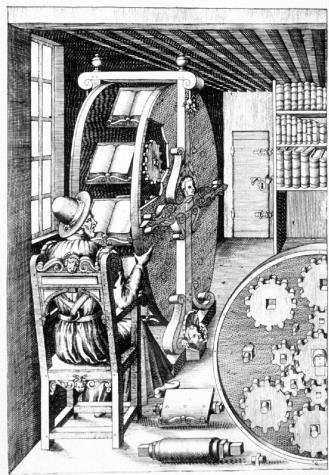


# **Structure and Interpretation of Computer Programs**



**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).

# 目次

非公式 Texinfo フォーマット	ix
非公式日本語版	xi
献辞	xiii
前書き	xiv
第二版 序文	xix
第一版 序文	xxi
謝辞	xxiv
<b>1 手続を用いた抽象化の構築</b>	<b>1</b>
1.1 プログラミングの要素 . . . . .	4
1.1.1 式 . . . . .	5
1.1.2 名前付けと環境 . . . . .	8
1.1.3 組み合わせの評価 . . . . .	9
1.1.4 複合手続 . . . . .	12
1.1.5 手続適用の置換モデル . . . . .	14
1.1.6 条件式と述語 . . . . .	17
1.1.7 例: ニュートン法による平方根 . . . . .	21
1.1.8 ブラックボックス抽象化としての手続 . . . . .	26

1.2	手続とそれが生成するプロセス . . . . .	31
1.2.1	線形再帰と反復 . . . . .	32
1.2.2	木再帰 . . . . .	37
1.2.3	増加のオーダー . . . . .	42
1.2.4	指數計算 . . . . .	44
1.2.5	最大公約数 . . . . .	48
1.2.6	例: 素数判定 . . . . .	50
1.3	高階手続による抽象の形式化 . . . . .	57
1.3.1	引数としての手續 . . . . .	58
1.3.2	<code>lambda</code> を用いた手續の構築 . . . . .	64
1.3.3	汎用手法としての手續 . . . . .	69
1.3.4	返り値としての手續 . . . . .	74
<b>2</b>	<b>データを用いた抽象化の構築</b>	<b>82</b>
2.1	データ抽象化のイントロダクション . . . . .	86
2.1.1	例: 分数のための数値演算命令 . . . . .	86
2.1.2	抽象化バリア . . . . .	91
2.1.3	データにより何が意味されるのか . . . . .	93
2.1.4	延長課題: 区間演算 . . . . .	96
2.2	階層データと閉包性 . . . . .	101
2.2.1	列の表現 . . . . .	103
2.2.2	階層構造 . . . . .	112
2.2.3	慣習的インターフェイスとしての列 . . . . .	118
2.2.4	例: ピクチャ一言語 . . . . .	132
2.3	記号データ . . . . .	149
2.3.1	クオート . . . . .	149
2.3.2	例: 記号微分 . . . . .	153
2.3.3	例: 集合を表現する . . . . .	158
2.3.4	例: ハフマン符号化木 . . . . .	169
2.4	抽象データの多重表現 . . . . .	178
2.4.1	複素数のための表現 . . . . .	180
2.4.2	タグ付きデータ . . . . .	184
2.4.3	データ適従プログラミングと付加性 . . . . .	188

2.5	ジェネリック命令を持つシステム . . . . .	197
2.5.1	ジェネリックな数値演算命令 . . . . .	199
2.5.2	異なる型のデータを組み合わせる . . . . .	204
2.5.3	例: 記号代数 . . . . .	213
<b>3</b>	<b>モジュール方式、オブジェクト、状態 . . . . .</b>	<b>228</b>
3.1	代入と局所状態 . . . . .	229
3.1.1	局所状態変数 . . . . .	230
3.1.2	代入導入の利点 . . . . .	237
3.1.3	代入導入のコスト . . . . .	241
3.2	評価の環境モデル . . . . .	248
3.2.1	評価のルール . . . . .	250
3.2.2	単純な手続の適用 . . . . .	253
3.2.3	局所状態のレポジトリとしてのフレーム . . . . .	256
3.2.4	内部定義 . . . . .	261
3.3	ミュータブルデータによるモデリング . . . . .	264
3.3.1	ミュータブルなリスト構造 . . . . .	265
3.3.2	キューの表現 . . . . .	276
3.3.3	テーブルの表現 . . . . .	282
3.3.4	デジタル回路のシミュレータ . . . . .	290
3.3.5	制約伝播 . . . . .	303
3.4	並行性: 時間が本質 . . . . .	315
3.4.1	並行システム内の時間の性質 . . . . .	317
3.4.2	並行性制御のための仕組み . . . . .	322
3.5	ストリーム . . . . .	337
3.5.1	ストリームとは遅延化リスト . . . . .	338
3.5.2	無限ストリーム . . . . .	347
3.5.3	ストリームパラダイムの利用 . . . . .	356
3.5.4	ストリームと遅延評価 . . . . .	370
3.5.5	関数型プログラムのモジュール化とオブジェクト のモジュール化 . . . . .	377

<b>4 メタ言語抽象化</b>	<b>383</b>
<b>4.1 メタ循環評価機</b>	386
4.1.1 評価機の核	388
4.1.2 式の表現	393
4.1.3 評価機のデータ構造	401
4.1.4 評価機をプログラムとして実行する	406
4.1.5 プログラムとしてのデータ	409
4.1.6 内部定義	413
4.1.7 構文分析を実行から分離する	419
<b>4.2 Scheme 上でのバリエーション --- 遅延評価</b>	424
4.2.1 正規順と適用順	425
4.2.2 遅延評価を持つインタプリタ	427
4.2.3 遅延化リストとしてのストリーム	436
<b>4.3 Scheme 上でのバリエーション --- 非決定性演算</b>	438
4.3.1 amb と検索	441
4.3.2 非決定性プログラムの例	445
4.3.3 Amb 評価機の実装	454
<b>4.4 論理プログラミング</b>	467
4.4.1 演繹的情報検索	471
4.4.2 クエリシステムの働き方	484
4.4.3 論理プログラミングは記号論理学なのか?	495
4.4.4 クエリシステムの実装	501
4.4.4.1 ドライバループとインスタンス化	501
4.4.4.2 評価機	503
4.4.4.3 パターンマッチングによりアサーション を見つける	506
4.4.4.4 ルールとユニフィケーション	509
4.4.4.5 データベースの保守	514
4.4.4.6 ストリーム命令	517
4.4.4.7 クエリ構文手続	518
4.4.4.8 フレームと束縛	521

<b>5 レジスタマシンによる演算</b>	<b>526</b>
5.1 レジスタマシンの設計 . . . . .	528
5.1.1 レジスタマシンを記述するための言語 . . . . .	531
5.1.2 機械設計における抽象化 . . . . .	535
5.1.3 サブルーチン . . . . .	538
5.1.4 再帰実装にスタックを使用する . . . . .	542
5.1.5 命令の要約 . . . . .	549
5.2 レジスタマシンシミュレータ . . . . .	550
5.2.1 マシンモデル . . . . .	552
5.2.2 アセンブラー . . . . .	557
5.2.3 各命令に対する実行手続の生成 . . . . .	561
5.2.4 機械のパフォーマンスの監視 . . . . .	569
5.3 記憶域の割当とガベージコレクション . . . . .	572
5.3.1 ベクタとしてのメモリ . . . . .	573
5.3.2 無限のメモリの幻想を維持する . . . . .	579
5.4 明示的制御評価機 . . . . .	587
5.4.1 明示制御評価機の核 . . . . .	589
5.4.2 列の評価と末尾再帰 . . . . .	596
5.4.3 条件文、代入、定義 . . . . .	599
5.4.4 評価機を実行する . . . . .	602
5.5 コンパイル . . . . .	608
5.5.1 コンパイラの構造 . . . . .	612
5.5.2 式のコンパイル . . . . .	617
5.5.3 組み合わせのコンパイル . . . . .	625
5.5.4 命令列のコンパイル . . . . .	633
5.5.5 コンパイルされたコードの例 . . . . .	636
5.5.6 レキシカルアドレッシング . . . . .	648
5.5.7 コンパイル済みコードと評価機の連結 . . . . .	653
<b>参考文献</b>	<b>662</b>
<b>課題リスト</b>	<b>670</b>
<b>図一覧</b>	<b>672</b>

索引	673
奥付	683

# 非公式 Texinfo フォーマット

これは SICP の第二版非公式 Texinfo 版です。

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なっています。

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。

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。

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”の記号を <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。

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。

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*

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.

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*

# 非公式日本語版

SICP はかつて第一版、第二版共に日本にて公式に翻訳が商業出版されていました。第二版を出版していたピアソン桐原が 2013 年 8 月に [ピアソングループ](#)から撤退し技術書の取扱を終了したため、日本語で SICP を読む機会は失われました。このことがこの翻訳を行うことの契機となりました。

実際にはその後、2014 年 1 月付近に、寛大にも第二版の訳者、和田英一先生がオンライン上にて SICP の訳書、「計算機プログラムの構造と解釈」全文を公開して下さいました。この時点でこの非公式日本語版の価値は随分と小さくなりました。

しかし、その時、既に 3 章まで翻訳していたこと、そして非公式 TexInfo 版が 2013 年 11 月に大改訂を行い、当初の日本語には正式に対応していない texi2pdf から変更を行い、XeLaTeX を採用したために、日本語でも美しい組版ができる可能性が出てきたことが、この原稿を廃棄することを押し止めました。

SICP のライセンスについてはインターネットアーカイブにて調べてみました。2001 年 1 月に MIT が SICP を寛大にもオンラインで無料で読むことができるよう公開された時にはライセンスが指定されていませんでした。

2008 年 4 月に MIT は SICP のライセンスを CC BY-NC と指定しました。その後ライセンスは 2011 年 10 月に一旦 CC BY-SA に変更されます。そして 2 年後の 2013 年 9 月に再び CC BY-NC へと戻されました。この事実が SICP 原文のライセンスの解釈を難しくしています。ライセンスの変更はオーナーの自由ですが、ライセンサーはコンテンツ取得時のライセンスを尊重すれば良いからです。

最初に非公式 TexInfo 版を作成した Lytha Ayth はライセンス指定の無い SICP 公開を Web 文化に基づくものだと理解しました。次に LaTeX の組版を開発した Andres Raba は CC BY-SA に基き正式な許諾の下、PDF 版を作成しました。私の翻訳は PDF 版のライセンスである CC BY-SA に従うことが求め

られます。しかし、現在の MIT が非商業を求めていることを鑑みて、Raba 氏に許可を頂いた上で非商業制約を追加した [CC BY-NC-SA 3.0](#) にてリリースすることにしました。

CC BY-NC、及び BY-SA は共に翻訳の許可を明記しています。従ってこの翻訳には Lytha が心配したような法的問題は起こらないと信じています。しかし同時に、法的問題は常に一方的に起こされることもあることもまた現実です。従って読者の皆様には常にネットワーク上のデータは(そしてプログラムも!)消えてなくなってしまうシャボン玉であることを忘れずに御用心願います。

TeX、LaTeX 環境の日本語対応を進めて下さった全ての関係者の皆様に感謝します。特に最新の情報を常に更新し続けて下さっている [TeX Wiki](#) の奥村 晴彦氏、W32TeX を自動でインストールし更新可能な [TeX インストーラ](#) 作者の阿部 紀行氏、XeLaTeX 向け日本語パッケージ “[ZXjatype](#)” を開発して下さった八登 崇之氏に感謝致します。

海外では SICP の新しい形の開発が非常に盛んです。PDF はもちろん、epub やインタラクティブ版、Kindle 版 (mobi 形式)、Clojure や JavaScript による SICP 等が公開されています。この翻訳は CC BY-NC-SA ですので非商業であればそのような派生や翻案に利用することが可能です。日本でも SICP の世界が広がっていくことを期待しています。

※ 校正御協力者様 (順不同、敬称略)

- Kei Shiratsuchi
- Kimura, Koichi
- のな
- Naoki Ainoya

## 献辞

この本を、尊敬と賛美を込めて、コンピュータの中に住む妖精に捧げます。

“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っていました。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っ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)

# 前書き

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影響を与えません。

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つ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！

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とも可能です。(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くいため、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。

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“機械の”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…！

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なって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。

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。プログラム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ムの責務です。

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ーリ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と同等に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の遺伝子を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築物です。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 プログラマ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。

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。

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。

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んと障害になりません。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

## 第二版 序文

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

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ることを喜ばしく思っています。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。

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 実装がそれら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。

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

てまとめることを試みました。

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。

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。

# 第一版序文

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”(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。私達はこの科目的開発を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、ほんの少數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。

私達のこの計算機科学の入門科目的設計は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。

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。

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。

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方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。

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。手続き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始めた呼び名)に感謝します。

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單一の表現、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。

