリ内の連続した位置にコピーすることでガベージコレクションを実行します。(使い道のあるペアは機械のレジスタから始めて、全ての `car` と `cdr` のポインタを追跡することにより探し出します)。ゴミはコピーしないため、推定上、新しいペアを割り当てるための利用できる追加の空きメモリが存在するはずです。加えて、ワーキングメモリ内の全てが必要ありません。その中の使い道のあるペアはコピーされています。従ってワーキングメモリと空きメモリの役割を交換すれば、処理を続けることができます。新しいペアは新しいワーキングメモリ(空きメモリだった物)の中に割り当てられます。これがいっぱいになったなら、使い道のあるペアを新しい空きメモリ(ワーキングメモリだったもの)の中にコピーできます。<sup>28</sup>

<sup>28</sup>This idea was invented and first implemented by Minsky, as part of the implementation of Lisp for the PDP-1 at the MIT Research Laboratory of Electronics. It was further developed by Fenichel and Yochelson (1969) for use in the Lisp implementation for the Multics time-sharing system. Later, Baker (1978) developed a “real-time” version of the method, which does not require the computation to stop during garbage collection. Baker’s idea was extended by Hewitt, Lieberman, and Moon (see Lieberman and Hewitt 1983) to take advantage of the fact that some structure is more volatile and other structure is more permanent.

この考えは Minsky(ミンスキー) により発明され、MIT 研究所の電子工学ラボの PDP-1 に対する Lisp の実装の一部として実装されました。Fenichel and Yochelson (1969) により Multics 時分割システムの Lisp 実装で使用するために、さらに開発が進められました。後に、Baker (1978) はこの手法の“リアルタイム”版を開発しました。これはガベージコレクションの間に演算を停止する必要がありません。Baker の考えは Hewitt, Lieberman, Moon により拡張され (Lieberman and Hewitt 1983 参照)、ある構造は volatile(揮発性) であり、別の構造はより永続的であるといった事実を活用する様になりました。

An alternative commonly used garbage-collection technique is the *mark-sweep* method. This consists of tracing all the structure accessible from the machine registers and marking each pair we reach. We then scan all of memory, and any location that is unmarked is “swept up” as garbage and made available for reuse. A full discussion of the mark-sweep method can be found in Allen 1978.

一般に利用される代替的なガベージコレクションの技術は *mark-sweep*(マークアンドスイープ) の手法です。これは回帰のレジスタからアクセス可能な全ての構造の追跡と辿り着く各ペアへのマーキングから成り立ちます。次に全てのメモリを走査し、マークの無い全てのメモリはゴミとして“掃き出し”、再使用可能とされます。マークアンドスイープの十分な議論は Allen 1978 の中に見つけられます。

## Implementation of a stop-and-copy garbage collector

We now use our register-machine language to describe the stop-and-copy algorithm in more detail. We will assume that there is a register called `root` that contains a pointer to a structure that eventually points at all accessible data. This can be arranged by storing the contents of all the machine registers in a pre-allocated list pointed at by `root` just before starting garbage collection.<sup>29</sup> We also assume that, in addition to the current working memory, there is free memory available into which we can copy the useful data. The current working memory consists of vectors whose base addresses are in registers called `the-cars` and `the-cdrs`, and the free memory is in registers called `new-cars` and `new-cdrs`.

今から私達はレジスタマシン言語を用いて stop-and-copy アルゴリズムをより詳細に記述します。私達は `root` と呼ばれるレジスタが存在し、ある構造体へのポインタを保持し、そのポインタから最終的には全てのアクセス可能なデータを指示示すことができるという前提を行います。これはガベージコレクションを行う直前に全てのレジスタの中身を事前に割り当てられたリストに格

---

The Minsky-Fenichel-Yochelson algorithm is the dominant algorithm in use for large-memory systems because it examines only the useful part of memory. This is in contrast to mark-sweep, in which the sweep phase must check all of memory. A second advantage of stop-and-copy is that it is a *compacting* garbage collector. That is, at the end of the garbage-collection phase the useful data will have been moved to consecutive memory locations, with all garbage pairs compressed out. This can be an extremely important performance consideration in machines with virtual memory, in which accesses to widely separated memory addresses may require extra paging operations.

Minsky-Fenichel-Yochelson アルゴリズムは巨大なメモリシステムに対する使用における支配的なアルゴリズムです。メモリの使い道のある部分のみを調査するためです。これはスイープの段階で全てのメモリを確認しなければならない mark-and-sweep とは対照的です。stop-and-copy の 2 つ目の強みは *compacting*(圧縮) ガベージコレクタであることです。つまり、ガベージコレクションの段階の終わりには使い道のあるデータは連続したメモリ位置に移動され、全てのゴミペアは圧縮の仮定で外に出されます。このことが仮想メモリを使用する機械におけるパフォーマンス上の考慮において非常に重要と成ります。仮想メモリを使用する機械は広範囲に分離されたメモリアドレスへのアクセスに余計なページング処理が必要となるかもしれません。

<sup>29</sup>This list of registers does not include the registers used by the storage-allocation system—`root`, `the-cars`, `the-cdrs`, and the other registers that will be introduced in this section.

納し、`root` により指し示させることで準備が行えます。<sup>30</sup> 私達はまた現在のワーキングメモリに加えて、使い道のあるデータをコピーできる空きメモリが存在すると前提します。現在のワーキングメモリはベースアドレスが `the-cars` と `the-cdrs` と呼ばれるレジスタに格納されるベクタから成り立ち、そして空きメモリは同様に `new-cars` と `new-cdrs` と呼ばれるレジスタに格納されます。

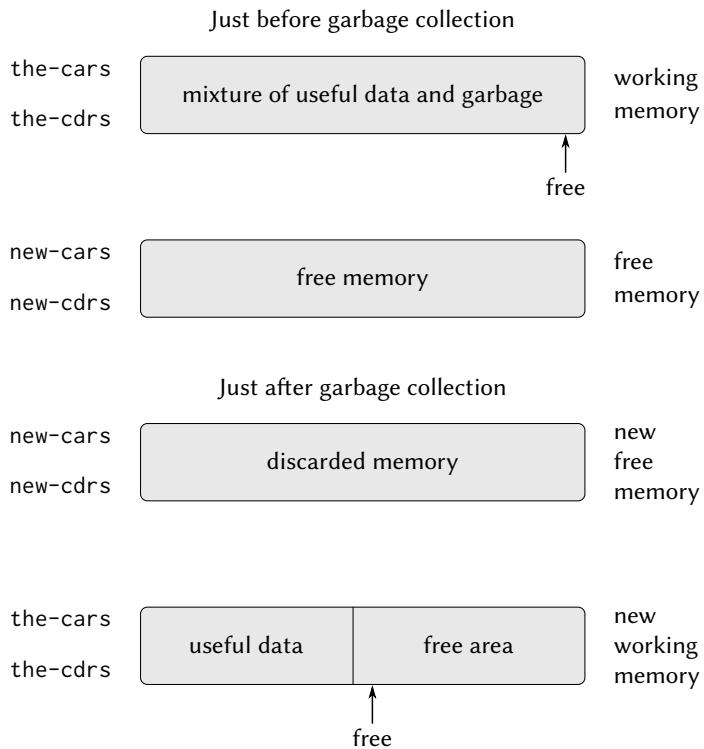
Garbage collection is triggered when we exhaust the free cells in the current working memory, that is, when a `cons` operation attempts to increment the `free` pointer beyond the end of the memory vector. When the garbage-collection process is complete, the `root` pointer will point into the new memory, all objects accessible from the `root` will have been moved to the new memory, and the `free` pointer will indicate the next place in the new memory where a new pair can be allocated. In addition, the roles of working memory and new memory will have been interchanged—new pairs will be constructed in the new memory, beginning at the place indicated by `free`, and the (previous) working memory will be available as the new memory for the next garbage collection. Figure 5.15 shows the arrangement of memory just before and just after garbage collection.

ガベージコレクションは現在のワーキングメモリ内の空きセルが枯渇した時に引き起こされます。それはつまり、`cons` 命令が `free` ポインタをメモリベクタの終端を越えてインクリメントしようとした時です。ガベージコレクションの処理が完了した時、`root` ポインタは新しいメモリの中を指し示し、`root` からアクセス可能な全てのオブジェクトは新しいメモリに移動されています。そして `free` ポインタは新しいメモリ内の新しいペアを割り当てる次の位置を示します。加えて、ワーキングメモリと新しいメモリの役割が交換されます。新しいペアは `free` により指し示される位置から始まる新しいメモリ内に構築され、(以前の)ワーキングメモリは次のガベージコレクションに対する新しいメモリとして使用可能となります。Figure 5.15はガベージコレクション直前、直後のメモリの割り振りを示します。

The state of the garbage-collection process is controlled by maintaining two pointers: `free` and `scan`. These are initialized to point to the beginning of the new memory. The algorithm begins by relocating the pair pointed at by `root` to the beginning of the new memory. The pair is copied, the `root` pointer is adjusted to point to the new location, and the `free` pointer is incremented. In addition, the old location of the pair is marked to show that its contents have been moved. This marking is done as follows: In the `car` position, we place a

---

<sup>30</sup> このレジスタのリストは記憶域割当システムのレジスタ — `root`, `the-cars`, `the-cdrs`, それにこの節で紹介される他のレジスタは含みません。



**Figure 5.15:** Reconfiguration of memory by the garbage-collection process.

special tag that signals that this is an already-moved object. (Such an object is traditionally called a *broken heart*).<sup>31</sup> In the `cdr` position we place a *forwarding address* that points at the location to which the object has been moved.

ガベージコレクション処理の状態は 2 つのポインタを管理することによりコントロールされています。`free` と `scan` です。これらは新しいメモリの開始位置を指し示すように初期化されます。アルゴリズムは `root` により指し示されるペアの新しいメモリの開始位置への再配置から開始されます。ペアはコピーされ、`root` ポインタは新しい位置を指すように調整されます。そして `free` ポインタがインクリメントされます。併せて、ペアの古い位置はその中身が移動されたことを示すマークが付けられます。このマーキングは次のように行われます。`car` の位置にはこれが既に移動されたオブジェクトであることを示す特別なタグを置きます。(そのようなオブジェクトは伝統的に *broken heart*(失恋) と呼ばれます。)<sup>32</sup> `cdr` の位置には *forwarding address*(転送先) を置きます。これはオブジェクトの移動先の位置を指し示します。

After relocating the root, the garbage collector enters its basic cycle. At each step in the algorithm, the `scan` pointer (initially pointing at the relocated root) points at a pair that has been moved to the new memory but whose `car` and `cdr` pointers still refer to objects in the old memory. These objects are each relocated, and the `scan` pointer is incremented. To relocate an object (for example, the object indicated by the `car` pointer of the pair we are scanning) we check to see if the object has already been moved (as indicated by the presence of a broken-heart tag in the `car` position of the object). If the object has not already been moved, we copy it to the place indicated by `free`, update `free`, set up a broken heart at the object's old location, and update the pointer to the object (in this example, the `car` pointer of the pair we are scanning) to point to the new location. If the object has already been moved, its forwarding address (found in the `cdr` position of the broken heart) is substituted for the pointer in the pair being scanned. Eventually, all accessible objects will have been moved and scanned, at which point the `scan` pointer will overtake the `free` pointer and the process will terminate.

`root` の再配置の後に、ガベージコレクタは基本となるサイクルに入ります。アルゴリズムの各ステップにおいて、`scan` ポインタ(初期値として再配置後の

---

<sup>31</sup>The term *broken heart* was coined by David Cressey, who wrote a garbage collector for MDL, a dialect of Lisp developed at MIT during the early 1970s.

<sup>32</sup>*broken heart* という用語は David Cressey により作られました。彼は 1970 年代初期の間に MIT で開発された Lisp の方言、MDL のためにガベージコレクタを書きました。

`root` を指す) は、新しいメモリに移動されたがその `car` と `cdr` のポインタが依然として古いメモリ内のオブジェクトを参照しているペアを指します。これらのオブジェクトはそれぞれが再配置され、`scan` ポインタはインクリメントされます。オブジェクト(例えば走査しているペアの `car` ポインタにより指されたオブジェクト)を再配置するためにはそのオブジェクトが既に移動されていないかを(そのオブジェクトの `car` の位置内に broken-heart タグが存在することにより示されていないか)確認します。もしオブジェクトがまだ移動されていなければ、それを `free` により示される位置にコピーし、`free` を更新し、オブジェクトの古い位置に broken-heart を設定し、そのオブジェクトへのポインタを(この礼では、走査しているペアの `car` ポインタを)更新し、新しい位置を指すようにします。もしオブジェクトが既に移動されている場合には、(broken heart の `cdr` の位置に見つかる)その移動先は走査中のペアのポインタに置き換えられます。最終的には、`scan` ポインタが `free` ポインタを追い越す時点まで、全てのアクセス可能なオブジェクトは移動され、精査されます。そして処理は停止します。

We can specify the stop-and-copy algorithm as a sequence of instructions for a register machine. The basic step of relocating an object is accomplished by a subroutine called `relocate-old-result-in-new`. This subroutine gets its argument, a pointer to the object to be relocated, from a register named `old`. It relocates the designated object (incrementing `free` in the process), puts a pointer to the relocated object into a register called `new`, and returns by branching to the entry point stored in the register `relocate-continue`. To begin garbage collection, we invoke this subroutine to relocate the `root` pointer, after initializing `free` and `scan`. When the relocation of `root` has been accomplished, we install the new pointer as the new `root` and enter the main loop of the garbage collector.

stop-and-copy アルゴリズムをレジスタマシンの命令列として記述することができます。オブジェクトの再配置の基本的なステップは `relocate-old-result-in-new` と呼ばれるサブルーチンにて達成されます。このサブルーチンはその引数として再配置するオブジェクトのポインタを `old` という名のレジスタから取得します。これは指定されたオブジェクトを再配置し、(処理の間に `free` をインクリメントし)、再配置されたオブジェクトを指すポインタを `new` と呼ばれるレジスタに入れます。そして `relocate-continue` レジスタに格納されたエントリポイントへ分岐することで帰ります。ガベージコレクションを始めるために、このサブルーチンを起動して、`free` と `scan` を初期化した後に `root` ポインタを再配置します。`root` の再配置が完了した時に、`new` ポインタを

新しい `root` として導入し、ガベージコレクタのメインループに入ります。

```
begin-garbage-collection
  (assign free (const 0))
  (assign scan (const 0))
  (assign old (reg root))
  (assign relocate-continue (label reassign-root))
  (goto (label relocate-old-result-in-new))

reassgin-root
  (assign root (reg new))
  (goto (label gc-loop))
```

In the main loop of the garbage collector we must determine whether there are any more objects to be scanned. We do this by testing whether the `scan` pointer is coincident with the `free` pointer. If the pointers are equal, then all accessible objects have been relocated, and we branch to `gc-flip`, which cleans things up so that we can continue the interrupted computation. If there are still pairs to be scanned, we call the `relocate` subroutine to relocate the `car` of the next pair (by placing the `car` pointer in `old`). The `relocate-continue` register is set up so that the subroutine will return to update the `car` pointer.

ガベージコレクタのメインループでは走査すべきオブジェクトが残っているのか決定しなければなりません。これを `scan` ポインタが `free` ポインタと一致するかどうかを試験することで行います。もしポインタが等しければ、全てのアクセス可能なオブジェクトの再配置は完了し、`gc-flip`へと分岐します。ここは後片付けを行い、割り込みが行われた演算を継続します。もしまだ走査すべきペアが残っているのであれば、再配置 (`relocate`) のサブルーチンを呼び出し次のペアの `car` を (`old` 内の `car` ポインタを配置することで) 再配置します。`relocate-continue` レジスタの設定によりサブルーチンは `car` ポインタを更新するために帰ります。

```
gc-loop
  (test (op =) (reg scan) (reg free))
  (branch (label gc-flip))
  (assign old (op vector-ref) (reg new-cars) (reg scan))
  (assign relocate-continue (label update-car))
  (goto (label relocate-old-result-in-new))
```

At `update-car`, we modify the `car` pointer of the pair being scanned, then proceed to relocate the `cdr` of the pair. We return to `update-cdr` when that

relocation has been accomplished. After relocating and updating the `cdr`, we are finished scanning that pair, so we continue with the main loop.

`update-car` にて、精査しているペアの `car` ポインタを変更します。次にペアの `cdr` を再配置するために向かいます。再配置が完了すると `update-cdr` に帰ってきます。再配置と `cdr` の更新の後に、そのペアの精査を完了しメインループを継続します。

```
update-car
  (perform (op vector-set!)
    (reg new-cars)
    (reg scan)
    (reg new))
  (assign old (op vector-ref) (reg new-cdrs) (reg scan))
  (assign relocate-continue (label update-cdr))
  (goto (label relocate-old-result-in-new))

update-cdr
  (perform (op vector-set!)
    (reg new-cdrs)
    (reg scan)
    (reg new))
  (assign scan (op +) (reg scan) (const 1))
  (goto (label gc-loop))
```

The subroutine `relocate-old-result-in-new` relocates objects as follows: If the object to be relocated (pointed at by `old`) is not a pair, then we return the same pointer to the object unchanged (in `new`). (For example, we may be scanning a pair whose `car` is the number 4. If we represent the `car` by `n4`, as described in Section 5.3.1, then we want the “relocated” `car` pointer to still be `n4`.) Otherwise, we must perform the relocation. If the `car` position of the pair to be relocated contains a broken-heart tag, then the pair has in fact already been moved, so we retrieve the forwarding address (from the `cdr` position of the broken heart) and return this in `new`. If the pointer in `old` points at a yet-unmoved pair, then we move the pair to the first free cell in new memory (pointed at by `free`) and set up the broken heart by storing a broken-heart tag and forwarding address at the old location. `Relocate-old-result-in-new` uses a register `oldcr` to hold the `car` or the `cdr` of the object pointed at by

old.<sup>33</sup>

サブルーチン `relocate-old-result-in-new` はオブジェクトを次のように再配置します。もし (`old` により指し示される) 再配置すべきオブジェクトがペアでないなら、そのオブジェクトへの同じポインタを変更無しで (`new` の中で) 返します。(例えば、`car` が数値の 4 であるペアを精査しているとします。もし Section 5.3.1 にて説明されているように `n4` で `car` を表現するのなら、“再配置された”`car` のポインタも依然として `n4` であって欲しいと願うはずです)。そうでなければ、再配置を実行しなければなりません。もしペアの再配置すべき `car` の位置に `broken-heart` タグを持つのならば、そのペアは実際には既に移動されています。従って (`broken-heart` の `cdr` の位置から) 移動先を取得し、これを `new` に入れて返します。もし `old` 内のポインタがまだ移動されていないペアを指す場合、そのペアを (`free` が指し示す) 新しいメモリの最初の空きセルに移動させ、`broken-heart` タグと移動先を元の位置に格納することで `broken-heart` を設定します。`relocate-old-result-in-new` はレジスタ `oldcr` を用いて `old` により指し示されるオブジェクトの `car` または `cdr` を保持します。<sup>34</sup>

```
relocate-old-result-in-new
  (test (op pointer-to-pair?) (reg old))
  (branch (label pair))
  (assign new (reg old))
  (goto (reg relocate-continue))

pair
  (assign oldcr (op vector-ref) (reg the-cars) (reg old))
  (test (op broken-heart?) (reg oldcr))
  (branch (label already-moved))
```

---

<sup>33</sup>The garbage collector uses the low-level predicate `pointer-to-pair?` instead of the list-structure `pair?` operation because in a real system there might be various things that are treated as pairs for garbage-collection purposes. For example, in a Scheme system that conforms to the IEEE standard a procedure object may be implemented as a special kind of “pair” that doesn’t satisfy the `pair?` predicate. For simulation purposes, `pointer-to-pair?` can be implemented as `pair?`.

<sup>34</sup>ガベージコレクタは低レベルの述語 `pointer-to-pair?` をリスト構造 `pair?` 命令の代わりに使用します。実際のシステムでは様々な物がガベージコレクションの目的のためにペアとして扱われるためです。例えば、IEEE 標準に準拠する Scheme システムでは手続オブジェクトは特別な種類の“ペア”として実装されても良くこれは述語 `pair?` は満たしません。シミュレーションの目的には、`pointer-to-pair?` は `pair?` として実装できます。

```

(assign new (reg free)) ; ペアの新しい位置
;; free ポインタを更新する
(assign free (op +) (reg free) (const 1))
;; car と cdr を新しいメモリにコピーする。
(perform (op vector-set!)
  (reg new-cars) (reg new) (reg oldcr))
(assign oldcr (op vector-ref) (reg the-cdrs) (reg old))
(perform (op vector-set!)
  (reg new-cdrs) (reg new) (reg oldcr))
;; ブローケンハートの構築
(perform (op vector-set!)
  (reg the-cars) (reg old) (const broken-heart))
(perform
  (op vector-set!) (reg the-cdrs) (reg old) (reg new))
(goto (reg relocate-continue))

already-moved
  (assign new (op vector-ref) (reg the-cdrs) (reg old))
  (goto (reg relocate-continue))

```

At the very end of the garbage-collection process, we interchange the role of old and new memories by interchanging pointers: interchanging `the-cars` with `new-cars`, and `the-cdrs` with `new-cdrs`. We will then be ready to perform another garbage collection the next time memory runs out.

ガベージコレクション処理の最後に、メモリの新旧の役割をポインタを交換することにより交代します。`the-cars` と `new-cars`、そして `the-cdrs` を `new-cdrs` を交換します。これで次回メモリが枯渇した時にもう一度ガベージコレクションを行う準備ができます。

```

gc-flip
  (assign temp (reg the-cdrs))
  (assign the-cdrs (reg new-cdrs))
  (assign new-cdrs (reg temp))
  (assign temp (reg the-cars))
  (assign the-cars (reg new-cars))
  (assign new-cars (reg temp))

```

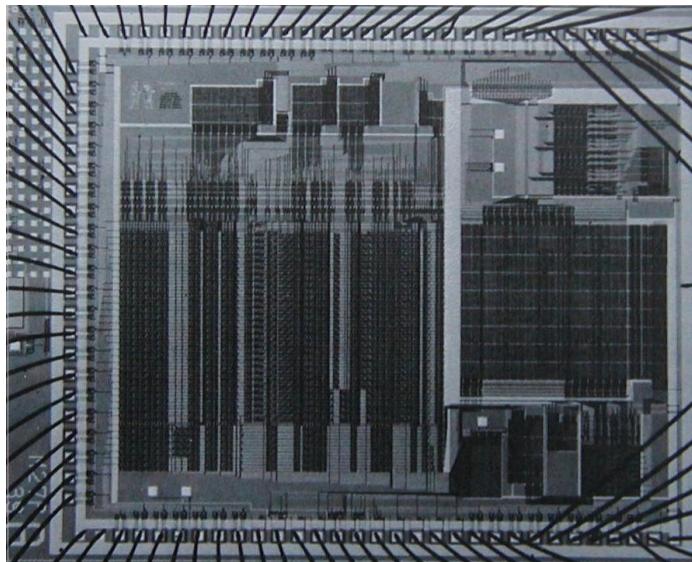
## 5.4 The Explicit-Control Evaluator

In Section 5.1 we saw how to transform simple Scheme programs into descriptions of register machines. We will now perform this transformation on a more complex program, the metacircular evaluator of Section 4.1.1–Section 4.1.4, which shows how the behavior of a Scheme interpreter can be described in terms of the procedures `eval` and `apply`. The *explicit-control evaluator* that we develop in this section shows how the underlying procedure-calling and argument-passing mechanisms used in the evaluation process can be described in terms of operations on registers and stacks. In addition, the explicit-control evaluator can serve as an implementation of a Scheme interpreter, written in a language that is very similar to the native machine language of conventional computers. The evaluator can be executed by the register-machine simulator of Section 5.2. Alternatively, it can be used as a starting point for building a machine-language implementation of a Scheme evaluator, or even a special-purpose machine for evaluating Scheme expressions. Figure 5.16 shows such a hardware implementation: a silicon chip that acts as an evaluator for Scheme. The chip designers started with the data-path and controller specifications for a register machine similar to the evaluator described in this section and used design automation programs to construct the integrated-circuit layout.<sup>35</sup>

Section 5.1では簡単な Scheme プログラムをどのようにレジスタマシンの記述に変形するかについて学びました。ここではこの変形をより複雑なプログラム上で実行します。Section 4.1.1–Section 4.1.4のメタ循環評価機です。メタ循環評価機は Scheme インタプリタの振舞が手続 `eval` と `apply` と用いてどのように説明できるかを示しました。この節で開発する *explicit-control evaluator*(明示的制御評価機)は評価過程にて使用される潜在的な手続呼出と引数受け渡しの仕組みがレジスタとスタックの命令を用いてどのように説明できるかを示します。付け加えて、明示的制御評価機は Scheme インタプリタの実装としての役割を果たすことができ、従来の計算機の生來の機械語とともにそつくりな言語で書かれています。この評価機は Section 5.2 のレジスタマシンシミュレータにより実行することができます。あるいは、Scheme 評価機の機械語実装を構築するための開始点として使用することができます。または Scheme の式を評価するための特殊用途の機械にすら使用できるでしょう。Figure 5.16はそのようなハードウェア実装を示しています。Scheme の評価機として働くシリコン

---

<sup>35</sup>See Batali et al. 1982 for more information on the chip and the method by which it was designed.



**Figure 5.16:** A silicon-chip implementation of an evaluator for Scheme.

チップです。このチップの設計者はこの節の中で説明される評価機に似たレジスタマシンに対するデータバスとコントローラの仕様から開始しました。そして IC(integrated-circuit、統合回路) を構築するための設計自動化プログラムを使用しました。<sup>36</sup>

### Registers and operations

In designing the explicit-control evaluator, we must specify the operations to be used in our register machine. We described the metacircular evaluator in terms of abstract syntax, using procedures such as `quoted?` and `make-`

---

<sup>36</sup> このチップとその設計手法についてのより多くの情報についてはBatali et al. 1982を参照して下さい。

**procedure**. In implementing the register machine, we could expand these procedures into sequences of elementary list-structure memory operations, and implement these operations on our register machine. However, this would make our evaluator very long, obscuring the basic structure with details. To clarify the presentation, we will include as primitive operations of the register machine the syntax procedures given in [Section 4.1.2](#) and the procedures for representing environments and other run-time data given in sections [Section 4.1.3](#) and [Section 4.1.4](#). In order to completely specify an evaluator that could be programmed in a low-level machine language or implemented in hardware, we would replace these operations by more elementary operations, using the list-structure implementation we described in [Section 5.3](#).

明示的制御評価機の設計では、私達のレジスタマシンで使用される命令を指定しなければなりません。私達は `quoted?` や `make-procedure` のような手続を用いることで抽象構文を用いたメタ循環評価機を説明しました。レジスタマシンの実装においてはこれらの手続を初步的なリスト構造メモリの命令列に展開することができました。そしてこれらの命令を私達のレジスタマシン上に実装しました。しかし、これは私達の評価機の基本的な構造をその詳細によりわかりにくくしながら、とても長くしてしまいます。その表現を明快にするために、レジスタマシンのプリミティブな命令として [Section 4.1.2](#)で与えられた構文手続と環境を表現するための手続、それに [Section 4.1.3](#)と [Section 4.1.4](#)で与えられた実行時データを含めることにします。低レベルのマシン語でプログラミングできる、またはハードウェアにて実装できる評価機の完全な仕様化を行うために、[Section 5.3](#)で説明したリスト構造の実装を用いてこれらの命令をより基本的な命令により置き換えることができるでしょう。

Our Scheme evaluator register machine includes a stack and seven registers: `exp`, `env`, `val`, `continue`, `proc`, `arg1`, and `unev`. `Exp` is used to hold the expression to be evaluated, and `env` contains the environment in which the evaluation is to be performed. At the end of an evaluation, `val` contains the value obtained by evaluating the expression in the designated environment. The `continue` register is used to implement recursion, as explained in [Section 5.1.4](#). (The evaluator needs to call itself recursively, since evaluating an expression requires evaluating its subexpressions.) The registers `proc`, `arg1`, and `unev` are used in evaluating combinations.

私達の Scheme 評価機レジスタマシンはスタックと 7 つのレジスタを含みます。`exp`, `env`, `val`, `continue`, `proc`, `arg1`, `unev` です。`exp` は評価される式の保持に仕様され、`env` は評価がその中で実行される環境を持ちます。評価の終

わりには、`val` が指定された環境における式の評価により得られた値を保持します。`continue` レジスタはSection 5.1.4で説明されたように再帰の実装に用いられます。(評価機はそれ自身を再帰的に呼び出す必要があります。式の評価はその部分式の評価を必要とするためです)。レジスタ `proc`, `argl`, `unev` は組み合わせの評価に用いられます。

We will not provide a data-path diagram to show how the registers and operations of the evaluator are connected, nor will we give the complete list of machine operations. These are implicit in the evaluator's controller, which will be presented in detail.

私達はデータパス図を評価機のレジスタと命令がどのように接続されているかを示すために提供はしません。また機械の命令の完全なリストの提供も行いません。これらは評価機のコントローラに暗黙的に存在し、コントローラの詳細が与えられます。

#### 5.4.1 The Core of the Explicit-Control Evaluator

The central element in the evaluator is the sequence of instructions beginning at `eval-dispatch`. This corresponds to the `eval` procedure of the metacircular evaluator described in Section 4.1.1. When the controller starts at `eval-dispatch`, it evaluates the expression specified by `exp` in the environment specified by `env`. When evaluation is complete, the controller will go to the entry point stored in `continue`, and the `val` register will hold the value of the expression. As with the metacircular `eval`, the structure of `eval-dispatch` is a case analysis on the syntactic type of the expression to be evaluated.<sup>37</sup>

評価機の中心的な要素は `eval-dispatch` で始まる命令列です。これはSection 4.1.1で説明されたメタ循環評価機の `eval` 手続に対応します。コントローラが `eval-dispatch` から開始する時、`exp` により指定された式を、`env` により指定された環境にて評価します。評価が完了した時には、コントローラは `continue` に格納されたエントリポイントに飛びます。その時、`val` レジスタが式の値を保持しています。メタ循環の `eval` と同様に、`eval-dispatch` の構造は評価さ

---

<sup>37</sup>In our controller, the dispatch is written as a sequence of `test` and `branch` instructions. Alternatively, it could have been written in a data-directed style (and in a real system it probably would have been) to avoid the need to perform sequential tests and to facilitate the definition of new expression types. A machine designed to run Lisp would probably include a `dispatch-on-type` instruction that would efficiently execute such data-directed dispatches.

れる式の構文型上の事例分析です。<sup>38</sup>

```
eval-dispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
  (test (op quoted?) (reg exp))
  (branch (label ev-quoted))
  (test (op assignment?) (reg exp))
  (branch (label ev-assignment))
  (test (op definition?) (reg exp))
  (branch (label ev-definition))
  (test (op if?) (reg exp))
  (branch (label ev-if))
  (test (op lambda?) (reg exp))
  (branch (label ev-lambda))
  (test (op begin?) (reg exp))
  (branch (label ev-begin))
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type))
```

## Evaluating simple expressions

Numbers and strings (which are self-evaluating), variables, quotations, and `lambda` expressions have no subexpressions to be evaluated. For these, the evaluator simply places the correct value in the `val` register and continues execution at the entry point specified by `continue`. Evaluation of simple expressions is performed by the following controller code:

---

<sup>38</sup>私達のコントローラにおいては、ディスパッチ (dispatch、割り振り) は `test` と `branch` の命令列として書かれています。代替法として、データ適従スタイルで書くこともできるでしょう（そして実際のシステムは恐らくそうされているでしょう）。連続したテストの実行の必要を防ぎ、新しい式の型の定義を用意にするためです。Lisp を実行するように設計された機械は恐らく `dispatch-on-type` 命令を含むことでしょう。これはそのようなデータに従った割り振りを効率的に実行します。

数値と文字列(これらは自己評価です)、変数、クオーテーション、そして `lambda` 式は評価するべき部分式がありません。これらのために、評価機は単純に正しい値を `val` レジスタに配置し、`continue` により指定されたエントリポイントから実行を継続します。単純な式の評価は以下のコントローラのコードにより実行されます。

```
ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))

ev-variable
  (assign
    val (op lookup-variable-value) (reg exp) (reg env))
  (goto (reg continue))

ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))

ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  (assign val (op make-procedure)
    (reg unev) (reg exp) (reg env))
  (goto (reg continue))
```

Observe how `ev-lambda` uses the `unev` and `exp` registers to hold the parameters and body of the lambda expression so that they can be passed to the `make-procedure` operation, along with the environment in `env`.

`ev-lambda` がどのように `unev` と `exp` レジスタを用いてラムダ式のパラメタとボディを保持し、`env` の中の環境と共に `make-procedure` 命令に引き渡されるのか観察して下さい。

## Evaluating procedure applications

A procedure application is specified by a combination containing an operator and operands. The operator is a subexpression whose value is a procedure, and the operands are subexpressions whose values are the arguments to which the procedure should be applied. The metacircular `eval` handles applications by calling itself recursively to evaluate each element of the combination, and

then passing the results to `apply`, which performs the actual procedure application. The explicit-control evaluator does the same thing; these recursive calls are implemented by `goto` instructions, together with use of the stack to save registers that will be restored after the recursive call returns. Before each call we will be careful to identify which registers must be saved (because their values will be needed later).<sup>39</sup>

手続の適用はオペレータとオペランドを含む組み合わせにより指定します。オペレータはその値が手続となる部分式であり、オペランドはその値が引数となる部分式で、その引数に対して手続が適用されねばなりません。メタ循環の `eval` は適用をそれ自身を再帰的に呼び出すことで扱い、組み合わせの各要素を評価し、そして結果を `apply` に渡します。これが実際の手続適用を実行します。明示的制御評価機も同じことを行います。これらの再帰呼出は `goto` 命令と共に、スタックを使用して再帰呼出から戻った時に再格納されるようにレジスタを保存することで実装されます。各呼出の前にどのレジスタが保存されなければならないのかの確認に注意をしなければなりません。(なぜならこれらの値が後で必要になるからです)。<sup>40</sup>

We begin the evaluation of an application by evaluating the operator to produce a procedure, which will later be applied to the evaluated operands. To evaluate the operator, we move it to the `exp` register and go to `eval-dispatch`. The environment in the `env` register is already the correct one in which to evaluate the operator. However, we save `env` because we will need it later to evaluate the operands. We also extract the operands into `unev` and save this on

---

<sup>39</sup>This is an important but subtle point in translating algorithms from a procedural language, such as Lisp, to a register-machine language. As an alternative to saving only what is needed, we could save all the registers (except `val`) before each recursive call. This is called a *framed-stack* discipline. This would work but might save more registers than necessary; this could be an important consideration in a system where stack operations are expensive. Saving registers whose contents will not be needed later may also hold onto useless data that could otherwise be garbage-collected, freeing space to be reused.

<sup>40</sup>これは重要ですが、アルゴリズムを Lisp の様な手続型の言語からレジスタマシンの言語へ翻訳する場合において微妙な点です。必要な物だけを保存することの代替法として、各再帰呼出の前に全てのレジスタ (`val` を除く) を保存することができます。これは *framed-stack*(スタックフレーム) の統制と呼ばれます。これはうまく行きますがしかし必要以上のレジスタを保存します。このことはスタック命令が高価であるというシステム内の懸念点に成り得ます。後に使用される必要のないレジスタの保存はまた使用価値の無いデータを手放さないことも成り得ます。これはそうでなければガーベージコレクションされ、再使用されるために領域が解法されたはずです。

the stack. We set up `continue` so that `eval-dispatch` will resume at `ev-appl-did-operator` after the operator has been evaluated. First, however, we save the old value of `continue`, which tells the controller where to continue after the application.

適用の評価はオペレータを評価し手続を生成することから開始します。手続は後に評価されたオペランドに適用されます。オペレータを評価するためには、それを `exp` レジスタに移動させ、`eval-dispatch` へ飛びます。`env` レジスタ内の環境は既にその中でオペレータを評価するために適切な物になっていますが、それでも `env` を保存します。オペランドの評価にも必要なためです。またオペランドを `unev` の中に展開し、スタック上にこれを保存します。`continue` に対し `eval-dispatch` がオペレータの評価が完了した後に `ev-appl-did-operator` にて `resume`(再開) できるように設定します。しかし、最初に `continue` の古い値は保存します。これがコントローラに対し適用後にどこから続行するのかを告げるためです。

```
ev-application
  (save continue)
  (save env)
  (assign unev (op operands) (reg exp))
  (save unev)
  (assign exp (op operator) (reg exp))
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
```

Upon returning from evaluating the operator subexpression, we proceed to evaluate the operands of the combination and to accumulate the resulting arguments in a list, held in `argl`. First we restore the unevaluated operands and the environment. We initialize `argl` to an empty list. Then we assign to the `proc` register the procedure that was produced by evaluating the operator. If there are no operands, we go directly to `apply-dispatch`. Otherwise we save `proc` on the stack and start the argument-evaluation loop:

オペレータ部分式の評価からの帰還すると、組み合わせのオペランドの評価と、結果としての引数を `argl` に保持されるリストの中への蓄積へと進みます。最初に未評価のオペランドと環境を戻します。`argl` を空リストに初期化します。そして `proc` レジスタにオペレータの評価により生成された手続を割り当てます。もしオペランドが無ければ、直接 `apply-dispatch` へと進みます。

そうでなければ、`proc` をスタックに保存し引数評価ループを開始します。<sup>41</sup>

```
ev-app1-did-operator
  (restore unev) ; the operands
  (restore env)
  (assign argl (op empty-arglist))
  (assign proc (reg val)) ; the operator
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))
  (save proc)
```

Each cycle of the argument-evaluation loop evaluates an operand from the list in `unev` and accumulates the result into `argl`. To evaluate an operand, we place it in the `exp` register and go to `eval-dispatch`, after setting `continue` so that execution will resume with the argument-accumulation phase. But first we save the arguments accumulated so far (held in `argl`), the environment (held in `env`), and the remaining operands to be evaluated (held in `unev`). A special case is made for the evaluation of the last operand, which is handled at `ev-app1-last-arg`.

引数評価ループの各サイクルは `unev` の中のリストからオペランドを評価し、その結果を `argl` の中に蓄積します。オペランドを評価するために、それを `exp` レジスタの中に入れ、実行が引数蓄積段階から再開できるよう `continue` を設定した後に `eval-dispatch` に飛びます。しかし、最初に私達はそれまでに (`argl` に保持されている) 蓄積された引数、環境 (`env` に維持)、評価されていな

---

<sup>41</sup>We add to the evaluator data-structure procedures in Section 4.1.3 the following two procedures for manipulating argument lists:

Section 4.1.3の評価機データ構造の手続に以下の 2 つの手続を引数リストの操作のために追加します。

```
(define (empty-arglist) '())
(define (adjoin-arg arg arglist) (append arglist (list arg)))
```

We also use an additional syntax procedure to test for the last operand in a combination:

また追加の構文手続を使用して組み合わせの最後のオペランドであるかのテストを行います。

```
(define (last-operand? ops) (null? (cdr ops)))
```

い残りのオペランド (`unev` が保持) を保存します。最後のオペランドの評価は特別な場合として扱われ `ev-appl-last-arg` により取り扱われます。

```
ev-appl-operand-loop
  (save argl)
  (assign exp (op first-operand) (reg unev))
  (test (op last-operand?) (reg unev))
  (branch (label ev-appl-last-arg))
  (save env)
  (save unev)
  (assign continue (label ev-appl-accumulate-arg))
  (goto (label eval-dispatch))
```

When an operand has been evaluated, the value is accumulated into the list held in `argl`. The operand is then removed from the list of unevaluated operands in `unev`, and the argument-evaluation continues.

オペランドが評価された時に、その値は `argl` にて保持されるリストの中に蓄積されます。その後 `unev` 中の未評価オペランドのリストから消され、引数評価が続行されます。

```
ev-appl-accumulate-arg
  (restore unev)
  (restore env)
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (assign unev (op rest-operands) (reg unev))
  (goto (label ev-appl-operand-loop))
```

Evaluation of the last argument is handled differently. There is no need to save the environment or the list of unevaluated operands before going to `eval-dispatch`, since they will not be required after the last operand is evaluated. Thus, we return from the evaluation to a special entry point `ev-appl-accumulate-last-arg`, which restores the argument list, accumulates the new argument, restores the saved procedure, and goes off to perform the application.<sup>42</sup>

---

<sup>42</sup>The optimization of treating the last operand specially is known as *evil's tail recursion* (see Wand 1980). We could be somewhat more efficient in the argument evaluation loop if we made evaluation of the first operand a special case too. This would permit us to postpone initializing `argl` until after evaluating the first operand, so as to

最後の引数の評価は異なる扱いを受けます。環境や未評価のオペランドのリストを `eval-dispatch` に飛ぶ前に保存する必要がありません。最後のオペランドが評価された後にそれらは必要が無いためです。従って評価から特別なエントリポイント `ev-appl-accum-last-arg` に帰ります。これは引数リストを戻し、新しい引数を蓄積し、保存された手続を戻し、適用を実行するために飛びます。<sup>43</sup>

```
ev-appl-last-arg
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))
ev-appl-accum-last-arg
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore proc)
  (goto (label apply-dispatch))
```

The details of the argument-evaluation loop determine the order in which the interpreter evaluates the operands of a combination (e.g., left to right or right to left—see [Exercise 3.8](#)). This order is not determined by the metacircular evaluator, which inherits its control structure from the underlying Scheme in which it is implemented.<sup>44</sup> Because the `first-operand` selector (used in `ev-appl-operand-loop` to extract successive operands from `unev`) is implemented as `car` and the `rest-operands` selector is implemented as `cdr`, the explicit-control evaluator will evaluate the operands of a combination in left-to-right order.

引数評価ループの詳細はインタプリタが組み合わせのオペランドを評価する順を決定します。(例えば、左から右や右から左—[Exercise 3.8](#)参照)。この

---

avoid saving `argl` in this case. The compiler in [Section 5.5](#) performs this optimization. (Compare the `construct-arglist` procedure of [Section 5.5.3](#).)

<sup>43</sup> 最後のオペランドの処理の特別な最適化は *evil tail recursion*(エブリス末尾再帰)として知られています([Wand 1980](#)参照)。最初のオペランドも特別な場合とすれば、私達は引数評価ループをいくらかより効率的良くできたでしょう。これは `argl` の初期化を最初のオペランドの評価の後まで延期することができ、この場合に `argl` を保存することを防げたでしょう。[Section 5.5](#)のコンパイラはこの最適化を実行します。[\(Section 5.5.3の `construct-arglist` 手続と比較して下さい。\)](#)

<sup>44</sup> The order of operand evaluation in the metacircular evaluator is determined by the order of evaluation of the arguments to `cons` in the procedure `list-of-values` of [Section 4.1.1](#) (see [Exercise 4.1](#)).

順はメタ循環評価機では決定されません。メタ循環評価機はその制御構造をその基礎を成し実装を行う Scheme から継承します。<sup>45</sup>(`ev-appl-operand-loop` 内で一連のオペランドを `unev` から抽出するために使用された) `first-operand` セレクタは `car` として実装され、`rest-operands` は `cdr` として実装され、明示的制御評価機は組み合わせのオペランドを左から右への順で評価します。

## Procedure application

The entry point `apply-dispatch` corresponds to the `apply` procedure of the metacircular evaluator. By the time we get to `apply-dispatch`, the `proc` register contains the procedure to apply and `argl` contains the list of evaluated arguments to which it must be applied. The saved value of `continue` (originally passed to `eval-dispatch` and saved at `ev-application`), which tells where to return with the result of the procedure application, is on the stack. When the application is complete, the controller transfers to the entry point specified by the saved `continue`, with the result of the application in `val`. As with the metacircular `apply`, there are two cases to consider. Either the procedure to be applied is a primitive or it is a compound procedure.

エントリポイント `apply-dispatch` はメタ循環評価機の `apply` 手続に対応します。`apply-dispatch` に到達する時に、`proc` レジスタは適用するための手続を持ち、`argl` は適用するべき評価された引数のリストを持ちます。(元々は `eval-dispatch` に渡され、`ev-application` で保存された)`continue` の保存された値は手続適用の結果と共に帰る場所を伝えますが、スタック上に存在します。適用が完了した時に、コントローラは保存された `continue` により指示されたエントリポイントへ、`val` 内の適用の結果と共に移動します。メタ循環の `apply` と同様に、考慮すべき 2 つの場合が存在します。適用すべき手続はプリミティブであるか、または複合手続であるかです。

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknown-procedure-type))
```

---

<sup>45</sup> メタ循環評価機のオペランドの評価順は Section 4.1.1 の手続 `list-of-values` 内の `cons` への引数の評価順により決定されます (Exercise 4.1 参照)。

We assume that each primitive is implemented so as to obtain its arguments from `argl` and place its result in `val`. To specify how the machine handles primitives, we would have to provide a sequence of controller instructions to implement each primitive and arrange for `primitive-apply` to dispatch to the instructions for the primitive identified by the contents of `proc`. Since we are interested in the structure of the evaluation process rather than the details of the primitives, we will instead just use an `apply-primitive-procedure` operation that applies the procedure in `proc` to the arguments in `argl`. For the purpose of simulating the evaluator with the simulator of Section 5.2 we use the procedure `apply-primitive-procedure`, which calls on the underlying Scheme system to perform the application, just as we did for the metacircular evaluator in Section 4.1.4. After computing the value of the primitive application, we restore `continue` and go to the designated entry point.

各プリミティブは引数を `argl` から取得し、その結果を `val` 内に置くように実装されていると想定されます。機械がどのようにプリミティブを扱うかを指定するためには、それぞれのプリミティブを実装するための一連のコントローラ命令を提供しなければならず、`proc` の中身により判別されたプリミティブのための命令への割り振りを行うように `primitive-apply` を準備しなければなりません。私達はプリミティブの詳細ではなく、評価処理の構造に興味があるため、それらの代わりに単に `apply-primitive-procedure` を使用します。これは `proc` 内の手続を `argl` 内の引数に対して適用します。Section 5.2 のシミュレータを用いて評価機のシミュレーションを行う目的のために、私達は手続 `apply-primitive-procedure` を使用します。これは根底にある Scheme システムを適用を実行するために呼び出します。私達が Section 4.1.4 のメタ循環評価機で行ったのと全く同じです。プリミティブの適用の値を計算した後に、`continue` を戻して指定されたエントリポイントに飛びます。

```
primitive-apply
  (assign val (op apply-primitive-procedure)
         (reg proc)
         (reg argl))
  (restore continue)
  (goto (reg continue))
```

To apply a compound procedure, we proceed just as with the metacircular evaluator. We construct a frame that binds the procedure's parameters to the arguments, use this frame to extend the environment carried by the procedure,

and evaluate in this extended environment the sequence of expressions that forms the body of the procedure. **Ev-sequence**, described below in Section 5.4.2, handles the evaluation of the sequence.

複合手続を適用するためには、メタ循環評価機と全く同様に進行します。手続のパラメタを引数に束縛するフレームを構築し、このフレームを用いて手続により運ばれた環境を拡張し、この拡張環境の中で手続のボディを形成する式の列を評価します。Section 5.4.2で説明される **ev-sequence** は列の評価を取り扱います。

```
compound-apply
  (assign unev (op procedure-parameters) (reg proc))
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment)
    (reg unev) (reg argl) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))
```

**Compound-apply** is the only place in the interpreter where the **env** register is ever assigned a new value. Just as in the metacircular evaluator, the new environment is constructed from the environment carried by the procedure, together with the argument list and the corresponding list of variables to be bound.

**compound-apply** は **env** レジスタが新しい値を割り当てられるインタプリタ内で唯一の場所です。メタ循環評価機と同様に、新しい環境は手続により運ばれた環境から引数リストと対応する束縛される変数のリストと共に構築されます。

## 5.4.2 Sequence Evaluation and Tail Recursion

The portion of the explicit-control evaluator at **ev-sequence** is analogous to the metacircular evaluator's **eval-sequence** procedure. It handles sequences of expressions in procedure bodies or in explicit **begin** expressions.

明示的制御評価機の **ev-sequence** の部分はメタ循環評価機の **eval-sequence** 手続と同等です。手続のボディ内の式、または明示的な **begin** 式内の列を取り扱います。

Explicit **begin** expressions are evaluated by placing the sequence of expressions to be evaluated in **unev**, saving **continue** on the stack, and jumping to

### **ev-sequence.**

明示的な `begin` 式は `unev` 内に評価されるべき式の列を配置し、`continue` をスタック上に保存し、`ev-sequence` に飛ぶことで評価されます。

```
ev-begin
  (assign unev (op begin-actions) (reg exp))
  (save continue)
  (goto (label ev-sequence))
```

The implicit sequences in procedure bodies are handled by jumping to `ev-sequence` from `compound-apply`, at which point `continue` is already on the stack, having been saved at `ev-application`.

手続きのボディ内の暗黙的な列は `compound-apply` から `ev-sequence` へと飛ぶことで扱われます。この時点で `continue` は既に `ev-application` で保存され、スタック上に存在します。

The entries at `ev-sequence` and `ev-sequence-continue` form a loop that successively evaluates each expression in a sequence. The list of unevaluated expressions is kept in `unev`. Before evaluating each expression, we check to see if there are additional expressions to be evaluated in the sequence. If so, we save the rest of the unevaluated expressions (held in `unev`) and the environment in which these must be evaluated (held in `env`) and call `eval-dispatch` to evaluate the expression. The two saved registers are restored upon the return from this evaluation, at `ev-sequence-continue`.

`ev-sequence` と `ev-sequence-continue` のエントリポイントはループを形成し、連続して列内の各式を評価します。未評価の式のリストは `unev` に保持されています。各式の評価の前に、列内にさらなる評価すべき式が存在しないかどうか確認します。もしそうであれば、(`unev` に保持された) 未評価の式の残りと、(`env` に保持された) 式の残りが評価される環境を保存し、その式を評価するために `eval-dispatch` を呼びます。2つの保存されたレジスタはこの評価からの帰還時に `ev-sequence-continue` にて戻されます。

The final expression in the sequence is handled differently, at the entry point `ev-sequence-last-exp`. Since there are no more expressions to be evaluated after this one, we need not save `unev` or `env` before going to `eval-dispatch`. The value of the whole sequence is the value of the last expression, so after the evaluation of the last expression there is nothing left to do except `continue` at the entry point currently held on the stack (which was saved by `ev-application` or `ev-begin`.) Rather than setting up `continue` to arrange

for `eval-dispatch` to return here and then restoring `continue` from the stack and continuing at that entry point, we restore `continue` from the stack before going to `eval-dispatch`, so that `eval-dispatch` will continue at that entry point after evaluating the expression.

列内の最後の式は `ev-sequence-last-exp` にて異なる取扱を行います。この後には評価すべき式は無いため、`unev` と `env` を `eval-dispatch` に行く前に保存する必要はありません。列全体の値は最後の式の値であるため、最後の式の評価の後にスタック上に現時点で保存されている (`ev-application` または `ev-begin` にて保存された) エントリポイントから続行すること以外に必要なことはありません。`continue` を設定して `eval-dispatch` からここに帰るよう準備し、次にスタックから `continue` の値を戻してそのエントリポイントから続行するのではなく、`eval-dispatch` へ行く前にスタックから `continue` を戻します。そうすることで `eval-dispatch` は式を評価した後にそのエントリポイントから続行します。

```
ev-sequence
  (assign exp (op first-exp) (reg unev))
  (test (op last-exp?) (reg unev))
  (branch (label ev-sequence-last-exp))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))

ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))

ev-sequence-last-exp
  (restore continue)
  (goto (label eval-dispatch))
```

## Tail recursion

In Chapter 1 we said that the process described by a procedure such as Chapter 1 にて以下のような手続により記述されるプロセスは、

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

is an iterative process. Even though the procedure is syntactically recursive (defined in terms of itself), it is not logically necessary for an evaluator to save information in passing from one call to `sqrt-iter` to the next.<sup>46</sup> An evaluator that can execute a procedure such as `sqrt-iter` without requiring increasing storage as the procedure continues to call itself is called a *tail-recursive* evaluator. The metacircular implementation of the evaluator in Chapter 4 does not specify whether the evaluator is tail-recursive, because that evaluator inherits its mechanism for saving state from the underlying Scheme. With the explicit-control evaluator, however, we can trace through the evaluation process to see when procedure calls cause a net accumulation of information on the stack.

反復プロセスだと述べました。例えこの手続が構文的に(それ自身の単語の定義において)再帰であっても、論理的には評価機にとって、1つの `sqrt-iter` の呼出から次の呼出への横断において情報を保存する必要がありません。<sup>47</sup> `sqrt-iter`のような手続を、手續がそれ自身を呼び出すことを続けるに従い記憶域を増加させる必要無しに実行することが可能な評価機は、*tail-recursive*(末尾再帰)評価機と呼ばれます。Chapter 4のメタ循環評価機の実装は評価機が末尾再帰であるかどうかを指定しませんでした。その評価機が状態を保存するための仕組みをその基礎に横たわる Scheme から継承していたためです。しかし明示的制御評価機と用いる場合、私達は評価の過程を追跡し、いつ手続呼出が正味の情報集積をスタック上に引き起すのかを確認することができます。

Our evaluator is tail-recursive, because in order to evaluate the final expression of a sequence we transfer directly to `eval-dispatch` without saving any information on the stack. Hence, evaluating the final expression in a sequence—even if it is a procedure call (as in `sqrt-iter`, where the `if` expression, which is the last expression in the procedure body, reduces to a call to `sqrt-iter`)—will

---

<sup>46</sup>We saw in Section 5.1 how to implement such a process with a register machine that had no stack; the state of the process was stored in a fixed set of registers.

<sup>47</sup>Section 5.1にてそのようなプロセスをどのようにスタックを持たないレジスタマシンにて実装するのかを学びました。プロセスの状態は固定長のレジスタ集合に格納されます。

not cause any information to be accumulated on the stack.<sup>48</sup>

私達の評価機は末尾再帰です。なぜなら列の最後の式を評価するために、スタック上に何の情報も保存すること無く eval-dispatch へと直接飛びます。従って、列の最後の式—例えもしそれが手続呼出であっても (sqrt-iter のように、手続のボディの最後の式が if 式であっても、sqrt-iter への呼出へと簡約されます)—の評価がスタック上に何の情報の蓄積も起こしません。<sup>49</sup>

If we did not think to take advantage of the fact that it was unnecessary to save information in this case, we might have implemented eval-sequence by treating all the expressions in a sequence in the same way—saving the registers, evaluating the expression, returning to restore the registers, and repeating this until all the expressions have been evaluated:

もしこの場合に情報を保存する必要が無いという事実を活用することを考えなかった場合、列内の全ての式を同じように取り扱うように eval-sequence を実装していたことでしょう。レジスタの保存、式の評価、レジスタを戻すために帰る、これらを全ての式が評価されるまで繰り返したことでしょう。<sup>50</sup>

```
ev-sequence
  (test (op no-more-exp?) (reg unev))
  (branch (label ev-sequence-end))
  (assign exp (op first-exp) (reg unev))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
```

---

<sup>48</sup>This implementation of tail recursion in ev-sequence is one variety of a well-known optimization technique used by many compilers. In compiling a procedure that ends with a procedure call, one can replace the call by a jump to the called procedure's entry point. Building this strategy into the interpreter, as we have done in this section, provides the optimization uniformly throughout the language.

<sup>49</sup>この ev-sequence における末尾再帰の実装は多くのコンパイラで使用されている良く知られた最適化の技術の一種です。手続呼出で終了する手続のコンパイルでは、呼出を呼び出された手続のエントリポイントへのジャンプで置き換えることができます。この節で行ったように、この戦略をインタプリタの中に構築することは言語の至る所に均一に最適化を提供します。

<sup>50</sup>We can define no-more-exp? as follows:

no-more-exp? を以下のように定義することができます。

```
(define (no-more-exp? seq) (null? seq))
```

```

(goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exp) (reg unev))
  (goto (label ev-sequence))
ev-sequence-end
  (restore continue)
  (goto (reg continue))

```

This may seem like a minor change to our previous code for evaluation of a sequence: The only difference is that we go through the save-restore cycle for the last expression in a sequence as well as for the others. The interpreter will still give the same value for any expression. But this change is fatal to the tail-recursive implementation, because we must now return after evaluating the final expression in a sequence in order to undo the (useless) register saves. These extra saves will accumulate during a nest of procedure calls. Consequently, processes such as `sqrt-iter` will require space proportional to the number of iterations rather than requiring constant space. This difference can be significant. For example, with tail recursion, an infinite loop can be expressed using only the procedure-call mechanism:

恐らくこれは列の評価のための以前のコードに対する軽微な変更のように見えるでしょう。唯一の違いは保存と再格納のサイクルを他と同様に列の最後の式でも通すことです。インタプリタは依然としてどの式に対しても同じ値を与えます。しかし、この変更是末尾再帰の実装に対しては致命的です。なぜなら、これで私達は列の最後の式の評価の後にも（使用価値の無い）レジスタの保存を戻すために帰らねばなりません。これらの余分な保存は入れ子の手続の呼出の間で蓄積されます。その結果として、`sqrt-iter`のようなプロセスは一定容量を必要とするだけでなく、繰り返しの回数に比例する記憶域を必要とします。この違いは重大に成ります。例えば、末尾再帰を用いれば、無限ループは手続呼出の仕組みだけを用いて表現できます。

```

(define (count n)
  (newline) (display n) (count (+ n 1)))

```

Without tail recursion, such a procedure would eventually run out of stack space, and expressing a true iteration would require some control mechanism other than procedure call.

末尾再帰が無ければ、そのような手続はいつかはスタック領域を使いつくします。そして真に反復を表現することは手続呼出以外の何らかの制御の仕組みを必要とします。

### 5.4.3 Conditionals, Assignments, and Definitions

As with the metacircular evaluator, special forms are handled by selectively evaluating fragments of the expression. For an `if` expression, we must evaluate the predicate and decide, based on the value of predicate, whether to evaluate the consequent or the alternative.

メタ循環評価機と同様に、特殊形式は選択的に式の部分部分を評価することで取り扱われます。`if` 式に対しては、述語を評価して、その値を元に、結果部 (consequent) と代替部 (alternative) のどちらを評価するか決定します。

Before evaluating the predicate, we save the `if` expression itself so that we can later extract the consequent or alternative. We also save the environment, which we will need later in order to evaluate the consequent or the alternative, and we save `continue`, which we will need later in order to return to the evaluation of the expression that is waiting for the value of the `if`.

述語を評価する前に、`if` 式自身を保存します。そうすることで後に結果部か代替部を抽出することができます。また後に結果部か代替部を評価するため必要となるので環境を保存します。そして後に `if` の値を待っている式の評価に戻るために必要なため `continue` も保存します。

```
ev-if
  (save exp) ; 後のため式を保存する
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg exp))
  (goto (label eval-dispatch)) ; 述語を評価する
```

When we return from evaluating the predicate, we test whether it was true or false and, depending on the result, place either the consequent or the alternative in `exp` before going to `eval-dispatch`. Notice that restoring `env` and `continue` here sets up `eval-dispatch` to have the correct environment and to continue at the right place to receive the value of the `if` expression.

述語の評価から戻る時、真か偽であるかをテストし、結果に依り `eval-dispatch` に飛ぶ前に `exp` に結果部か代替部を配置します。`env` と `continue` をここで戻すことが `eval-dispatch` に正しい環境を持たせ、正しい場所から継続し `if` 式の値を受けとるように設定していることに注意して下さい。

```
ev-if-decide
  (restore continue)
  (restore env)
  (restore exp)
  (test (op true?) (reg val))
  (branch (label ev-if-consequent))
ev-if-alternative
  (assign exp (op if-alternative) (reg exp))
  (goto (label eval-dispatch))
ev-if-consequent
  (assign exp (op if-consequent) (reg exp))
  (goto (label eval-dispatch))
```

## Assignments and definitions

Assignments are handled by `ev-assignment`, which is reached from `eval-dispatch` with the assignment expression in `exp`. The code at `ev-assignment` first evaluates the value part of the expression and then installs the new value in the environment. `Set-variable-value!` is assumed to be available as a machine operation.

代入と定義は `ev-assignment` により扱われます。ここには `eval-dispatch` から代入式が `exp` の中にある状態で到達します。`ev-assignment` の最初のコードは式の部分の値を評価し、次に新しい値を環境に導入します。`Set-variable-value!` が機械語命令として必要可能であると前提します。

```
ev-assignment
  (assign unev (op assignment-variable) (reg exp))
  (save unev) ; 後のため変数を保存
  (assign exp (op assignment-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-assignment-1))
```

```

(goto (label eval-dispatch)) ; 代入値を評価する
ev-assignment-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
    (op set-variable-value!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue)))

```

Definitions are handled in a similar way:

定義も同様に扱われます。

```

ev-definition
  (assign unev (op definition-variable) (reg exp))
  (save unev) ; 変数を後のため保存
  (assign exp (op definition-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-definition-1))
  (goto (label eval-dispatch)) ; 定義値を評価する
ev-definition-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
    (op define-variable!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue)))

```

**Exercise 5.23:** Extend the evaluator to handle derived expressions such as `cond`, `let`, and so on (Section 4.1.2). You may “cheat” and assume that the syntax transformers such as `cond->if` are available as machine operations.<sup>51</sup>

---

<sup>51</sup>This isn’t really cheating. In an actual implementation built from scratch, we would use our explicit-control evaluator to interpret a Scheme program that performs source-level transformations like `cond->if` in a syntax phase that runs before execution.

評価機を拡張し、`cond`, `let`, 等の派生式を取り扱うようにせよ ([Section 4.1.2](#))。`cond->if` の様な構文変換器が機械語命令として使用可能と前提して“インチキ”しても良い。<sup>52</sup>

**Exercise 5.24:** Implement `cond` as a new basic special form without reducing it to `if`. You will have to construct a loop that tests the predicates of successive `cond` clauses until you find one that is true, and then use `ev-sequence` to evaluate the actions of the clause.

`cond` を新しい基本的な特殊形式として `if` に簡約すること無しに実装せよ。連続する `cond` 節の述語を真になるものを見つけるまでテストするループを構築する必要がある。次に `ev-sequence` を使用してその節のアクションを評価する。

**Exercise 5.25:** Modify the evaluator so that it uses normal-order evaluation, based on the lazy evaluator of [Section 4.2](#).

評価機を変更し、[Section 4.2](#)の遅延評価機を基にした正規順評価を使用するようにせよ。

#### 5.4.4 Running the Evaluator

With the implementation of the explicit-control evaluator we come to the end of a development, begun in [Chapter 1](#), in which we have explored successively more precise models of the evaluation process. We started with the relatively informal substitution model, then extended this in [Chapter 3](#) to the environment model, which enabled us to deal with state and change. In the metacircular evaluator of [Chapter 4](#), we used Scheme itself as a language for making more explicit the environment structure constructed during evaluation of an expression. Now, with register machines, we have taken a close look at the evaluator's mechanisms for storage management, argument passing, and control. At each new level of description, we have had to raise issues and resolve ambiguities that were not apparent at the previous, less precise treatment of evaluation. To understand the behavior of the explicit-control evaluator, we can simulate it and monitor its performance.

---

<sup>52</sup>これは本当はインチキではありません。実際のゼロからの実装においても、Schemeを解釈する明示的制御評価機を用いて `cond->if` のようなソースレベル変換を実行前の構文フェーズにて実行するでしょう。

明示的制御評価機の実装と共に、私達はChapter 1から始まった開発の終わりにやってきました。ここまで私達は引き続きより的確な評価過程のモデルを探求してきました。比較的、略式な置換モデルから開始し、次にこれをChapter 3で環境モデルに拡張しました。これは状態と変更を扱うことを可能にしました。Chapter 4のメタ循環評価機では Scheme 自身を式の評価の間に構築される、より明確な環境構造のための言語として使用しました。ここでは、レジスタマシンを用いてメモリ管理、引数渡し、制御のための評価機の仕組みについてつぶさに見てきました。それぞれの新しいレベルの説明にて、直前の、明確さで劣る評価処理の見ることはできない曖昧さに関して問題を提起し、解決する必要がありました。明示的制御評価機の振舞を理解するために、そのシミュレーションを行い、パフォーマンスを監視することができます。

We will install a driver loop in our evaluator machine. This plays the role of the `driver-loop` procedure of Section 4.1.4. The evaluator will repeatedly print a prompt, read an expression, evaluate the expression by going to `eval-dispatch`, and print the result. The following instructions form the beginning of the explicit-control evaluator's controller sequence:

私達の評価機にドライバループを導入します。これはSection 4.1.4の `driver-loop` 手続の役割を果たします。この評価機は繰り返しプロンプトを表示し、式を読み込み、`eval-dispatch`へ飛ぶことで式を評価し、結果を表示します。以下の命令は明示的制御評価機のコントローラシーケンスの開始を形づくります。<sup>53</sup>