# 謝辞

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。

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計算上の素晴らしい科目、“6.231”の知的末裔です。

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。

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 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され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 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。

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。

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。

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 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。

他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。

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, 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。

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 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。

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 TeX 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。

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

心がその中で、その力を単純な考えの上に發揮する“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)

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。

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

つたり、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。

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、初心者のプログラム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。

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霊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。例えれば 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。

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を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

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。

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>1</sup> Lisp はその名前を List Processing(リスト処理)の頭文字から取つており、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 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>2</sup>

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

<sup>1</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。

<sup>2</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) といっ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。

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>3</sup> Lisp は今でもどうしようもなく非効率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。

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(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 プログラミングの要素

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

---

<sup>3</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)

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、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、これにより複合要素は名前を付けて個体として扱える

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。

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>4</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。

### 1.1.1 式

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。

---

<sup>4</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。

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

インタプリタは以下を表示することで応答します。<sup>5</sup>

486

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

(- 1000 334)

666

(\* 5 99)

495

(/ 10 5)

2

(+ 2.7 10)

12.7

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。

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。

---

<sup>5</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。

```
(+ 21 35 12 7)
```

```
75
```

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

```
1200
```

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。

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
```

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))
```

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))
```

*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>6</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

---

<sup>6</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。

ードはしばしばインタプリタが*read-eval-print loop*(REPL:レプル)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>7</sup>

### 1.1.2 名前付けと環境

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。

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)
```

インタプリタは名前 `size` と値 2 を関連付けます。<sup>8</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
```

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
```

---

<sup>7</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 Perils 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”

<sup>8</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。

`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。

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) と呼ばれます。(より正確には *global environment*(グローバル環境、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)<sup>9</sup>

### 1.1.3 組み合わせの評価

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。

組み合わせを評価するため、以下を行います

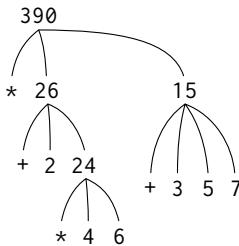
1. 組み合わせの部分式を評価する
2. 最も左の部分式(オペレータ)の値である手続を他の部分式の値である引数(オペランド)に対し適用する

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实上 *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>10</sup>

---

<sup>9</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なことを示します。

<sup>10</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します。後に組み合



**Figure 1.1:** 部分的組合せの値を示す木表現

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))
```

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とても強力な技術です。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。

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ことが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。

- 数字の値はそれが意味する値です
- 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。

せのオペレータそのものが組み合わ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。

- 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。

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てい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。

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。）

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。

11

---

<sup>11</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します。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。こ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”

#### 1.1.4 複合手続

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。

- 数値と算術命令はプリミティブなデータと手続です。
- 組み合わせのネストは演算の結合手法を提供します。
- 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。

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る、1つの単位としてアクセス可能になります。

“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))
```

これを以下のように理解することが可能です。

```
(define (square x) (* x x))
|   |   |
定義 二乗する xを   かける xを    xで.
```

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す。<sup>12</sup>

一般的な手続の定義形式は以下の通りです。

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

`<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>13</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

---

<sup>12</sup> 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ずに作成すること、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。

<sup>13</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。

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>14</sup>  $\langle name \rangle$  と  $\langle formal\ parameters \rangle$  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。

`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
```

`square` を構築要素として他の手続の構築に用いることも可能です。例えば、 $x^2 + y^2$  は次のように表現できます。

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

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
```

`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
```

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。

---

<sup>14</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。

### 1.1.5 手続適用の置換モデル

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。

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。

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。

過程を説明するために以下のコンビネーションを評価してみましょう。

(f 5)

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))

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))

従って問題は 2 つのオペランドとオペレータ 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、オペランド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 手續を 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 と y を置き換え、式は以下のように置換されます。

(+ (square 6) (square 10))

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))

乗算を置換することで以下になります。

(+ 36 100)

最終的には次のとおりです。

136

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、代入モデル)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。

- 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。
- 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うにつれ、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>15</sup>

## 適用順 対 正規順

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

---

<sup>15</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。

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)))
```

簡約が続きます。

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

136

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。

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 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 $(* 5 2)$  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>16</sup>

---

<sup>16</sup> Chapter 3 では stream processing(ストリーム処理) 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序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。

## 1.1.6 条件式と述語

この時点では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のです。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。

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

この考えは*case analysis*(ケース分析、事例分析)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))))
```

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)))
```

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))
```

*clauses*(クローズ、節)と呼ばれます。各ペアの最初の式は*predicate*(述語) — 値が真か偽になる式です。<sup>17</sup>

条件式は次のように評価されます。まず述語 *p*<sub>1</sub> が最初に評価されます。もしその値が偽であれば次に *p*<sub>2</sub> が評価されます。もし *p*<sub>2</sub> の値もまた偽で

<sup>17</sup>“真か偽のどちらかに解釈される”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。

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 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。

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>18</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。

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)))
```

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。

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))
```

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 <predicate> <consequent> <alternative>)
```

*if* 式を評価するためにインタプリタは式の  $\langle predicate \rangle$  の部分を評価することから始めます。もし  $\langle predicate \rangle$  の評価が真値になる場合、インタプリタは次

---

<sup>18</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。

に  $\langle \text{consequent} \rangle$  を評価しその値を返します。そうでなければ  $\langle \text{alternative} \rangle$  を評価しその値を返します。<sup>19</sup>

<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。

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

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**  $\langle e_1 \rangle \dots \langle e_n \rangle$ )

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**  $\langle e \rangle$ )

**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。

**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。

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))
```

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)))
```

または代替法として

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

---

<sup>19</sup>**if** と **cond** の小さな違いは **cond** の各クローズの  $\langle e \rangle$  は連続する式になってしまって良いことです。もし対応する  $\langle p \rangle$  が真になる場合、 $\langle e \rangle$  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  $\langle \text{consequent} \rangle$  と  $\langle \text{alternative} \rangle$  は単一の式でなければなりません

**Exercise 1.1:** 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:** 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:** 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:** 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 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))
```

次に以下の式を評価する。

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

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 例: ニュートン法による平方根

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。

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。

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

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))))
```

これはただ問題を提起するだけです。

関数と手続の間の対称性は事物の属性の説明と行い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>20</sup>

人はどのようにして平方根を求めること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>21</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	...	...

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。

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

<sup>20</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>21</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。

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)))
```

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))
```

“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>22</sup>

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

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>23</sup>

---

<sup>22</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>23</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。

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

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
```

`sqrt` プログラムはまた私達がここまで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に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>24</sup>

**Exercise 1.6:** 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 は Alyssa に対してプログラムのデモを行った。

---

<sup>24</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

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

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)))
```

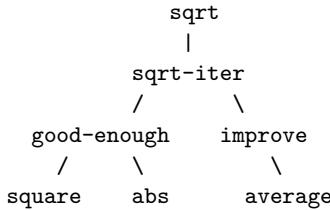
Alyssa が平方根の計算にこれの使用を試した時に何が起こるだろうか？

**Exercise 1.7:** 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け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である。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:** 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。

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

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。)



**Figure 1.2:** `sqrt` プログラムの手続分解

### 1.1.8 ブラックボックス抽象化としての手続

`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。

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。

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可能です。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のです。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。

従って返り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>25</sup>

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

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。

## ローカル名

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))
```

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える結論は重要です。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))
```

---

<sup>25</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

`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。

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。従つて `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。

手続の形式パラメタは手続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。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定義の意味は変わりません。<sup>26</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。手続定義においてはその手続の形式パラメタとして宣言された束縛変数はその手続のボディをそのスコープとします。

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るので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。

---

<sup>26</sup> 静的なり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

## 内部定義とブロック構造

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)))
```

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に存し、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))

```

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。しかしそれが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。このような規律を *lexical scoping*(レキシカルスコープ) と呼びます。<sup>27</sup>

```

(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))

```

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範囲にて利用されます。<sup>28</sup> ブロック構造の考え方に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。

<sup>27</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>28</sup> 組込み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

## 1.2 手続とそれが生成するプロセス

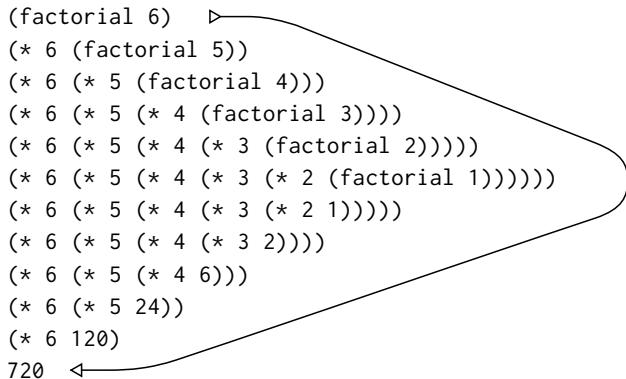
私達はプログラミングの要素について考えてきました。プリミティブな算術演算子を用いこれらの演算子を組み合わせ、その合成演算子を複合手続として定義することで抽象化を行ってきました。しかしそれらは私達がプログラムをどのように書くか知っていると言えるためには十分ではありません。私達の状況はチェスにおいて各駒がどのように動くのかルールを覚えたが典型的な序盤や戦術、戦略について何も知らない人に似ています。チェスの初心者棋士のように、私達はまだこの領域での慣習としての一般的なパターンを知りません。私達はどの手が打つ価値があるのか(どの手続が定義する価値があるのか)の知識を欠いています。打った手の(手続実行の)結果を予想する経験を欠いています。

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。

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なパターンについて説明することを試してみましょう。

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。



**Figure 1.3:**  $6!$  を求めるための線形再帰プロセス

### 1.2.1 線形再帰と反復

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。

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1.$$

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。

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

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)))))


```

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。

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
```

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。

再び、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>29</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)))
```

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。

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。両者は同じ数学的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

---

<sup>29</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:**  $6!$  を求めるための線形反復プロセス

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がります。

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。

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。

2つのプロセスの対称性は他の見方もあります。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>30</sup>

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に、ただ 3 つの変数を追跡することのみが必要です。

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い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>31</sup>

<sup>30</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。

<sup>31</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 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。

**Exercise 1.9:** 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))))
```

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:** 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))))))
```

以下の式の値はいくつであるか？

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

`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))
```

$n$  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。

---

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。

## 1.2.2 木再帰

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。

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$$

一般に、フィボナッチ数は次のルールにて定義可能です。

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

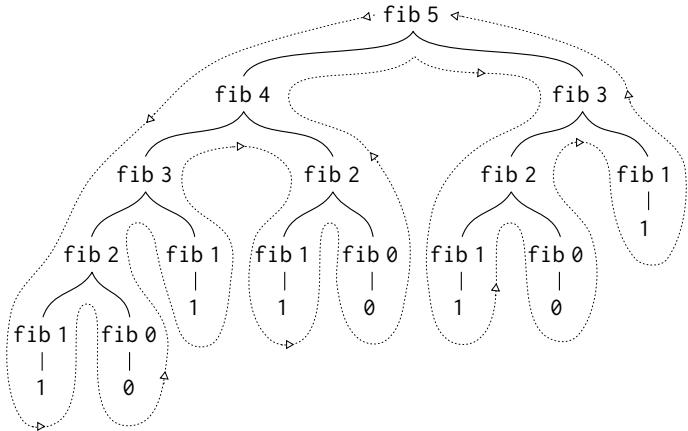
私達は直ぐにこの定義をフィボナッチ数を計算する再帰手続の定義に翻訳が可能です。

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

この計算のパターンについて考えてみましょう。 $(\text{fib } 5)$  を計算するには  $(\text{fib } 4)$  と  $(\text{fib } 3)$  を計算します。 $(\text{fib } 4)$  を計算するには  $(\text{fib } 3)$  と  $(\text{fib } 2)$  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  $\text{fib}$  手続が実行される度に毎回、自身を二回呼び出す事実を反映しています。

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  $(\text{fib } 3)$  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  $(\text{fib } 1)$  や  $(\text{fib } 0)$  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。この方法の酷さを知るために、 $\text{Fib}(n)$  の値が  $n$  に対し指數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:** (fib 5) を求める際に生成された木再帰プロセス

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

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

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。

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実行するというものです。

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

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))))
```

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ともその差は非常に大きくなります。

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>32</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。

## 例: 両替方法を数える

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？

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を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。

$n$  種類の硬貨を用いた場合、金額  $a$  の両替方法の数は

- 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、

プラス

---

<sup>32</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

- $d$  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

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。

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>33</sup>

- もし  $a$  が 0 である場合、両替方法は 1 と数える
- もし  $a$  が 0 未満の場合、両替方法は 0 と数える
- もし  $n$  が 0 の場合、両替方法は 0 と数える

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)))
```

---

<sup>33</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

(first-denomination 手続は利用可能な硬貨の種類の数を入力に取り、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)  
292
```

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>34</sup>

**Exercise 1.11:** 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。

---

<sup>34</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。手続を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)

**Exercise 1.12:** 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。

$$\begin{array}{ccccccc}
 & & 1 & & & & \\
 & & 1 & 1 & & & \\
 & & 1 & 2 & 1 & & \\
 & & 1 & 3 & 3 & 1 & \\
 1 & 4 & 6 & 4 & 1 & & \\
 & \dots & & & & &
 \end{array}$$

三角の端の数値は全て 1 であり、三角内部の各数値はその上 2 つの数値の和である。<sup>35</sup> パスカルの三角形の要素を再帰プロセスを用いて求める手続を書け。

**Exercise 1.13:**  $\text{Fib}(n)$  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。帰納法と斐波ナッチ数の定義 (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 増加のオーダー

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り得ることを示しました。この違いを説明する 1 つの便利な方法は、*order of growth*(増加のオーダー) の記法を用いて入力が大きくなるに従い、プロセスが要求するリソース(資源)の総体的量を得ることです。

$n$  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であれば、 $n$  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

---

<sup>35</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 Bhāskara Āchārya についても同様です。

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。

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、 $R(n)$  は増加のオーダー  $\Theta(f(n))$  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。

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  $\Theta(1)$ —一定数です。<sup>36</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。この時  $\varphi$  はSection 1.2.2で示したとおりの黄金比です。

増加のオーダー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増加のオーダーは  $\Theta(n^2)$  になります。一方で増加のオーダー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 の最後にて増加のオーダーが対数である 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:** 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増加のオーダーはいくつか?

**Exercise 1.15:** (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

<sup>36</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”をプ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。こ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。

三角法の恒等式、

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

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 手続  $p$  は  $(\text{sine } 12.15)$  を評価した時、何回適用されるか?
- b  $(\text{sine } a)$  が評価された時、 $\text{sine}$  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オーダーを求めよ。

#### 1.2.4 指数計算

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。

$$b^n = b \cdot b^{n-1}, \\ b^0 = 1,$$

早速、手続に翻訳します。

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

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))))
```

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。

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、

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

3回の乗算で求めることが可能です。

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

この方法は2の幕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。

$$\begin{array}{ll} b^n = (b^{n/2})^2 & n \text{ が偶数の場合,} \\ b^n = b \cdot b^{n-1} & n \text{ が奇数の場合.} \end{array}$$

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))))))
```

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))
```

`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で増加します。このプロセスは  $\Theta(\log n)$  で増加します。<sup>37</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>38</sup> 連続する二乗の考え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できません。<sup>39</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:** 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*(不変量) 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:** 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` 手續の類似である。

<sup>37</sup> より正確に言えば、必要とされる乗算の数は 1 から  $n$  の基數 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  $n$  の基數 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、その対数の基數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>38</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

<sup>39</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)))))
```

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:** 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>40</sup>

**Exercise 1.19:** 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ということである。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>41</sup>

```
(define (fib n)
  (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count))
```

---

<sup>40</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、リンドパピ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。

<sup>41</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

```

(cond ((= count 0) b)
      ((even? count)
       (fib-iter a
                  b
                  ;;; 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 最大公約数

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。

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。

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

引き続いでも 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、

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

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、 $\text{GCD}$  の値はもう 1 つの値です。この  $\text{GCD}$  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>42</sup>

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

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。

ユーク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 の定理:** もしユークリッドの互除法があるペアの  $\text{GCD}$  を求めのに  $k$  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>43</sup>

---

<sup>42</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

<sup>43</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 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

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:** 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 例: 素数判定

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。

#### 約数を探す

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。

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。

```
(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))
```

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)))
```

`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>44</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。

## フェルマー・テスト

$\Theta(\log n)$  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>45</sup>

**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。

---

<sup>44</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>45</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、同様の証明がライプ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) *Arithmetica* 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一つとなりました。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。

(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  $n$  に関する合同) と呼ばれます。また  $a$  を  $n$  で割った時の余りは  $a$  modulo  $n$  の*remainder*(剩余)、または単純に  $a$  modulo  $n$  と呼ばれます。)

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  $a^n$  modulo  $n$  の剩余を求めます。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。

フェルマーテストを実装するには(ある数値の指數関数 modulo 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))))
```

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”指數”引数の対数になります。<sup>46</sup>

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

---

<sup>46</sup>指數  $e$  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  $m$ に関する剩余を求ることで( $x$ 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  $m$ に関する剩余を得ます。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)

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)))))
```

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

## 確率的手法

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。

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、 $a^n \equiv n \pmod{a}$  となる  $a$  に合同であるような数値  $n$ 。そのよ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>47</sup>

---

<sup>47</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つていません。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

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。

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>48</sup>

**Exercise 1.21:** `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:** 多くの Lisp 実装は `runtime` と呼ばれるプリミティブな手続を持っ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った時間を表示する。

```
(define (timed-prime-test n)
```

---

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>48</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で現実的な適用例を持つことが判明しました。

```

(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))

```

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:** 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が予測される。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:** [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積もるだろうか？ 実際との相違をどのように説明できるか？

**Exercise 1.25:** Alyssa P. Hacker 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))
```

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 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))))
```

“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:** 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:** 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  $n$  に対して 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  $n$  に対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ような方法で  $a^{n-1}$  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 高階手続による抽象の形式化

私達はここまでで手続が事実上、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))
```

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)
```

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、その後抽象化の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。

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ることができるか、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

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))))
```

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))))
```

3つ目は以下の級数の一連の項の合計を計算します。

$$\frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \dots,$$

これは  $\pi/8$  に (とてもゆっくりと) 収束します。<sup>49</sup>

```
(define (pi-sum a b)
  (if (> a b)
    0
    (+ (/ 1.0 (* a (+ a 2)))
        (pi-sum (+ a 4) b))))
```

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))))
```

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、

$$\sum_{n=a}^b f(n) = f(a) + \dots + f(b),$$

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。

---

<sup>49</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。

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))))
```

`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。例えばそれを(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))
```

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
```

`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))
```

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
```

`pi-sum` も同様に定義可能です。<sup>50</sup>

---

<sup>50</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

```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
```

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
```

`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。

$$\int_a^b f = \left[ f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \dots \right] dx$$

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)
.2499998750000001
```

(`cube` の 0 から 1 の実際の定積分の値は  $1/4$  です。)

**Exercise 1.29:** 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

---

有用ではなかろうためです。それらを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。

$b$  の間の  $f$  の定積分は次のように近似される。

$$\frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \dots + 2y_{n-2} + 4y_{n-1} + y_n),$$

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:** 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 `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>51</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>52</sup>

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \dots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \dots}.$$

---

<sup>51</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>52</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。

- b 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. `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` 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. 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:** 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  $a$  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 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 lambda を用いた手続の構築

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っ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))))
```

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))
```

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))
```

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>)
```

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))
```

上記は以下と等価です。

```
(define plus4 (lambda (x) (+ x 4)))
```

lambda式は以下のように読むことができます。

(lambda (x) (+ x 4))  
| | | |  
手続は 引数xを持ち 足す x と 4

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

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>53</sup>

## ローカル変数使用のため let を用いる

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。

$$f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y),$$

これは以下のようにも表現できます。

$$\begin{aligned} a &= 1 + xy, \\ b &= 1 - y, \\ f(x, y) &= xa^2 + yb + ab. \end{aligned}$$

---

<sup>53</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λ-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λ計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。λ計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。

$f$  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。

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (f-helper (+ 1 (* x y))
            (- 1 y)))
```

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)))
```

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))))
```

`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>)
```

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。)

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>)
```

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。

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 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)
```

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。

- 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))
```

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。

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)))
```

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>54</sup>

**Exercise 1.34:** 以下の手続を定義したとする。

```
(define (f g) (g 2))
```

すると以下の結果を得る。

```
(f square)
4
(f (lambda (z) (* z (+ z 1))))
```

もし(天邪鬼にも) インタプリタに `(f f)` の組み合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>54</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實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 汎用手法としての手続

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。

#### 半区間手法により方程式の根を求める

*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  $x$  の間に持ります。もし  $f(x) < 0$  なら  $f$  は 0 を  $x$  と  $b$  の間に持ります。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)))))))
```

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。

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>55</sup>

```
(define (close-enough? x y) (< (abs (- x y)) 0.001))
```

search は直接利用するのは扱いにくいでs。 $f$  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 手続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>56</sup>

```
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (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))))))
```

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
```

---

<sup>55</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>56</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。

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
```

## 関数の不動点を求める

数値  $x$  は  $x$  が等式  $f(x) = x$  を満たす時、関数  $f$  の *fixed point*(不動点) 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ができます。

$$f(x), \quad f(f(x)), \quad f(f(f(x))), \quad \dots,$$

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))
```

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>57</sup>

```
(fixed-point cos 1.0)
.7390822985224023
```

---

<sup>57</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。

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
```

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方を基にしています。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>58</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ができます。

```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y)) 1.0))
```

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。

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。

```
(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y))) 1.0))
```

( $y = \frac{1}{2}(y + x/y)$  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。得るために等式の両辺に  $y$  を足し、2 で割ります。)

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:** 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を示せ。この比率を用いて  $\varphi$  を fixed-point 手続を用いて求めよ。

---

<sup>58</sup>→ (“maps to”(写す) と読みます。) は数学者による 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。

**Exercise 1.36:** `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 無限 *continued fraction*(連分数) とは以下の形式の式である。

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \dots}}}.$$

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。

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{\ddots + \cfrac{N_k}{D_k}}}.$$

`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)
```

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 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:** 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:** 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。

$$\tan x = \frac{x}{1 - \frac{x^2}{3 - \frac{x^2}{5 - \dots}}},$$

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 返り値としての手続

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。

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。

平均減衰の考え方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` 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) 手續を返します。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` 手續に適用した時、生成された手續の値は、ある値 `x` において  $x$  と  $x^2$  の平均となります。この結果の手續に 10 を適用すれば 10 と 100 の平均として 55 を返します。<sup>59</sup>

```
((average-damp square) 10)  
55
```

`average-damp` を用いて `square-root` 手續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))
```

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。これらの手續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` 手續を立方根を求める手續に汎化することが可能です。<sup>60</sup>

```
(define (cube-root x)  
  (fixed-point (average-damp (lambda (y) (/ x (square y))))  
 1.0))
```

---

<sup>59</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—高階手續により値として返されることで得られた手續をいつ適用するのかについて学び始めます。

<sup>60</sup> さらなる一般化については Exercise 1.45 を参照して下さい。

## ニュートン法

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  $x \mapsto f(x)$  の不動点となります。

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

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の程式の解の近似を求めます。<sup>61</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>62</sup>

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)

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}.$$

従って微分の考えを( $dx$  を例えれば 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)))
```

次の定義と一緒に用います。

```
(define dx 0.00001)
```

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。

<sup>61</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を用いることで手法の説明を平易にできます。

<sup>62</sup> ニュートン法は常に解へと収束しません。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。そのような場合で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。

```
(define (cube x) (* x x x))
((deriv cube) 5)
75.00014999664018
```

`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))
```

`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、 $x$  の平方根を見つける時、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>63</sup>

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))
```

## 抽象化と一級手続

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))
```

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。

---

<sup>63</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。

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))
```

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))
```

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。

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。

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>64</sup>

- 変数により名付けることが可能
- 手続に対し引数として渡すことが可能

---

<sup>64</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
- データ構造に含まれることが可能<sup>65</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>66</sup>

**Exercise 1.40:** `newtons-method` 手続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)
```

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:** 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？

```
((double (double double)) inc) 5)
```

**Exercise 1.42:**  $f$  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:**  $f$  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<sup>65</sup>  $f(f(\dots(f(x))\dots))$  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ぬ。

<sup>65</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>66</sup> 第一級手続の主な実装コストは、手続が値として返る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。

```
((repeated square 2) 5)  
625
```

ヒント：[Exercise 1.42](#)の `compose` を使うと便利でしょう。

**Exercise 1.44:** 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:** [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:** この章にて説明された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` 手續を `iterative-improve` を用いて書き直せ。

# 2

## データを用いた抽象化の構築

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*

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。また高階手續が、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。

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続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。

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。

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。多くの分数に対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。

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。

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)))
```

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)))
```

`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>1</sup>

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

---

<sup>1</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。

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。

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“手續”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。

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異なる演算処理により計算され得ると同様に、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。

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(データ適従プログラミング)を個別のデータ表現に対し分離した設計とadditively(付加的)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 データ抽象化のイントロダクション

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、手續がどのように使われるかをその手續がどのようによりプリミティブな手續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。

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 例: 分数のための数値演算命令

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。

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 <n> <d>`) 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 <x>`) は分数  $\langle x \rangle$  の分子を返す。