```
read-eval-print-loop
```

---

<sup>53</sup>We assume here that `read` and the various printing operations are available as primitive machine operations, which is useful for our simulation, but completely unrealistic in practice. These are actually extremely complex operations. In practice, they would be implemented using low-level input-output operations such as transferring single characters to and from a device.

ここでは `read` と多様な表示命令がプリミティブな機械語命令として使用可能であると前提します。このことは私達のシミュレーションには便利ですが、実際には完全に非現実的です。これらは本当はかなり複雑な命令です。実際には、それらは単一の文字を端末との間で双方向に転送するような低レベルの入出力命令を用いて実装されるでしょう。

To support the `get-global-environment` operation we define

`get-global-environment` 命令をサポートするためには以下を定義します。

```
(define the-global-environment (setup-environment))  
(define (get-global-environment) the-global-environment)
```

```

(perform (op initialize-stack))
(perform
  (op prompt-for-input) (const ";;EC-Eval input:"))
(assign exp (op read))
(assign env (op get-global-environment))
(assign continue (label print-result))
(goto (label eval-dispatch))

print-result
  (perform (op announce-output) (const ";;EC-Eval value:"))
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))

```

When we encounter an error in a procedure (such as the “unknown procedure type error” indicated at `apply-dispatch`), we print an error message and return to the driver loop.<sup>54</sup>

手続の中で `(apply-dispatch` で指摘される “未知の手続型エラー” の様な) エラーに遭遇した時、エラーメッセージを表示し、ドライバループへと戻ります。<sup>55</sup>

```

unknown-expression-type
  (assign val (const unknown-expression-type-error))
  (goto (label signal-error))
unknown-procedure-type
  (restore continue)      ; clean up stack (from apply-dispatch)
  (assign val (const unknown-procedure-type-error))
  (goto (label signal-error))
signal-error
  (perform (op user-print) (reg val))
  (goto (label read-eval-print-loop))

```

For the purposes of the simulation, we initialize the stack each time through the driver loop, since it might not be empty after an error (such as an undefined variable) interrupts an evaluation.<sup>56</sup>

<sup>54</sup>There are other errors that we would like the interpreter to handle, but these are not so simple. See [Exercise 5.30](#).

<sup>55</sup>インタプリタに取り扱って欲しいと願うかもしれない他のエラーも存在します。しかしこれらはあまり単純ではありません。[Exercise 5.30](#)を参照して下さい。

<sup>56</sup>We could perform the stack initialization only after errors, but doing it in the

シミュレーションの目的のために、ドライバループを通る度にスタックを初期化します。(未定義変数の様な) エラーが評価を割り込みした後には空でない可能性があるためです。<sup>57</sup>

If we combine all the code fragments presented in Section 5.4.1–Section 5.4.4, we can create an evaluator machine model that we can run using the register-machine simulator of Section 5.2.

Section 5.4.1からSection 5.4.4の間のコードの断片を組合せれば、Section 5.2のレジスタマシンシミュレータを用いて実行することができる評価機の機械モデルを作ることができます。

```
(define eceval
  (make-machine
    '(exp env val proc argl continue unev)
    eceval-operations
    '(read-eval-print-loop
      (entire machine controller as given above) )))
```

We must define Scheme procedures to simulate the operations used as primitives by the evaluator. These are the same procedures we used for the metacircular evaluator in Section 4.1, together with the few additional ones defined in footnotes throughout Section 5.4.

評価機によりプリミティブとして使用される命令をシミュレートするための Scheme 手続を定義しなければなりません。これらはSection 4.1でメタ循環評価機のために使用したものと同じ手続と、Section 5.4の至る所の脚注にて定義されたいいくつかの追加の物があります。

```
(define eceval-operations
  (list (list 'self-evaluating? self-evaluating)
        (complete list of operations for eceval machine)))
```

Finally, we can initialize the global environment and run the evaluator:

最後に、グローバル環境を初期化し、評価機を実行します。

```
(define the-global-environment (setup-environment))
```

---

driver loop will be convenient for monitoring the evaluator's performance, as described below.

<sup>57</sup> スタックの初期化をエラーの後にのみ行うことも可能でしょう。しかしドライバループの中で行うことは評価機のパフォーマンスを監視するために便利です。この先で説明されます。

```

(start eceval)
;; EC-Eval input:
(define (append x y)
  (if (null? x) y (cons (car x) (append (cdr x) y))))
;; EC-Eval value:
ok
;; EC-Eval input:
(append '(a b c) '(d e f))
;; EC-Eval value:
(a b c d e f)

```

Of course, evaluating expressions in this way will take much longer than if we had directly typed them into Scheme, because of the multiple levels of simulation involved. Our expressions are evaluated by the explicit-control-evaluator machine, which is being simulated by a Scheme program, which is itself being evaluated by the Scheme interpreter.

もちろん、この方法の式の評価は Scheme に直接入力した場合よりもずっと長くかかります。複数レベルのシミュレーションが関与するためです。式は明示的制御評価器の機械により評価されます。これは Scheme プログラムによりシミュレートされ、Scheme プログラム自身は Scheme インタプリタにより評価されています。

## Monitoring the performance of the evaluator

Simulation can be a powerful tool to guide the implementation of evaluators. Simulations make it easy not only to explore variations of the register-machine design but also to monitor the performance of the simulated evaluator. For example, one important factor in performance is how efficiently the evaluator uses the stack. We can observe the number of stack operations required to evaluate various expressions by defining the evaluator register machine with the version of the simulator that collects statistics on stack use ([Section 5.2.4](#)), and adding an instruction at the evaluator's `print-result` entry point to print the statistics:

シミュレーションは評価器の実装を案内するのに強力なツールです。シミュレーションはレジスタマシンの設計の多様性を探求することだけでなく、シミュレートされた評価器のパフォーマンスを観察することも簡単にします。例えば、パフォーマンスにおける 1 つの重要な要因はどれだけ効率良く評価機が

スタックを使用するかがあります。スタック利用上の統計を集めるシミュレータの版を用いて評価機のレジスタマシンを設計すること、評価機の `print-result` エントリポイントに統計を表示する命令を追加することでにより、様々な式を評価するのに必要とされるスタック命令の数を観察することができます (Section 5.2.4)。

```
print-result
  (perform (op print-stack-statistics)) ; 追加された命令
  (perform
    (op announce-output) (const ";; EC-Eval value:")
    ... ; 以前と同じ
```

Interactions with the evaluator now look like this:

評価機との応答はこれで以下のように見えます。

```
;;; EC-Eval input:
(define (factorial n)
  (if (= n 1) 1 (* (factorial (- n 1)) n)))
(total-pushes = 3 maximum-depth = 3)
;;; EC-Eval value:
ok
;;; EC-Eval input:
(factorial 5)
(total-pushes = 144 maximum-depth = 28)
;;; EC-Eval value:
120
```

Note that the driver loop of the evaluator reinitializes the stack at the start of each interaction, so that the statistics printed will refer only to stack operations used to evaluate the previous expression.

評価機のドライバループが全ての応答の開始にスタックを再度初期化することに注意して下さい。それにより表示された統計は直前の式の評価のために使用されたスタック命令のみを参照します。

**Exercise 5.26:** Use the monitored stack to explore the tail-recursive property of the evaluator (Section 5.4.2). Start the evaluator and define the iterative `factorial` procedure from Section 1.2.1:

監視付きのスタックを用いて評価機 (Section 5.4.2) の末尾再帰の特性を調査せよ。評価機を開始し、Section 1.2.1の反復 `factorial`

手続を定義せよ。

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product) (+ counter 1))))
  (iter 1 1))
```

Run the procedure with some small values of  $n$ . Record the maximum stack depth and the number of pushes required to compute  $n!$  for each of these values.

手続をいくつかの  $n$  の小さな値で実行せよ。これらの値に対する  $n!$  を求めるのに必要な最大スタック深度と push の数を記録せよ。

- a You will find that the maximum depth required to evaluate  $n!$  is independent of  $n$ . What is that depth?  
 $n!$  を評価するために必要な最大深度が  $n$  から独立していることを発見するだろう。この深さは何か?
- b Determine from your data a formula in terms of  $n$  for the total number of push operations used in evaluating  $n!$  for any  $n \geq 1$ . Note that the number of operations used is a linear function of  $n$  and is thus determined by two constants.  
あなたのデータから任意の  $n \geq 1$  に対して  $n!$  を評価するのに使用される push 命令の総数を求める  $n$  の方程式を求めよ。使用される命令数は  $n$  の線形関数であり、従って 2 つの定数から決定されることに注意せよ。

**Exercise 5.27:** For comparison with [Exercise 5.26](#), explore the behavior of the following procedure for computing factorials recursively:

[Exercise 5.26](#)との比較として、以下の階乗を再帰的に求める手続の振舞を調査せよ。

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

By running this procedure with the monitored stack, determine, as a function of  $n$ , the maximum depth of the stack and the total number of pushes used in evaluating  $n!$  for  $n \geq 1$ . (Again, these functions will be linear.) Summarize your experiments by filling in the following table with the appropriate expressions in terms of  $n$ : この手続を監視付きのスタックを用いて実行し、任意の  $n \geq 1$  に対して  $n!$  を評価するために使用される、スタックの最大深度とプッシュの総数を  $n$  の関数として求めよ。(再び、これらの関数は線形になる)。あなたの経験を以下の表に適切な  $n$  の式を埋めることでまとめよ。

	最大深度	push の総数
再帰		
階乗		
反復		
階乗		

The maximum depth is a measure of the amount of space used by the evaluator in carrying out the computation, and the number of pushes correlates well with the time required.

最大深度は演算の実行において評価機により使用された記憶域の量の尺度である。push の総数は必要な時間に良く関連している。

**Exercise 5.28:** Modify the definition of the evaluator by changing `eval-sequence` as described in Section 5.4.2 so that the evaluator is no longer tail-recursive. Rerun your experiments from Exercise 5.26 and Exercise 5.27 to demonstrate that both versions of the `factorial` procedure now require space that grows linearly with their input.

Section 5.4.2 にて説明されているように `eval-sequence` を変えることで評価機の定義を変更し、評価機がもはや末尾再帰ではないようにせよ。Exercise 5.26 と Exercise 5.27 の実験を再実行し、`factorial` 手続の両版が今では必要とされる記憶域がそれらの入力に対し線形に増加することを実演せよ。

**Exercise 5.29:** Monitor the stack operations in the tree-recursive Fibonacci computation:

木再帰斐波那ッチ数の演算におけるスタック命令を監視せよ。

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

- a Give a formula in terms of  $n$  for the maximum depth of the stack required to compute  $\text{Fib}(n)$  for  $n \geq 2$ . Hint: In Section 1.2.2 we argued that the space used by this process grows linearly with  $n$ .

$n \geq 2$  に対して  $\text{Fib}(n)$  を求まる場合に必要なスタックの最大深度に対する  $n$  の方程式を与えよ。ヒント：Section 1.2.2において私達はこの処理により使用される記憶域は  $n$  に対し線形に増加することを議論した。

- b Give a formula for the total number of pushes used to compute  $\text{Fib}(n)$  for  $n \geq 2$ . You should find that the number of pushes (which correlates well with the time used) grows exponentially with  $n$ . Hint: Let  $S(n)$  be the number of pushes used in computing  $\text{Fib}(n)$ . You should be able to argue that there is a formula that expresses  $S(n)$  in terms of  $S(n - 1)$ ,  $S(n - 2)$ , and some fixed “overhead” constant  $k$  that is independent of  $n$ . Give the formula, and say what  $k$  is. Then show that  $S(n)$  can be expressed as  $a \cdot \text{Fib}(n + 1) + b$  and give the values of  $a$  and  $b$ .

$n \geq 2$  に対して  $\text{Fib}(n)$  を求めるのに使用された push の総数に対する方程式を与えよ。(費やされた時間に良く関連する)push の総数は  $n$  の指数関数的に増加することを見付けなければならない。ヒント： $S(n)$  を  $\text{Fib}(n)$  を求めるのに使用された push の総数とせよ。 $S(n - 1)$ ,  $S(n - 2)$ , それにある固定の“オーバヘッド”として  $n$  から独立した定数  $k$  を用いて  $S(n)$  を表現する方程式が存在することを主張することができるはずだ。方程式を与えて、 $k$  が何であるかを述べよ。次に  $S(n)$  が  $a \cdot \text{Fib}(n + 1) + b$  として表現できることを示し、 $a$  と  $b$  の値を与えよ。

**Exercise 5.30:** Our evaluator currently catches and signals only two kinds of errors—unknown expression types and unknown proce-

dure types. Other errors will take us out of the evaluator read-eval-print loop. When we run the evaluator using the register-machine simulator, these errors are caught by the underlying Scheme system. This is analogous to the computer crashing when a user program makes an error.<sup>58</sup> It is a large project to make a real error system work, but it is well worth the effort to understand what is involved here.

私達の評価機は現在は 2 つの種類のエラー—未知の型の式と未知の型の手続—のみを発しキャッチする。他のエラーは評価機の REPL から抜けてしまう。評価機をレジスタマシンシミュレータを用いて実行した時に、これらのエラーはその下に横たわる Scheme システムによりキャッチされるこれはユーザプログラムがエラーを発生させた時にコンピュータが強制終了するのと同様である。<sup>59</sup> 本物のエラーシステムを働かせることは大きなプロジェクトである。しかし、ここで何が関与しているのかを理解する努力には大きな価値がある。

- a Errors that occur in the evaluation process, such as an attempt to access an unbound variable, could be caught by changing the lookup operation to make it return a distinguished condition code, which cannot be a possible value of any user variable. The evaluator can test for this condition code and then do what is necessary to go to `signal-error`. Find all of the places in the evaluator where such a change is necessary and fix them. This is lots of work.

未定義の変数にアクセスを試すような評価プロセスで発生するエラーは `lookup` 命令を変更し識別可能な状態コードを返

---

<sup>58</sup>Regrettably, this is the normal state of affairs in conventional compiler-based language systems such as C. In UNIX(tm) the system “dumps core,” and in DOS/Windows(tm) it becomes catatonic. The Macintosh(tm) displays a picture of an exploding bomb and offers you the opportunity to reboot the computer—if you’re lucky.

<sup>59</sup>残念ながら、C 言語のような従来のコンパイラベース言語システムにおいてはこれが普通の状況です。UNIX(tm) ではシステムは“コア (core) をダンプ (dump)”し、DOS/Windows(tm) では catatonic(硬まる、フリーズ) します。Macintosh(tm) は—もしラッキーな場合は—爆発する爆弾の絵を表示しコンピュータをリブートするよう提案します。

すように変更することで捕まえることができるだろう。状態コードは全てのユーザ変数が取り得ない値でなければならぬ。評価機はこの状態コードに対しテストを行い、`signal-error` に飛ぶために必要なことを行う。評価機の中でそのような変更が必要な全ての箇所を見つけ修正せよ。これはとても大変な作業だ。

- b Much worse is the problem of handling errors that are signaled by applying primitive procedures, such as an attempt to divide by zero or an attempt to extract the `car` of a symbol. In a professionally written high-quality system, each primitive application is checked for safety as part of the primitive. For example, every call to `car` could first check that the argument is a pair. If the argument is not a pair, the application would return a distinguished condition code to the evaluator, which would then report the failure. We could arrange for this in our register-machine simulator by making each primitive procedure check for applicability and returning an appropriate distinguished condition code on failure. Then the `primitive-apply` code in the evaluator can check for the condition code and go to `signal-error` if necessary. Build this structure and make it work. This is a major project.

ゼロで割ることやシンボルから `car` を抽出するような試行により、プリミティブな手続の適用上で発せられるエラーの取り扱いの問題はずっと酷い物だ。専門的に記述された高品質なシステムにおいては、プリミティブの適用全てはプリミティブの一部として安全性が確認されている。例えば全ての `car` の呼出は最初に引数がペアであるかを確認する。もし引数がペアでなければ、適用は区別可能な状態コードを評価機に返す。すると評価機は失敗を報告する。私達はこれを私達のレジスタマシンシミュレータに全てのプリミティブ手続の適用性チェックを行い適切な識別可能な状態コードを失敗時に返すようにすることで手筈を整えることができるだろう。すると評価機の `primitive-apply` コードが状態コードをチェックし必要なら `signal-error` へ飛ぶことができる。この構造を構築し、働くようにせよ。これは巨大なプロジェクトである。

## 5.5 Compilation

The explicit-control evaluator of Section 5.4 is a register machine whose controller interprets Scheme programs. In this section we will see how to run Scheme programs on a register machine whose controller is not a Scheme interpreter.

Section 5.4の明示的制御評価機はコントローラが Scheme プログラムを解釈するレジスタマシンです。この節では Scheme プログラムをコントローラが Scheme インタプリタではないレジスタマシン上にてどのように実行するのかについて学びます。

The explicit-control evaluator machine is universal—it can carry out any computational process that can be described in Scheme. The evaluator's controller orchestrates the use of its data paths to perform the desired computation. Thus, the evaluator's data paths are universal: They are sufficient to perform any computation we desire, given an appropriate controller.<sup>60</sup>

明示的制御評価機械は普遍的です。Scheme で記述できるどんな演算プロセスも実行できます。評価機のコントローラは望んだ演算を実行するためのデータパスの使用を調整します。従って、評価機のデータパスは普遍的です。適切なコントローラを与えられた場合に、私達が望む任意の演算を実行するのに十分です。<sup>61</sup>

Commercial general-purpose computers are register machines organized around a collection of registers and operations that constitute an efficient and convenient universal set of data paths. The controller for a general-purpose machine is an interpreter for a register-machine language like the one we have been using. This language is called the *native language* of the machine, or simply *machine language*. Programs written in machine language are sequences of instructions that use the machine's data paths. For example, the explicit-control evaluator's instruction sequence can be thought of as a machine-language pro-

---

<sup>60</sup>This is a theoretical statement. We are not claiming that the evaluator's data paths are a particularly convenient or efficient set of data paths for a general-purpose computer. For example, they are not very good for implementing high-performance floating-point calculations or calculations that intensively manipulate bit vectors.

<sup>61</sup>これは理論的な発言です。この評価機のデータパスが一般的なコンピュータのため特に便利な、または効率的なデータパスの集合であると主張している訳ではありません。例えば、これらは高いパフォーマンスの浮動小数点演算や、激しくビットベクタを操作する演算の実装にはあまり向いていません。

gram for a general-purpose computer rather than as the controller for a specialized interpreter machine.

商業上の汎用なコンピュータはレジスタと効率的で便利なデータパスの普遍的な集合を構成する命令の周りに構築されるレジスタマシンです。汎用目的の機械は私達がここまで使用しているようなレジスタマシン言語のためのインタプリタです。この言語は機械の*native language*(ネイティブ言語)、または単純に*machine language*(機械語)と呼ばれます。機械語で記述されたプログラムはその機械のデータパスを用いた命令列です。例えば、明示的制御評価機の命令列は専門のインタプリタマシンのためのコントローラではなく、汎用目的のコンピュータのための機械語プログラムだと考えることができます。

There are two common strategies for bridging the gap between higher-level languages and register-machine languages. The explicit-control evaluator illustrates the strategy of interpretation. An interpreter written in the native language of a machine configures the machine to execute programs written in a language (called the *source language*) that may differ from the native language of the machine performing the evaluation. The primitive procedures of the source language are implemented as a library of subroutines written in the native language of the given machine. A program to be interpreted (called the *source program*) is represented as a data structure. The interpreter traverses this data structure, analyzing the source program. As it does so, it simulates the intended behavior of the source program by calling appropriate primitive subroutines from the library.

2つの共通な戦略が高水準の言語とレジスタマシンの言語の間のギャップを橋渡しします。明示的制御評価機は逐次翻訳(interpretation)上の戦略を説明します。機械のネイティブ言語で書かれたインタプリタは、評価を実行する機械のネイティブ言語とは異なっても良いある言語(*source language*(ソース言語))で書かれたプログラムを実行するように機械を構成します。ソース言語のプリミティブ手続は与えられた機械のネイティブ言語により記述されたサブルーチンのライブラリとして実装されます。*(source program*(ソースプログラム)と呼ばれる)逐次翻訳するプログラムはデータ構造として表現されます。インタプリタはこのデータ構造を横断し、ソースプログラムを分析します。それを行うにつれ、ソースプログラムの意図された振舞を適切なプリミティブのサブルーチンをライブラリから呼ぶことによりシミュレートします。

In this section, we explore the alternative strategy of *compilation*. A compiler for a given source language and machine translates a source program into an equivalent program (called the *object program*) written in the machine's

native language. The compiler that we implement in this section translates programs written in Scheme into sequences of instructions to be executed using the explicit-control evaluator machine's data paths.<sup>62</sup>

この節では、*compilation*(コンパイル) という代替的な戦略を探求します。与えられたソース言語と機械に対するコンパイラはソースプログラムを機械のネイティブ言語で書かれた (*object program*(オブジェクトプログラム) と呼ばれる) 等価なプログラムに翻訳します。この節で実装するコンパイラは Scheme で書かれたプログラムを明示的制御評価機のデータパスを用いて実行される命令列へと翻訳します。<sup>63</sup>

Compared with interpretation, compilation can provide a great increase in the efficiency of program execution, as we will explain below in the overview of the compiler. On the other hand, an interpreter provides a more powerful environment for interactive program development and debugging, because the source program being executed is available at run time to be examined and modified. In addition, because the entire library of primitives is present, new programs can be constructed and added to the system during debugging.

逐次翻訳と比べた時、コンパイルはプログラム実行の効率性において大きな向上を与えられます。このことは下記にてコンパイラの概観において説明して行きます。一方で、インタプリタはより強力な環境を対話式のプログラム開発とデバッグのために提供します。実行するソースプログラムが実行時にも試験し、変更するために使用可能なためです。それに加えて、プリミティブのライブラリ全体が存在し、新しいプログラムがデバッグの間に構築し、追加することができることも挙げられます。

---

<sup>62</sup> Actually, the machine that runs compiled code can be simpler than the interpreter machine, because we won't use the `exp` and `unev` registers. The interpreter used these to hold pieces of unevaluated expressions. With the compiler, however, these expressions get built into the compiled code that the register machine will run. For the same reason, we don't need the machine operations that deal with expression syntax. But compiled code will use a few additional machine operations (to represent compiled procedure objects) that didn't appear in the explicit-control evaluator machine.

<sup>63</sup> 実際には、コンパイルされたコードを実行する機械はインタプリタマシンよりもより単純に成り得ます。`exp` と `unev` のレジスタを使用しないためです。これらを使用するインタプリタは未評価の式の部分を保持します。しかしコンパイラを用いる場合には、これらの式はレジスタマシンが実行するコンパイルされたコードの中に組込まれます。同じ理由により、式の構文を扱う機械語命令を必要としません。しかしコンパイルされたコードは明示的制御評価機械では存在しなかつたいくつかの追加の(コンパイルされた手続オブジェクトを表現するための)機械語命令を使用します。

In view of the complementary advantages of compilation and interpretation, modern program-development environments pursue a mixed strategy. Lisp interpreters are generally organized so that interpreted procedures and compiled procedures can call each other. This enables a programmer to compile those parts of a program that are assumed to be debugged, thus gaining the efficiency advantage of compilation, while retaining the interpretive mode of execution for those parts of the program that are in the flux of interactive development and debugging. In [Section 5.5.7](#), after we have implemented the compiler, we will show how to interface it with our interpreter to produce an integrated interpreter-compiler development system.

コンパイルと逐次翻訳の相補的な利点の視点において、最新のプログラム開発環境は入り交じった戦略を追求しています。Lisp インタプリタは一般的に逐次翻訳された手続とコンパイルされた手続がお互いを呼びだせるように構築されています。これはプログラマがデバッグすることを想定されているこれらのプログラムの部品をコンパイルすることを可能にします。従ってコンパイルの効率上の利点を得ながら、プログラムのそれらの部品に対して対話式開発とデバッグの流動的な、実行の解釈的なモードを維持することもできます。[Section 5.5.7](#)において、コンパイラを実装した後にはインタプリタとどのように接続して統合的なインタプリタ・コンパイラ開発システムを生成するかを示します。

## An overview of the compiler

Our compiler is much like our interpreter, both in its structure and in the function it performs. Accordingly, the mechanisms used by the compiler for analyzing expressions will be similar to those used by the interpreter. Moreover, to make it easy to interface compiled and interpreted code, we will design the compiler to generate code that obeys the same conventions of register usage as the interpreter: The environment will be kept in the `env` register, argument lists will be accumulated in `arg1`, a procedure to be applied will be in `proc`, procedures will return their answers in `val`, and the location to which a procedure should return will be kept in `continue`. In general, the compiler translates a source program into an object program that performs essentially the same register operations as would the interpreter in evaluating the same source program.

私達のコンパイラは私達のインタプリタに両者のその構造と実行する機能

においてとても良く似ています。従って、コンパイラにより式の解析のために使用される仕組みはインタプリタにて使用されたものと同様になります。さらに、コンパイルされたコードと逐次翻訳されたコードの接続を簡単にするために、インタプリタと同じレジスタ使用法の仕様に従うコードを生成する様にコンパイラを設計します。環境は `env` レジスタに保持され、引数リストは `argl` に蓄積され、適用される手続は `proc` に入り、手続はそれらの回答を `val` に入れて戻り、手続が戻らなければいけない位置は `continue` に維持されます。一般的に、コンパイラはソースプログラムをインタプリタが同じソースプログラムを評価する場合に行うのと本質的に同じレジスタ命令を実行するオブジェクトプログラムに翻訳します。

This description suggests a strategy for implementing a rudimentary compiler: We traverse the expression in the same way the interpreter does. When we encounter a register instruction that the interpreter would perform in evaluating the expression, we do not execute the instruction but instead accumulate it into a sequence. The resulting sequence of instructions will be the object code. Observe the efficiency advantage of compilation over interpretation. Each time the interpreter evaluates an expression—for example, `(f 84 96)`—it performs the work of classifying the expression (discovering that this is a procedure application) and testing for the end of the operand list (discovering that there are two operands). With a compiler, the expression is analyzed only once, when the instruction sequence is generated at compile time. The object code produced by the compiler contains only the instructions that evaluate the operator and the two operands, assemble the argument list, and apply the procedure (in `proc`) to the arguments (in `argl`).

この説明ではとても基本的なコンパイラを実装するための戦略を提案します。式をインタプリタと同じ方法で横断します。インタプリタが式の評価で実行するだろうレジスタ命令に遭遇したら、その命令を実行はしませんがその代わりに列に蓄積します。結果としての命令列はオブジェクトコードになります。逐次翻訳に対するコンパイルの効率上の利点を注意して下さい。インタプリタが式、例えば `(f 84 96)` を評価する度に、式の分類(手続の適用であるかを見出す)とオペラントリストの終端の検査(2つのオペランドが残っているかを見出す)を行います。コンパイラを用いる場合、式は命令列がコンパイル時に生成された時に一度しか解析されません。コンパイラにより生成されたオブジェクトコードはオペレータと 2 つのオペランドを評価する命令しか含んでおらず、引数リストを組み立て、(`proc` 内の) 手続を (`argl` 内の) その引数に適用します。

This is the same kind of optimization we implemented in the analyzing eval-

uator of Section 4.1.7. But there are further opportunities to gain efficiency in compiled code. As the interpreter runs, it follows a process that must be applicable to any expression in the language. In contrast, a given segment of compiled code is meant to execute some particular expression. This can make a big difference, for example in the use of the stack to save registers. When the interpreter evaluates an expression, it must be prepared for any contingency. Before evaluating a subexpression, the interpreter saves all registers that will be needed later, because the subexpression might require an arbitrary evaluation. A compiler, on the other hand, can exploit the structure of the particular expression it is processing to generate code that avoids unnecessary stack operations.

これはSection 4.1.7の解析評価機で実装したものと同じ種類の最適化です。しかし、コンパイルされたコード中で効率を良くするためのさらなる機会が存在します。インタプリタが実行するにしたがって、インタプリタは言語の任意の式に必ず当てはまる過程を追います。対照的に、与えられたコンパイル済みコードの断片はある特定の式を実行することを意味します。これは例えばスタックを用いてレジスタを保存する場合等に大きな違いを生みます。インタプリタが式を評価する時には、任意の偶発性に対して準備をしなければなりません。部分式を評価する前に、インタプリタは後で必要となる全てのレジスタを保存します。部分式が無原則な評価を要求するかもしれません。一方、コンパイラは処理対象の特定の式の構造を利用して不必要的スタック命令を回避するコードを生成することができます。

As a case in point, consider the combination (`f` 84 96). Before the interpreter evaluates the operator of the combination, it prepares for this evaluation by saving the registers containing the operands and the environment, whose values will be needed later. The interpreter then evaluates the operator to obtain the result in `val`, restores the saved registers, and finally moves the result from `val` to `proc`. However, in the particular expression we are dealing with, the operator is the symbol `f`, whose evaluation is accomplished by the machine operation `lookup-variable-value`, which does not alter any registers. The compiler that we implement in this section will take advantage of this fact and generate code that evaluates the operator using the instruction

その一例として、組み合わせ (`f` 84 96) について考えてみます。インタプリタが組み合わせのオペレータを評価する前に、値が後で必要になるオペランドと環境を持つレジスタを保存することでこの評価のための準備を行います。次にインタプリタはオペレータを評価してその結果を `val` に取得し、保存した

レジスタを戻し、最後に結果を `val` から `proc` に移します。しかし、私達が評価しているこの式では、オペレータがシンボルの `f` であり、その評価は機械語の `lookup-variable-value` にて達成され、これはどのレジスタの値も変化させません。この節で実装するコンパイラはこの事実を活用し、オペレータをこの命令を使用して評価するコードを生成します。

```
(assign proc (op lookup-variable-value)
       (const f)
       (reg env))
```

This code not only avoids the unnecessary saves and restores but also assigns the value of the lookup directly to `proc`, whereas the interpreter would obtain the result in `val` and then move this to `proc`.

このコードは不必要的保存と復元を回避するだけでなく、`lookup` の値を直接 `proc` に割り当てます。一方でインタプリタは結果を `val` の中に取得し、その後に `proc` へと移します。

A compiler can also optimize access to the environment. Having analyzed the code, the compiler can in many cases know in which frame a particular variable will be located and access that frame directly, rather than performing the `lookup-variable-value` search. We will discuss how to implement such variable access in [Section 5.5.6](#). Until then, however, we will focus on the kind of register and stack optimizations described above. There are many other optimizations that can be performed by a compiler, such as coding primitive operations “in line” instead of using a general `apply` mechanism (see [Exercise 5.38](#)); but we will not emphasize these here. Our main goal in this section is to illustrate the compilation process in a simplified (but still interesting) context.

コンパイラはまた環境へのアクセスを最適化することができます。コードを解析した後に、コンパイラは多くの場合において、どのフレームの中に特定の変数が位置するかを知り、`lookup-variable-value` による検索を実行するのではなくに、直接アクセスすることができます。そのような変数のアクセスをどのように実装するかについての議論は[Section 5.5.6](#)にて行います。

### 5.5.1 Structure of the Compiler

In [Section 4.1.7](#) we modified our original metacircular interpreter to separate analysis from execution. We analyzed each expression to produce an execution procedure that took an environment as argument and performed the

required operations. In our compiler, we will do essentially the same analysis. Instead of producing execution procedures, however, we will generate sequences of instructions to be run by our register machine.