- (`(denom <x>)`) は分数  $\langle x \rangle$  の分母を返す。

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。

$$\begin{aligned}\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}, \\ \frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}, \\ \frac{n_1}{d_1} \cdot \frac{n_2}{d_2} &= \frac{n_1 n_2}{d_1 d_2}, \\ \frac{n_1/d_1}{n_2/d_2} &= \frac{n_1 d_2}{d_1 n_2}, \\ \frac{n_1}{d_1} &= \frac{n_2}{d_2} \quad \text{if and only if } n_1 d_2 = n_2 d_1.\end{aligned}$$

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)))))
```

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。

## ペア

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、プリミティブな手続 `car` と `cdr` を用いてその部分を抽出することができます。<sup>2</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
```

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
```

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

---

<sup>2</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。

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。

## 分数を表現する

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>3</sup>

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
```

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>4</sup>

```
(define (print-rat x)
  (newline))
```

---

<sup>3</sup>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挙げられます。

```
(define make-rat cons)
(define numer car)
(define denom cdr)
```

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。

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。

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>4</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)))
```

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
```

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))))
```

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
```

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。

引数を扱えるより良い版の `make-rat` を定義せよ。**Exercise 2.1:** 正と負の両方の `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 抽象化バリア

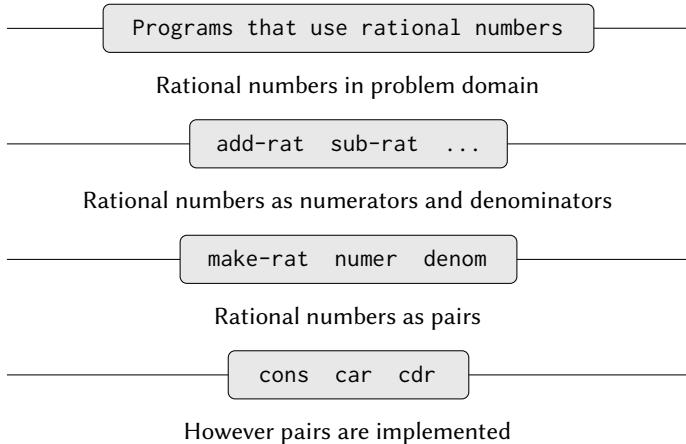
さらなる複合データとデータ抽象化の例を続ける前に、分数の例にて持ち上がったいくつかの問題について考えてみましょう。私達は分数操作をコンストラクタ `make-rat` とセレクタ `numer` と `denom` を用いて定義しました。一般的にデータ抽象化の基となる考えはデータオブジェクトの各型に対し、その型のデータオブジェクトの全ての操作が表される命令を用いて、命令の基本的な集合を判断し、そのデータを操作する時にそれらの命令のみを用いることです。

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。

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。

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))))
```



**Figure 2.1:** 分数パッケージ内の抽象化バリア

```

(/ (car x) g)))
(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g)))
    
```

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、手続き、`add-rat`, `sub-rat`, その他は全く変更する必要がありません。

表現上の依存対象を少ないインターフェイス手続き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:** 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:** 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 データにより何が意味されるのか

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。

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` で割のとの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。

$$\frac{(\text{numer } x)}{(\text{denom } x)} = \frac{n}{d}.$$

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>5</sup>

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 (`cons x y`) であるなら (`car z`) は `x` であり、(`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

---

<sup>5</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。

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))
```

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續が上で与えられた制約を満たすことです。

注意すべき微細な点は `(cons x y)` により返される値は手續—すなわち内部で定義された手續 `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)` により作られた手續であるのなら、`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續を 1 に適用し、`y` を返します。従ってこのペアの手續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。

ペアの手續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。手續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。この例はまた手續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ませんが、しかし手續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 にてモデリングとシ

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:** 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)))
```

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:** 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:** 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、手続を操作可能で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ともやつていける(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)))))
```

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。`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 延長課題: 区間演算

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。

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。

$$R_p = \frac{1}{1/R_1 + 1/R_2}.$$

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 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 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、データ構造 `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 つの区間を足す手順を書きました。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 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))))
```

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。以下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))
```

セレクタ `upper-bound` と `lower-bound` を定義し実装を完成させよ。

**Exercise 2.8:** Alyssaの考え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:** 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と、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:** 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:** 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。

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で直した。

```
(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))
```

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:** 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:** 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。

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。

$$\frac{R_1 R_2}{R_1 + R_2}$$

と、

$$\frac{1}{1/R_1 + 1/R_2}.$$

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)))

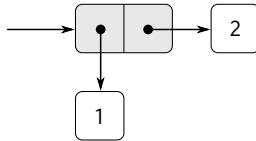
(define (par2 r1 r2)
  (let ((one (make-interval 1 1)))
    (div-interval
      one (add-interval (div-interval one r1)
                         (div-interval one r2)))))
```

Lem 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:** 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 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:** 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)



**Figure 2.2:** (`cons 1 2`) の箱とポインタ表現

## 2.2 階層データと閉包性

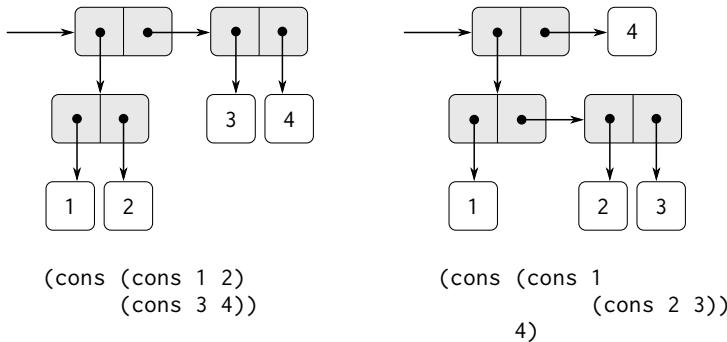
ここまで学んだように、ペアは私達が複合データオブジェクトを構築するのに利用可能なプリミティブな“糊”を提供します。Figure 2.2はペア—この場合は(`cons 1 2`)にて形成されたペアを図示する標準的な方法を示しています。この*box-and-pointer notation*(箱と点表記法)と呼ばれる表現において、各オブジェクトは箱へのpointer(ポインタ)として表わされています。プリミティブオブジェクトの箱はオブジェクトの表現を持っています。例えば数値の箱は数字を持っています。ペアの箱は実際には二重の箱で、左部分はペアの `car`(へのポインタ)を持っており、右部分は `cdr` を持っています。

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では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プ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。

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>6</sup>閉包はどのような目的の組み合わせをも強力にする鍵となります。なぜなら *hierarchical*(階層)構造—複数のパートから成る構造

---

<sup>6</sup>ここでの“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つ目の意味では用いません。



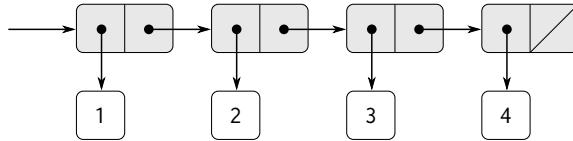
**Figure 2.3:** ペアを用いて 1,2,3,4 を組み合わせる 2 つの方法

であり、ペア自身も複数のペア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。

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>7</sup>

---

<sup>7</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せん。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。”



**Figure 2.4:** ペアの鎖として表現された列 1, 2, 3, 4

### 2.2.1 列の表現

ペアを用いて構築可能な便利な構造の 1 つが *sequence(列)*—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* は鎖内で相対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))))
```

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を手助けします。<sup>8</sup> 上の列は (*list* 1 2 3 4) により生成可能です。

```
(list <a1> <a2> ... <an>)
```

is equivalent to

は以下と等価です。

```
(cons <a1>
  (cons <a2>
```

---

<sup>8</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。

```
(cons ...
  (cons ⟨an
```

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を表します。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)
```

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。

`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>9</sup>

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)
(2 3 4)
(car (cdr one-through-four))
2
(cons 10 one-through-four)
(10 1 2 3 4)
```

---

<sup>9</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。

```
(cons 5 one-through-four)
(5 1 2 3 4)
```

codenil 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>10</sup>

## リスト命令

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。

- *n* = 0 の場合、`list-ref` はリストの `car` を返す。
- 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

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?` の使用的典型的なパターンを説明します。

---

<sup>10</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カーです。`(true` が普通の変数であり、真の値を持つとのと同様です)。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。

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))
(define odds (list 1 3 5 7))
(length odds)
4
```

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。

- 任意のリストの `length` はリストの `cdr` の `length` に 1 を足した値

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

- 空リストの `length` は 0

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))
```

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` 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。

- もし `list1` が空リストであれば、結果は単に `list2`
- 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:** 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:** 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:** 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。

`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にしたいと考える。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))
```

次に `cc` を以下のように呼び出す。

```
(cc 100 us-coins)
292
```

これを行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))))))
```

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-denomination, 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:** 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))
```

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)
```

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>)
```

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)
```

`g` のボディでは `w` はリスト `(1 2 3 4 5 6)` となる。<sup>11</sup>

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)
```

## リストに渡る `map`

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)
```

---

<sup>11</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>))
```

私達はSection 1.3のように、この一般的な考えを抽象化し、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` と呼ばれます。`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>12</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)
```

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` 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

---

<sup>12</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)
```

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つ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:** T 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)
```

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 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))
```

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 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))
```

これもまたうまく行かない。説明せよ。

**Exercise 2.23:** 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
```

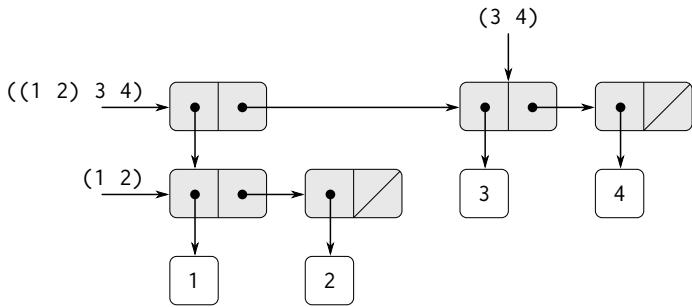
(上では示されていない) `for-each` 呼出による返り値は真のような不定な何かである。`for-each` の実装を与える。

## 2.2.2 階層構造

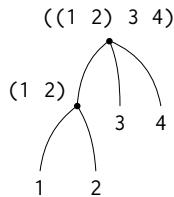
リストを用いた列の表現は要素が列自身である列を表現することを自然に一般化する。例えば以下の様に構築されたオブジェクト `((1 2) 3 4)` を

```
(cons (list 1 2) (list 3 4))
```

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



**Figure 2.5:** (`cons (list 1 2) (list 3 4)`) により形作られた構造



**Figure 2.6:** 木として見たFigure 2.5のリスト構造

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。

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。

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

```

`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。

- リスト `x` の `length` は `x` の `cdr` の `length` に 1 を足した物
- 空リストの `length` は 0

`count-leaves` も同様えす。空リストの値は同じで

- 空リストの `count-leaves` は 0

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

- 木 `x` の `count-leaves` は `x` の `car` の `count-leaves` と、`x` の `cdr` の `count-leaves` の和

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。

- 葉の `count-leaves` は 1

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。以下に完成した手續を置きます。<sup>13</sup>

```

(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                  (count-leaves (cdr x)))))))

```

---

<sup>13</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:** 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:** 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:** 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))
```

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:** Exercise 2.18の 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:** 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:** バイナリモバイル (binary mobile)<sup>14</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))
```

枝は `length`(数値であること) と `structure` から組み立てられ、`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))
```

- 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。
- 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。
- 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。

---

<sup>14</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。

d 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と考える。

```
(define (make-mobile left right) (cons left right))
(define (make-branch length structure)
  (cons length structure))
```

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?

## 木に渡る map

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))
```

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))
```

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:** 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))
```

`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:** 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:** 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 (??) rest)))))
```

### 2.2.3 慣習的インターフェイスとしての列

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。

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` 手続に類似しており、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)))))))
```

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。

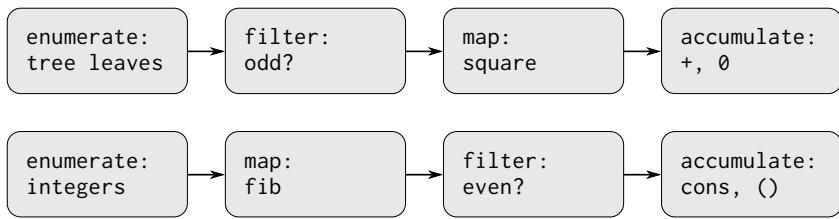
```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

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

- 木の葉を列挙する
- フィルタを通して奇数のみを選ぶ
- 選択された数の二乗を求める
- 初期値 0 にて + を用いて集積する。

2 つ目のプログラムは

- 0 から  $n$  を列挙する
- 各整数のフィボナッチ数を求める



**Figure 2.7:** 手続 `sum-odd-squares`(上) と `even-fibs`(下) の信号の流れの計画が二つのプログラムの間の共通点を明かす

- フィルタを通して偶数を選択する
- 初期値は空リストにて `cons` を用いて結果を集積する

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。

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のは失敗しています。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。

## 列命令

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)
```

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))))))
```

例として、

```
(filter odd? (list 1 2 3 4 5))  
(1 3 5)
```

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)
```

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)
```

木の葉を列挙するには、以下の様にして可能です。<sup>15</sup>

```
(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree)))))))
(enumerate-tree (list 1 (list 2 (list 3 4)) 5))
(1 2 3 4 5)
```

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)))))
```

`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)))))
```

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

---

<sup>15</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。

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。

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に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)
```

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
```

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たいとします。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))))
```

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>16</sup>

---

<sup>16</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

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:** 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:**  $x$  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。

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

ホーナー法は上記の計算を以下のような構造にする。

$$(\dots (a_n x + a_{n-1}) x + \dots + a_1) x + a_0.$$

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>17</sup>

---

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。

<sup>17</sup>Knuth 1981によるこの方法は 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

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))
```

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。

```
(horner-eval 2 (list 1 3 0 5 0 1))
```

**Exercise 2.35:** 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:** 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、 $(\text{accumulate}-n\ +\ 0\ s)$  の値は列  $(22\ 26\ 30)$  にならなければならない。以下の  $\text{accumulate}-n$  の定義の欠けた式を補え。

```
(define (\text{accumulate}-n op init seqs)
  (if (\text{null}?) (\text{car} seqs))
      nil
```

---

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。

```
(cons (accumulate op init (??))
  (accumulate-n op init (??))))
```

**Exercise 2.37:** 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、

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{pmatrix}$$

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。

<pre>(dot-product v w) (matrix-*-vector m v) (matrix-*-matrix m n) (transpose m)</pre>	<pre>returns the sum <math>\sum_i v_i w_i</math>, returns the vector <math>\mathbf{t}</math>, where <math>t_i = \sum_j m_{ij} v_j</math>, returns the matrix <math>\mathbf{p}</math>, where <math>p_{ij} = \sum_k m_{ik} n_{kj}</math>, returns the matrix <math>\mathbf{n}</math>, where <math>n_{ij} = m_{ji}</math>.</pre>
--	---

ドット積を次のように定義できる。<sup>18</sup>

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

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。)

```
(define (matrix-*-vector m v)
  (map (??) m))
(define (transpose mat)
  (accumulate-n (??) (??) mat))
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (??) m)))
```

---

<sup>18</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

**Exercise 2.38:** `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))
```

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))
```

`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:** 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))
```

## 入れ子の `map`

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>19</sup> 次の問題について考えてみて下さい：正の整数  $n$  を

<sup>19</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。

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

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。

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>20</sup>

```
(accumulate
  append nil (map (lambda (i)
    (map (lambda (j) (list i j))
      (enumerate-interval 1 (- i 1))))
    (enumerate-interval 1 n)))
```

`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。

```
(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))
```

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になります。

```
(define (prime-sum? pair)
  (prime? (+ (car pair) (cadr pair))))
```

---

<sup>20</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。

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))))
```

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)))))
```

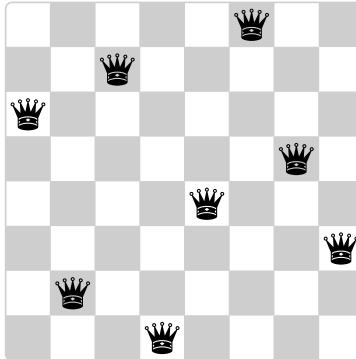
入れ子の map は区間を列挙するもの以外の列に対しても便利です。ある集合  $S$  の全ての順列を生成したいとします。つまり集合内の項目の全ての並べ方です。例えば  $\{1, 2, 3\}$  の順列は  $\{1, 2, 3\}$ ,  $\{1, 3, 2\}$ ,  $\{2, 1, 3\}$ ,  $\{2, 3, 1\}$ ,  $\{3, 1, 2\}$ , and  $\{3, 2, 1\}$  です。ここに集合  $S$  の順列を生成するための計画を上げます :  $S$  中の全て項目  $x$  に対し再帰的に  $S - x$  の順列の列を生成し、<sup>21</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>22</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)))
```

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。境界条件に關

<sup>21</sup>集合  $S - x$  は  $S$  の全ての要素から  $x$  を除いた集合

<sup>22</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。



**Figure 2.8:** 8 クイーンパズルの解の一例

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:** 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:** 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:**

“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。

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)
    (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)))
```

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

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 は Exercise 2.42 を行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持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)))
```

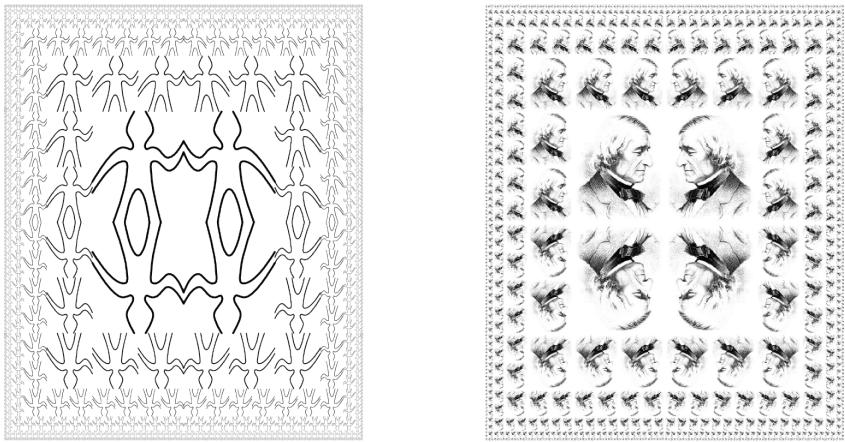
この交換がなぜプログラムの実行を遅くするのか説明せよ。Louis のプログラムが 8 クイーンパズルを解くのにどれだけの時間がかかるか推察せよ。Exercise 2.42 のプログラムが同じパズルを解くのに必要な時間が  $T$  であるとの前提で行え。

## 2.2.4 例: ピクチャー言語

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るよう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>23</sup> こ

---

<sup>23</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



**Figure 2.9:** ピクチャー言語を用いて生成したデザイン

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。

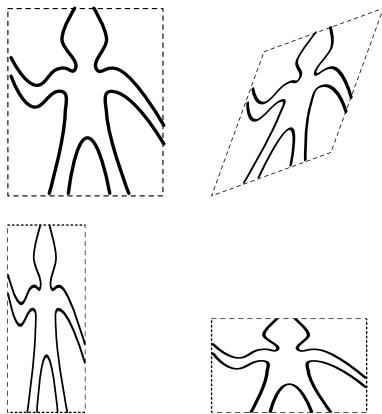
### この本のピクチャー言語

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。

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

---

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` 手続を用いて描かれた配置と似ております。



**Figure 2.10:** `wave` により生成されたイメージ

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>24</sup> Figure 2.11 の 4 つのイメージ

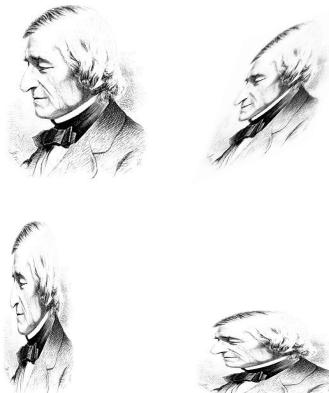
<sup>24</sup>William Barton Rogers (1804-1882) は MIT の創始者であり、かつ初代学長です。地質学者であり、才能溢れる教師である彼は William and Mary College と University of Virginia にて教鞭を取りました。1859 年に彼はボストンへ移りそこでより研究に打ち込み、“技術専門の研究所”を設立する計画を進めました。またマサチューセッツ州の最初のガスマータの州検査官も務めました。

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。彼に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。

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 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 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。



**Figure 2.11:** William Barton Rogers のイメージ

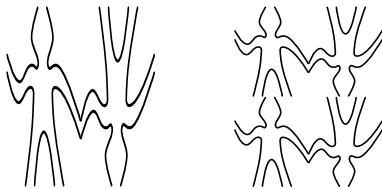
はFigure 2.10の `wave` のイメージと同じ 4 つのフレームを考慮して描かれてています。

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  
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。

“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とした小論文を  
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でいて、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。

Francis A. Walker(MIT の三代目の学長)の言葉では

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 2.12:** Figure 2.10の wave ペインタから始めて複雑な図を作成する

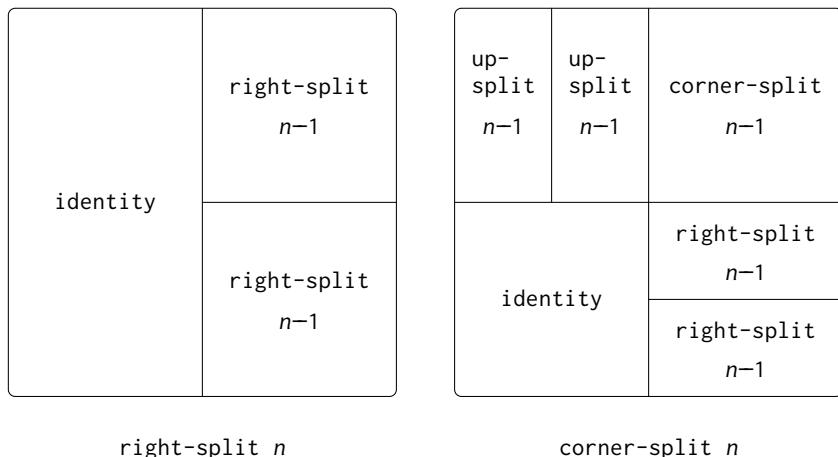
い最初のペインタのイメージをフレームの左半分に、2つ目のペインタのイメージをフレームの右半分に描く複合ペインタを生成します。同様に `below` は2つのペインタを取り、1つ目のペインタのイメージを2つ目のペインタのイメージの下に描きます。いくつかの命令は単一のペインタを変換し新しいペインタを生成します。例えば `flip-vert` はペインタを取りそのイメージを上下逆さに描くペインタを生成し、`flip-horiz` は元のペインタのイメージを左右逆に描くペインタを生成します。

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))
```

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 `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。

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ーランゲージ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。



**Figure 2.13:** right-split と corner-split の再帰計画

```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))
```

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))
```

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)))))
```

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)))))))
```

`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。

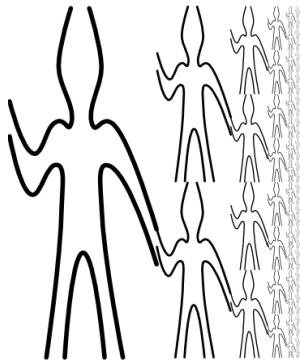
```
(define (square-limit painter n)
  (let ((quarter (corner-split painter n)))
    (let ((half (beside (flip-horiz quarter) quarter)))
      (below (flip-vert half) half))))
```

**Exercise 2.44:** `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。

## 高階命令

ペインタ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タ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。

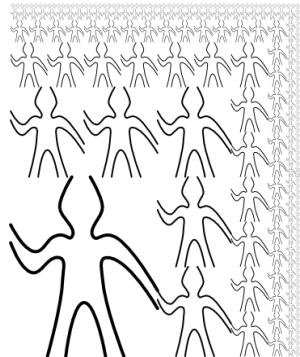
例として、`flipped-pairs` と `square-limit` はペインタのイメージを四角のパターン内に 4 つのコピーをそれぞれが準備します。それらはどのような位置と向きに置くかということのみにおいて異なります。このペインタ接続のパターンを抽象化する 1 つの方法は以下のプロシージャを用いて、4 つの 1 引数ペインタ命令を取り与えられたペインタをそれら 4 つの命令で変換する命令を生成し、結果を四角の中に配置します。`tl`, `tr`, `bl`, and `br` は左上、右上、左下、右下のコピーに対応する変換です。



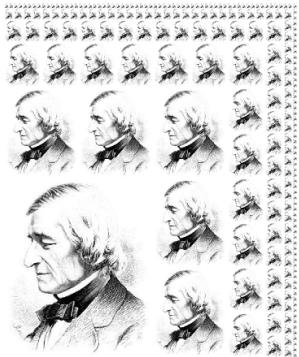
(right-split wave 4)



(right-split rogers 4)



(corner-split wave 4)



(corner-split rogers 4)

**Figure 2.14:** ペインタ wave と rogers に適用された再帰命令 right-split と corner-split。4つの図 corner-split を組み合わせることでFigure 2.9で示された対照的な square-limit のデザインを生成する。

```
(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))))
```

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>25</sup>

```
(define (flipped-pairs painter)
  (let ((combine4 (square-of-four identity flip-vert
                                    identity flip-vert)))
    (combine4 painter)))
```

そして `square-limit` は以下の様に表現可能です。<sup>26</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` 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))
```