Section 4.1.7において、私達は元のメタ循環インタプリタを変更して分析を実行から分離しました。各式を分析して環境を引数として取り必要とされる命令を実行する実行手続を生成しました。私達のコンパイラでは、本質的には同じ分析を行います。しかし、実行手続を生成する代わりに、私達のレジスタマシンにより実行される命令列を生成します。

The procedure `compile` is the top-level dispatch in the compiler. It corresponds to the `eval` procedure of Section 4.1.1, the `analyze` procedure of Section 4.1.7, and the `eval-dispatch` entry point of the explicit-control-evaluator in Section 5.4.1. The compiler, like the interpreters, uses the expression-syntax procedures defined in Section 4.1.2.<sup>64</sup> `Compile` performs a case analysis on the syntactic type of the expression to be compiled. For each type of expression, it dispatches to a specialized *code generator*:

手続 `compile` はコンパイラ内でのトップレベルの割り振りです。これはSection 4.1.1の `eval` 手続、Section 4.1.7 の `analyze` 手続、そして Section 5.4.1 の明示的制御評価機のエントリーポイント `eval-dispatch` に対応します。コンパイラはインタプリタと同様に、Section 4.1.2における式の構文手続を用います。<sup>65</sup> `compile` はコンパイルされる式の構文の型の事例分析を実行します。各式の型に対し、特別な *code generator*(コード生成器) を割り振ります。

```
(define (compile exp target linkage)
  (cond ((self-evaluating? exp)
          (compile-self-evaluating exp target linkage))
        ((quoted? exp) (compile-quoted exp target linkage))
        ((variable? exp)
```

<sup>64</sup>Notice, however, that our compiler is a Scheme program, and the syntax procedures that it uses to manipulate expressions are the actual Scheme procedures used with the metacircular evaluator. For the explicit-control evaluator, in contrast, we assumed that equivalent syntax operations were available as operations for the register machine. (Of course, when we simulated the register machine in Scheme, we used the actual Scheme procedures in our register machine simulation.)

<sup>65</sup>しかし、私達のコンパイラが Scheme プログラムであり、式を操作するためにそれが用いる構文手続がメタ循環評価機により仕様される実際の Scheme 手続であることに注意して下さい。一方で、明示的制御評価機では等価な構文命令がレジスタマシンに対する命令として使用可能であると前提しました。(もちろん、Scheme でレジスタマシンをシミュレートした時には、実際の Scheme の手続を使用しました。)

```

  (compile-variable exp target linkage))
((assignment? exp)
  (compile-assignment exp target linkage))
((definition? exp)
  (compile-definition exp target linkage))
((if? exp) (compile-if exp target linkage))
((lambda? exp) (compile-lambda exp target linkage))
((begin? exp)
  (compile-sequence
    (begin-actions exp) target linkage))
((cond? exp)
  (compile (cond->if exp) target linkage))
((application? exp)
  (compile-application exp target linkage))
(else
  (error "Unknown expression type: COMPILE" exp)))

```

## Targets and linkages

`Compile` and the code generators that it calls take two arguments in addition to the expression to compile. There is a *target*, which specifies the register in which the compiled code is to return the value of the expression. There is also a *linkage descriptor*, which describes how the code resulting from the compilation of the expression should proceed when it has finished its execution. The linkage descriptor can require that the code do one of the following three things:

`compile` とそれが呼ぶコード生成器はコンパイル対象の式に加えて 2 つの引数を取ります。コンパイルされたコードがその中で式の値を返すレジスタを指定する *target*(ターゲット) と実行が完了した時に、式のコンパイルの結果としてのコードがどのように続けるべきかを説明する *linkage descriptor*(リンク記述子) です。リンク記述子はコードが以下の 3 つの内 1 つを行うよう要求することができます。

- continue at the next instruction in sequence (this is specified by the linkage descriptor `next`),  
列の次の命令を続ける（これはリンク記述子 `next` により指定されます）

- return from the procedure being compiled (this is specified by the linkage descriptor `return`), or  
コンパイルしている手続から戻る (これはリンク記述子 `return` により指定されます)
- jump to a named entry point (this is specified by using the designated label as the linkage descriptor).  
名前付きエントリポイントへ飛ぶ (これはリンク記述子として指定したラベルを用いて指定します)

For example, compiling the expression 5 (which is self-evaluating) with a target of the `val` register and a linkage of `next` should produce the instruction

例えば、式 5(これは自己評価型) をターゲットをレジスタ `val`、リンク記述子を `next` でコンパイルする時、以下の命令を生成しなければなりません。

```
(assign val (const 5))
```

Compiling the same expression with a linkage of `return` should produce the instructions

同じ式をリンク記述子 `return` でコンパイルする時には以下の命令を生成しなければなりません。

```
(assign val (const 5))
(goto (reg continue))
```

In the first case, execution will continue with the next instruction in the sequence. In the second case, we will return from a procedure call. In both cases, the value of the expression will be placed into the target `val` register.

最初の場合には、実行は列内の次の命令と共に続行します。2つ目の場合には、手続呼出から戻ります。両者の場合において、式の値はターゲットレジスタ `val` に配置されます。

## Instruction sequences and stack usage

Each code generator returns an *instruction sequence* containing the object code it has generated for the expression. Code generation for a compound expression is accomplished by combining the output from simpler code generators for component expressions, just as evaluation of a compound expression is accomplished by evaluating the component expressions.

各コード生成器は式のために生成したオブジェクトコードを含む *instruction sequence*(命令列) を返します。複合式に対するコード生成は部分式のためのよ  
り単純なコード生成器からの出力を組み合わせることにより達成されます。これは複合式の評価が部分式を評価することにより達成されるのと同じです。

The simplest method for combining instruction sequences is a procedure called **append-instruction-sequences**. It takes as arguments any number of instruction sequences that are to be executed sequentially; it appends them and returns the combined sequence. That is, if  $\langle seq_1 \rangle$  and  $\langle seq_2 \rangle$  are sequences of instructions, then evaluating

命令列を組み合わせる最も単純な手法は **append-instruction-sequences** という手続です。これは引数として順に実行されるべき任意の数の命令列を取り、それらを接続し、組み合わされた列を返します。つまり、もし  $\langle seq_1 \rangle$  と  $\langle seq_2 \rangle$  が命令列であるならば、以下の評価は、

```
(append-instruction-sequences <seq1> <seq2>)
```

produces the sequence

次の列を生成します。

```
<seq1>  
<seq2>
```

Whenever registers might need to be saved, the compiler's code generators use **preserving**, which is a more subtle method for combining instruction sequences. **Preserving** takes three arguments: a set of registers and two instruction sequences that are to be executed sequentially. It appends the sequences in such a way that the contents of each register in the set is preserved over the execution of the first sequence, if this is needed for the execution of the second sequence. That is, if the first sequence modifies the register and the second sequence actually needs the register's original contents, then **preserving** wraps a **save** and a **restore** of the register around the first sequence before appending the sequences. Otherwise, **preserving** simply returns the appended instruction sequences. Thus, for example, `(preserving (list <reg1> <reg2>) <seq1> <seq2>)` produces one of the following four sequences of instructions, depending on how  $\langle seq_1 \rangle$  and  $\langle seq_2 \rangle$  use  $\langle reg_1 \rangle$  and  $\langle reg_2 \rangle$ :

レジスタが保存される必要がある度に、コンパイラのコード生成器は **preserving** を使用します。これは命令列を組み立てるための、より芸が細か

い手法です。`preserving` は 3 つの引数を取ります。レジスタの集合と 2 つの命令列です。これは列をレジスタ集合内の各レジスタの中身が、2 つ目の列の実行に必要ならば、最初の列の実行の間は維持 (preserve) されるような方法で接続します。言い換えれば、もし最初の命令列がレジスタを変更し、2 つ目の列が実際にそのレジスタの元の中身を必要とするならば、`preserving` は列を接続する前に最初の列をそのレジスタの `save` と `restore` で包みます。そうでなければ、`preserving` は単純に接続した命令列を返します。従って、例えば `(preserving (list <reg1> <reg2>) <seq1> <seq2>)` は、`<seq1>` と `<seq2>` がどのように `<reg1>` と `<reg2>` を使用するかに依存して以下の 4 つの命令列の内 1 つを生成します。

<code>&lt;seq<sub>1</sub>&gt;</code>	<code>(save &lt;reg<sub>1</sub>&gt;)</code>	<code>(save &lt;reg<sub>2</sub>&gt;)</code>	<code>(save &lt;reg<sub>2</sub>&gt;)</code>
<code>&lt;seq<sub>2</sub>&gt;</code>	<code>&lt;seq<sub>1</sub>&gt;</code>	<code>&lt;seq<sub>1</sub>&gt;</code>	<code>(save &lt;reg<sub>1</sub>&gt;)</code>
	<code>(restore &lt;reg<sub>1</sub>&gt;)</code>	<code>(restore &lt;reg<sub>2</sub>&gt;)</code>	<code>&lt;seq<sub>1</sub>&gt;</code>
	<code>&lt;seq<sub>2</sub>&gt;</code>	<code>&lt;seq<sub>2</sub>&gt;</code>	<code>(restore &lt;reg<sub>1</sub>&gt;)</code>

By using `preserving` to combine instruction sequences the compiler avoids unnecessary stack operations. This also isolates the details of whether or not to generate `save` and `restore` instructions within the `preserving` procedure, separating them from the concerns that arise in writing each of the individual code generators. In fact no `save` or `restore` instructions are explicitly produced by the code generators.

`preserving` を用いて命令列を組み立てることにより、コンパイラは不要なスタック命令を回避することが可能になります。これはまた `save` と `restore` の命令を `preserving` 手続の中で生成するか、しないかの詳細を分離し、個別のコード生成器それぞれを書く場合に浮かび上がる考慮点から隔離します。実際に `save` と `restore` の命令は明示的にはコード生成器により生成されることはありません。

In principle, we could represent an instruction sequence simply as a list of instructions. `Append-instruction-sequences` could then combine instruction sequences by performing an ordinary list `append`. However, `preserving` would then be a complex operation, because it would have to analyze each instruction sequence to determine how the sequence uses its registers. `Preserving` would be inefficient as well as complex, because it would have to analyze each of

its instruction sequence arguments, even though these sequences might themselves have been constructed by calls to `preserving`, in which case their parts would have already been analyzed. To avoid such repetitious analysis we will associate with each instruction sequence some information about its register use. When we construct a basic instruction sequence we will provide this information explicitly, and the procedures that combine instruction sequences will derive register-use information for the combined sequence from the information associated with the component sequences.

原理上は、命令列を単純に命令のリストとして表現できるでしょう。`append-instruction-sequences` はそうすると命令列の組み立てを通常のリストの `append` にて行うことができます。するとしかし、`preserving` は複雑な命令になります。それが各命令列に対し、レジスタをどのように使用するかの分析を行わなければならないためです。また複雑であると同様に `preserving` が非効率にもなります。各命令列の引数をも分析しなければならなくなるためです。例えこれらの列自身が `preserving` の呼出により構築されていて、それらの部品が既に分析されていてもです。そのような分析の繰り返しを防ぐために、各命令列とそのレジスタ使用に関する情報を結び付けます。基本的な命令列を構築する時に、私達はこの情報を明示的に与えます。そして命令列を接続する手続は列の組み合わせのために、レジスタ使用の情報を構成部品である列に結び付けられた情報から引き出します。

An instruction sequence will contain three pieces of information:

命令列は 3 つの情報を持ちます。

- the set of registers that must be initialized before the instructions in the sequence are executed (these registers are said to be *needed* by the sequence),

命令列内の命令が実行される前に初期化しなければならないレジスタ集合 (これらのレジスタは命令列により *needed*(必要とされる) と述べられる)

- the set of registers whose values are modified by the instructions in the sequence, and

列内の命令によりその値が変更されるレジスタ集合

- the actual instructions (also called *statements*) in the sequence.

列内の実際の命令 (命令文 (*statements*) とも呼ばれる)

We will represent an instruction sequence as a list of its three parts. The constructor for instruction sequences is thus

命令列をその3つの部品として表現します。命令列のコンストラクタは従って以下のようになります。

```
(define (make-instruction-sequence
           needs modifies statements)
  (list needs modifies statements))
```

For example, the two-instruction sequence that looks up the value of the variable `x` in the current environment, assigns the result to `val`, and then returns, requires registers `env` and `continue` to have been initialized, and modifies register `val`. This sequence would therefore be constructed as

喻えれば、現在の環境内で変数 `x` の値を探し、その結果を `val` に割り当てて戻る2つの命令の列はレジスタ `env` と `continue` が初期化される必要があります、そしてレジスタ `val` を変更します。この列は従って以下のように構築されます。

```
(make-instruction-sequence
  '(env continue)
  '(val)
  '(((assign val
             (op lookup-variable-value) (const x) (reg env))
    (goto (reg continue)))))
```

We sometimes need to construct an instruction sequence with no statements:  
時々、命令文が無い命令列を構築する必要があります。

```
(define (empty-instruction-sequence)
  (make-instruction-sequence '() '() '()))
```

The procedures for combining instruction sequences are shown in [Section 5.5.4](#).  
命令列を組み立てる手続は [Section 5.5.4](#) に示します。

**Exercise 5.31:** In evaluating a procedure application, the explicit-control evaluator always saves and restores the `env` register around the evaluation of the operator, saves and restores `env` around the evaluation of each operand (except the final one), saves and restores `argl` around the evaluation of each operand, and saves and restores `proc` around the evaluation of the operand sequence. For each of the following combinations, say which of these `save` and `restore` operations are superfluous and thus could be eliminated by the compiler's `preserving` mechanism:

手続適用を評価する場合において、明示的制御評価機は常にオペレータの評価の周りで `env` レジスタの保存と復元を行う。また各オペランドの評価の評価の周りでも（最後の 1 つを除いて）`env` の保存と復元を行う。そえいオペランド列の評価の周りでは `proc` の保存と復元を行う。以下の各組み合わせに対し、これらの `save` と `restore` 命令のどれが余分であり、従ってコンパイラの `preserving` の仕組みにより削減できるかを述べよ。

```
(f 'x 'y)
((f) 'x 'y)
(f (g 'x) y)
(f (g 'x) 'y)
```

**Exercise 5.32:** Using the `preserving` mechanism, the compiler will avoid saving and restoring `env` around the evaluation of the operator of a combination in the case where the operator is a symbol. We could also build such optimizations into the evaluator. Indeed, the explicit-control evaluator of Section 5.4 already performs a similar optimization, by treating combinations with no operands as a special case.

`preserving` の仕組みを用いた場合、コンパイラは組み合わせのオペレータの評価の周りでオペレータがシンボルの場合に `env` の保存と復元を削減することができる。またそのような最適化を評価機の中に構築することもできるだろう。実際に、Section 5.4 の明示的制御評価機は既に似たような最適化をオペランドの無い組み合わせを特別な場合として扱うことで実行している。

- a Extend the explicit-control evaluator to recognize as a separate class of expressions combinations whose operator is a symbol, and to take advantage of this fact in evaluating such expressions.

明示的制御評価機を拡張しオペレータがシンボルである組み合わせを別のクラスの式として認識するようにせよ。そしてこの事実をそのような式の評価において活用するようにせよ。

- b Alyssa P. Hacker suggests that by extending the evaluator to recognize more and more special cases we could incorporate

all the compiler's optimizations, and that this would eliminate the advantage of compilation altogether. What do you think of this idea?

Alyssa P. Hacker は評価機を拡張し、組込むことができる全てのコンパイラの最適化をより多くの特別な場合として認識することで、コンパイルの利点全体を無くすことができる提案した。あなたはこの考え方をどう思うか?

### 5.5.2 Compiling Expressions

In this section and the next we implement the code generators to which the `compile` procedure dispatches.

この節と次の節では、`compile` 手続が割り振るコード生成器を実装します。

#### Compiling linkage code

In general, the output of each code generator will end with instructions—generated by the procedure `compile-linkage`—that implement the required linkage. If the linkage is `return` then we must generate the instruction (`goto (reg continue)`). This needs the `continue` register and does not modify any registers. If the linkage is `next`, then we needn't include any additional instructions. Otherwise, the linkage is a label, and we generate a `goto` to that label, an instruction that does not need or modify any registers.

一般的に、各コード生成器の出力は手続 `compile-linkage` により生成された、要求されたリンク記述子を実装した命令で終わります。もしリンク記述子が `return` なら、命令 (`goto (reg continue)`) を生成せねばなりません。これは `continue` レジスタを必要とし、他のレジスタを変更はしません。もしリンク記述子が `next` なら、何の追加の命令も必要ありません。さもなければ、リンク記述子はラベルであり、そのラベルへの `goto` を生成します。この命令はレジスタを必要とせず、変更もしません。<sup>66</sup>

---

<sup>66</sup>This procedure uses a feature of Lisp called *backquote* (or *quasiquote*) that is handy for constructing lists. Preceding a list with a backquote symbol is much like quoting it, except that anything in the list that is flagged with a comma is evaluated.

この手続は *backquote*(バッククオート、または *quasiquote*(擬似クオート)) と呼ばれる Lisp の機能を使用します。これはリストを構築するのに便利です。リストの前にバッククオート記号を置くことはクオートすることにとても似ていますが、リスト内のカンマ

```
(define (compile-linkage linkage)
  (cond ((eq? linkage 'return)
    (make-instruction-sequence '(continue) '()
     `((goto (reg continue)))))
   ((eq? linkage 'next)
    (empty-instruction-sequence))
   (else
    (make-instruction-sequence '() '()
     `((goto (label ,linkage)))))))
```

The linkage code is appended to an instruction sequence by **preserving** the **continue** register, since a **return** linkage will require the **continue** register: If the given instruction sequence modifies **continue** and the linkage code needs it, **continue** will be saved and restored.

リンクのコードが命令列に対し **preserving** により **continue** レジスタを維持しながら追加されます。リンク記述子 **return** が **continue** レジスタを必要とするためです。もし与えられた命令列が **continue** を変更し、リンクのコードがそれを必要とする場合、**continue** は保存と復元が行われます。

```
(define (end-with-linkage linkage instruction-sequence)
```

で合図された物全てを評価することが異なります。

For example, if the value of **linkage** is the symbol **branch25**, then the expression  
例えば、もし **linkage** の値がシンボル **branch25** の場合、以下の式は

```
`((goto (label ,linkage)))
```

evaluates to the list

次のリストとして評価されます。

```
((goto (label branch25)))
```

Similarly, if the value of **x** is the list (**a** **b** **c**), then

同様に、もし **x** の値がリスト (**a** **b** **c**) ならば、以下の式は

```
`(1 2 ,(car x))
```

evaluates to the list

次のリストとして評価されます。

```
(1 2 a)
```

```
(preserving '(continue)
  instruction-sequence
  (compile-linkage linkage)))
```

## Compiling simple expressions

The code generators for self-evaluating expressions, quotations, and variables construct instruction sequences that assign the required value to the target register and then proceed as specified by the linkage descriptor.

自己評価型式、クオート、変数に対するコード生成器は必要な値をターゲットのレジスタに割り当てリンク記述子により指示されたように進める命令列を構築します。

```
(define (compile-self-evaluating exp target linkage)
  (end-with-linkage linkage
    (make-instruction-sequence '() (list target)
      `((assign ,target (const ,exp))))))
(define (compile-quoted exp target linkage)
  (end-with-linkage linkage
    (make-instruction-sequence '() (list target)
      `((assign ,target (const ,(text-of-quotation exp)))))))
(define (compile-variable exp target linkage)
  (end-with-linkage linkage
    (make-instruction-sequence '(env) (list target)
      `((assign ,target
        (op lookup-variable-value)
        (const ,exp)
        (reg env))))))
```

All these assignment instructions modify the target register, and the one that looks up a variable needs the `env` register.

これら全ての代入命令はターゲットレジスタを変更します。また変数の検索を行う物は `env` レジスタを必要とします。

Assignments and definitions are handled much as they are in the interpreter. We recursively generate code that computes the value to be assigned to the variable, and append to it a two-instruction sequence that actually sets or defines the variable and assigns the value of the whole expression (the symbol

`ok`) to the target register. The recursive compilation has target `val` and linkage `next` so that the code will put its result into `val` and continue with the code that is appended after it. The appending is done preserving `env`, since the environment is needed for setting or defining the variable and the code for the variable value could be the compilation of a complex expression that might modify the registers in arbitrary ways.

代入と定義はインタプリタの物と同様に扱われます。再帰的に変数に割り当てる値を求めるコードを生成し、それに対して実際に変数の設定、または定義を行う物と式全体の値(シンボル `ok`)を割り当てる物の2つの命令列を接続します。再帰的なコンパイルはターゲット `val` とリンク記述子 `next` を持つのでコードはその結果を `val` に入れ、その後に接続されたコードを用いて続けられます。接続は `env` を維持(preserving)している間に行われます。環境が変数の設定、または定義のため必要なためです。また変数の値のためのコードは複雑な式のコンパイルと成り得るため任意の方法でレジスタを変更する可能性があります。

```
(define (compile-assignment exp target linkage)
  (let ((var (assignment-variable exp)))
    (get-value-code
      (compile (assignment-value exp) 'val 'next)))
  (end-with-linkage linkage
    (preserving '(env)
      get-value-code
      (make-instruction-sequence '(env val) (list target)
        `((perform (op set-variable-value!)
                  (const ,var)
                  (reg val)
                  (reg env))
          (assign ,target (const ok))))))))
  
(define (compile-definition exp target linkage)
  (let ((var (definition-variable exp)))
    (get-value-code
      (compile (definition-value exp) 'val 'next)))
  (end-with-linkage linkage
    (preserving '(env)
      get-value-code
```

```
(make-instruction-sequence '(env val) (list target)
  `((perform (op define-variable!)
              (const ,var)
              (reg val)
              (reg env))
    (assign ,target (const ok)))))))
```

The appended two-instruction sequence requires `env` and `val` and modifies the target. Note that although we preserve `env` for this sequence, we do not preserve `val`, because the `get-value-code` is designed to explicitly place its result in `val` for use by this sequence. (In fact, if we did preserve `val`, we would have a bug, because this would cause the previous contents of `val` to be restored right after the `get-value-code` is run.)

接続された2つの命令列は `env` と `val` を必要とし、ターゲットを変更します。例え `env` をこの列のために維持したとしても、`val` は維持しません。`get-value-code` が明示的にその結果をこの命令列のために `val` に配置するように設計されているためです。(実際には、もし `val` を維持した場合、バグを持つことになります。これにより直前の `val` の中身が `get-value-code` の実行直後に復元されるためです。)

## Compiling conditional expressions

The code for an `if` expression compiled with a given target and linkage has the form

与えられたターゲットとリンク記述子と共にコンパイルされる `if` 式のためのコードは以下の形式を持ちます。

```
<述語のコンパイル, ターゲット val, リンク記述子 next>
  (test (op false?) (reg val))
  (branch (label false-branch))
true-branch
<結果部与えられたターゲット与えられたリンク記述子又は after-if>
false-branch
<代替部与えられたターゲットリンク記述子>
after-if
```

To generate this code, we compile the predicate, consequent, and alternative, and combine the resulting code with instructions to test the predicate result

and with newly generated labels to mark the true and false branches and the end of the conditional.

このコードを生成するために、述語、結果部、代替部をコンパイルし、結果のコードを述語の結果をテストするコードと新しく生成された真と偽の分岐をマークするラベルと条件文の最後と共に組み立てます。<sup>67</sup> In this arrangement of code, we must branch around the true branch if the test is false. The only slight complication is in how the linkage for the true branch should be handled. If the linkage for the conditional is `return` or a label, then the true and false branches will both use this same linkage. If the linkage is `next`, the true branch ends with a jump around the code for the false branch to the label at the end of the conditional.

このコードの準備では、テストが偽である場合、真の分岐へと飛ぶ必要があります。唯一、微妙に複雑なのは真の分岐がリンク記述子をどのように扱うかについてです。もし条件文のリンク記述子が `return`、またはラベルの場合、真と偽の分岐は両方共これと同じリンク記述子を用います。もしリンク記述子が `next` なら、真の分岐は偽の分岐を飛び越し条件文の最後へと飛ぶコードで

---

<sup>67</sup>We can't just use the labels `true-branch`, `false-branch`, and `after-if` as shown above, because there might be more than one `if` in the program. The compiler uses the procedure `make-label` to generate labels. `Make-label` takes a symbol as argument and returns a new symbol that begins with the given symbol. For example, successive calls to `(make-label 'a)` would return `a1`, `a2`, and so on. `Make-label` can be implemented similarly to the generation of unique variable names in the query language, as follows:

私達は単にラベル `true-branch`, `false-branch`, `after-if` を上で示されたように使うことはできません。なぜならプログラム中に `if` 文は複数存在する可能性があるためです。`make-label` はシンボルを引数として、与えられたシンボルで始まる取り新しいシンボルを返します。例えば、`(make-label 'a)` に対する連続した呼出は `a1`, `a2`, ... を返します。`make-label` はクエリ言語における一意の変数名の生成と同様に、以下の様に実装することができます。

```
(define label-counter 0)
(define (new-label-number)
  (set! label-counter (+ 1 label-counter))
  label-counter)
(define (make-label name)
  (string->symbol
    (string-append (symbol->string name)
                  (number->string (new-label-number)))))
```

終わります。

```
(define (compile-if exp target linkage)
  (let ((t-branch (make-label 'true-branch))
        (f-branch (make-label 'false-branch))
        (after-if (make-label 'after-if)))
    (let ((consequent-linkage
           (if (eq? linkage 'next) after-if linkage)))
      (let ((p-code (compile (if-predicate exp) 'val 'next))
            (c-code
              (compile
                (if-consequent exp) target
                consequent-linkage)))
        (a-code
          (compile (if-alternative exp) target linkage)))
      (preserving '(env continue)
        p-code
        (append-instruction-sequences
          (make-instruction-sequence '(val) '()
            `((test (op false?) (reg val))
              (branch (label ,f-branch)))))

        (parallel-instruction-sequences
          (append-instruction-sequences t-branch c-code)
          (append-instruction-sequences f-branch a-code)))
      after-if)))))
```

`Env` is preserved around the predicate code because it could be needed by the true and false branches, and `continue` is preserved because it could be needed by the linkage code in those branches. The code for the true and false branches (which are not executed sequentially) is appended using a special combiner `parallel-instruction-sequences` described in [Section 5.5.4](#).

`env` は述語コードの間維持されます。真と偽の分岐で必要になるかもしれませんいためです。そして `continue` もそれら分岐内でリンクのためのコードにて使用されるかもしれないため維持されます。真と偽の分岐のためのコード(順には実行されません)は[Section 5.5.4](#)で説明される専用の結合器、`parallel-instruction-sequences` を使用して接続されます。

Note that `cond` is a derived expression, so all that the compiler needs to

do handle it is to apply the `cond->if` transformer (from [Section 4.1.2](#)) and compile the resulting `if` expression.

`cond` は派生式であることに注意してください。そのためコンパイラが取り扱いのために必要なこと全ては ([Section 4.1.2の](#))`cond->if` 変換器を適用して、結果の `if` 式をコンパイルするだけです。

## Compiling sequences

The compilation of sequences (from procedure bodies or explicit `begin` expressions) parallels their evaluation. Each expression of the sequence is compiled—the last expression with the linkage specified for the sequence, and the other expressions with linkage `next` (to execute the rest of the sequence). The instruction sequences for the individual expressions are appended to form a single instruction sequence, such that `env` (needed for the rest of the sequence) and `continue` (possibly needed for the linkage at the end of the sequence) are preserved.

列のコンパイル (手続のボディ、または明示的な `begin` 式) はそれらの評価を並列化します。列の各式は次の条件でコンパイルされます。最後の式は列に対して指示されたリンク記述子を用いて。他の式はリンク記述子 `next` を用いて (列の残りを実行するために)。個別の式の命令列は接続され单一の命令列を形成します。(列の残りのために必要な)`env` と (列の終わりのリンクコードで必要な可能性のある)`continue` は維持されます。

```
(define (compile-sequence seq target linkage)
  (if (last-exp? seq)
      (compile (first-exp seq) target linkage)
      (preserving '(env continue)
        (compile (first-exp seq) target 'next)
        (compile-sequence (rest-exps seq) target linkage))))
```

## Compiling lambda expressions

`Lambda` expressions construct procedures. The object code for a `lambda` expression must have the form

`lambda` 式は手続を構築します。`lambda` 式のためのオブジェクトコードは以下の形式に従わねばなりません。

⟨手続オブジェクトの構築  
　ターゲットレジスタにそれを割り当てる⟩  
⟨リンク⟩

When we compile the `lambda` expression, we also generate the code for the procedure body. Although the body won't be executed at the time of procedure construction, it is convenient to insert it into the object code right after the code for the `lambda`. If the linkage for the `lambda` expression is a label or `return`, this is fine. But if the linkage is `next`, we will need to skip around the code for the procedure body by using a linkage that jumps to a label that is inserted after the body. The object code thus has the form

`lambda` 式をコンパイルする時、手続のボディのためのコードも生成します。例えボディが手続構築時に実行されなくても、オブジェクトコードの中の `lambda` 式のコードの直後に挿入しておくことは便利です。もし `lambda` 式に対するリンク記述子がラベルか `return` ならば、このことに問題はありません。しかし、もしリンク記述子が `next` ならば、手続のボディの後ろに挿入されたラベルへ飛ぶリンク記述子を使用することによりボディに対するコードを回避する必要があります。従ってオブジェクトコードは以下の形式になります。

⟨手続オブジェクトの構築  
　それをターゲットレジスタに割当⟩  
⟨与えられたリンク記述子に対するコード⟩ または (`goto (label after-lambda)`)  
⟨手続ボディのコンパイル後コード⟩  
`after-lambda`

`Compile-lambda` generates the code for constructing the procedure object followed by the code for the procedure body. The procedure object will be constructed at run time by combining the current environment (the environment at the point of definition) with the entry point to the compiled procedure body (a newly generated label).

`compile-lambda` は手続のボディのコードが続く手続オブジェクトを構築するためのコードを生成します。手続オブジェクトは実行時に現在の環境(定義時点での環境)をコンパイルされた手続ボディのエントリポイント(新しく生成されたラベル)と共に組み立てることで構築されます。<sup>68</sup>

---

<sup>68</sup> We need machine operations to implement a data structure for representing compiled procedures, analogous to the structure for compound procedures described in

```

(define (compile-lambda exp target linkage)
  (let ((proc-entry (make-label 'entry))
        (after-lambda (make-label 'after-lambda)))
    (let ((lambda-linkage
           (if (eq? linkage 'next) after-lambda linkage)))
      (append-instruction-sequences
       (tack-on-instruction-sequence
        (end-with-linkage lambda-linkage
                           (make-instruction-sequence '(env) (list target)
                                         `((assign ,target
                                                   (op make-compiled-procedure)
                                                   (label ,proc-entry)
                                                   (reg env))))))
       (compile-lambda-body exp proc-entry)
       after-lambda))))

```

Compile-lambda uses the special combiner `tack-on-instruction-sequence` rather than `append-instruction-sequences` (Section 5.5.4) to append the procedure body to the `lambda` expression code, because the body is not part of the sequence of instructions that will be executed when the combined sequence is entered; rather, it is in the sequence only because that was a convenient place to put it.

`compile-lambda` は `append-instruction-sequences` (Section 5.5.4) ではなく、特別な結合器 `tack-on-instruction-sequence` を手続のボディと `lambda` 式のコードを接続するのに利用します。ボディは組み立てられた列が入力された時に実行される命令列の一部ではないためです。そうではなく、それはただ、

---

#### Section 4.1.3:

Section 4.1.3で説明した複合手続のための構造と同様に、コンパイル後の手続を表現するためのデータ構造を実装するための機械語命令を必要とします。

```

(define (make-compiled-procedure entry env)
  (list 'compiled-procedure entry env))
(define (compiled-procedure? proc)
  (tagged-list? proc 'compiled-procedure))
(define (compiled-procedure-entry c-proc) (cadr c-proc))
(define (compiled-procedure-env c-proc) (caddr c-proc))

```

そこに置くことが便利だから、その列の中に入ります。

`Compile-lambda-body` constructs the code for the body of the procedure. This code begins with a label for the entry point. Next come instructions that will cause the run-time evaluation environment to switch to the correct environment for evaluating the procedure body—namely, the definition environment of the procedure, extended to include the bindings of the formal parameters to the arguments with which the procedure is called. After this comes the code for the sequence of expressions that makes up the procedure body. The sequence is compiled with linkage `return` and target `val` so that it will end by returning from the procedure with the procedure result in `val`.