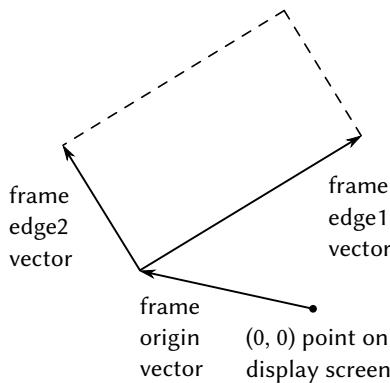
既に定義済みのものと全く同じ振舞を行う手続 `right-split` と  
`up-split` を生成するよう定義せよ。

---

<sup>25</sup> 同等に、こうも書けます。

```
(define flipped-pairs
  (square-of-four identity flip-vert identity flip-vert))
```

<sup>26</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。



**Figure 2.15:** 3 つのベクトル — 始点と 2 つの辺により記述されたフレーム

## フレーム

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 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はありません。[\(Exercise 2.47\) を参照して下さい。](#)

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。

$$\text{Origin(Frame)} + x \cdot \text{Edge}_1(\text{Frame}) + y \cdot \text{Edge}_2(\text{Frame}).$$

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>27</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))))))
```

`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))
```

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:** 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-vect` と関連するセレクタ `xcor-vect` と `ycor-vect` 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-vect`, `sub-vect`, `scale-vect` を実装せよ。

$$\begin{aligned} (x_1, y_1) + (x_2, y_2) &= (x_1 + x_2, y_1 + y_2), \\ (x_1, y_1) - (x_2, y_2) &= (x_1 - x_2, y_1 - y_2), \\ s \cdot (x, y) &= (sx, sy). \end{aligned}$$

---

<sup>27</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。

**Exercise 2.47:** 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)))
```

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。

## ペインタ

ペインタはフレームを引数として与え、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。

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>28</sup>

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
      (lambda (segment)
        (draw-line
          ((frame-coord-map frame)
           (start-segment segment)))
        ((frame-coord-map frame)
         (end-segment segment))))))
  segment-list)))
```

---

<sup>28</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` を使ってています。

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。

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>29</sup>

**Exercise 2.48:** 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:** `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 指定したフレームの外枠を描くペインタ
- b フレームの反対の角を繋いで“X”を描くペインタ
- c 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 `wave` ペインタ

## 変形とペインタの組み合わせ

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く

<sup>29</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。

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。

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)))))))
```

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
```

`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)))
```

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>30</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)))
```

イメージをフレームの中心に向けて潰したりします。<sup>31</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)))
```

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
```

---

<sup>30</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>31</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 1.0 0.0)
(make-vect 0.5 1.0)))
(lambda (frame)
(paint-left frame)
(paint-right 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` 手續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:** 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:** 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) 回転命令を利用せよ。

## 堅牢な設計のための言語のレベル

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、抽象化手續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。

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。

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>32</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。

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。

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:** 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 Exercise 2.49 のプリミティブな `wave` ペインタにいくつか線分を加えよ。(例えば笑顔を追加せよ)
- b `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)

---

<sup>32</sup>Section 3.3.4がそのような言語について記述します。

c `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 記号データ

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 クオート

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))
```

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)))))
```

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>33</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>34</sup>

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)
(a b)
(list 'a b)
(a 2)
```

---

<sup>33</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>34</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。

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なります。<sup>35</sup>

```
(car '(a b c))  
a  
(cdr '(a b c))  
(b c)
```

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。

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>36</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))))))
```

例えば、次の式の値は

---

<sup>35</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単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。

<sup>36</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。

```
(memq 'apple '(pear banana prune))
```

falseになります。そして次の式の値は

```
(memq 'apple '(x (apple sauce) y apple pear))
```

(apple pear)です。

**Exercise 2.53:** 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:** 2つのリストはそれぞれが同じ要素を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))
```

は真であるが

```
(equal? '(this is a list) '(this (is a) list))
```

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。この考え方を用いて equal? を手続として実装せよ。<sup>37</sup>

**Exercise 2.55:** Eva Lu Ator はインタプリタに次の式を入力した。

---

<sup>37</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。

```
(car ''abracadabra)
```

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 例: 記号微分

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。

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。

#### 抽象データを用いた微分プログラム

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。

$$\begin{aligned}\frac{dc}{dx} &= 0, \quad c \text{ は定数か、 } x \text{ と異なる変数,} \\ \frac{dx}{dx} &= 1, \\ \frac{d(u+v)}{dx} &= \frac{du}{dx} + \frac{dv}{dx}, \\ \frac{d(uv)}{dx} &= u \frac{dv}{dx} + v \frac{du}{dx}.\end{aligned}$$

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最初に項の導関数を求めて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。

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

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))))
```

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。

## 代数式を表現する

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。

- 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))
```

- 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)))
```

- 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))
```

- 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) '+)))
```

- 加数は和のリストの二つ目の項だ。

```
(define (addend s) (cadr s))
```

- 被加数は和のリストの三つめの項だ。

```
(define (augend s) (caddr s))
```

- 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) '*)))
```

- 乗数は積のリストの二つ目の項だ。

```
(define (multiplier p) (cadr p))
```

- 被乗数は積のリストの三つめの項だ。

```
(define (multiplicand p) (caddr p))
```

従って記号微分プログラムを得るためにには、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)))
```

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す。

$$\frac{d(xy)}{dx} = x \cdot 0 + 1 \cdot y,$$

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。

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))))
```

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)))
```

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))))
```

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)
y
(deriv '(* x y) 'x)
y
(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* x y) (* y (+ x 3)))
```

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:** 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。

$$\frac{d(u^n)}{dx} = n u^{n-1} \frac{du}{dx}$$

`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:** 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)
```

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:** 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。

- $(x + (3 * (x + (y + 2))))$  の様な接中辞で表される代数式を微分することをどのように行うのか示せ。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。
- $(x + 3 * (x + y + 2))$  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？

### 2.3.3 例: 集合を表現する

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。

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>38</sup>

## 順序無しリストとしての集合

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)))))
```

---

<sup>38</sup>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。

任意の集合 `S` と任意のオブジェクト `x` に対し、`(element-of-set? x (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”）

任意の集合 `S` と `T` と任意のオブジェクト `x` に対し、`(element-of-set? x (union-set S T))` は `(or (element-of-set? x S) (element-of-set? x T))` に等しい（非公式には“(union S T) の要素は S または T に存在する要素”）

任意のオブジェクト `x` に対し `(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”）

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。

```
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))
```

`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))))
```

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:** 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:** 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 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續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？

## 順序有りリストとしての集合

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。例えば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。

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、集合全体をスキヤ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)))))
```

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。しかし一方でもし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。

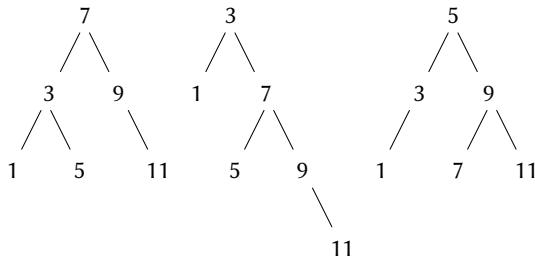
平均では集合の項目数の半分近くを試験しなければいけないことが期待できるはずです。従って必要とされる平均のステップ数は約  $n/2$  になります。これはそれでも  $\Theta(n)$  で増加しますが、以前の実装に対して平均的にはステップ数において半分に節約します。

`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。そうではなく `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)))))))
```

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  $\Theta(n^2)$  ではなく、 $\Theta(n)$  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:** 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



**Figure 2.16:** 集合  $\{1, 3, 5, 7, 9, 11\}$  を表現するさまざまな二分木

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。

**Exercise 2.62:** 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。

## 二分木としての集合

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。

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>39</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。

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>40</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))
```

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)))))
```

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 '() '())))


---


```

<sup>39</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>40</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。

```

((= 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))))))

```

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素持っていることになります。しかしそのようにすれば私達が構築した木がバランスが取れている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ればバランスが取れてい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けば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>41</sup>

**Exercise 2.63:** 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。

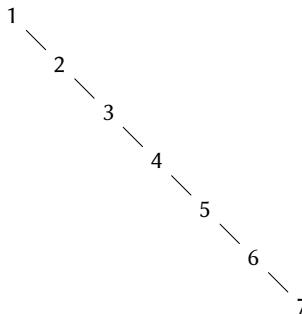
```

(define (tree->list-1 tree)
  (if (null? tree)
      '()
      (append (tree->list-1 (left-branch tree))
              (list (entry tree))
              (tree->list-1 (right-branch tree))))

```

---

<sup>41</sup> そのような構造の例には *B-trees*(B 木) や *red-black trees*(赤黒木) 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。



**Figure 2.17:** 1 から 7 まで順に adjoin することで生成した不均衡な木

```

(define (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 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 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:** 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` が木に含まれなかつ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 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 `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んオーダはいくらか?

**Exercise 2.65:** 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>42</sup>

## 集合と情報検索

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。

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。

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示するためには手続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))))))
```

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

---

<sup>42</sup>Exercise 2.63からExercise 2.65はPaul Hilfingerによるものである。

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可能です。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:** 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。

### 2.3.4 例: ハフマン符号化木

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

この符号を用いて、以下のメッセージは

BACADAEAFABBAAGAH

54 ビットの列として符号化されます。

001000010000011000100000101000001001000000000110000111

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と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

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。

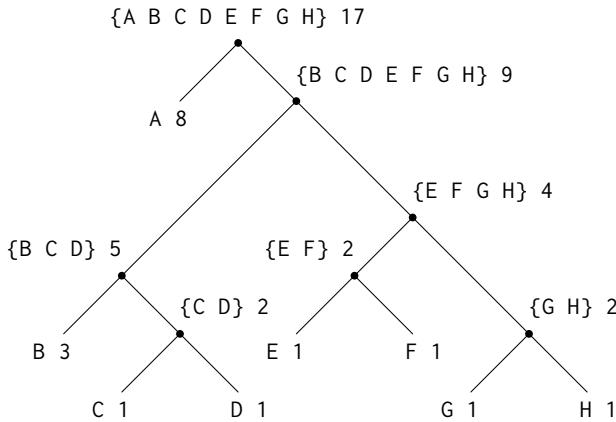
100010100101101100011010100100000111001111

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。

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。

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 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。



**Figure 2.18:** ハフマン符号化木

ハフマン木を与えられることで、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。

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。従ってメッセージ全体は BAC です。

## ハフマン木の生成

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>43</sup>

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します。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うまず重みが最低の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
```

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

---

<sup>43</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。

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)。

## ハフマン木の表現

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。

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))
```

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))))
```

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)
    (cadddr tree)))

```

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。

## 複合化手続

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))))

```

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。

## 重み付き要素の集合

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。

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)))))))
```

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:** 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 0))
```

`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:** `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` 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:** 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` 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:** 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”の“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

`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

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:**  $n$  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:** 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。この質問の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るよ。

## 2.4 抽象データの多重表現

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こ—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。

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。

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。

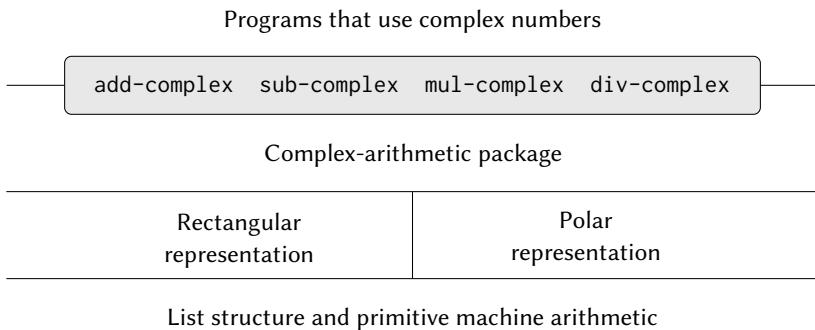
より重要なこととして、プログラミングシステムは時折、長期間に渡り多くの人々が働くことにより設計されるため、長い間仕様変更にさらされます。そのような環境では、データ表現の選択を前もって同意することが誰にとって

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なります。

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。

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、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。

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て用いるかについて示します。



**Figure 2.19:** 複素数システムのデータ抽象化バリエーション

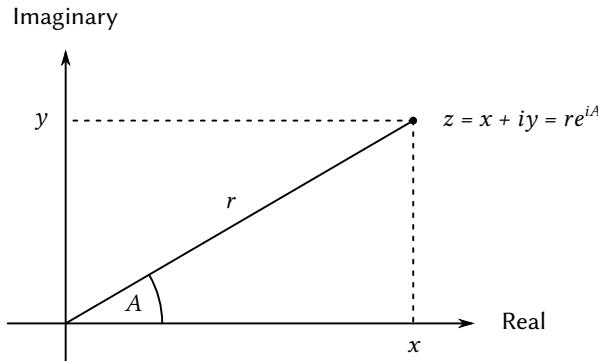
### 2.4.1 複素数のための表現

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 つのもっとも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>44</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。

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>44</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:** 平面上の点としての複素数

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。

$$\begin{aligned} \text{Magnitude}(z_1 \cdot z_2) &= \text{Magnitude}(z_1) \cdot \text{Magnitude}(z_2), \\ \text{Angle}(z_1 \cdot z_2) &= \text{Angle}(z_1) + \text{Angle}(z_2). \end{aligned}$$

$$\begin{aligned} \text{大きさ}(z_1 \cdot z_2) &= \text{大きさ}(z_1) \cdot \text{大きさ}(z_2), \\ \text{角度}(z_1 \cdot z_2) &= \text{角度}(z_1) + \text{角度}(z_2). \end{aligned}$$

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。

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))
```