`compile-lambda-body` は手続のボディのためのコードを構築します。このコードはエントリーポイントに対するラベルで開始します。次に来るるのは実行時の環境を手続のボディの評価を評価するために正しい環境へとスイッチする命令列です。即ち、手続の定義環境であり、これは手続が呼ばれる時に利用される引数に対する形式パラメタの束縛を含むように拡張されています。これの後には、式の列のコードが来ます。これが手続のボディを作り上げます。この列はリンク記述子 `return` とターゲット `val` と共にコンパイルさるため、手続の結果は `val` に入れられた状態で手続から戻ることで終わります。

```
(define (compile-lambda-body exp proc-entry)
  (let ((formals (lambda-parameters exp)))
    (append-instruction-sequences
      (make-instruction-sequence '(env proc argl) '(env)
        `(~(proc-entry
          (assign env
            (op compiled-procedure-env)
            (reg proc))
          (assign env
            (op extend-environment)
            (const ,formals)
            (reg argl)
            (reg env)))))

      (compile-sequence (lambda-body exp) 'val 'return))))
```

### 5.5.3 Compiling Combinations

The essence of the compilation process is the compilation of procedure applications. The code for a combination compiled with a given target and linkage has the form

コンパイル処理の本質は手続適用のコンパイルです。与えられたターゲットとリンク記述子と共にコンパイルされた組み合わせのコードは以下の形式を持ちます。

```
<演算子のコンパイル, ターゲット proc, リンク記述子 next>
<オペランドを評価し、argl 内に引数リストを構築>
<手続呼出のコンパイル
  与えられたターゲットとリンク記述子と共に>
```

The registers `env`, `proc`, and `argl` may have to be saved and restored during evaluation of the operator and operands. Note that this is the only place in the compiler where a target other than `val` is specified.

レジスタ `env`, `proc`, `argl` はオペレータ (演算子) とオペランドの評価の間に保存と復元を行う必要があるかもしれません。ここだけがこのコンパイラにおいて `val` 以外のターゲットが指定される箇所であることに注意して下さい。

The required code is generated by `compile-application`. This recursively compiles the operator, to produce code that puts the procedure to be applied into `proc`, and compiles the operands, to produce code that evaluates the individual operands of the application. The instruction sequences for the operands are combined (by `construct-arglist`) with code that constructs the list of arguments in `argl`, and the resulting argument-list code is combined with the procedure code and the code that performs the procedure call (produced by `compile-procedure-call`). In appending the code sequences, the `env` register must be preserved around the evaluation of the operator (since evaluating the operator might modify `env`, which will be needed to evaluate the operands), and the `proc` register must be preserved around the construction of the argument list (since evaluating the operands might modify `proc`, which will be needed for the actual procedure application). `Continue` must also be preserved throughout, since it is needed for the linkage in the procedure call.

必要なコードは `compile-application` により生成されます。これは再帰的にオペレータをコンパイルして `proc` に適用する手続を配置するコードを生成し、オペランドをコンパイルして個別の適用の個々のオペランドを評価するコードを生成します。オペランドの命令列は (`construct-arglist` により)`argl`

に引数リストを構築するコードと共に組み合わされます。そして結果となる引数リストのコードは手続のコードと (`compile-procedure-call` により生成された) 手続呼出を実行するコードと共に組み合わされます。コードの列の接続において、`env` レジスタはオペレータの評価の周りにおいて維持 (preserving) されなければなりません。(オペレータの評価がオペランドの評価で必要となる `env` を変更する可能性があるため)。そして `proc` レジスタは引数リストの周りで維持されなければなりません。(オペランドの評価が実際の手続適用に必要な `proc` レジスタを変更するかもしれないため)。`Continue` もまたその間中、維持されなければなりません。手続呼出のリンクコードが必要とするためです。

```
(define (compile-application exp target linkage)
  (let ((proc-code (compile (operator exp) 'proc 'next))
        (operand-codes
         (map (lambda
                  (operand) (compile operand 'val 'next))
              (operands exp))))
    (preserving '(env continue)
      proc-code
      (preserving '(proc continue)
        (construct-arglist operand-codes)
        (compile-procedure-call target linkage))))))
```

The code to construct the argument list will evaluate each operand into `val` and then `cons` that value onto the argument list being accumulated in `argl`. Since we `cons` the arguments onto `argl` in sequence, we must start with the last argument and end with the first, so that the arguments will appear in order from first to last in the resulting list. Rather than waste an instruction by initializing `argl` to the empty list to set up for this sequence of evaluations, we make the first code sequence construct the initial `argl`. The general form of the argument-list construction is thus as follows:

引数リストを構築するためのコードは `val` 内に評価して、次にその値を `argl` に蓄積される引数リスト上に `cons` します。`argl` 上に順に引数を `cons` するため、最後の引数から開始し、最初のもので終わらなければなりません。そうすることで引数は結果リストの中に最初から最後の順で現れることになります。この一連の評価のための設定を行うため、`argl` を空に初期化することで命令を無駄にするのではなく、`argl` の初期値を構築する最初のコード列を作成します。従って、引数リスト構築の一般的な形式は以下になります。

```

⟨最後のオペランドのコンパイル, ターゲットは val⟩
(assign argl (op list) (reg val))
⟨次のオペランドのコンパイル, ターゲットは val⟩
(assign argl (op cons) (reg val) (reg argl))
...
⟨最初のオペランドのコンパイル, ターゲットは val⟩
(assign argl (op cons) (reg val) (reg argl))

```

`Argl` must be preserved around each operand evaluation except the first (so that arguments accumulated so far won't be lost), and `env` must be preserved around each operand evaluation except the last (for use by subsequent operand evaluations).

`argl` は各オペランドの評価の間、最初の 1 つを除いて維持しなければなりません。(そうすることで、そこまで蓄積した引数を失わないように)。そして `env` は(続きのオペランド評価での使用のため) 各オペランドの評価の周りで、最後の 1 つを除いて維持されなければなりません。

Compiling this argument code is a bit tricky, because of the special treatment of the first operand to be evaluated and the need to preserve `argl` and `env` in different places. The `construct-arglist` procedure takes as arguments the code that evaluates the individual operands. If there are no operands at all, it simply emits the instruction

この引数コードのコンパイルは少しだけ巧妙です。評価する最初のオペランドの特別な扱いと、`argl` と `env` を異なる箇所にて維持する必要性のためです。`construct-arglist` 手続は引数として個々のオペランドを評価するコードを取ります。もしオペランドが全く無ければ、単純に以下の命令を発行します。

```
(assign argl (const ()))
```

Otherwise, `construct-arglist` creates code that initializes `argl` with the last argument, and appends code that evaluates the rest of the arguments and adjoins them to `argl` in succession. In order to process the arguments from last to first, we must reverse the list of operand code sequences from the order supplied by `compile-application`.

そうでなければ、`construct-arglist` は `argl` を最後の引数で初期化するコードを生成し、引数の残りを評価するコードを接続し、それらを相次いで `argl` の中に隣接させていきます。引数を最後から最初へ処理するために、オペランドのコード列のリストを `compile-application` により提供された順から逆順(reverse) にする必要があります。

```

(define (construct-arglist operand-codes)
  (let ((operand-codes (reverse operand-codes)))
    (if (null? operand-codes)
        (make-instruction-sequence '() '(argl)
          '((assign argl (const ()))))
        (let ((code-to-get-last-arg
              (append-instruction-sequences
                (car operand-codes)
                (make-instruction-sequence '(val) '(argl)
                  '((assign argl (op list) (reg val)))))))
          (if (null? (cdr operand-codes))
              code-to-get-last-arg
              (preserving '(env)
                code-to-get-last-arg
                (code-to-get-rest-args
                  (cdr operand-codes)))))))

(define (code-to-get-rest-args operand-codes)
  (let ((code-for-next-arg
        (preserving '(argl)
          (car operand-codes)
          (make-instruction-sequence '(val argl) '(argl)
            '((assign argl
                      (op cons) (reg val) (reg argl)))))))
    (if (null? (cdr operand-codes))
        code-for-next-arg
        (preserving '(env)
          code-for-next-arg
          (code-to-get-rest-args (cdr operand-codes)))))))

```

## Applying procedures

After evaluating the elements of a combination, the compiled code must apply the procedure in `proc` to the arguments in `argl`. The code performs essentially the same dispatch as the `apply` procedure in the metacircular evaluator of Section 4.1.1 or the `apply-dispatch` entry point in the explicit-control

evaluator of Section 5.4.1. It checks whether the procedure to be applied is a primitive procedure or a compiled procedure. For a primitive procedure, it uses `apply-primitive-procedure`; we will see shortly how it handles compiled procedures. The procedure-application code has the following form:

組み合わせの要素を評価した後に、コンパイルされたコードは `proc` 内の手続を `argl` 内の引数に適用しなければなりません。このコードは本質的に Section 4.1.1 のメタ循環評価機の `apply` 手続、または Section 5.4.1 の明示的制御評価機の `apply-dispatch` エントリーポイントと同じ割り振りを実行します。適用する手続がプリミティブな手続であるか複合手続であるかを確認します。プリミティブな手続に対しては、`apply-primitive-procedure` を使用します。簡潔にこれがどのようにコンパイルされた手續を取り扱うのかについて見てていきます。手續適用のコードは以下の形式を持ちます。

```
(test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch))
  compiled-branch
  <与えられたターゲットと適切なリンク記述子と共に手続をコンパイルするコード>
  primitive-branch
  (assign <target>
    (op apply-primitive-procedure)
    (reg proc)
    (reg argl))
  <リンクコード>
after-call
```

Observe that the compiled branch must skip around the primitive branch. Therefore, if the linkage for the original procedure call was `next`, the compound branch must use a linkage that jumps to a label that is inserted after the primitive branch. (This is similar to the linkage used for the true branch in `compile-if`.)

コンパイルされた分岐は `primitive-branch` をスキップしなければならないことに注意して下さい。従って、もし元の手続呼出のリンク記述子が `next` ならば、複合分岐は `primitive-branch` の後に挿入されたラベルへと飛ぶリンクコードを使用しなければなりません。(これは `compile-if` において、真の分岐のために使用されたリンクコードと同様です。)

```
(define (compile-procedure-call target linkage)
```

```

(let ((primitive-branch (make-label 'primitive-branch))
      (compiled-branch (make-label 'compiled-branch))
      (after-call (make-label 'after-call)))

(let ((compiled-linkage
        (if (eq? linkage 'next) after-call linkage)))
  (append-instruction-sequences
   (make-instruction-sequence '(proc) '()
     `((test (op primitive-procedure?) (reg proc)
              (branch (label ,primitive-branch))))
   (parallel-instruction-sequences
    (append-instruction-sequences
     compiled-branch
     (compile-proc-appl target compiled-linkage))
    (append-instruction-sequences
     primitive-branch
     (end-with-linkage linkage
       (make-instruction-sequence '(proc argl)
                                 (list target)
         `((assign ,target
                   (op apply-primitive-procedure)
                   (reg proc)
                   (reg argl)))))))
   after-call)))))


```

The primitive and compound branches, like the true and false branches in `compile-if`, are appended using `parallel-instruction-sequences` rather than the ordinary `append-instruction-sequences`, because they will not be executed sequentially.

`compile-if` の真と偽の分岐のような、プリミティブかつ、複合な分岐は通常の `append-instruction-sequences` ではなく `parallel-instruction-sequences` を用いて接続されます。それらは順には実行されないためです。

## Applying compiled procedures

The code that handles procedure application is the most subtle part of the compiler, even though the instruction sequences it generates are very short. A

compiled procedure (as constructed by `compile-lambda`) has an entry point, which is a label that designates where the code for the procedure starts. The code at this entry point computes a result in `val` and returns by executing the instruction (`goto (reg continue)`). Thus, we might expect the code for a compiled-procedure application (to be generated by `compile-proc-appl`) with a given target and linkage to look like this if the linkage is a label

手続の適用を取り扱うコードはコンパイラの最も微妙な部分です。例えそれが生成する命令列がとても短くても変わりません。`(compile-lambda` により構築されたような) コンパイルされた手続は手続が開始する場所を指定するラベルであるエントリポイントを持ちます。このエントリポイントにてコードは `val` に結果を求め、命令 (`goto (reg continue)`) を実行することにより戻ります。従って与えられたターゲットとリンク記述子を伴なう (`compile-proc-appl` により生成される) コンパイルされた手続の適用はリンク記述子がラベルであれば以下のようになります。

```
(assign continue (label proc-return))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
proc-return
(assign <target> (reg val)) ; ターゲットが val でなければ含まれる
(goto (label <リンク記述子>)) ; リンクコード
```

or like this if the linkage is `return`.

またはリンク記述子が `return` の場合は次のとおりです。

```
(save continue)
(assign continue (label proc-return))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
proc-return
(assign <target> (reg val)) ; ターゲットが val でなければ含まれる
(restore continue)
(goto (reg continue)) ; リンクコード
```

This code sets up `continue` so that the procedure will return to a label `proc-return` and jumps to the procedure's entry point. The code at `proc-return`

transfers the procedure's result from `val` to the target register (if necessary) and then jumps to the location specified by the linkage. (The linkage is always `return` or a label, because `compile-procedure-call` replaces a `next` linkage for the compound-procedure branch by an `after-call` label.)

このコードは手続が `proc-return` に戻るよう `continue` を設定し、手続のエントリポイントへと飛びます。`proc-return` のコードは手続の結果を `val` からターゲットレジスタへと (もし必要なら) 転送し、次にリンク記述子により指定された位置へと飛びます。(リンク記述子は常に `return` かラベルです。なぜなら `compile-procedure-call` が複合手続の分岐のためのリンク記述子 `next` を `after-call` ラベルに置き換えるためです。)

In fact, if the target is not `val`, that is exactly the code our compiler will generate.<sup>69</sup> Usually, however, the target is `val` (the only time the compiler specifies a different register is when targeting the evaluation of an operator to `proc`), so the procedure result is put directly into the target register and there is no need to return to a special location that copies it. Instead, we simplify the code by setting up `continue` so that the procedure will “return” directly to the place specified by the caller's linkage:

実際には、もしターゲットが `val` でなければ、それはまさに私達のコンパイラが生成するコードです。<sup>70</sup> しかし、通常はターゲットは `val` であり (コンパイラが異なるレジスタを指定する唯一の場合はオペレータの評価のターゲットを `proc` にする時です)、そのため手続の結果は直接ターゲットレジスタに入れられ、コピーを行う特別な位置へ戻る必要はありません。その代わりに、手続が直接呼び出し元のリンク記述子により指定される場所へ直接“戻る”ように `continue` を設定します。

```
(continue にリンク記述子を設定)
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

If the linkage is a label, we set up `continue` so that the procedure will return to that label. (That is, the `(goto (reg continue))` the procedure ends with becomes equivalent to the `(goto (label <linkage>))` at `proc-return` above.)

---

<sup>69</sup> Actually, we signal an error when the target is not `val` and the linkage is `return`, since the only place we request `return` linkages is in compiling procedures, and our convention is that procedures return their values in `val`.

<sup>70</sup> 実際に、ターゲットが `val` でなく、リンク記述子が `return` である場合にはエラーを発します。私達がリンク記述子 `return` を要求する箇所は手続のコンパイル内のみです。そして私達の仕様は、手続はその値を `val` にて返す、です。

もしリンク記述子がラベルならば、手続がそのラベルに戻るよう continue を設定します。(つまり、上記の proc-return において手続の終端 (goto (reg continue)) が (goto (label <linkage>)) と等価になります。)

```
(assign continue (label <リンク記述子>))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

If the linkage is **return**, we don't need to set up **continue** at all: It already holds the desired location. (That is, the (goto (reg continue)) the procedure ends with goes directly to the place where the (goto (reg continue)) at proc-return would have gone.)

もしリンク記述子が **return** なら、**continue** を設定する必要は全くありません。それは既に望まれた位置を持っています。(言い換えれば、手続の終端 (goto (reg continue)) は proc-return の (goto (reg continue)) が飛ぶはずだった場所へ直接飛びます。)

```
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
```

With this implementation of the **return** linkage, the compiler generates tail-recursive code. Calling a procedure as the final step in a procedure body does a direct transfer, without saving any information on the stack.

このリンク記述子 **return** の実装を用いて、コンパイラは末尾再帰のコードを生成します。手続のボディの最後ステップとしての手続の呼出は直接移動を行いスタック上にどのような情報も保存しません。

Suppose instead that we had handled the case of a procedure call with a linkage of **return** and a target of **val** as shown above for a non-**val** target. This would destroy tail recursion. Our system would still give the same value for any expression. But each time we called a procedure, we would save **continue** and return after the call to undo the (useless) save. These extra saves would accumulate during a nest of procedure calls.<sup>71</sup>

---

<sup>71</sup>Making a compiler generate tail-recursive code might seem like a straightforward idea. But most compilers for common languages, including C and Pascal, do not do this, and therefore these languages cannot represent iterative processes in terms of procedure call alone. The difficulty with tail recursion in these languages is that their implementations use the stack to store procedure arguments and local variables as well as return addresses. The Scheme implementations described in this book store

その代わりに手続呼出の場合をリンク記述子 `return` とターゲット `val` を用いて、上記で示されたように `val` 以外のターゲットに対しても取り扱ったと仮定します。これは末尾再帰を損うでしょう。それでも、私達のシステムは任意の式に対して同じ値を与えます。しかし、私達が手続を呼ぶ度に、`continue` を保存し、呼出の後に（必要な無い）保存の取消を呼び出すことになります。これらの余分な保存が入れ子の手続呼出の間に蓄積されます。<sup>72</sup>

`Compile-proc-app1` generates the above procedure-application code by considering four cases, depending on whether the target for the call is `val` and whether the linkage is `return`. Observe that the instruction sequences are declared to modify all the registers, since executing the procedure body can change the registers in arbitrary ways.

`compile-proc-app1` は上記の手続適用のコードを生成します。これは呼出のためのターゲットが `val` であるか、そしてリンク記述子が `return` であるかについてに依存する 4 つの場合について考慮します。命令列が全てのレジスタを変更するために宣言されることについて注意して下さい。手続のボ

---

arguments and variables in memory to be garbage-collected. The reason for using the stack for variables and arguments is that it avoids the need for garbage collection in languages that would not otherwise require it, and is generally believed to be more efficient. Sophisticated Lisp compilers can, in fact, use the stack for arguments without destroying tail recursion. (See [Hanson 1990](#) for a description.) There is also some debate about whether stack allocation is actually more efficient than garbage collection in the first place, but the details seem to hinge on fine points of computer architecture. (See [Appel 1987](#) and [Miller and Rozas 1994](#) for opposing views on this issue.)

<sup>72</sup> コンパイラに末尾再帰のコードを生成させることは簡単な考えのように見えるかもしれません。しかし一般的な言語のための多くのコンパイラは C 言語と Pascal を含めて、これを行いません。従ってこれらの言語は反復プロセスを手続呼出のみを用いて表現することができません。これらの言語における末尾再帰の困難さはそれらの実装がスタックを用いて手続の引数とローカル変数と同様にリターンアドレスをも格納しているためです。この本で説明されている Scheme の実装は引数と変数をガベージコレクションされるようにメモリに保存します。変数と引数に対してスタックを使用する理由は他のやり方によりガベージコレクションの必要の無い言語内で、その必要性を回避するからです。そして一般的にはより効率的になると信じられています。実際には、最新の Lisp コンパイラは末尾再帰を無効化せずにスタックを引数のために使用することができます。（このことの説明に関しては [Hanson 1990](#) を参照して下さい）。またスタックの割当がそもそもガベージコレクションより効率的であるかどうかについての討論もいくつか存在します。しかし、詳細はコンピュータアーキテクチャの委細に依存しているように見えます。（この問題の反対の立場からの視点については [Appel 1987](#) と [Miller and Rozas 1994](#) を参照して下さい。）

ディの実行が自由な形でレジスタを変更することができるためです。<sup>73</sup> Also note that the code sequence for the case with target `val` and linkage `return` is declared to need `continue`: Even though `continue` is not explicitly used in the two-instruction sequence, we must be sure that `continue` will have the correct value when we enter the compiled procedure.

またターゲットが `val` であり、リンク記述子が `return` の場合に対するコードの列は `continue` を必要とすると宣言されていることに注意して下さい。例え `continue` が明示的に 2 つの命令列の中で使用されていなくとも、私達がコンパイルされた手続を入力した時に `continue` が正しい値を持つことを確実にしなければなりません。

```
(define (compile-proc-appl target linkage)
  (cond ((and (eq? target 'val) (not (eq? linkage 'return)))
         (make-instruction-sequence '(proc) all-reg
           `((assign continue (label ,linkage))
             (assign val (op compiled-procedure-entry)
                   (reg proc))
             (goto (reg val)))))

        ((and (not (eq? target 'val))
              (not (eq? linkage 'return))))
        (let ((proc-return (make-label 'proc-return)))
          (make-instruction-sequence '(proc) all-reg
            `((assign continue (label ,proc-return))
              (assign val (op compiled-procedure-entry)
                    (reg proc))
              (goto (reg val))
              ,proc-return
              (assign ,target (reg val))
              (goto (label ,linkage)))))

        ((and (eq? target 'val) (eq? linkage 'return)))
        (make-instruction-sequence
          '(proc continue)
```

---

<sup>73</sup>The variable `all-reg`s is bound to the list of names of all the registers:  
変数 `all-reg`s は全てのレジスタの名前のリストに対して束縛されます。

```
(define all-regs '(env proc val arg1 continue))
```

```

all-reg
'((assign val (op compiled-procedure-entry)
  (reg proc)
  (goto (reg val))))))
((and (not (eq? target 'val))
  (eq? linkage 'return))
(error "return linkage, target not val: COMPILE"
  target)))

```

## 5.5.4 Combining Instruction Sequences

This section describes the details on how instruction sequences are represented and combined. Recall from [Section 5.5.1](#) that an instruction sequence is represented as a list of the registers needed, the registers modified, and the actual instructions. We will also consider a label (symbol) to be a degenerate case of an instruction sequence, which doesn't need or modify any registers. So to determine the registers needed and modified by instruction sequences we use the selectors

この節では命令列がどのように表現され、組み合わされるのかについての詳細を説明します。[Section 5.5.1](#)から命令列が必要なレジスタのリスト、変更されるレジスタ、実際の命令のリストとして表現されたことを思い出して下さい。またラベル（シンボル）を命令列の退化した場合だと考慮します。これはどのレジスタも必要とせず、また変更しません。故に、命令列により必要とされる、または変更されるレジスタを決定するために以下のセレクタを用います。

```

(define (registers-needed s)
  (if (symbol? s) '() (car s)))
(define (registers-modified s)
  (if (symbol? s) '() (cadr s)))
(define (statements s)
  (if (symbol? s) (list s) (caddr s)))

```

and to determine whether a given sequence needs or modifies a given register we use the predicates

また与えられた命令列が与えられたレジスタを必要とするか、変更するかを決定するために以下の述語を用います。

```
(define (needs-register? seq reg)
  (memq reg (registers-needed seq)))
(define (modifies-register? seq reg)
  (memq reg (registers-modified seq)))
```

In terms of these predicates and selectors, we can implement the various instruction sequence combiners used throughout the compiler.

これらの述語とセレクタを用いて、コンパイラを通して使用される様々な命令列の結合器 (combiner) を実装することができます。

The basic combiner is `append-instruction-sequences`. This takes as arguments an arbitrary number of instruction sequences that are to be executed sequentially and returns an instruction sequence whose statements are the statements of all the sequences appended together. The subtle point is to determine the registers that are needed and modified by the resulting sequence. It modifies those registers that are modified by any of the sequences; it needs those registers that must be initialized before the first sequence can be run (the registers needed by the first sequence), together with those registers needed by any of the other sequences that are not initialized (modified) by sequences preceding it.

基本的な結合器は `append-instruction-sequences` です。これは引数として順に実行される任意の数の命令列を取り、命令文 (statement) が全ての命令列の命令文を一緒に接続した命令文である命令列を返します。結果の命令列により必要とされる、または変更されるレジスタの決定が繊細な点になります。これは命令列のどれかにより変更されるレジスタが変更されます。またこれは最初の命令列が実行する前に初期化されなければならないレジスタ (最初の命令列で必要とされるレジスタ) に加えて、それに続く命令列により初期化されない (変更されない) 他の命令列により必要とされるレジスタ全てです。

The sequences are appended two at a time by `append-2-sequences`. This takes two instruction sequences `seq1` and `seq2` and returns the instruction sequence whose statements are the statements of `seq1` followed by the statements of `seq2`, whose modified registers are those registers that are modified by either `seq1` or `seq2`, and whose needed registers are the registers needed by `seq1` together with those registers needed by `seq2` that are not modified by `seq1`. (In terms of set operations, the new set of needed registers is the union of the set of registers needed by `seq1` with the set difference of the registers needed by `seq2` and the registers modified by `seq1`.) Thus, `append-instruction-sequences` is implemented as follows:

命令列は `append-2-sequences` により一度に 2 つが接続されます。これは 2 つの命令列 `seq1` と `seq2` を取り、命令文が `seq1` の命令文の後に `seq2` の命令文が置かれる命令列を返します。この変更されたレジスタは `seq1` か `seq2` のどちらかにより変更されたレジスタです。そして必要とされるレジスタは `seq1` により必要とされるレジスタと `seq2` で必要とされ `seq1` で変更されないレジスタを加えたものです。(集合の命令を用いて、必要なレジスタの新しい集合は `seq1` により必要とされるレジスタの集合と、`seq2` により必要とされるレジスタと `seq1` により変更されたレジスタの差集合との、和集合です。)

```
(define (append-instruction-sequences . seqs)
  (define (append-2-sequences seq1 seq2)
    (make-instruction-sequence
      (list-union
        (registers-needed seq1)
        (list-difference (registers-needed seq2)
          (registers-modified seq1))))
    (list-union (registers-modified seq1)
      (registers-modified seq2)))
  (append (statements seq1) (statements seq2))))
(define (append-seq-list seqs)
  (if (null? seqs)
    (empty-instruction-sequence)
    (append-2-sequences
      (car seqs)
      (append-seq-list (cdr seqs))))))
(append-seq-list seqs))
```

This procedure uses some simple operations for manipulating sets represented as lists, similar to the (unordered) set representation described in [Section 2.3.3](#):

この手続はリストとして表現された集合を操作するためのいくつかの簡単な命令を使います。[Section 2.3.3](#)で説明された(順序無し)集合表現と同様です。

```
(define (list-union s1 s2)
  (cond ((null? s1) s2)
        ((memq (car s1) s2) (list-union (cdr s1) s2))
        (else (cons (car s1) (list-union (cdr s1) s2)))))
(define (list-difference s1 s2)
  (cond ((null? s1) '())
        ((memq (car s2) s1) (list-difference (cdr s2) s1))
        (else (cons (car s2) (list-difference s1 (cdr s2))))))
```

```
((memq (car s1) s2) (list-difference (cdr s1) s2))
(else (cons (car s1)
             (list-difference (cdr s1) s2)))))
```

`Preserving`, the second major instruction sequence combiner, takes a list of registers `regs` and two instruction sequences `seq1` and `seq2` that are to be executed sequentially. It returns an instruction sequence whose statements are the statements of `seq1` followed by the statements of `seq2`, with appropriate `save` and `restore` instructions around `seq1` to protect the registers in `regs` that are modified by `seq1` but needed by `seq2`. To accomplish this, `preserving` first creates a sequence that has the required `saves` followed by the statements of `seq1` followed by the required `restores`. This sequence needs the registers being saved and restored in addition to the registers needed by `seq1`, and modifies the registers modified by `seq1` except for the ones being saved and restored. This augmented sequence and `seq2` are then appended in the usual way. The following procedure implements this strategy recursively, walking down the list of registers to be preserved:<sup>74</sup>

`preserving` は 2 つ目の主な命令列結合器ですが、レジスタのリスト `regs` と順に実行する 2 つの命令列 `seq1` と `seq2` を取ります。これは `seq1` の命令文 (statements) のその後に `seq2` の命令文が続く命令文を持つ命令列を返します。この命令文には `seq1` により変更されるが `seq2` で必要とされる `regs` 内のレジスタを守るために `seq1` の周りに適切な `save` と `restore` が追加されます。これを達成するために、`preserving` は最初に必要とされる `save` とそれに続く `seq1`、それに続く必要とされる `restore` を持つ命令列を作ります。この命令列は `seq1` により必要とされるレジスタに加えてレジスタの保存と復元を必要とします。そして `seq1` で変更されたレジスタを保存と回復が行われるものを探して変更します。次に、この増補された命令列と `seq2` が通常の方法で接続されます。以下の手続はこの戦略を、維持されるべきレジスタのリストを横断しながら再帰的に実装します。<sup>75</sup>

```
(define (preserving regs seq1 seq2)
```

---

<sup>74</sup>Note that `preserving` calls `append` with three arguments. Though the definition of `append` shown in this book accepts only two arguments, Scheme standardly provides an `append` procedure that takes an arbitrary number of arguments.

<sup>75</sup>`preserving` が `append` を 3 つの引数と共に呼び出すことに注意して下さい。この本に表われる `append` の定義は 2 つの引数しか受け付けませんが、Scheme の標準は任意の数の引数を取る `append` 手續を提供します。

```

(if (null? regs)
  (append-instruction-sequences seq1 seq2)
  (let ((first-reg (car regs)))
    (if (and (needs-register? seq2 first-reg)
              (modifies-register? seq1 first-reg))
        (preserving (cdr regs)
          (make-instruction-sequence
            (list-union (list first-reg)
              (registers-needed seq1))
            (list-difference (registers-modified seq1)
              (list first-reg)))
          (append `((save ,first-reg))
            (statements seq1)
            `((restore ,first-reg))))
        seq2)
      (preserving (cdr regs) seq1 seq2))))))

```

Another sequence combiner, `tack-on-instruction-sequence`, is used by `compile-lambda` to append a procedure body to another sequence. Because the procedure body is not “in line” to be executed as part of the combined sequence, its register use has no impact on the register use of the sequence in which it is embedded. We thus ignore the procedure body’s sets of needed and modified registers when we tack it onto the other sequence.

別の命令列結合器である `tack-on-instruction-sequence` は `compile-lambda` により手続のボディを他の命令列に接続するために使用されます。手続のボディは組み合わされた列の一部として実行されるための“インライン”形式ではないため、それによるレジスタの使用はそれが組込まれる命令列のレジスタ使用に影響を与えません。従って手続ボディの必要な、また変更されるレジスタの集合は別の命令列に接続する時に無視されます。

```

(define (tack-on-instruction-sequence seq body-seq)
  (make-instruction-sequence
    (registers-needed seq)
    (registers-modified seq)
    (append (statements seq)
      (statements body-seq))))

```

`Compile-if` and `compile-procedure-call` use a special combiner called `parallel-instruction-sequences` to append the two alternative branches that follow a test. The two branches will never be executed sequentially; for any particular evaluation of the test, one branch or the other will be entered. Because of this, the registers needed by the second branch are still needed by the combined sequence, even if these are modified by the first branch.

`compile-if` と `compile-procedure-call` は `parallel-instruction-sequences` と呼ばれる特別な結合器を使用してテストに続く二者択一の分岐を接続します。2つの分岐は絶対に順には実行されません。どんなテストの評価に対しても、一方か、別の一方に入ります。このため、2つ目の分岐により必要とされるレジスタは例えもしこれらが1つ目の分岐により変更されようとも依然として結合後の命令列でも必要とします。

```
(define (parallel-instruction-sequences seq1 seq2)
  (make-instruction-sequence
    (list-union (registers-needed seq1)
                (registers-needed seq2))
    (list-union (registers-modified seq1)
                (registers-modified seq2)))
  (append (statements seq1)
          (statements seq2))))
```

### 5.5.5 An Example of Compiled Code

Now that we have seen all the elements of the compiler, let us examine an example of compiled code to see how things fit together. We will compile the definition of a recursive `factorial` procedure by calling `compile`:

これでコンパイラの全ての要素について学び終えました。ここまでのがどのように御互いに組合せられるのかを見るためにコンパイル済みのコードの例を試してみましょう。再帰 `factorial` 手続の定義を `compile` を呼ぶことでコンパイルしてみます。

```
(compile
  '(define (factorial n)
     (if (= n 1)
         1
         (* (factorial (- n 1)) n))))
```

```
'val  
'next)
```

We have specified that the value of the `define` expression should be placed in the `val` register. We don't care what the compiled code does after executing the `define`, so our choice of `next` as the linkage descriptor is arbitrary.

`define` 式の値はレジスタ `val` に配置されなければならないと指定しました。私達は `define` を実行した後にコンパイル済みコードが何を行うのか気にはしません。そのためリンク記述子に対する `next` の選択は気まぐれです。

`Compile` determines that the expression is a definition, so it calls `compile-definition` to compile code to compute the value to be assigned (targeted to `val`), followed by code to install the definition, followed by code to put the value of the `define` (which is the symbol `ok`) into the target register, followed finally by the linkage code. `Env` is preserved around the computation of the value, because it is needed in order to install the definition. Because the linkage is `next`, there is no linkage code in this case. The skeleton of the compiled code is thus

`compile` は式が定義であるかを判断します。そのため `compile-definition` を呼び出し (ターゲット `val` に対し) 割り当てられるべき値を求めるコードをコンパイルします。続いて定義を導入するコード、さらに `define` の値 (シンボル `ok`) をターゲットレジスタに入れるコード、最後にリンクコードが続きます。`env` は値の演算の周りで維持されます。定義の導入のために必要とされるためです。今回のリンク記述子は `next` ですから、リンクコードは存在しません。従ってコンパイルされたコードの骨格は以下のようになります。

```
<値を求めるコードで変更されるなら env を保存>  
<定義値、ターゲット val、リンク記述子 next のコンパイル>  
<上で保存したなら env の復元>  
(perform (op define-variable!)  
         (const factorial)  
         (reg val)  
         (reg env))  
(assign val (const ok))
```

The expression that is to be compiled to produce the value for the variable `factorial` is a `lambda` expression whose value is the procedure that computes factorials. `Compile` handles this by calling `compile-lambda`, which compiles the procedure body, labels it as a new entry point, and generates the instruction

that will combine the procedure body at the new entry point with the run-time environment and assign the result to `val`. The sequence then skips around the compiled procedure code, which is inserted at this point. The procedure code itself begins by extending the procedure's definition environment by a frame that binds the formal parameter `n` to the procedure argument. Then comes the actual procedure body. Since this code for the value of the variable doesn't modify the `env` register, the optional `save` and `restore` shown above aren't generated. (The procedure code at `entry2` isn't executed at this point, so its use of `env` is irrelevant.) Therefore, the skeleton for the compiled code becomes

変数 `factorial` に対する値を生成するためにコンパイルされる式は、値が階乗を計算する手続である `lambda` 式です。`compile` は `compile-lambda` を呼ぶことによりこれを扱います。`compile-lambda` は手続のボディをコンパイルし、それに新しいエントリポイントとしてラベル付けを行い、新しいエントリポイントの手続ボディを実行時環境と組み合わせ、結果を `val` に割り当てるコードを生成します。次に命令列はこの時点で挿入された、このコンパイルされたコードをスキップします。手続のコードそれ自体は手続定義環境を形式パラメタ `n` を手続の引数に束縛するフレームにより拡張することから始めます。その次に実際の手続のボディが来ます。変数の値のためのこのコードは `env` レジスタを変更しませんので、上で示された任意の `save` と `restore` は生成されません。`(entry2` における手続のコードはこの時点では実行されません。そのため、その `env` の使用は無関係です)。従って、コンパイルされたコードの骨組は以下のようになります。

```
(assign val
      (op make-compiled-procedure)
      (label entry2)
      (reg env))
(goto (label after-lambda1))
entry2
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env
          (op extend-environment)
          (const (n))
          (reg arg1)
          (reg env))
  <手続ボディのコンパイル>
after-lambda1
```

```
(perform (op define-variable!)
         (const factorial)
         (reg val)
         (reg env))
(assign val (const ok))
```

A procedure body is always compiled (by `compile-lambda-body`) as a sequence with target `val` and linkage `return`. The sequence in this case consists of a single `if` expression:

手続のボディは常に (`compile-lambda-body` により)、ターゲット `val` とリンク記述子 `return` を用いる命令列としてコンパイルされます。今回の場合の命令列は単一の `if` 式から成り立ちます。

```
(if (= n 1)
    1
    (* (factorial (- n 1)) n))
```

`Compile-if` generates code that first computes the predicate (targeted to `val`), then checks the result and branches around the true branch if the predicate is false. `Env` and `continue` are preserved around the predicate code, since they may be needed for the rest of the `if` expression. Since the `if` expression is the final expression (and only expression) in the sequence making up the procedure body, its target is `val` and its linkage is `return`, so the true and false branches are both compiled with target `val` and linkage `return`. (That is, the value of the conditional, which is the value computed by either of its branches, is the value of the procedure.)

`compile-if` は最初に述語を演算し (ターゲットは `val`)、次にその結果を確認して述語が偽であれば真の分岐を回避します。`env` と `continue` が述語のコードの周りで維持されます。それらが `if` 式の残りの部分で必要となる可能性があるためです。`if` 式が手続のボディを構成する命令列内の最後の式であるため (そしてただ 1 つの式であるため)、そのターゲットは `val` で、リンク記述子は `return` になります。そのため真と偽の両方の分岐がターゲット `val` とリンク記述子 `return` と共にコンパイルされます。(言い換えれば、どちらかの分岐により値が演算される条件文の値がその手続の値です。)

〈述語により変更され、分岐により必要とされるなら `continue`, `env` を保存する〉

〈述語、ターゲット `val`、リンク記述子 `next` のコンパイル〉

```

<上で保存したなら continue, env を復元する>
(test (op false?) (reg val))
(branch (label false-branch4))

true-branch5
<真の分岐, ターゲット val, リンク記述子 return のコンパイル>
false-branch4
<偽の分岐, ターゲット val, リンク記述子 return のコンパイル>
after-if3

```

The predicate (`= n 1`) is a procedure call. This looks up the operator (the symbol `=`) and places this value in `proc`. It then assembles the arguments `1` and the value of `n` into `arg1`. Then it tests whether `proc` contains a primitive or a compound procedure, and dispatches to a primitive branch or a compound branch accordingly. Both branches resume at the `after-call` label. The requirements to preserve registers around the evaluation of the operator and operands don't result in any saving of registers, because in this case those evaluations don't modify the registers in question.

述語 (`= n 1`) は手続の呼出です。これはオペレータ (シンボル `=`) を探し、その値を `proc` 内に配置します。次に引数 `1` と変数 `n` を `arg1` に集めます。そして `proc` がプリミティブ、または複合手続を含むかどうかをテストし、それに応じてプリミティブの分岐か複合の分岐へ飛びます。両方の分岐がラベル `after-call` にて再開します。オペレータとオペランドの評価の周りでレジスタを維持する必要性はどのレジスタも保存することにはなりません。今回の場合はそれらの評価は問題となるレジスタを変更しないためです。

```

(assign proc
        (op lookup-variable-value) (const =) (reg env))
(assign val (const 1))
(assign arg1 (op list) (reg val))
(assign val
        (op lookup-variable-value) (const n) (reg env))
(assign argl (op cons) (reg val) (reg arg1))
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch17))

compiled-branch16
(assign continue (label after-call15))
(assign val (op compiled-procedure-entry) (reg proc))

```

```

(goto (reg val))
primitive-branch17
(assign val
       (op apply-primitive-procedure)
       (reg proc)
       (reg argl))
after-call15

```

The true branch, which is the constant 1, compiles (with target `val` and linkage `return`) to

真の分岐は定数 1 ですが、(ターゲット `val` とリンク記述子 `return` と共に) 以下のようにコンパイルされます。

```

(assign val (const 1))
(goto (reg continue))

```

The code for the false branch is another procedure call, where the procedure is the value of the symbol `*`, and the arguments are `n` and the result of another procedure call (a call to `factorial`). Each of these calls sets up `proc` and `argl` and its own primitive and compound branches. Figure 5.17 shows the complete compilation of the definition of the `factorial` procedure. Notice that the possible `save` and `restore` of `continue` and `env` around the predicate, shown above, are in fact generated, because these registers are modified by the procedure call in the predicate and needed for the procedure call and the `return` linkage in the branches.

偽の分岐のコードは別の手続呼出です。手続はシンボル `*` で、その引数は `n` と別の手続呼出の結果 (`factorial` の呼出) です。これらの呼出の全てが `proc` と `argl`、それ自身のプリミティブと複合の分岐の準備を行います。Figure 5.17 は手続 `factorial` の定義の完全なコンパイルを示します。述語の周りで可能性のある `continue` と `env` の `save` と `restore` が実際に生成されていることに注意して下さい。これらのレジスタが述語内の手続呼出にて変更され、また分岐内の手続呼び出しと `return` のリンクコードにより必要とされるためです。

**Exercise 5.33:** Consider the following definition of a factorial procedure, which is slightly different from the one given above:

上で与えられたものとは微妙に異なる以下の階乗手続の定義について考えよ。

```
(define (factorial-alt n)
```

```
(if (= n 1)
    1
    (* n (factorial-alt (- n 1)))))
```

Compile this procedure and compare the resulting code with that produced for `factorial`. Explain any differences you find. Does either program execute more efficiently than the other?

この手続をコンパイルし結果のコードを `factorial` に対して生成されたコードと比べよ。見つけた全ての違いについて説明せよ。どちらのプログラムが他方よりもより効率的に実行するだろうか?

**Exercise 5.34:** Compile the iterative factorial procedure  
反復階乗手続をコンパイルせよ

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Annotate the resulting code, showing the essential difference between the code for iterative and recursive versions of `factorial` that makes one process build up stack space and the other run in constant stack space.

結果のコードに注釈を付け、一方のプロセスがスタック領域を増進させ、他方が一定のスタック領域で実行される元となる、`factorial` の反復版と再帰版のコードの間の本質的な違いを示せ。

**Figure 5.17:** ↓ `factorial` 手続定義のコンパイル結果

```
; 手続を構築し、手続のボディのコードを飛ばす
(assign val
  (op make-compiled-procedure)
  (label entry2)
  (reg env))
(goto (label after-lambda1))
entry2      ; factorial の呼出はここから入る
(assign env (op compiled-procedure-env) (reg proc))
```

```

(assign env
        (op extend-environment)
        (const (n))
        (reg argl)
        (reg env))
;; 実際の手続きのボディを開始する
(save continue)
(save env)
;; (= n 1) を求める
(assign proc
        (op lookup-variable-value)
        (const =)
        (reg env))
(assign val (const 1))
(assign argl (op list) (reg val))
(assign val
        (op lookup-variable-value)
        (const n)
        (reg env))
(assign argl (op cons) (reg val) (reg argl))
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch17))
compiled-branch16
(assign continue (label after-call15))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
primitive-branch17
(assign val
        (op apply-primitive-procedure)
        (reg proc)
        (reg argl))
after-call15 ; ここで val は (= n 1) の結果を持つ
	restore env
	restore continue
(test (op false?) (reg val))
(branch (label false-branch4))
true-branch5 ; return 1
(assign val (const 1))
(goto (reg continue))
false-branch4
;; (* (factorial (- n 1)) n) を求めて返す
(assign proc

```

```

        (op lookup-variable-value)
        (const *)
        (reg env))
(save continue)
(save proc)    ;* 手続を保存する
(assign val
        (op lookup-variable-value)
        (const n)
        (reg env))
(assign argl (op list) (reg val))
(save argl)   ;* の引数リストの一部を保存
;; (factorial (- n 1)) を求める。これは*のもう一方の引数
(assign proc
        (op lookup-variable-value)
        (const factorial)
        (reg env))
(save proc)  ; factorial 手続を保存
;; (- n 1) を求める。これは factorial に対する引数
(assign proc
        (op lookup-variable-value)
        (const -)
        (reg env))
(assign val (const 1))
(assign argl (op list) (reg val))
(assign val
        (op lookup-variable-value)
        (const n)
        (reg env))
(assign argl (op cons) (reg val) (reg argl))
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch8))
compiled-branch7
(assign continue (label after-call16))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
primitive-branch8
(assign val
        (op apply-primitive-procedure)
        (reg proc)
        (reg argl))
after-call16  ; ここで val は (- n 1) の結果を持つ
(assign argl (op list) (reg val))

```

```

    (restore proc) ; factorial に戻す
;; factorial の適用
    (test (op primitive-procedure?) (reg proc))
    (branch (label primitive-branch11))
compiled-branch10
    (assign continue (label after-call19))
    (assign val (op compiled-procedure-entry) (reg proc))
    (goto (reg val))
primitive-branch11
    (assign val
        (op apply-primitive-procedure)
        (reg proc)
        (reg argl))
after-call19      ; ここで val は (factorial (- n 1)) の結果を持つ
    (restore argl) ; *の引数リストの一部を復元
    (assign argl (op cons) (reg val) (reg argl))
    (restore proc) ; *に戻す
    (restore continue)
;; *を適用しその値を返す
    (test (op primitive-procedure?) (reg proc))
    (branch (label primitive-branch14))
compiled-branch13
;; この複合手続は末尾再帰で呼ばれることに注意すること
    (assign val (op compiled-procedure-entry) (reg proc))
    (goto (reg val))
primitive-branch14
    (assign val
        (op apply-primitive-procedure)
        (reg proc)
        (reg argl))
    (goto (reg continue))
after-call12
after-if3
after-lambda1
;; 手続を変数 factorial に割り当てる
    (perform (op define-variable!)
        (const factorial)
        (reg val)
        (reg env))
    (assign val (const ok))

```

**Exercise 5.35:** What expression was compiled to produce the code

shown in Figure 5.18?

どの式がコンパイルされるとFigure 5.18に示されるコードを生成するか?

Figure 5.18: ↓ コンパイラ出力の例。Exercise 5.35参照

```
(assign val
      (op make-compiled-procedure)
      (label entry16)
      (reg env))
  (goto (label after-lambda15))
entry16
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env
    (op extend-environment)
    (const (x))
    (reg argl)
    (reg env)))
  (assign proc
    (op lookup-variable-value)
    (const +)
    (reg env))
  (save continue)
  (save proc)
  (save env)
  (assign proc
    (op lookup-variable-value)
    (const g)
    (reg env)))
  (save proc)
  (assign proc
    (op lookup-variable-value)
    (const +)
    (reg env)))
  (assign val (const 2))
  (assign argl (op list) (reg val))
  (assign val
    (op lookup-variable-value)
    (const x)
    (reg env)))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
```

```

  (branch (label primitive-branch19))
compiled-branch18
  (assign continue (label after-call17))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch19
  (assign val
    (op apply-primitive-procedure)
    (reg proc)
    (reg argl))
after-call17
  (assign argl (op list) (reg val))
  (restore proc)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch22))
compiled-branch21
  (assign continue (label after-call20))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch22
  (assign val
    (op apply-primitive-procedure)
    (reg proc)
    (reg argl))
after-call20
  (assign argl (op list) (reg val))
  (restore env)
  (assign val
    (op lookup-variable-value)
    (const x)
    (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)
  (restore continue)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch25))
compiled-branch24
  (assign val
    (op compiled-procedure-entry)
    (reg proc))
  (goto (reg val))
primitive-branch25

```

```

(assign val
        (op apply-primitive-procedure)
        (reg proc)
        (reg argl))
(goto (reg continue))
after-call23
after-lambda15
(perform (op define-variable!)
          (const f)
          (reg val)
          (reg env))
(assign val (const ok))

```

**Exercise 5.36:** What order of evaluation does our compiler produce for operands of a combination? Is it left-to-right, right-to-left, or some other order? Where in the compiler is this order determined? Modify the compiler so that it produces some other order of evaluation. (See the discussion of order of evaluation for the explicit-control evaluator in [Section 5.4.1](#).) How does changing the order of operand evaluation affect the efficiency of the code that constructs the argument list?

私達のコンパイラが生成する組み合わせのオペランドに対する評価の順はどれか? 左から右であるか、右から左であるか、または何らかの他の順であるか? コンパイラの中のどこがこの順を決定するか? コンパイラを変更し、それが何らかの別の評価順を生成するようにせよ。[\(Section 5.4.1における明示的制御評価機の評価順の議論を参考にせよ\)](#)。オペランドの評価順を変更することが引数リストを構築するコードの効率にどのような影響があるか?

**Exercise 5.37:** One way to understand the compiler's **preserving** mechanism for optimizing stack usage is to see what extra operations would be generated if we did not use this idea. Modify **preserving** so that it always generates the **save** and **restore** operations. Compile some simple expressions and identify the unnecessary stack operations that are generated. Compare the code to that generated with the **preserving** mechanism intact.

スタック使用の最適化のためのコンパイラの **preserving** の仕組みを理解する 1 つの方法はこの考えを用いなかった場合にどんな

余分な命令が生成されるかを見てみることだ。`preserving` を変更し、常に `save` と `restore` の命令を生成するようにせよ。いくつかの簡単な式をコンパイルし、生成された不必要的スタック命令を確認せよ。`preserving` の仕組みが失われていないものから生成されたコードと比較せよ。

**Exercise 5.38:** Our compiler is clever about avoiding unnecessary stack operations, but it is not clever at all when it comes to compiling calls to the primitive procedures of the language in terms of the primitive operations supplied by the machine. For example, consider how much code is compiled to compute  $(+ a 1)$ : The code sets up an argument list in `arg1`, puts the primitive addition procedure (which it finds by looking up the symbol `+` in the environment) into `proc`, and tests whether the procedure is primitive or compound. The compiler always generates code to perform the test, as well as code for primitive and compound branches (only one of which will be executed). We have not shown the part of the controller that implements primitives, but we presume that these instructions make use of primitive arithmetic operations in the machine's data paths. Consider how much less code would be generated if the compiler could *open-code* primitives—that is, if it could generate code to directly use these primitive machine operations. The expression  $(+ a 1)$  might be compiled into something as simple as <sup>76</sup>

私達のコンパイラは不必要的スタック命令を防ぐことに関して賢いものだ。しかし、機械により提供されるプリミティブな命令を用いて言語のプリミティブな手続の呼出をコンパイルすることに関しては全く賢くない。例えば、 $(+ a 1)$  を求めるためにどれだけのコードがコンパイルされるか考えてみる。このコードは引数リストを `arg1` に準備し、(環境内でシンボル `+` を探すことにより見つけた) プリミティブな加算手続を `proc` に入れる。そしてこの手続がプリミティブであるか複合であるかをテストする。コンパイラは常にこのテストを実行するコードと、同様にプリミティブと

---

<sup>76</sup>We have used the same symbol `+` here to denote both the source-language procedure and the machine operation. In general there will not be a one-to-one correspondence between primitives of the source language and primitives of the machine.

複合の分岐のためのコード（内、一方のみが実行される）が生成される。私達はコントローラのプリミティブを実装する部品を示さなかった。しかし、これらの命令が機械のデータパス内のプリミティブな数値演算命令を利用することは仮定した。もしコンパイラがプリミティブを *open-code* できたらどれだけ少ないコードが生成されたか考えよ。これはつまり、もしこれらのプリミティブな機械語命令を直接使用するコードを生成することができれば、である。式  $(+ \ a \ 1)$  は以下と同じくらい単純なものにコンパイルされるだろう。<sup>77</sup>

```
(assign
  val (op lookup-variable-value) (const a) (reg env))
(assign val (op +) (reg val) (const 1))
```

In this exercise we will extend our compiler to support open coding of selected primitives. Special-purpose code will be generated for calls to these primitive procedures instead of the general procedure-application code. In order to support this, we will augment our machine with special argument registers `arg1` and `arg2`. The primitive arithmetic operations of the machine will take their inputs from `arg1` and `arg2`. The results may be put into `val`, `arg1`, or `arg2`.

この課題では私達のコンパイラを拡張し、選択されたプリミティブの *open-code* をサポートする。特別な目的のコードがこれらのプリミティブな手続の呼出に対し、一般的な手続適用のコードの代わりに生成される。これをサポートするためには、私達の機械に特別な引数レジスタ、`arg1` と `arg2` を追加する。機械のプリミティブな数値演算子は入力を `arg1` と `arg2` から得る。その結果は `val`, `arg1`, `arg2` のどれかに入れて良い。

The compiler must be able to recognize the application of an open-coded primitive in the source program. We will augment the dispatch in the `compile` procedure to recognize the names of these primitives in addition to the reserved words (the special forms) it currently recognizes.<sup>78</sup> For each special form our compiler has a

---

<sup>77</sup>私達は同じシンボル `+` をソース言語の手続と機械語命令の両方を示すためにここで使用しました。一般的に、ソース言語のプリミティブと機械のプリミティブの間に 1 対 1 の対応はありません。

<sup>78</sup>Making the primitives into reserved words is in general a bad idea, since a user

code generator. In this exercise we will construct a family of code generators for the open-coded primitives.

コンパイラはソースプログラム内の open-code なプリミティブの適用を認識できなければならない。compile 手続に割り振りを追加し、現在認識可能な予約語(特殊形式)に加えてこれらのプリミティブの名前を認識できるようにする。<sup>79</sup> 特殊形式のそれぞれに対してコンパイラはコード生成器を持つ。この課題では open-code なプリミティブのためのコード生成器の仲間を構築する。

- a The open-coded primitives, unlike the special forms, all need their operands evaluated. Write a code generator `spread-arguments` for use by all the open-coding code generators. `Spread-arguments` should take an operand list and compile the given operands targeted to successive argument registers. Note that an operand may contain a call to an open-coded primitive, so argument registers will have to be preserved during operand evaluation.

open-code なプリミティブ全ては特殊形式とは異なり、オペランドが評価されることを必要とする。全ての open-code のコード生成器から使用されるコード生成器 `spread-arguments` を書け。`spread-arguments` はオペランドのリストを取り、与えられたオペランドを次に続く引数レジスタをターゲットにコンパイルしなければならない。オペランドが open-code なプリミティブへの呼出を含んでも良いことに注意すること。そのため引数レジスタはオペランド評価の間に維持されなければならない。

- b For each of the primitive procedures `=`, `*`, `-`, and `+`, write a code generator that takes a combination with that operator,

---

cannot then rebind these names to different procedures. Moreover, if we add reserved words to a compiler that is in use, existing programs that define procedures with these names will stop working. See [Exercise 5.44](#) for ideas on how to avoid this problem.

<sup>79</sup> プリミティブを予約語に入れることは一般的には悪い考えです。そうするとユーザがこれらの名前を異なる手続に束縛し直すことができなくなるためです。さらに、もし使用中のコンパイラに予約語を追加すると、これらの名前で手續を定義した既存のプログラムが動作しなくなります。この問題をどのように回避するかの見解については [Exercise 5.44](#) を参照して下さい。

together with a target and a linkage descriptor, and produces code to spread the arguments into the registers and then perform the operation targeted to the given target with the given linkage. You need only handle expressions with two operands. Make `compile` dispatch to these code generators.

プリミティブな手続 `=`, `*`, `-`, `+` のそれぞれに対してそのオペレータとターゲット、リンク記述子の組み合わせを取り引数をレジスタに入れ、与えられたターゲットをターゲットに取り、与えられたリンク記述子と共に命令を実行するコードを生成するコード生成器を書け。2つのオペラントを扱う式を扱うのみで良い。これらのコード生成器に対する割り振りを作成せよ。

- c Try your new compiler on the `factorial` example. Compare the resulting code with the result produced without open coding.

貴方の新しいコンパイラを階乗の例を用いて試してみよ。結果のコードを open-code 無しで生成した結果と比較せよ。

- d Extend your code generators for `+` and `*` so that they can handle expressions with arbitrary numbers of operands. An expression with more than two operands will have to be compiled into a sequence of operations, each with only two inputs.

`+` と `*` のコード生成器を拡張し任意の数のオペラントを持つ式を取り扱えるようにせよ。3つ以上のオペラントを持つ式は、それぞれが2つだけ入力を持つ命令の列にコンパイルしなければならない。

### 5.5.6 Lexical Addressing

One of the most common optimizations performed by compilers is the optimization of variable lookup. Our compiler, as we have implemented it so far, generates code that uses the `lookup-variable-value` operation of the evaluator machine. This searches for a variable by comparing it with each variable that is currently bound, working frame by frame outward through the run-time environment. This search can be expensive if the frames are deeply nested or if there are many variables. For example, consider the problem of looking up

the value of `x` while evaluating the expression `(* x y z)` in an application of the procedure that is returned by

コンパイラにより実行される最も一般的な最適化の1つは変数検索の最適化です。ここまで実装した私達のコンパイラは評価機の `lookup-variable-value` 命令を用いるコードを生成します。これは実行時環境を通してフレーム毎に取り組みながら、変数を現在束縛されている全ての変数と比較することで変数の検索を行う。この検索はもしフレームが深く入れ子になったり、変数の数が多い場合には高コストに成ります。例えば以下の式を評価した結果の適用において、式 `(* x y z)` の評価の間に `x` の値を探す問題について考えましょう。

```
(let ((x 3) (y 4))
  (lambda (a b c d e)
    (let ((y (* a b x)) (z (+ c d x)))
      (* x y z))))
```

Since a `let` expression is just syntactic sugar for a `lambda` combination, this expression is equivalent to

`let` 式は `lambda` の組み合わせのための単なる構文糖ですので、この式は以下と等価です。

```
((lambda (x y)
  (lambda (a b c d e)
    ((lambda (y z) (* x y z))
     (* a b x)
     (+ c d x))))
```

3  
4)

Each time `lookup-variable-value` searches for `x`, it must determine that the symbol `x` is not `eq?` to `y` or `z` (in the first frame), nor to `a`, `b`, `c`, `d`, or `e` (in the second frame). We will assume, for the moment, that our programs do not use `define`—that variables are bound only with `lambda`. Because our language is lexically scoped, the run-time environment for any expression will have a structure that parallels the lexical structure of the program in which the expression appears.<sup>80</sup> Thus, the compiler can know, when it analyzes the

---

<sup>80</sup>This is not true if we allow internal definitions, unless we scan them out. See Exercise 5.43.

above expression, that each time the procedure is applied the variable `x` in `(* x y z)` will be found two frames out from the current frame and will be the first variable in that frame.

`lookup-variable-value` が `x` を検索する度に、シンボル `x` は `y`、または `z` に `eq?` でないことを（最初のフレームで）確認しなければなりません。また（2つ目のフレームにて）`a, b, c, d, e` についても同様に必要です。差し当たり、私達のプログラムは `define` を使用しないと仮定します。つまり変数は `lambda` の使用にのみ束縛されます。私達の言語はレキシカルスコープであるため、任意の式のための実行時環境は式が現れるプログラムのレキシカルな（語彙的な）構造を並列化する構造を持ちます。<sup>81</sup> 従って、コンパイラは上の式を分析した時に、手続が適用される度に `(* x y z)` 内の変数 `x` が現在のフレームから 2 つ外のフレームの最初の変数として見つかる 것을 알 수 있습니다.

We can exploit this fact by inventing a new kind of variable-lookup operation, `lexical-address-lookup`, that takes as arguments an environment and a *lexical address* that consists of two numbers: a *frame number*, which specifies how many frames to pass over, and a *displacement number*, which specifies how many variables to pass over in that frame. `Lexical-address-lookup` will produce the value of the variable stored at that lexical address relative to the current environment. If we add the `lexical-address-lookup` operation to our machine, we can make the compiler generate code that references variables using this operation, rather than `lookup-variable-value`. Similarly, our compiled code can use a new `lexical-address-set!` operation instead of `set-variable-value!`.

私達は新しい種類の変数検索命令、`lexical-address-lookup` を発明することにより、この事実を利用することができます。この命令は引数として環境と 2 つの数値から成る *lexical address*（レキシカルアドレス）を取ります。2 つの数値は、いくつのフレームを見送るかを指定する *frame number* とそのフレーム内でいくつの変数を見送るかを指定する *displacement number* です。`lexical-address-lookup` は現在のフレームに対して相対的なレキシカルアドレスに格納された変数の値を生成します。もし私達の機械に `lexical-address-lookup` 命令を追加したなら、コンパイラに対して `lookup-variable-value` ではなく、この命令を使用して変数を参照するコードを生成させることができます。同様に、コンパイルされたコードは `set-variable-value!` の代わりに新しい `lexical-address-set!` 命令を使用することができます。

---

<sup>81</sup> これはもし内部定義を許可するのであれば、それら全てを走査しない限りは正しくありません。Exercise 5.43を参照して下さい。

In order to generate such code, the compiler must be able to determine the lexical address of a variable it is about to compile a reference to. The lexical address of a variable in a program depends on where one is in the code. For example, in the following program, the address of `x` in expression `(e1)` is  $(2, 0)$ —two frames back and the first variable in the frame. At that point `y` is at address  $(0, 0)$  and `c` is at address  $(1, 2)$ . In expression `(e2)`, `x` is at  $(1, 0)$ , `y` is at  $(1, 1)$ , and `c` is at  $(0, 2)$ .

そのようなコードを生成するためには、コンパイラは参照をコンパイルしようとする変数のレキシカルアドレスを決定できなければなりません。プログラム中の変数のレキシカルアドレスはそれがコードのどこにあるのかに依存します。例えば、以下のプログラムでは式 `(e1)` のアドレスは  $(2, 0)$  です。つまり、2 フレーム後ろでそのフレームの最初の変数です。同じ地点で `y` はアドレス  $(0, 0)$  であり、`c` はアドレス  $(1, 2)$  に存在します。式 `(e2)` においては、`x` は  $(1, 0)$  に、`y` は  $(1, 1)$  に、`c` は  $(0, 2)$  に存在します。

```
((lambda (x y)
  (lambda (a b c d e)
    ((lambda (y z) (e1))
     (e2)
     (+ c d x))))  
3  
4)
```

One way for the compiler to produce code that uses lexical addressing is to maintain a data structure called a *compile-time environment*. This keeps track of which variables will be at which positions in which frames in the run-time environment when a particular variable-access operation is executed. The compile-time environment is a list of frames, each containing a list of variables. (There will of course be no values bound to the variables, since values are not computed at compile time.) The compile-time environment becomes an additional argument to `compile` and is passed along to each code generator. The top-level call to `compile` uses an empty compile-time environment. When a `lambda` body is compiled, `compile-lambda-body` extends the compile-time environment by a frame containing the procedure's parameters, so that the sequence making up the body is compiled with that extended environment. At each point in the compilation, `compile-variable` and `compile-assignment` use the compile-time environment in order to generate the appropriate lexical

addresses.

コンパイラにとってレキシカルアドレスを使用する 1 つの方法は *compile-time environment*(コンパイル時環境) と呼ばれるデータ構造を管理することです。これは実行時環境内にて特定の変数アクセス命令が実行された時に、どの変数がどのフレーム内のどの位置に存在することになるのかを追跡します。コンパイル時環境はフレームのリストであり、各フレームが変数の変数のリストを保持します。(もちろん値が束縛されない変数も存在します。値はコンパイル時には計算されないためです)。コンパイル時環境は `compile` の追加の引数になり、各コード生成器に渡されます。`lambda` のボディがコンパイルされる時、`compile-lambda-body` がコンパイル時環境を手続のパラメータを持つフレームにより拡張し、ボディを構成する命令列がその拡張された環境を用いてコンパイルされます。コンパイルの各時点にて、`compile-variable` と `compile-assignment` は適切なレキシカルアドレスを生成するためにコンパイル時環境を使用します。

**Exercise 5.39** through **Exercise 5.43** describe how to complete this sketch of the lexical-addressing strategy in order to incorporate lexical lookup into the compiler. **Exercise 5.44** describes another use for the compile-time environment.

**Exercise 5.39**から**Exercise 5.43**はコンパイラにレキシカルな検索を組込むためにこのレキシカルアドレス付けの戦略の草案をどのようにして完了させるかについて説明します。**Exercise 5.44**はコンパイル時環境の別の使用法を説明します。

**Exercise 5.39:** Write a procedure `lexical-address-lookup` that implements the new lookup operation. It should take two arguments—a lexical address and a run-time environment—and return the value of the variable stored at the specified lexical address. `Lexical-address-lookup` should signal an error if the value of the variable is the symbol `*unassigned*`.<sup>82</sup> Also write a procedure `lexical-address-set!` that implements the operation that changes the value of the variable at a specified lexical address.