の両方が `z` に等しい複素数を生成します。

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)))))
```

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?

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。

$$\begin{aligned}x &= r \cos A, & r &= \sqrt{x^2 + y^2}, \\y &= r \sin A, & A &= \arctan(y, x),\end{aligned}$$

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>45</sup> Ben の表現は従っ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)
  (cons (* r (cos a)) (* r (sin a))))
```

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には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))
```

---

<sup>45</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。

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 タグ付きデータ

データ抽象化を考え方の 1 つは“最小責務の原則”的適用としてです。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。

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。

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)))
```

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))
```

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共存す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)))))
```

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)))

```

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))))
```

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。

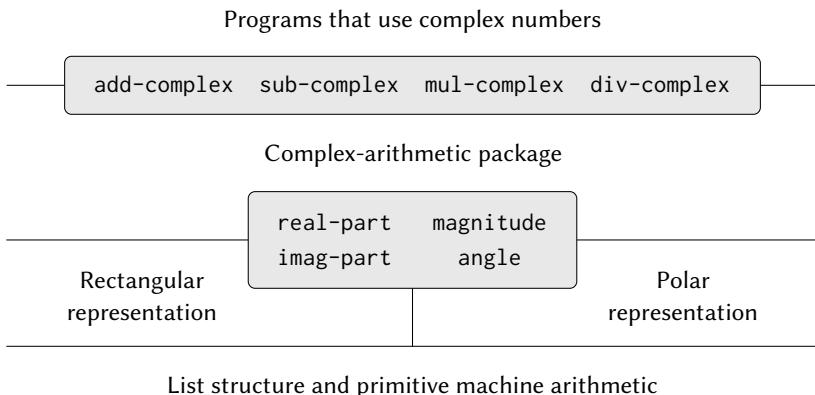
```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2))))
```

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))
```

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できます。

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



**Figure 2.21:** ジェネリックな複素数演算システムの構造

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することができます。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 データ適従プログラミングと付加性

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っている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。

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。

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的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は簡単ですが、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。しかしだけ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。

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。

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。

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: 複素数システムの命令表

- `(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

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。

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 `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)

```

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続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>46</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>47</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 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)))

```

---

<sup>46</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>47</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。

```

(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)))
        (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)

```

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てようとも、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こりません。

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>48</sup>

---

```
(define (apply-generic op . args)
```

48

`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。

`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))
```

は 10 を返します。

```
(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))))))
```

`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))
```

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。

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、実数部と虚数部か大きさ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))))
```

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。このチュートリアルではデータの“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))
```

- 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？
- 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。
- 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。
- 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)
```

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.(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。

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 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 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` 手續を実装せよ。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 本社のために `find-employee-record` 手續を実装せよ。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 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？

## メッセージパッシング

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づ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。

代替的な実装戦略はテーブルを列に分解し、データ型に基き呼び出しを行う“知的命令”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)
```

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>49</sup>

```
(define (apply-generic op arg) (arg op))
```

---

<sup>49</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。

`make-from-real-imag` により返される値は手続—内部手続 `dispatch` の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。

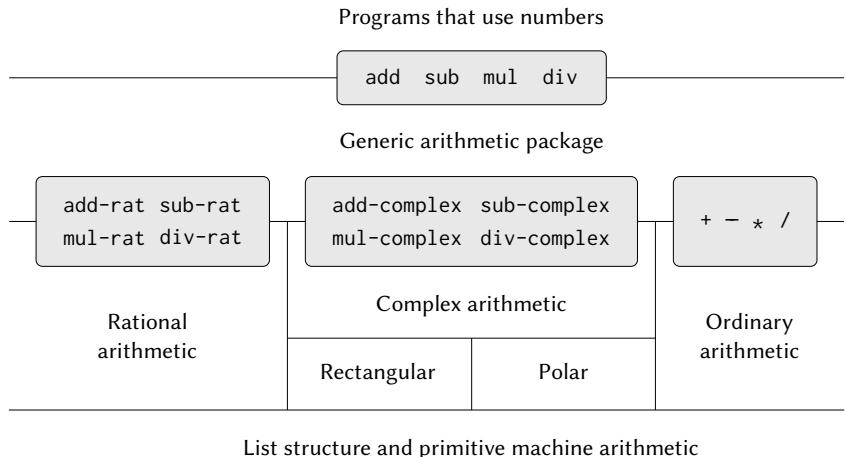
このプログラミングスタイルは *message passing*(メッセージパッシング) と呼ばれます。その名前はデータオブジェクトが要求された命令の名前を“メッセージ”として受け取った要素であるというイメージから来ています。私達は既にメッセージパッシングの例を [Section 2.1.3](#) にて見ています。その時は `cons`, `car`, `cdr` がデータオブジェクト無し、手続のみでどのように定義され得るかを学びました。ここではメッセージパッシングは数学上のトリックではなくジェネリック命令を用いてシステムを構造化するのに便利なテクニックであることを学びます。この章の残りでは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:** 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:** 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ません。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、メッセージパッシング—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 ジェネリック命令を持つシステム

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を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



**Figure 2.23:** ジェネリックな数値演算システム

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 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単一の手続 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) ジェネリックな手續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 ジェネリックな数値演算命令

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。

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))
```

*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)
```

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))
```

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))

```

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)

```

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。

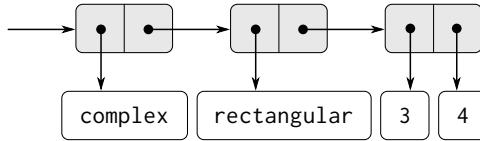
```

(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))

```

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。

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



**Figure 2.24:** 直行形式による  $3 + 4i$  の表現

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 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)
```

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? どの手続が各ケースに対して呼び出されるか?

**Exercise 2.78:** `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:** 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:** 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 異なる型のデータを組み合わせ

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こ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。

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>50</sup>

---

<sup>50</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。

```

;; 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))))

```

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、クロスタイプの命令を実装する手続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プ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。

## 型の強制

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。

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))
```

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)
```

(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。

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>51</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。

```
(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)))
```

---

<sup>51</sup>一般化については Exercise 2.82 を参照して下さい。

```

(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))))))

```

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、型に関係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>52</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。

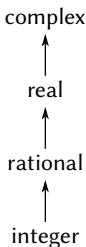
一方で、私達の強制スキームが十分に汎用ではないアプリケーションが存在するかもしれません。たとえ合成されるオブジェクトの両方ともが他方に変換できないとしても両者を第三の型に変換することで命令を実行することが可能になるかもしれません。そのような複雑さに対処するため、そしてそれでもプログラムのモジュール方式を維持するために、通常はより一層タイプ間の関係の構造の利点を得るシステムを構築することが、次で議論するように必要です。

## 型の階層

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

---

<sup>52</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:** 型の塔

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。

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。

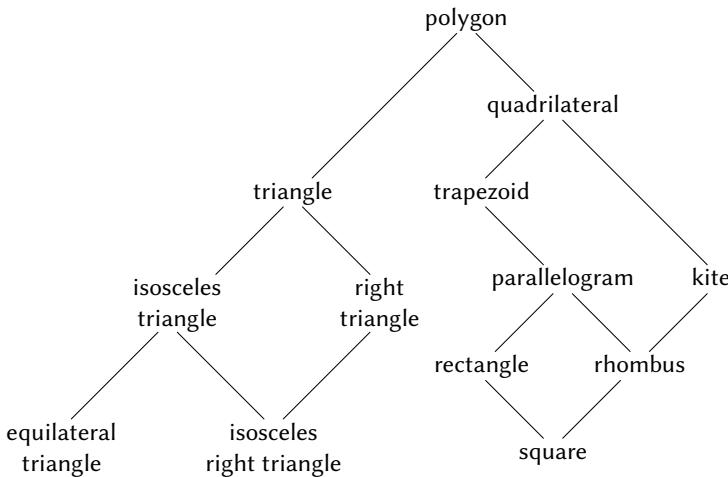
`apply-generic` 手続を以下のように再設計することもできます。各型に対して `raise` 手続を与える必要があります。これはある型のオブジェクトをタワーにおいて 1 レベル上げます。そうすればシステムが異なる型のオブジェクト上にて操作する必要がある時、全てのオブジェクトがタワー内にて同じレベルになるまで連続して上げることができます (Exercise 2.83 と Exercise 2.84 がそのような戦略の実装の詳細について考察しています)。

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ままのスーパー タ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。

別に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)。

## 階層の不十分さ

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、ここまで見えてきた通りに、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](#) は雑多な型のより複雑な配置を図示しています。この図は幾何学的图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あります。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 タイプ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 タイプ問題は特に困難で、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 タイプを求めるることは `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



**Figure 2.26:** 幾何学的図形の型の間の関係

多くの研究領域です。<sup>53</sup>

**Exercise 2.81:** 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

<sup>53</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。実用的で汎用的な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。

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 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))
```

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
```

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 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 *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:** *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:** 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:** 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を行え。

**Exercise 2.85:** この節では可能な限りタワー内の型レベルを下げるこ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>54</sup> また Exercise 2.79 で説明した等価

---

<sup>54</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ます。

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:** 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 例: 記号代数

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

$$x^2 \sin(y^2 + 1) + x \cos 2y + \cos(y^3 - 2y^2)$$

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。

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。

## 多項式の計算

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>55</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、

$$5x^2 + 3x + 7$$

は簡単な  $x$  の多項式であり、

$$(y^2 + 1)x^3 + (2y)x + 1$$

は係数が  $y$  の多項式である  $x$  の多項式です。

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“多項式を純粋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べきでしょうか?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>56</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。

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

<sup>55</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>56</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対応する点の多項式の当たいを足すだけです。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られた場合に  $n$  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。

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。

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))))
```

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)

```

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。

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-term-list` 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-term-list?` 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。

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>57</sup>

```
(define (add-terms L1 L2)
  (cond ((empty-term-list? L1) L2)
        ((empty-term-list? 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))))
                  (else
                    (adjoin-term
                      (make-term (order t1)
                                 (add (coeff t1) (coeff t2)))
                      (add-terms (rest-terms L1)
                                 (rest-terms L2))))))))))
```

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つの項の係数と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。

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となる項を形成します。

```
(define (mul-terms L1 L2)
  (if (empty-term-list? L1)
      (the-empty-term-list)
      (add-terms (mul-term-by-all-terms (first-term L1) L2)
```

---

<sup>57</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。

```

(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)))))))

```

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。

$$[3x^2 + (2 + 3i)x + 7] \cdot \left[ x^4 + \frac{2}{3}x^2 + (5 + 3i) \right].$$

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できます。

$$\left[ (y + 1)x^2 + (y^2 + 1)x + (y - 1) \right] \cdot \left[ (y - 2)x + (y^3 + 7) \right].$$

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` を通して呼出を行います。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”のような物で、例えば `mul-poly` の呼出は係数の乗算のために `mul-poly` の再帰呼出しに帰着します。もし係数の係数がそれ自身多項式（多項式を 3 変数で表現した場合）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し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>58</sup>

---

<sup>58</sup>

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

## 項リストの表現

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と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。

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、

$$A : \quad x^5 + 2x^4 + 3x^2 - 2x - 5$$

は密多項式です。

$$B : \quad x^{100} + 2x^2 + 1$$

は疎です。

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>59</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>60</sup>

---

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。

$$[x^2 + (y + 1)x + 5] + [x^2 + 2x + 1],$$

これは係数  $y + 1$  を係数 2 に対し足す必要があります。

<sup>59</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>60</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

```

(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons term term-list)))

(define (the-empty-term-list) '())
(define (first-term term-list) (car term-list))
(define (rest-terms term-list) (cdr term-list))
(define (empty-term-list? term-list) (null? term-list))

(define (make-term order coeff) (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))

```

`=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)。

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:** 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:** 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:** 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:** 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

---

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。

リスト表現をシステム内にて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:** 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、

$$\frac{x^5 - 1}{x^2 - 1} = x^3 + x, \text{ remainder } x - 1.$$

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。

`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。除算の商と剰余の両者を求める `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剰余の項リストのリストを返す。

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剰余の多項式のリストを返す。

```
(define (div-terms L1 L2)
  (if (empty-termlist? L1)
      (list (the-empty-termlist) (the-empty-termlist))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-termlist) 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)
  )))))

```

## 記号代数の型の階層

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。

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。

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:** 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!)

### 延長課題: 分数関数

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。

$$\frac{x+1}{x^3-1}.$$

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、

$$\frac{x+1}{x^3-1} + \frac{x}{x^2-1} = \frac{x^3+2x^2+3x+1}{x^4+x^3-x-1}.$$

(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。)

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な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:** 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))
```

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。

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>61</sup> 整数版は以下のとおりです。

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

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))))
```

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剩余部を取り出します。

**Exercise 2.94:** `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。

---

<sup>61</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のに必要な条件です。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 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)
```

次にその結果を手でチェックせよ。

**Exercise 2.95:** 以下の多項式  $P_1, P_2, P_3$  を定義せよ。

$$\begin{aligned}P_1 &: x^2 - 2x + 1, \\P_2 &: 11x^2 + 7, \\P_3 &: 13x + 5.\end{aligned}$$

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>62</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。

**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ることで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。

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:**

---

<sup>62</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。

- a 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 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。

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。

- Exercise 2.96 の gcd-terms の版を用いて、分子と分母の GCD を求める
- 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。
- 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 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 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))))
```

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)
```

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。

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>63</sup>

---

<sup>63</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

## モジュール方式、オブジェクト、状態

Μεταβάλλον ἀναπαύεται

(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.