新しい検索命令を実装する `lexical-address-lookup` 手續を書け。2 つの引数、レキシカルアドレスと実行時環境を取ること。そして指定したレキシカルアドレスに格納された変数の値を

---

<sup>82</sup>This is the modification to variable lookup required if we implement the scanning method to eliminate internal definitions (**Exercise 5.43**). We will need to eliminate these definitions in order for lexical addressing to work.

返すこと。`lexical-address-lookup` はもし変数の値がシンボル `*unassigned*` ならばエラーを発する。<sup>83</sup>また指定したレキシカルアドレスの変数の値を変更する操作を実装する手続 `lexical-address-set!` を書け。

**Exercise 5.40:** Modify the compiler to maintain the compile-time environment as described above. That is, add a compile-time-environment argument to `compile` and the various code generators, and extend it in `compile-lambda-body`.

コンパイラを変更し、上で説明されたコンパイル時環境を保存するようにせよ。つまり、`compile` と多様なコード生成器の引数に `compile-time-environment` を追加し、それを `compile-lambda-body` の中に拡張せよ。

**Exercise 5.41:** Write a procedure `find-variable` that takes as arguments a variable and a compile-time environment and returns the lexical address of the variable with respect to that environment. For example, in the program fragment that is shown above, the compile-time environment during the compilation of expression  $\langle e1 \rangle$  is  $((y\ z)\ (a\ b\ c\ d\ e)\ (x\ y))$ . `Find-variable` should produce

引数として変数とコンパイル時環境を取り、その環境に関するその変数のレキシカルアドレスを返す手続 `find-variable` を書け。例えば、上で示されたプログラムの断片において、式  $\langle e1 \rangle$  をコンパイルしている間のコンパイル時環境は  $((y\ z)\ (a\ b\ c\ d\ e)\ (x\ y))$  である。`find-variable` は以下を生成しなければならない。

```
(find-variable 'c '((y z) (a b c d e) (x y)))
(1 2)
(find-variable 'x '((y z) (a b c d e) (x y)))
(2 0)
(find-variable 'w '((y z) (a b c d e) (x y)))
not-found
```

---

<sup>83</sup>これはもし内部定義を削除するためにこの検索手法を実装するのであれば必要となる、変数検索に対する変更です (Exercise 5.43)。レキシカルアドレスをうまく動かすためにはこれらの定義を排除する必要があります。

**Exercise 5.42:** Using `find-variable` from Exercise 5.41, rewrite `compile-variable` and `compile-assignment` to output lexical-address instructions. In cases where `find-variable` returns `not-found` (that is, where the variable is not in the compile-time environment), you should have the code generators use the evaluator operations, as before, to search for the binding. (The only place a variable that is not found at compile time can be is in the global environment, which is part of the run-time environment but is not part of the compile-time environment.<sup>84</sup> Thus, if you wish, you may have the evaluator operations look directly in the global environment, which can be obtained with the operation (`op get-global-environment`), instead of having them search the whole run-time environment found in `env`.) Test the modified compiler on a few simple cases, such as the nested `lambda` combination at the beginning of this section.

Exercise 5.41の`find-variable`を使用して、`compile-variable`と`compile-assignment`を書き直し、レキシカルアドレス命令を出力するようにせよ。`find-variable`が`not-found`を返す場合には(つまり、変数がコンパイル時環境内には存在しない場合には)、コード生成器に対して以前と同じ環境命令を使用させることで束縛を検索させなければならない。(コンパイル時に変数が見つからない唯一の場所はグローバル環境である。これは実行時環境の一部であり、コンパイル時環境の一部ではない。<sup>85</sup>従って、もしあなたが望むなら、それらに対し `env` 内の全ての実行時環境を探させる代わりに、環境の命令に、命令 (`op get-global-environment`)により獲得できるグローバル環境を直接探させてもかまわない)。

---

<sup>84</sup>Lexical addresses cannot be used to access variables in the global environment, because these names can be defined and redefined interactively at any time. With internal definitions scanned out, as in Exercise 5.43, the only definitions the compiler sees are those at top level, which act on the global environment. Compilation of a definition does not cause the defined name to be entered in the compile-time environment.

<sup>85</sup>レキシカルアドレスはグローバル環境内の変数をアクセスするためには利用できません。なぜなら、これらの名前は対話形式的に任意の時点で定義と再定義が可能なためです。Exercise 5.43の内部定義走査を用いてコンパイラが知ることができる定義は、グローバル環境に従うトップレベルのものだけです。定義のコンパイルは、定義された名前がコンパイル時環境に入れることにはなりません。

変更したコンパイラをこの節の最初の入れ子の `lambda` の組み合わせのような、いくつかの簡単な事例を用いてテストせよ。

**Exercise 5.43:** We argued in [Section 4.1.6](#) that internal definitions for block structure should not be considered “real” `defines`. Rather, a procedure body should be interpreted as if the internal variables being defined were installed as ordinary `lambda` variables initialized to their correct values using `set!`. [Section 4.1.6](#) and [Exercise 4.16](#) showed how to modify the metacircular interpreter to accomplish this by scanning out internal definitions. Modify the compiler to perform the same transformation before it compiles a procedure body.

[Section 4.1.6](#)においてブロック構造に対する内部定義は“実際の”`define`だと考慮されるべきでないと主張した。そうではなく、手続のボディは通常の `set!` を用いて正しい値に初期化された `lambda` の変数のように、内部変数定義が導入されたかのように解釈されるべきである。[Section 4.1.6](#)と[Exercise 4.16](#)はどのようにメタ循環インタプリタを変更して内部定義を走査することで、これを達成するかを示した。コンパイラを変更し、手続のボディをコンパイルする前にこれと同じ変形を実行するようにせよ。

**Exercise 5.44:** In this section we have focused on the use of the compile-time environment to produce lexical addresses. But there are other uses for compile-time environments. For instance, in [Exercise 5.38](#) we increased the efficiency of compiled code by open-coding primitive procedures. Our implementation treated the names of open-coded procedures as reserved words. If a program were to rebind such a name, the mechanism described in [Exercise 5.38](#) would still open-code it as a primitive, ignoring the new binding. For example, consider the procedure

この節ではレキシカルアドレスを生成するためのコンパイル時環境の使用に焦点を合わせた。しかしコンパイル時環境の他の使用法も存在する。例として、[Exercise 5.38](#)ではコンパイルされたコードの効率を open-code なプリミティブ手続により向上させた。私達の実装は open-code な手続を予約語として扱った。もしプログラムがそのような名前を再束縛するなら、[Exercise 5.38](#)にて説明された仕組みは依然としてプリミティブとして open-code し、新

しい束縛を無視するだろう。例えば、以下の手続について考えてみる。

```
(lambda (+ * a b x y)
  (+ (* a x) (* b y)))
```

which computes a linear combination of  $x$  and  $y$ . We might call it with arguments `+matrix`, `*matrix`, and four matrices, but the open-coding compiler would still open-code the `+` and the `*` in  $(+ (* a x) (* b y))$  as primitive `+` and `*`. Modify the open-coding compiler to consult the compile-time environment in order to compile the correct code for expressions involving the names of primitive procedures. (The code will work correctly as long as the program does not `define` or `set!` these names.)

これは  $x$  と  $y$  の一次結合を求める。これを引数 `+matrix`, `*matrix`, それに 4 つの行列 (matrix) と共に呼ぶこともあるだろう。しかし、open-code なコンパイラは依然として  $(+ (* a x) (* b y))$  内の `+` と `*` をプリミティブな `+` と `*` として open-code してしまうだろう。open-code なコンパイラを変更し、プリミティブな手続の名前を含む式に対して正しいコードをコンパイルするために、コンパイル時環境を参考にするようにせよ。(このコードはプログラムがこれらの名前に対して `define` や `set!` を行わない限り正しく動くようになる。)

### 5.5.7 Interfacing Compiled Code to the Evaluator

We have not yet explained how to load compiled code into the evaluator machine or how to run it. We will assume that the explicit-control-evaluator machine has been defined as in [Section 5.4.4](#), with the additional operations specified in [Footnote 38](#). We will implement a procedure `compile-and-go` that compiles a Scheme expression, loads the resulting object code into the evaluator machine, and causes the machine to run the code in the evaluator global environment, print the result, and enter the evaluator's driver loop. We will also modify the evaluator so that interpreted expressions can call compiled procedures as well as interpreted ones. We can then put a compiled procedure into the machine and use the evaluator to call it:

私達はまだコンパイルされたコードを評価機にどのようにロードするか、またはどのように実行するかについて説明していません。ここでは明示的制御評価機がSection 5.4.4の時点にて定義された状態であると仮定します。Footnote 38で指定された追加の命令も含みます。Scheme 式をコンパイルし、結果としてのオブジェクトコードを評価機にロードし、評価機にグローバル環境の中で実行させ、結果を表示し、評価機のドライバープヘル入る手続 `compile-and-go` を実装します。また評価機を変更し、逐次翻訳された式がコンパイルされた手続を逐次翻訳されたものと同じように呼ぶことができるようにもします。するとコンパイルされた手続を機械に入れてそれを呼び出すことができます。

```
(compile-and-go
  '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n))))
;;; EC-Eval value:
ok
;;; EC-Eval input:
(factorial 5)
;;; EC-Eval value:
120
```

To allow the evaluator to handle compiled procedures (for example, to evaluate the call to `factorial` above), we need to change the code at `apply-dispatch` (Section 5.4.1) so that it recognizes compiled procedures (as distinct from compound or primitive procedures) and transfers control directly to the entry point of the compiled code:<sup>86</sup>

評価機にコンパイルされた手続の取り扱いを可能にするには(例えば上記の `factorial` の呼出を評価すること)、`apply-dispatch`(Section 5.4.1) のコードを変更して、それがコンパイルされた手続を(複合、またはプリミティブな手続から区別可能なものとして)認識し、制御を直接コンパイルされたコードのエントリポイントへと移動させる必要があります。<sup>87</sup>

---

<sup>86</sup>Of course, compiled procedures as well as interpreted procedures are compound (nonprimitive). For compatibility with the terminology used in the explicit-control evaluator, in this section we will use “compound” to mean interpreted (as opposed to compiled).

<sup>87</sup>もちろん、逐次翻訳された手続と同様にコンパイルされた手続も複合 (compound,

```

apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (test (op compiled-procedure?) (reg proc))
  (branch (label compiled-apply))
  (goto (label unknown-procedure-type))

compiled-apply
  (restore continue)
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))

```

Note the restore of `continue` at `compiled-apply`. Recall that the evaluator was arranged so that at `apply-dispatch`, the continuation would be at the top of the stack. The compiled code entry point, on the other hand, expects the continuation to be in `continue`, so `continue` must be restored before the compiled code is executed.

`compiled-apply` での `continue` の復元に注意して下さい。評価機は `apply-dispatch` にて継続がスタックの一番上になるように準備されています。一方で、コンパイルされたコードのエントリポイントは継続が `continue` の中にあることを期待しています。そのため、`continue` はコンパイルされたコードが実行される前に復元されなければなりません。

To enable us to run some compiled code when we start the evaluator machine, we add a `branch` instruction at the beginning of the evaluator machine, which causes the machine to go to a new entry point if the `flag` register is set.

評価機を開始した時にいくつかのコンパイルされたコードを実行することを可能にするために、`branch` 命令を評価機の最初に追加します。これはもし `flag` レジスタが設定されていれば、機械を新しいエントリポイントへと飛ばします。<sup>88</sup>

---

非プリミティブ) です。明示的制御評価機で使用された用語との互換性のために、この節では“複合”を逐次翻訳された(コンパイルされたの逆を)意味するものとして使用します。

<sup>88</sup>Now that the evaluator machine starts with a `branch`, we must always initialize the `flag` register before starting the evaluator machine. To start the machine at its ordinary read-eval-print loop, we could use

今や評価機は `branch` を用いて開始するので、私達は常に評価機を開始する前に `flag`

```

(branch (label external-entry))           ; flag が立っていれば飛ぶ
read-eval-print-loop
  (perform (op initialize-stack))
  ...

```

`External-entry` assumes that the machine is started with `val` containing the location of an instruction sequence that puts a result into `val` and ends with `(goto (reg continue))`. Starting at this entry point jumps to the location designated by `val`, but first assigns `continue` so that execution will return to `print-result`, which prints the value in `val` and then goes to the beginning of the evaluator's read-eval-print loop.

`external-entry` は機械が結果を `val` に入れ (`goto (reg continue)`) で終わる命令列の位置を持つ `val` と共に開始すると仮定します。このエントリポイントで開始する場合、`val` で指定された位置へ飛びます。しかし、最初に `continue` に実行が `print-result` に戻るように設定します。`print-result` は `val` 内の値を表示し、次に評価機の REPL の最初へと飛びます。<sup>89</sup>

レジスタを初期化しなければなりません。機械を通常の REPL にて開始するためには、以下を用いることができます。

```

(define (start-eceval)
  (set! the-global-environment (setup-environment))
  (set-register-contents! eceval 'flag false)
  (start eceval))

```

<sup>89</sup>Since a compiled procedure is an object that the system may try to print, we also modify the system print operation `user-print` (from [Section 4.1.4](#)) so that it will not attempt to print the components of a compiled procedure:

コンパイルされた手続はシステムが表示しようとするかもしれないオブジェクトであるため、システムの表示命令 ([Section 4.1.4](#))の`user-print`も変更し、コンパイルされた手続の構成部品を表示しようとしないようにします。

```

(define (user-print object)
  (cond ((compound-procedure? object)
         (display (list 'compound-procedure
                       (procedure-parameters object)
                       (procedure-body object)
                       '<procedure-env>)))
        ((compiled-procedure? object)
         (display '<compiled-procedure>))
        (else (display object))))

```

```

external-entry
  (perform (op initialize-stack))
  (assign env (op get-global-environment))
  (assign continue (label print-result))
  (goto (reg val))

```

Now we can use the following procedure to compile a procedure definition, execute the compiled code, and run the read-eval-print loop so we can try the procedure. Because we want the compiled code to return to the location in `continue` with its result in `val`, we compile the expression with a target of `val` and a linkage of `return`. In order to transform the object code produced by the compiler into executable instructions for the evaluator register machine, we use the procedure `assemble` from the register-machine simulator ([Section 5.2.2](#)). We then initialize the `val` register to point to the list of instructions, set the `flag` so that the evaluator will go to `external-entry`, and start the evaluator.

これで以下手続を用いて手続定義をコンパイルし、コンパイルされたコードを実行し、手続を試行することができるよう REPL を実行することができます。コンパイルされたコードに `continue` 内の位置に、`val` 内の結果を持って戻って欲しいため、式をターゲット `val` とリンク記述子 `return` を用いてコンパイルします。コンパイラにより生成されたオブジェクトコードを評価機で実行可能な命令に変形するために、レジスタマシンシミュレータ ([Section 5.2.2](#)) の手続 `assemble` を使用します。次に `val` レジスタを命令のリストを指すように初期化し、`flag` を評価機が `external-entry` へ飛ぶように設定し、評価機を開始します。

```

(define (compile-and-go expression)
  (let ((instructions
         (assemble
           (statements
             (compile expression 'val 'return))
           eceval)))
    (set! the-global-environment (setup-environment))
    (set-register-contents! eceval 'val instructions)
    (set-register-contents! eceval 'flag true)
    (start eceval)))

```

---

If we have set up stack monitoring, as at the end of [Section 5.4.4](#), we can examine the stack usage of compiled code:

もし [Section 5.4.4](#) の終わりのように スタック監視を設定したなら、コンパイルされたコードのスタック使用量を調査できます。

```
(compile-and-go
'(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n))))
(total-pushes = 0 maximum-depth = 0)
;;; EC-Eval value:
ok
;;; EC-Eval input:
(factorial 5)
(total-pushes = 31 maximum-depth = 14)
;;; EC-Eval value:
120
```

Compare this example with the evaluation of `(factorial 5)` using the interpreted version of the same procedure, shown at the end of [Section 5.4.4](#). The interpreted version required 144 pushes and a maximum stack depth of 28. This illustrates the optimization that results from our compilation strategy.

この例を、[Section 5.4.4](#) の終わりで示された同じ手続の逐次翻訳された版を用いた `(factorial 5)` の評価と比べてみて下さい。逐次翻訳された版は 144 回の push と最大スタック深度 28 を必要としました。これは私達のコンパイル戦略に起因する最適化を説明しています。

## Interpretation and compilation

With the programs in this section, we can now experiment with the alternative execution strategies of interpretation and compilation.<sup>90</sup> An interpreter raises the machine to the level of the user program; a compiler lowers the user program to the level of the machine language. We can regard the Scheme language (or any programming language) as a coherent family of abstractions

---

<sup>90</sup>We can do even better by extending the compiler to allow compiled code to call interpreted procedures. See [Exercise 5.47](#).

erected on the machine language. Interpreters are good for interactive program development and debugging because the steps of program execution are organized in terms of these abstractions, and are therefore more intelligible to the programmer. Compiled code can execute faster, because the steps of program execution are organized in terms of the machine language, and the compiler is free to make optimizations that cut across the higher-level abstractions.

この節のプログラムを用いることで、今では逐次翻訳とコンパイルの代替的な実行戦略を実験することができます。<sup>91</sup> インタプリタは機械をユーザプログラムのレベルへと上げます。コンパイラはユーザプログラムを機械語のレベルへと下げます。私達は Scheme 言語を（またはどんなプログラミング言語も）機械語の上に構築された体系化の目的と同じとした仲間だと見做すことができます。インタプリタは対話的なプログラム開発とデバッグに最適です。プログラムのステップの実行がこれらの抽象化を用いて組織化され、そのため、プログラマにとってより理解しやすくなります。コンパイルされたコードはより速く実行することができます。プログラムのステップの実行が機械語を利用して体系化され、コンパイラは自由に高いレベルの抽象化を近道する最適化を作ることができます。<sup>92</sup>

---

<sup>91</sup> コンパイラを拡張してコンパイルされたコードに逐次翻訳された手続の呼び出しを許可することでさらにうまく行うことができます。Exercise 5.47を参照して下さい。

<sup>92</sup> Independent of the strategy of execution, we incur significant overhead if we insist that errors encountered in execution of a user program be detected and signaled, rather than being allowed to kill the system or produce wrong answers. For example, an out-of-bounds array reference can be detected by checking the validity of the reference before performing it. The overhead of checking, however, can be many times the cost of the array reference itself, and a programmer should weigh speed against safety in determining whether such a check is desirable. A good compiler should be able to produce code with such checks, should avoid redundant checks, and should allow programmers to control the extent and type of error checking in the compiled code.

実行戦略とは独立して、もしユーザプログラムを実行した場合にエラーに遭遇した時にシステムを殺すことや間違った値を生成するおおを許可するのではなく、エラーが発見され、その旨が伝えられることを望むのならば、明らかなオーバヘッドを経験することになります。例えば、配列の境界外参照は実行する前に参照の有効性をチェックすることで発見することができます。しかし、チェックのオーバヘッドは配列参照自体の何倍ものコストに成ります。そしてプログラマはそのようなチェックが望ましいかの決定において安全性よりもスピードに重きを置きます。良いコンパイラはそのようなチェックを行うコードを生成することが可能であるべきです。また冗長なチェックは回避し、プログラマにコンパイルされたコード内でのエラーチェックの範囲と型を制御できるようにするべきです。

Compilers for popular languages, such as C and C++, put hardly any error-checking

The alternatives of interpretation and compilation also lead to different strategies for porting languages to new computers. Suppose that we wish to implement Lisp for a new machine. One strategy is to begin with the explicit-control evaluator of Section 5.4 and translate its instructions to instructions for the new machine. A different strategy is to begin with the compiler and change the code generators so that they generate code for the new machine. The second strategy allows us to run any Lisp program on the new machine by first compiling it with the compiler running on our original Lisp system, and linking it with a compiled version of the run-time library.<sup>93</sup> Better yet, we can compile the compiler itself, and run this on the new machine to compile other Lisp programs.<sup>94</sup> Or we can compile one of the interpreters of Section 4.1 to produce an interpreter that runs on the new machine.

---

operations into running code, so as to make things run as fast as possible. As a result, it falls to programmers to explicitly provide error checking. Unfortunately, people often neglect to do this, even in critical applications where speed is not a constraint. Their programs lead fast and dangerous lives. For example, the notorious “Worm” that paralyzed the Internet in 1988 exploited the UNIX(tm) operating system’s failure to check whether the input buffer has overflowed in the finger daemon. (See Spafford 1989.)

C や C++ のような人気のある言語のコンパイラはほとんど何も実行コードの中にエラーチェックの命令を挿入しません。可能な限り速く実行するためです。結果として、プログラマに対して明示的にエラーチェックを提供させることに陥ります。残念なことに、人々は良くこのことを軽視します。例えスピードが制約ではない重要なアプリケーションにおいてもです。こうのような人々のプログラムは高速、かつ危険な生活へと導きます。例えば、1988 年にインターネットを麻痺させた悪名高い“Worm”(ワーム) は UNIX(tm) OS (オペレーティングシステム) の finger デーモンにおける入力バッファがオーバーフローしたかどうかのチェックミスを利用しました。(Spafford 1989 参照)

<sup>93</sup>Of course, with either the interpretation or the compilation strategy we must also implement for the new machine storage allocation, input and output, and all the various operations that we took as “primitive” in our discussion of the evaluator and compiler. One strategy for minimizing work here is to write as many of these operations as possible in Lisp and then compile them for the new machine. Ultimately, everything reduces to a small kernel (such as garbage collection and the mechanism for applying actual machine primitives) that is hand-coded for the new machine.

<sup>94</sup>This strategy leads to amusing tests of correctness of the compiler, such as checking whether the compilation of a program on the new machine, using the compiled compiler, is identical with the compilation of the program on the original Lisp system. Tracking down the source of differences is fun but often frustrating, because the results are extremely sensitive to minuscule details.

逐次翻訳とコンパイルの代替もまた、新しいコンピュータへ言語を移植するための異なる戦略へと導きます。新しい機械に Lisp を実装したいと願っていると仮定します。1つの戦略はSection 5.4の明示的制御評価機と共に始めて、その命令を新しい機械の命令へと翻訳することです。異なる戦略はコンパイラと共に始めてコード生成器を変更し、新しい機械のコードを生成するようにします。2つ目の戦略はどんな Lisp プログラムも最初に元の Lisp システム上で動くコンパイラを用いてコンパイルし、実行時ライブラリのコンパイル済みの版とリンクすることにより、新しい機械の上で実行させることができます。<sup>95</sup> もっと良いことには、コンパイラそれ自身をコンパイルすることができます。そしてこれを新しい機械の上で他の Lisp プログラムをコンパイルするために実行するのです。<sup>96</sup> または、Section 4.1のインタプリタの内1つをコンパイルして新しい機械上で実行できるインタプリタを生成することもできます。

**Exercise 5.45:** By comparing the stack operations used by compiled code to the stack operations used by the evaluator for the same computation, we can determine the extent to which the compiler optimizes use of the stack, both in speed (reducing the total number of stack operations) and in space (reducing the maximum stack depth). Comparing this optimized stack use to the performance of a special-purpose machine for the same computation gives some indication of the quality of the compiler.

コンパイルされたコードにより使用されたスタック命令と同じ演算のための評価機により使用されたスタック命令と比較することで、コンパイラのスタック使用の最適化の範囲を速さ(スタック命令の総数の削減)と記憶域(最大スタック深度の削減)の両方において判断することができる。この最適化されたスタックの使用を、

---

<sup>95</sup>もちろん、逐次翻訳とコンパイルの戦略のどちらを用いても、新しい機械の記憶域割り当て、入出力(I/O)、そして評価機とコンパイラの議論において“プリミティブ”として扱った全ての多彩な命令もまた新しい機械のために実装しなければなりません。ここで仕事量を最小化するための1つの方法としてはこれらの命令を可能な限り Lisp で書き、次に新しい機械のためにコンパイルすることが上げられます。究極的には、全てが新しい機械のために手で書かれた(ガベージコレクションや実際の機械のプリミティブを適用する仕組みの様な)小さなカーネルに縮小されます。

<sup>96</sup>この戦略は、コンパイルされたコンパイラを用いた、新しい機械上でのプログラムのコンパイルが元の Lisp システム上のプログラムのコンパイルと同一であるかどうかという、コンパイラの正確性の楽しいテストへと至ります。違いの原因の追跡は楽しいのですが、しばしばイライラもさせます。その結果はとても小さな詳細に非常に敏感なためです。

同じ演算のための特別な目的の機械と比較することでコンパイルの品質の何らかの指標を与えることができる。

- a [Exercise 5.27](#) asked you to determine, as a function of  $n$ , the number of pushes and the maximum stack depth needed by the evaluator to compute  $n!$  using the recursive factorial procedure given above. [Exercise 5.14](#) asked you to do the same measurements for the special-purpose factorial machine shown in [Figure 5.11](#). Now perform the same analysis using the compiled `factorial` procedure.

[Exercise 5.27](#)は、評価機が上で与えられた再帰階乗手続を用いて  $n!$  を求めるのに必要なプッシュの数と最大スタック深度を  $n$  の関数として決定するよう求めた。[Exercise 5.14](#)は[Figure 5.11](#)で示された特別な目的の階乗マシンに対しする同じ測定を求めた。ここでは同じ分析をコンパイルした `factorial` 手続を用いて実行する。

Take the ratio of the number of pushes in the compiled version to the number of pushes in the interpreted version, and do the same for the maximum stack depth. Since the number of operations and the stack depth used to compute  $n!$  are linear in  $n$ , these ratios should approach constants as  $n$  becomes large. What are these constants? Similarly, find the ratios of the stack usage in the special-purpose machine to the usage in the interpreted version.

コンパイルされた版のプッシュの数と逐次翻訳された版のプッシュの数との比率を取得せよ次に同じ事を最大スタック深度に対しても行なえ。 $n!$  を求めるために使用される命令数とスタック深度は  $n$  の線形であるために、これらの比率は  $n$  が巨大になるにつれ定数へと収束するはずである。これらの定数は何か? 同様に、特定目的マシンの使用量と逐次翻訳の版の使用量との比率も求めよ。

Compare the ratios for special-purpose versus interpreted code to the ratios for compiled versus interpreted code. You should find that the special-purpose machine does much better than the compiled code, since the hand-tailored controller code should be much better than what is produced by our

rudimentary general-purpose compiler.

特定目的と逐次翻訳されたコードとの間の比率と、コンパイルされたコードと逐次翻訳されたコードとの間の比率を比較せよ。特定目的マシンがコンパイルされたコードよりもとても良いことに気付くはずだ。手作りのコントローラのコードは基本的な汎用目的のコンパイラにより生成されたものよりも優れているはずだからである。

- b Can you suggest improvements to the compiler that would help it generate code that would come closer in performance to the hand-tailored version?

パフォーマンスにおいて手作り版により近いコードを生成することを手助けする、コンパイラに対する改善を提案できるだろうか？

**Exercise 5.46:** Carry out an analysis like the one in [Exercise 5.45](#) to determine the effectiveness of compiling the tree-recursive Fibonacci procedure

[Exercise 5.45](#)のような分析を木再帰フィボナッチ手続のコンパイルの効果を判断するために実行せよ。

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
          (fib (- n 2)))))
```

compared to the effectiveness of using the special-purpose Fibonacci machine of [Figure 5.12](#). (For measurement of the interpreted performance, see [Exercise 5.29](#).) For Fibonacci, the time resource used is not linear in  $n$ ; hence the ratios of stack operations will not approach a limiting value that is independent of  $n$ .

[Figure 5.12](#)の特定目的フィボナッチマシンを用いた場合の効果と比較せよ。(逐次翻訳のパフォーマンスの測定のために、[Exercise 5.29](#)を参照せよ)。フィボナッチ数では、使用された時間的リソースは  $n$  の線形にはならない。

**Exercise 5.47:** This section described how to modify the explicit-control evaluator so that interpreted code can call compiled procedures. Show how to modify the compiler so that compiled procedures can call not only primitive procedures and compiled procedures, but interpreted procedures as well. This requires modifying `compile-procedure-call` to handle the case of compound (interpreted) procedures. Be sure to handle all the same `target` and `linkage` combinations as in `compile-proc-app1`. To do the actual procedure application, the code needs to jump to the evaluator's `compound-apply` entry point. This label cannot be directly referenced in object code (since the assembler requires that all labels referenced by the code it is assembling be defined there), so we will add a register called `compapp` to the evaluator machine to hold this entry point, and add an instruction to initialize it:

この節では逐次翻訳されたコードがコンパイルされたコードを呼び出すことができるようにするために、明示的制御評価機をどのように変更するかを説明した。コンパイルされた手続がプリミティブとコンパイルされた手続のみでなく、逐次翻訳された手続も同様に呼び出すことができるようにするために、コンパイラをどのように変更するのか示せ。これは `compile-procedure-call` を複合(逐次翻訳)の場合を取り扱うように変更する必要がある。全ての同じ `target` と `linkage` の組み合わせを `compile-proc-app1` が行うように取り扱うよう気をつけよ。実際に手続適用を行うためには、コードは評価機の `compound-apply` エントリポイントへ飛ぶ必要がある。このラベルはオブジェクトコードの中では直接参照することができない。(アセンブラーが全てのラベルに対し、それがアセンブルしている、そこで定義されるコードにより参照されることを要求するためである)。従って、`compapp` と呼ばれるレジスタを評価機に追加し、このエントリポイントを持たせて、これを初期化する命令を追加する。

```
(assign compapp (label compound-apply))  
(branch (label external-entry)) ;flag が立つていれば  
飛び  
read-eval-print-loop ...
```

To test your code, start by defining a procedure `f` that calls a

procedure `g`. Use `compile-and-go` to compile the definition of `f` and start the evaluator. Now, typing at the evaluator, define `g` and try to call `f`.

あなたのコードをテストするために、手続 `g` を呼ぶ手続 `f` を定義することから始めよ。`compile-and-go` を用いて `f` の定義をコンパイルし、評価機を開始せよ。ここから評価機に対し入力を行い `g` を定義し `f` の呼出を試せ。

**Exercise 5.48:** The `compile-and-go` interface implemented in this section is awkward, since the compiler can be called only once (when the evaluator machine is started). Augment the compiler-interpreter interface by providing a `compile-and-run` primitive that can be called from within the explicit-control evaluator as follows:

この節で実装された `compile-and-go` インターフェイスは扱いにくい。コンパイラを(評価機が開始された時に)一度しか呼ぶことができないためだ。以下のように明示的制御評価機の中から呼び出すことができる `compile-and-run` を追加することでコンパイラ-インターフェイスを増補せよ。

```
; ; EC-Eval input:  
(compile-and-run  
  '(define (factorial n)  
    (if (= n 1) 1 (* (factorial (- n 1)) n))))  
; ; EC-Eval value:  
ok  
; ; EC-Eval input:  
(factorial 5)  
; ; EC-Eval value:  
120
```

**Exercise 5.49:** As an alternative to using the explicit-control evaluator's read-eval-print loop, design a register machine that performs a read-compile-execute-print loop. That is, the machine should run a loop that reads an expression, compiles it, assembles and executes the resulting code, and prints the result. This is easy to run in our simulated setup, since we can arrange to call the procedures `compile` and `assemble` as "register-machine operations."

明示的制御評価機の REPL を用いる代わりとして、read-compile-execute-print loop を実行するレジスタマシンを設計せよ。言い換えば、このマシンは式を読み込み、それをコンパイルし、その結果のコードをアセンブルして実行し、その結果を表示するループを実行する。これは私達のシミュレートされた構成内で簡単に実行できる。なぜなら、手続 `compile` と `assemble` を “レジスタマシンの命令” として呼ぶことを手配できるからだ。

**Exercise 5.50:** Use the compiler to compile the metacircular evaluator of [Section 4.1](#) and run this program using the register-machine simulator. (To compile more than one definition at a time, you can package the definitions in a `begin`.) The resulting interpreter will run very slowly because of the multiple levels of interpretation, but getting all the details to work is an instructive exercise.

コンパイラを用いて [Section 4.1](#) のメタ循環評価機をコンパイルし、レジスタマシンシミュレータと用いてこのプログラムを実行せよ。(一度に複数の定義をコンパイルするために、`begin` の中に定義を詰めることができる)。結果としてのインタプリタの実行は複数レベルの逐次翻訳のため、とても遅い。しかし、実行の詳細全てを理解することは教育的な課題である。

**Exercise 5.51:** Develop a rudimentary implementation of Scheme in C (or some other low-level language of your choice) by translating the explicit-control evaluator of [Section 5.4](#) into C. In order to run this code you will need to also provide appropriate storage-allocation routines and other run-time support.

C 言語 (またはあなたが選んだ何らかの他の低レベルな言語) による Scheme の基本的な実装を、[Section 5.4](#) の明示的制御評価機を C 言語に翻訳することで開発せよ。このコードを実行するためには、適切なメモリ割当ルーチンと他の実行時サポートも提供する必要がある。

**Exercise 5.52:** As a counterpoint to [Exercise 5.51](#), modify the compiler so that it compiles Scheme procedures into sequences of C instructions. Compile the metacircular evaluator of [Section 4.1](#) to produce a Scheme interpreter written in C.

[Exercise 5.51](#)に対する好対照として、コンパイラを変更して Scheme の手続を C 言語の命令列へとコンパイルするようにせよ。[Section](#)

4.1のメタ循環評価機をコンパイルして C 言語で書かれた Scheme インタプリタを生成せよ。

# References

- Abelson, Harold, Andrew Berlin, Jacob Katzenelson, William McAllister, Guillermo Rozas, Gerald Jay Sussman, and Jack Wisdom. 1992. The Supercomputer Toolkit: A general framework for special-purpose computing. *International Journal of High-Speed Electronics* 3(3): 337-361. ([Onl](#))
- Allen, John. 1978. *Anatomy of Lisp*. New York: McGraw-Hill.
- ANSI X3.226-1994. *American National Standard for Information Systems—Programming Language—Common Lisp*.
- Appel, Andrew W. 1987. Garbage collection can be faster than stack allocation. *Information Processing Letters* 25(4): 275-279. ([Online](#))
- Backus, John. 1978. Can programming be liberated from the von Neumann style? *Communications of the ACM* 21(8): 613-641. ([Online](#))
- Baker, Henry G., Jr. 1978. List processing in real time on a serial computer. *Communications of the ACM* 21(4): 280-293. ([Online](#))
- Batali, John, Neil Mayle, Howard Shrobe, Gerald Jay Sussman, and Daniel Weise. 1982. The Scheme-81 architecture—System and chip. In *Proceedings of the MIT Conference on Advanced Research in VLSI*, edited by Paul Penfield, Jr. Dedham, MA: Artech House.
- Borning, Alan. 1977. ThingLab—An object-oriented system for building simulations using constraints. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*. ([Online](#))
- Borodin, Alan, and Ian Munro. 1975. *The Computational Complexity of Algebraic and Numeric Problems*. New York: American Elsevier.
- Chaitin, Gregory J. 1975. Randomness and mathematical proof. *Scientific American* 232(5): 47-52.
- Church, Alonzo. 1941. *The Calculi of Lambda-Conversion*. Princeton, N.J.:

- Princeton University Press.
- Clark, Keith L. 1978. Negation as failure. In *Logic and Data Bases*. New York: Plenum Press, pp. 293-322. ([Online](#))
- Clinger, William. 1982. Nondeterministic call by need is neither lazy nor by name. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp. 226-234.
- Clinger, William, and Jonathan Rees. 1991. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pp. 155-162. ([Online](#))
- Colmérauer A., H. Kanoui, R. Pasero, and P. Roussel. 1973. Un système de communication homme-machine en français. Technical report, Groupe Intelligence Artificielle, Université d'Aix Marseille, Luminy.
- Cormen, Thomas, Charles Leiserson, and Ronald Rivest. 1990. *Introduction to Algorithms*. Cambridge, MA: MIT Press.
- Darlington, John, Peter Henderson, and David Turner. 1982. *Functional Programming and Its Applications*. New York: Cambridge University Press.
- Dijkstra, Edsger W. 1968a. The structure of the “THE” multiprogramming system. *Communications of the ACM* 11(5): 341-346. ([Online](#))
- Dijkstra, Edsger W. 1968b. Cooperating sequential processes. In *Programming Languages*, edited by F. Genuys. New York: Academic Press, pp. 43-112. ([Online](#))
- Dinesman, Howard P. 1968. *Superior Mathematical Puzzles*. New York: Simon and Schuster.
- deKleer, Johan, Jon Doyle, Guy Steele, and Gerald J. Sussman. 1977. AMORD: Explicit control of reasoning. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, pp. 116-125. ([Online](#))
- Doyle, Jon. 1979. A truth maintenance system. *Artificial Intelligence* 12: 231-272. ([Online](#))
- Feigenbaum, Edward, and Howard Shrobe. 1993. The Japanese National Fifth Generation Project: Introduction, survey, and evaluation. In *Future Generation Computer Systems*, vol. 9, pp. 105-117.
- Feeley, Marc. 1986. Deux approches à l'implantation du language Scheme. Masters thesis, Université de Montréal.
- Feeley, Marc and Guy Lapalme. 1987. Using closures for code generation. *Journal of Computer Languages* 12(1): 47-66. ([Online](#))

- Feller, William. 1957. *An Introduction to Probability Theory and Its Applications*, volume 1. New York: John Wiley & Sons.
- Fenichel, R., and J. Yochelson. 1969. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM* 12(11): 611-612.
- Floyd, Robert. 1967. Nondeterministic algorithms. *JACM*, 14(4): 636-644.
- Forbus, Kenneth D., and Johan deKleer. 1993. *Building Problem Solvers*. Cambridge, MA: MIT Press.
- Friedman, Daniel P., and David S. Wise. 1976. CONS should not evaluate its arguments. In *Automata, Languages, and Programming: Third International Colloquium*, edited by S. Michaelson and R. Milner, pp. 257-284. ([Online](#))
- Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 1992. *Essentials of Programming Languages*. Cambridge, MA: MIT Press/ McGraw-Hill.
- Gabriel, Richard P. 1988. The Why of Y. *Lisp Pointers* 2(2): 15-25. ([Online](#))
- Goldberg, Adele, and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley.
- Gordon, Michael, Robin Milner, and Christopher Wadsworth. 1979. *Edinburgh LCF*. Lecture Notes in Computer Science, volume 78. New York: Springer-Verlag.
- Gray, Jim, and Andreas Reuter. 1993. *Transaction Processing: Concepts and Models*. San Mateo, CA: Morgan-Kaufman.
- Green, Cordell. 1969. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 219-240. ([Online](#))
- Green, Cordell, and Bertram Raphael. 1968. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the ACM National Conference*, pp. 169-181.
- Griss, Martin L. 1981. Portable Standard Lisp, a brief overview. Utah Symbolic Computation Group Operating Note 58, University of Utah.
- Guttag, John V. 1977. Abstract data types and the development of data structures. *Communications of the ACM* 20(6): 396-404. ([Online](#))
- Hamming, Richard W. 1980. *Coding and Information Theory*. Englewood Cliffs, N.J.: Prentice-Hall.
- Hanson, Christopher P. 1990. Efficient stack allocation for tail-recursive languages. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pp. 106-118.

- Hanson, Christopher P. 1991. A syntactic closures macro facility. *Lisp Pointers*, 4(3). ([Online](#))
- Hardy, Godfrey H. 1921. Srinivasa Ramanujan. *Proceedings of the London Mathematical Society* XIX(2).
- Hardy, Godfrey H., and E. M. Wright. 1960. *An Introduction to the Theory of Numbers*. 4th edition. New York: Oxford University Press.
- Havender, J. 1968. Avoiding deadlocks in multi-tasking systems. *IBM Systems Journal* 7(2): 74-84.
- Hearn, Anthony C. 1969. Standard Lisp. Technical report AIM-90, Artificial Intelligence Project, Stanford University. ([Online](#))
- Henderson, Peter. 1980. *Functional Programming: Application and Implementation*. Englewood Cliffs, N.J.: Prentice-Hall.
- Henderson, Peter. 1982. Functional Geometry. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 179-187. ([Online](#)) ([2002 version](#))
- Hewitt, Carl E. 1969. PLANNER: A language for proving theorems in robots. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 295-301. ([Online](#))
- Hewitt, Carl E. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence* 8(3): 323-364. ([Online](#))
- Hoare, C. A. R. 1972. Proof of correctness of data representations. *Acta Informatica* 1(1).
- Hodges, Andrew. 1983. *Alan Turing: The Enigma*. New York: Simon and Schuster.
- Hofstadter, Douglas R. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books.
- Hughes, R. J. M. 1990. Why functional programming matters. In *Research Topics in Functional Programming*, edited by David Turner. Reading, MA: Addison-Wesley, pp. 17-42. ([Online](#))
- IEEE Std 1178-1990. 1990. *IEEE Standard for the Scheme Programming Language*.
- Ingerman, Peter, Edgar Irons, Kirk Sattley, and Wallace Feurzeig; assisted by M. Lind, Herbert Kanner, and Robert Floyd. 1960. THUNKS: A way of compiling procedure statements, with some comments on procedure declarations. Unpublished manuscript. (Also, private communication from Wallace Feurzeig.)

- Kaldewaij, Anne. 1990. *Programming: The Derivation of Algorithms*. New York: Prentice-Hall.
- Knuth, Donald E. 1973. *Fundamental Algorithms*. Volume 1 of *The Art of Computer Programming*. 2nd edition. Reading, MA: Addison-Wesley.
- Knuth, Donald E. 1981. *Seminumerical Algorithms*. Volume 2 of *The Art of Computer Programming*. 2nd edition. Reading, MA: Addison-Wesley.
- Kohlbecker, Eugene Edmund, Jr. 1986. Syntactic extensions in the programming language Lisp. Ph.D. thesis, Indiana University. ([Online](#))
- Konopasek, Milos, and Sundaresan Jayaraman. 1984. *The TK!Solver Book: A Guide to Problem-Solving in Science, Engineering, Business, and Education*. Berkeley, CA: Osborne/McGraw-Hill.
- Kowalski, Robert. 1973. Predicate logic as a programming language. Technical report 70, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh. ([Online](#))
- Kowalski, Robert. 1979. *Logic for Problem Solving*. New York: North-Holland.
- Lamport, Leslie. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7): 558-565. ([Online](#))
- Lampson, Butler, J. J. Horning, R. London, J. G. Mitchell, and G. K. Popek. 1981. Report on the programming language Euclid. Technical report, Computer Systems Research Group, University of Toronto. ([Online](#))
- Landin, Peter. 1965. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2): 89-101.
- Lieberman, Henry, and Carl E. Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* 26(6): 419-429. ([Online](#))
- Liskov, Barbara H., and Stephen N. Zilles. 1975. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering* 1(1): 7-19. ([Online](#))
- McAllester, David Allen. 1978. A three-valued truth-maintenance system. Memo 473, MIT Artificial Intelligence Laboratory. ([Online](#))
- McAllester, David Allen. 1980. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Laboratory. ([Online](#))
- McCarthy, John. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3(4): 184-195. ([Online](#))
- McCarthy, John. 1963. A basis for a mathematical theory of computation.

In *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg. North-Holland. ([Online](#))

McCarthy, John. 1978. The history of Lisp. In *Proceedings of the ACM SIGPLAN Conference on the History of Programming Languages*. ([Online](#))

McCarthy, John, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. 1965. *Lisp 1.5 Programmer's Manual*. 2nd edition. Cambridge, MA: MIT Press. ([Online](#))

McDermott, Drew, and Gerald Jay Sussman. 1972. Conniver reference manual. Memo 259, MIT Artificial Intelligence Laboratory. ([Online](#))

Miller, Gary L. 1976. Riemann's Hypothesis and tests for primality. *Journal of Computer and System Sciences* 13(3): 300-317. ([Online](#))

Miller, James S., and Guillermo J. Rozas. 1994. Garbage collection is fast, but a stack is faster. Memo 1462, MIT Artificial Intelligence Laboratory. ([Online](#))

Moon, David. 1978. MacLisp reference manual, Version 0. Technical report, MIT Laboratory for Computer Science. ([Online](#))

Moon, David, and Daniel Weinreb. 1981. Lisp machine manual. Technical report, MIT Artificial Intelligence Laboratory. ([Online](#))

Morris, J. H., Eric Schmidt, and Philip Wadler. 1980. Experience with an applicative string processing language. In *Proceedings of the 7th Annual ACM SIGACT/SIGPLAN Symposium on the Principles of Programming Languages*.

Phillips, Hubert. 1934. *The Sphinx Problem Book*. London: Faber and Faber.

Pitman, Kent. 1983. The revised MacLisp Manual (Saturday evening edition). Technical report 295, MIT Laboratory for Computer Science. ([Online](#))

Rabin, Michael O. 1980. Probabilistic algorithm for testing primality. *Journal of Number Theory* 12: 128-138.

Raymond, Eric. 1993. *The New Hacker's Dictionary*. 2nd edition. Cambridge, MA: MIT Press. ([Online](#))

Raynal, Michel. 1986. *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press.

Rees, Jonathan A., and Norman I. Adams IV. 1982. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 114-122. ([Online](#))

Rees, Jonathan, and William Clinger (eds). 1991. The revised<sup>4</sup> report on the algorithmic language Scheme. *Lisp Pointers*, 4(3). ([Online](#))

Rivest, Ronald, Adi Shamir, and Leonard Adleman. 1977. A method for obtaining digital signatures and public-key cryptosystems. Technical memo LC-

S/TM82, MIT Laboratory for Computer Science. ([Online](#))

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12(1): 23.

Robinson, J. A. 1983. Logic programming—Past, present, and future. *New Generation Computing* 1: 107-124.

Spafford, Eugene H. 1989. The Internet Worm: Crisis and aftermath. *Communications of the ACM* 32(6): 678-688. ([Online](#))

Steele, Guy Lewis, Jr. 1977. Debunking the “expensive procedure call” myth. In *Proceedings of the National Conference of the ACM*, pp. 153-62. ([Online](#))

Steele, Guy Lewis, Jr. 1982. An overview of Common Lisp. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp. 98-107.

Steele, Guy Lewis, Jr. 1990. *Common Lisp: The Language*. 2nd edition. Digital Press. ([Online](#))

Steele, Guy Lewis, Jr., and Gerald Jay Sussman. 1975. Scheme: An interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory. ([Online](#))

Steele, Guy Lewis, Jr., Donald R. Woods, Raphael A. Finkel, Mark R. Crispin, Richard M. Stallman, and Geoffrey S. Goodfellow. 1983. *The Hacker’s Dictionary*. New York: Harper & Row. ([Online](#))

Stoy, Joseph E. 1977. *Denotational Semantics*. Cambridge, MA: MIT Press.

Sussman, Gerald Jay, and Richard M. Stallman. 1975. Heuristic techniques in computer-aided circuit analysis. *IEEE Transactions on Circuits and Systems CAS-22(11)*: 857-865. ([Online](#))

Sussman, Gerald Jay, and Guy Lewis Steele Jr. 1980. Constraints—A language for expressing almost-hierarchical descriptions. *AI Journal* 14: 1-39. ([Online](#))

Sussman, Gerald Jay, and Jack Wisdom. 1992. Chaotic evolution of the solar system. *Science* 257: 256-262. ([Online](#))

Sussman, Gerald Jay, Terry Winograd, and Eugene Charniak. 1971. Microplanner reference manual. Memo 203A, MIT Artificial Intelligence Laboratory. ([Online](#))

Sutherland, Ivan E. 1963. SKETCHPAD: A man-machine graphical communication system. Technical report 296, MIT Lincoln Laboratory. ([Online](#))

Teitelman, Warren. 1974. Interlisp reference manual. Technical report, Xerox Palo Alto Research Center.

Thatcher, James W., Eric G. Wagner, and Jesse B. Wright. 1978. Data

type specification: Parameterization and the power of specification techniques. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 119-132.

Turner, David. 1981. The future of applicative languages. In *Proceedings of the 3rd European Conference on Informatics*, Lecture Notes in Computer Science, volume 123. New York: Springer-Verlag, pp. 334-348.

Wand, Mitchell. 1980. Continuation-based program transformation strategies. *Journal of the ACM* 27(1): 164-180. ([Online](#))

Waters, Richard C. 1979. A method for analyzing loop programs. *IEEE Transactions on Software Engineering* 5(3): 237-247.

Winograd, Terry. 1971. Procedures as a representation for data in a computer program for understanding natural language. Technical report AI TR-17, MIT Artificial Intelligence Laboratory. ([Online](#))

Winston, Patrick. 1992. *Artificial Intelligence*. 3rd edition. Reading, MA: Addison-Wesley.

Zabih, Ramin, David McAllester, and David Chapman. 1987. Non-deterministic Lisp with dependency-directed backtracking. AAAI-87, pp. 59-64. ([Online](#))

Zippel, Richard. 1979. Probabilistic algorithms for sparse polynomials. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT.

Zippel, Richard. 1993. *Effective Polynomial Computation*. Boston, MA: Kluwer Academic Publishers.

# List of Exercises

## Chapter 1

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10
1.11	1.12	1.13	1.14	1.15	1.16	1.17	1.18	1.19	1.20
1.21	1.22	1.23	1.24	1.25	1.26	1.27	1.28	1.29	1.30
1.31	1.32	1.33	1.34	1.35	1.36	1.37	1.38	1.39	1.40
1.41	1.42	1.43	1.44	1.45	1.46				

## Chapter 2

2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10
2.11	2.12	2.13	2.14	2.15	2.16	2.17	2.18	2.19	2.20
2.21	2.22	2.23	2.24	2.25	2.26	2.27	2.28	2.29	2.30
2.31	2.32	2.33	2.34	2.35	2.36	2.37	2.38	2.39	2.40
2.41	2.42	2.43	2.44	2.45	2.46	2.47	2.48	2.49	2.50
2.51	2.52	2.53	2.54	2.55	2.56	2.57	2.58	2.59	2.60
2.61	2.62	2.63	2.64	2.65	2.66	2.67	2.68	2.69	2.70
2.71	2.72	2.73	2.74	2.75	2.76	2.77	2.78	2.79	2.80
2.81	2.82	2.83	2.84	2.85	2.86	2.87	2.88	2.89	2.90
2.91	2.92	2.93	2.94	2.95	2.96	2.97			

### **Chapter 3**

3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10
3.11	3.12	3.13	3.14	3.15	3.16	3.17	3.18	3.19	3.20
3.21	3.22	3.23	3.24	3.25	3.26	3.27	3.28	3.29	3.30
3.31	3.32	3.33	3.34	3.35	3.36	3.37	3.38	3.39	3.40
3.41	3.42	3.43	3.44	3.45	3.46	3.47	3.48	3.49	3.50
3.51	3.52	3.53	3.54	3.55	3.56	3.57	3.58	3.59	3.60
3.61	3.62	3.63	3.64	3.65	3.66	3.67	3.68	3.69	3.70
3.71	3.72	3.73	3.74	3.75	3.76	3.77	3.78	3.79	3.80
3.81	3.82								

### **Chapter 4**

4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10
4.11	4.12	4.13	4.14	4.15	4.16	4.17	4.18	4.19	4.20
4.21	4.22	4.23	4.24	4.25	4.26	4.27	4.28	4.29	4.30
4.31	4.32	4.33	4.34	4.35	4.36	4.37	4.38	4.39	4.40
4.41	4.42	4.43	4.44	4.45	4.46	4.47	4.48	4.49	4.50
4.51	4.52	4.53	4.54	4.55	4.56	4.57	4.58	4.59	4.60
4.61	4.62	4.63	4.64	4.65	4.66	4.67	4.68	4.69	4.70
4.71	4.72	4.73	4.74	4.75	4.76	4.77	4.78	4.79	

### **Chapter 5**

5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10
5.11	5.12	5.13	5.14	5.15	5.16	5.17	5.18	5.19	5.20
5.21	5.22	5.23	5.24	5.25	5.26	5.27	5.28	5.29	5.30
5.31	5.32	5.33	5.34	5.35	5.36	5.37	5.38	5.39	5.40
5.41	5.42	5.43	5.44	5.45	5.46	5.47	5.48	5.49	5.50
5.51	5.52								

# List of Figures

## Chapter 1

1.1    1.2    1.3    1.4    1.5

## Chapter 2

2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10
2.11	2.12	2.13	2.14	2.15	2.16	2.17	2.18	2.19	2.20
2.21	2.22	2.23	2.24	2.25	2.26				

## Chapter 3

3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10
3.11	3.12	3.13	3.14	3.15	3.16	3.17	3.18	3.19	3.20
3.21	3.22	3.23	3.24	3.25	3.26	3.27	3.28	3.29	3.30
3.31	3.32	3.33	3.34	3.35	3.36	3.37	3.38		

## Chapter 4

4.1    4.2    4.3    4.4    4.5    4.6

## Chapter 5

5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10
5.11	5.12	5.13	5.14	5.15	5.16	5.17	5.18		

# Index

Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer.

この索引内のどんな間違いも、コンピュータの手助けにより準備されたという事実により説明できるだろう。

—Donald E. Knuth, *Fundamental Algorithms*  
(Volume 1 of *The Art of Computer Programming*)

*k*-term finite continued fraction, 122  
*n*-fold smoothed function, 134

abstract models, 158  
abstract syntax, 623  
abstraction barriers, 142, 152, 153  
accumulator, 198, 199, 385  
acquired, 529, 530  
action, 853, 854  
additive, 308  
additively, 144, 292  
address, 909  
address arithmetic, 909  
agenda, 478  
algebraic specification, 158  
aliasing, 402  
and-gate, 467  
and-gate, 467  
applicative-order, 681

applicative-order evaluation, 28  
applicative-order, 680  
arbiter, 533  
arguments, 11  
assembler, 879  
assertions, 753  
assignment operator, 377  
atomically, 532  
automatic storage allocation, 908, 909  
average damping, 121  
  
B-trees, 270, 271  
backbone, 456, 457  
backquote, 980  
backtracks, 708, 710  
balanced, 193  
barrier synchronization, 537  
base address, 910  
Bertrand's hypothesis, 563

bignum, 913  
bindings, 408  
binds, 48  
binomial coefficients, 70  
block structure, 50, 51  
bound variable, 48  
box-and-pointer notation, 170  
breakpoint, 906  
broken heart, 926  
bugs, 2, 3

cache-coherence, 509  
cache-coherence, 509  
call-by-name, 685, 686  
call-by-need, 685, 686  
call-by-name, 553  
call-by-name thunks, 553, 554  
call-by-need, 553, 554  
call-by-need thunks, 553, 554  
capturing, 48  
Carmichael numbers, 90, 91  
case analysis, 29  
cell, 530, 531  
chronological backtracking, 709, 710  
Church numerals, 162  
Church-Turing thesis, 657  
Church-Turing thesis, 656  
clauses, 30  
closed world assumption, 796  
closure, 142, 143  
closure property, 170  
code generator, 972  
coerce, 342, 343  
coercion, 333, 334  
combinations, 11  
comments, 213  
compacting, 923  
compilation, 966, 967  
compile-time environment, 1026, 1027  
composition, 133  
compound data, 138  
compound data object, 139, 140

compound procedure, 21  
computability, 656, 657  
computational process, 2  
concurrently, 505  
congruent modulo, 87  
connectors, 487  
consequent expression, 30  
constraint networks, 487  
constructors, 145  
continuation procedures, 728  
continued fraction, 122  
control structure, 790  
controller, 842  
conventional interfaces, 143, 196  
current time, 482

data, 2, 157  
data abstraction, 140, 145  
data paths, 842  
data-directed, 293  
data-directed programming, 144, 308, 309  
deadlock, 535  
deadlock-recovery, 535, 536  
debug, 3  
deep binding, 648  
deferred operations, 57, 58  
delayed argument, 593, 594  
delayed evaluation, 375, 541  
delayed object, 547  
dense, 357  
dependency-directed backtracking, 709, 710  
depth-first search, 709, 710  
deque, 455  
derived expressions, 637  
digital signals, 467  
dispatching on type, 307  
displacement number, 1025  
dotted-tail notation, 181  
driver loop, 652, 653

empty list, 176  
encapsulated, 381  
enclosing environment, 408  
entry points, 848  
enumerator, 198  
environment, 15  
environment model, 375  
environments, 408  
Euclid's Algorithm, 82  
Euclidean ring, 365  
evaluating, 10  
evaluation, 10  
evaluator, 615  
event-driven simulation, 466  
evlis tail recursion, 941, 942  
execution procedure, 672  
explicit-control evaluator, 932  
expression, 10  
  
failure continuation, 728  
FIFO, 447, 448  
filter, 107, 198  
first-class, 131  
fixed point, 118  
fixed-length, 277  
forcing, 685  
forwarding address, 926  
frame, 774, 775  
frame coordinate map, 230  
frame number, 1025  
framed-stack, 938  
frames, 408  
free, 48  
free list, 916  
front, 447, 448  
full-adder, 470  
function boxes, 467  
functional programming, 396  
functional programming languages,  
    608  
  
garbage, 920  
  
garbage collection, 908, 909, 921  
garbage collector, 434  
garbage-collected, 691  
generic operations, 144  
generic procedures, 284, 292, 293  
glitches, 2, 3  
global, 53, 408  
global environment, 15  
golden ratio, 64  
grammar, 719  
  
half-interval method, 115  
half-adder, 468  
Halting Theorem, 660, 661  
headed list, 456, 457  
hiding principle, 381  
hierarchical, 171, 172  
hierarchy of types, 337, 338  
higher-order procedures, 99  
Horner's rule, 205  
  
imperative programming, 404  
indeterminates, 348, 349  
index, 910, 911  
indexing, 776  
instantiated with, 759  
instruction counting, 905  
instruction execution procedure, 882  
instruction sequence, 974, 975  
instruction tracing, 905  
instructions, 840, 841, 848  
integerizing factor, 368  
integers, 9  
integrator, 585  
interning, 914, 915  
interpreter, 4, 615  
invariant quantity, 78  
inverter, 467  
iterative improvement, 135  
iterative process, 58  
  
key, 274, 275

labels, 848  
lazy evaluation, 680, 681  
lexical address, 1025  
lexical addressing, 648  
lexical scoping, 50, 51  
linear iterative process, 58  
linear recursive process, 57, 58  
linkage descriptor, 973  
list, 173, 174, 181  
list structure, 173, 174  
list-structured, 149  
list-structured memory, 907, 908  
local evolution, 53  
local state variables, 376, 377  
location, 909  
logic-programming, 618  
logic-programming, 618  
logical and, 467  
logical deductions, 769  
logical or, 467, 468

machine language, 965, 966  
macro, 637  
map, 198, 199  
mark-sweep, 922  
Memoization, 464  
memoization, 69, 465  
memoize, 685, 686  
merge, 609, 610  
message passing, 160, 161, 320, 321  
message-passing, 384, 385  
metacircular, 619  
Metalinguistic abstraction, 615  
Miller-Rabin test, 96, 97  
modular, 373, 374  
modulo, 87  
modus ponens, 790  
moments in time, 505  
Monte Carlo integration, 393  
Monte Carlo simulation, 389  
mutable data objects, 432  
mutators, 432

mutex, 529, 530  
mutual exclusion, 529, 530

native language, 965, 966  
needed, 977  
networks, 613  
Newton's method, 126  
nil, 176  
non-computable, 660, 661  
non-strict, 682  
nondeterministic, 513, 514  
nondeterministic choice point, 707, 708  
nondeterministic computing, 618, 702  
normal-order, 681  
normal-order evaluation, 28, 617, 618  
normal-order, 680

obarray, 913, 914  
object program, 966, 967  
objects, 374, 375  
open-code, 1020, 1021  
operand, 11  
operands, 11  
operator, 11, 670  
or-gate, 467  
or-gate, 468  
order of growth, 71  
ordinary, 324  
output prompt, 653

package, 310, 311  
painter, 219, 220  
pair, 148  
parse, 718, 719  
Pascal's triangle, 70  
pattern, 756  
pattern matcher, 774  
pattern matching, 773, 774  
pattern variable, 756, 757  
pipelining, 506  
pointer, 170

poly, 350  
power series, 567  
predicate, 30, 31  
prefix, 278  
prefix code, 278  
prefix notation, 11  
pretty-printing, 12, 13  
primitive constraints, 487  
probabilistic algorithms, 91, 92  
procedural abstraction, 45  
procedural epistemology, xxviii, xxix  
procedure, 59  
procedure definitions, 20, 21  
procedures, 7  
process, 59  
program, 2  
programming languages, 2  
prompt, 652, 653  
pseudo-random, 388, 389  
pseudodivision, 368  
pseudoremainder, 368

quasiquote, 980  
queries, 751, 752  
query language, 751, 752  
queue, 447  
quote, 243, 244

Ramanujan numbers, 584  
rational functions, 364  
RC circuit, 586, 587  
read-eval-print loop, 13  
reader macro characters, 829  
real numbers, 9  
rear, 447, 448  
recursion equations, 4  
Recursion theory, 657, 658  
recursive, 16, 44  
recursive process, 57, 58  
recursive;, 44  
red-black trees, 270, 271  
referentially transparent, 400

register machine, 840, 841  
register table, 881  
registers, 840, 841  
released, 529, 530  
remainder, 87  
remainder of, 87  
resolution principle, 747, 748  
ripple-carry adder, 474  
ripple-carry adder, 474  
robust, 241, 242  
RSA algorithm, 92  
RSA アルゴリズム, 92  
rules, 752, 763

satisfy, 759  
scope, 48  
selectors, 145  
semaphore, 529, 530  
separator code, 278  
sequence, 173  
sequence accelerator, 573  
sequences, 105  
serializer, 516  
serializers, 519  
series RLC circuit, 597  
shadow, 409  
shared, 440  
side-effect bugs, 402, 403  
sieve of Eratosthenes, 557, 558  
smoothing, 134  
source language, 966  
source program, 966  
sparse, 357  
special forms, 19  
stack, 59, 868  
state variables, 58, 376  
statements, 977  
stop-and-copy, 921, 922  
stratified design, 240  
stream processing, 28, 29  
streams, 374, 375, 539–541  
strict, 682

subroutine, 860  
substitution, 26  
substitution model, 25  
subtype, 337, 338  
success continuation, 728  
summation of a series, 100, 101  
summer, 585  
supertype, 338  
symbolic expressions, 143  
syntactic sugar, 19, 20  
syntax, 621, 622  
systematically search, 708, 709  
systems, 613

tableau, 574  
tabulation, 69, 464, 465  
tagged architectures, 911, 912  
tail-recursive, 60, 948  
target, 973  
thrashing, ix, x  
thunk, 684, 685  
thunks, 684  
time, 504  
time segments, 482  
tower, 338  
tree accumulation, 17  
tree recursion, 62  
trees, 188  
truth maintenance, 709, 710  
Turing machine, 656, 657  
type field, 911, 912  
type tag, 301  
type tags, 293  
type-inferencing, 600  
type-inferencing, 600

typed pointers, 911, 912  
unbound, 408  
unification, 747, 748, 773, 774, 782  
unification algorithm, 747, 748  
univariate polynomials, 348, 349  
universal machine, 656, 657  
upward-compatible extension, 696

value, 13, 14  
value of a variable, 408  
values, 243  
variable, 13, 14  
variable-length, 277  
vector, 909, 910

width, 165  
wires, 467  
wishful thinking, 146

zero crossings, 588

プロセス, 59

手続, 59  
手続の抽象化, 45  
正規順序評価, 28

特殊形式, 20  
置換, 27

述語, 31

適用順序評価, 28  
閉包性, 171

# Colophon

ON THE COVER PAGE is Agostino Ramelli's bookwheel mechanism from 1588. It could be seen as an early hypertext navigation aid. This image of the engraving is hosted by J. E. Johnson of [New Gottland](#).

The typefaces are Linux Libertine for body text and Linux Biolinum for headings, both by Philipp H. Poll. Typewriter face is Inconsolata created by Raph Levien and supplemented by Dimosthenis Kaponis and Takashi Tanigawa in the form of Inconsolata LGC.

Graphic design and typography are done by Andres Raba. Texinfo source is converted to LaTeX by a Perl script and compiled to PDF by XeLaTeX. Diagrams are drawn with Inkscape.