(より多くが変化する程、より同じであり続ける)

—Alphonse Karr

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。

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。

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。

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 代入と局所状態

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。

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。

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なります。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 局所状態変数

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
```

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れば場合の 2 度の呼出は常に同じ結果を生じました。<sup>1</sup>

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"))
```

balance を減らすのは次の式により行われます。

```
(set! balance (- balance amount))
```

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>)
```

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します。こ

---

<sup>1</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で作る方法は無かった訳です。

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>2</sup>

`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⟩)
```

⟨exp1⟩ から ⟨expk⟩ までの式は続けて評価され最後の式 ⟨expk⟩ が `begin` の形式全体の値として返ります。<sup>3</sup>

`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。

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"))))
```

---

2

`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。

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との同様です。

<sup>3</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。

ここで起きたのは `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>4</sup>

`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。

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>5</sup>

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
```

---

<sup>4</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なります。

<sup>5</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)

```
"Insufficient funds")))
```

`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
```

`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しは別の口座には影響しません。

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。従っ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)
```

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` 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
```

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))
```

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` を持ります。

**Exercise 3.1:** `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。アキュムレータ(累算器)を返す手続 `make-accumulator` を書け。アキュムレータはそれぞ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:** 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:** `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))
```

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:** 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 代入導入の利点

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。

“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)
```

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>6</sup>

`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 (let ((x random-init))
  (lambda ()
    (set! x (rand-update x))
    x)))
```

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

---

<sup>6</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。

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。

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>7</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。

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)
  (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)))))
```

---

<sup>7</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。

```
(iter trials 0))
```

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))
```

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` 手続を取る全体的な `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关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。

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。

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の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、このお話はそんなに簡単ではありません。

**Exercise 3.5:** *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。

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` 手続と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。

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>8</sup>

```
(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (random range))))
```

**Exercise 3.6:** 亂数生成器を与えられ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 代入導入のコスト

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。

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。

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)
```

---

<sup>8</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。

```
(lambda (amount)
  (set! balance (- balance amount))
  balance))
(define W (make-simplified-withdraw 25))
(W 20)
5
(W 10)
-5
```

この手続と以下の `set!` を使用しない `make-decrementer` 手続とを比べてみて下さい。

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))
```

`make-decrementer` 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-decrementer 25))
(D 20)
5
(D 10)
15
```

`make-decrementer` 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-decrementer 25) 20)
```

最初に結合のオペレータを `make-decrementer` 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)
```

`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)
```

最終的な答は 5 です。

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)
```

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>9</sup>

```
((lambda (amount) (set! balance (- 25 amount)) 25) 20)
```

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
```

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 `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。

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。

## 同一性と変更

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。

`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。

---

<sup>9</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。

```
(define D1 (make-decrementer 25))  
(define D2 (make-decrementer 25))
```

D1 と D2 は同じでしょうか? 無難な答は YES です。D1 と D2 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 D1 は任意の計算において結果を変えることなく D2 の代替にできます。

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))
```

W1 と W2 は同じでしょうか? もちろん違います。W1 と W2 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
```

例え W1 と W2 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、W1 が式の評価の結果を変えずに任意の式で W2 の代替になるかというのは正しくありません。

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。

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” の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。従つて “同一性” の何らかの *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。

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))
```

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)
```

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>10</sup>

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

---

<sup>10</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単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)

合データオブジェクトをまさにそ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。

## 命令型プログラミングの落とし穴

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))
```

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に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)))
```

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))
```

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に、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>11</sup>

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:** 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、

---

<sup>11</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るべきか?”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をしばしば負わせることになります。

```
(define paul-acc  
  (make-joint peter-acc 'open-sesame 'rosebud))
```

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:** [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))
```

`+` 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。

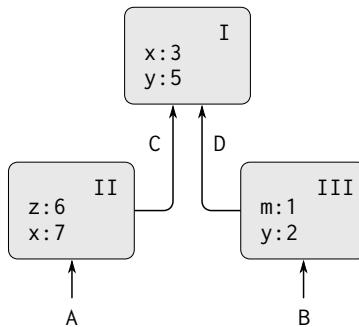
## 3.2 評価の環境モデル

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](#))。

- 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。

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。

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。



**Figure 3.1:** 単純な環境構造

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 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。

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言えるでしょう。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

的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 評価のルール

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。

- 組み合わせを評価するために
  1. 組み合わせの部分式を評価する。<sup>12</sup>
  2. 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。

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。

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。手続はただ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。

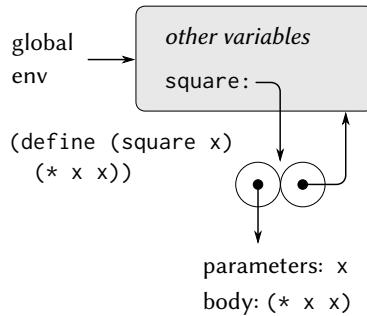
```
(define (square x)
  (* x x))
```

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>12</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。



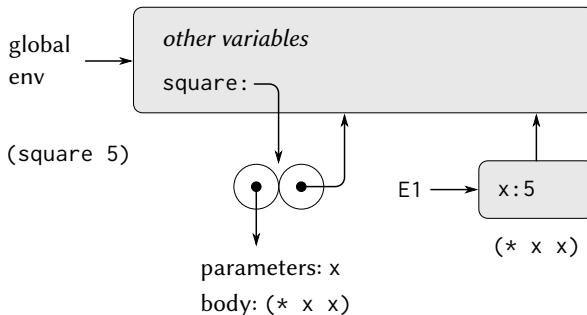
**Figure 3.2:** 大域環境内で `(define (square x) (* x x))` を評価することにより生成された環境構造

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。

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。

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



**Figure 3.3:** 大域環境内で `(square 5)` を評価することにより作られた環境

ローバル環境が選択されます。E1 の中では手続のボディ `(* x x)` を評価します。E1 中の `x` の値は 5 であるため結果は `(* 5 5)`、つまり 25 です。

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。

- 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。
- 手続は与えられた環境に関連する入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入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。

`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>13</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

<sup>13</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。

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。

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 単純な手続の適用

[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 5`) 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。

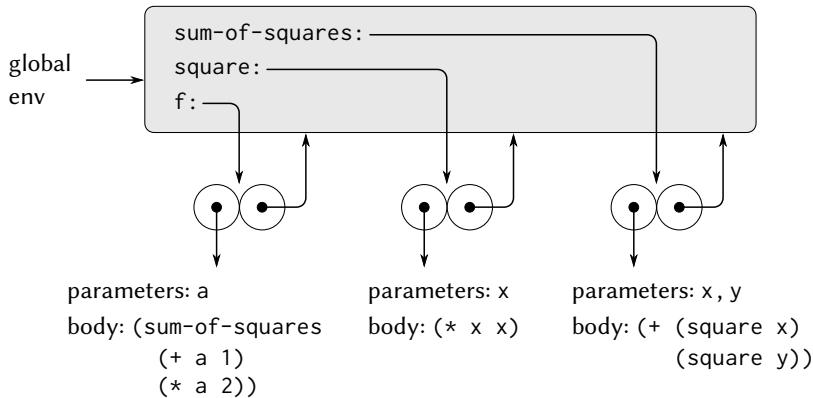
```
(define (square x)
  (* x x))
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

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。

[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))
```

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。

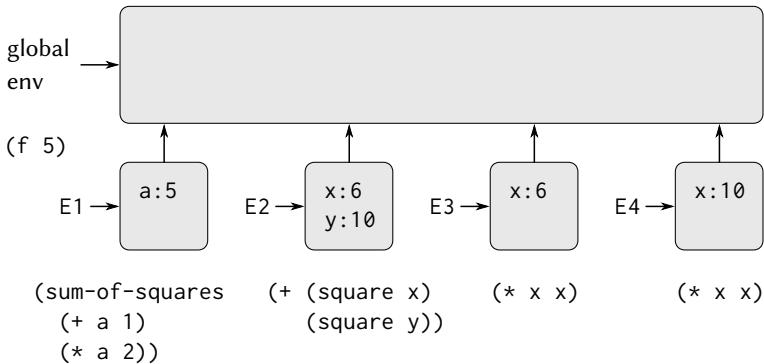


**Figure 3.4:** グローバルフレーム内の手続オブジェクト

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` のボディ `(* 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。

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。

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



**Figure 3.5:** Figure 3.4内の手続を用いて (f 5) を評価することで作られた環境

ます。

**Exercise 3.9:** 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)))))
```

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)))
```

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>14</sup>

---

<sup>14</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。未

### 3.2.3 局所状態のレポジトリとしてのフレーム

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あります。

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

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))
```

以下のように用いたとします。

```
(W1 50)  
50
```

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。今この所、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。

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。

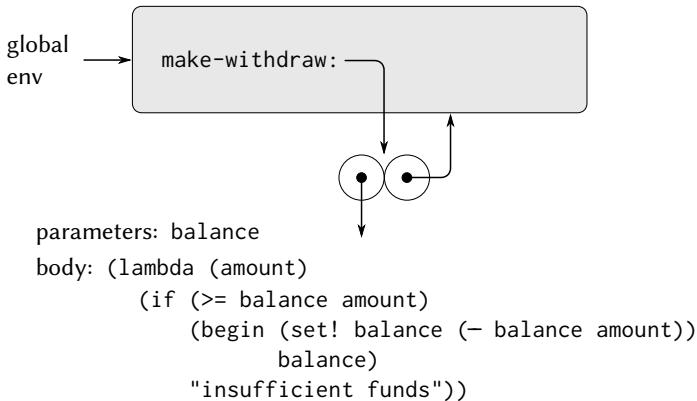
```
(define W1 (make-withdraw 100))
```

通常通りに形式パラメタ `balance` が引数 100 に束縛される環境 `E1` 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  $\lambda$  式を評価します。これがコードは `lambda` で指定され、環境が `E1` である新しい手続オブジェクトが構築されます。その `E1` 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 `W1` 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。

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。

---

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。



**Figure 3.6:** グローバル環境にて `make-withdraw` を定義した結果

(W1 50)

50

`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。

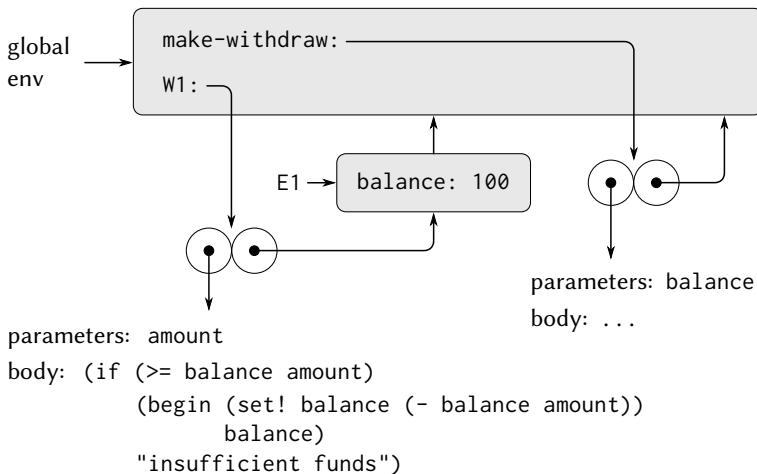
```

(if (>= balance amount)
  (begin (set! balance (- balance amount))
         balance)
  "Insufficient funds")

```

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。

`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



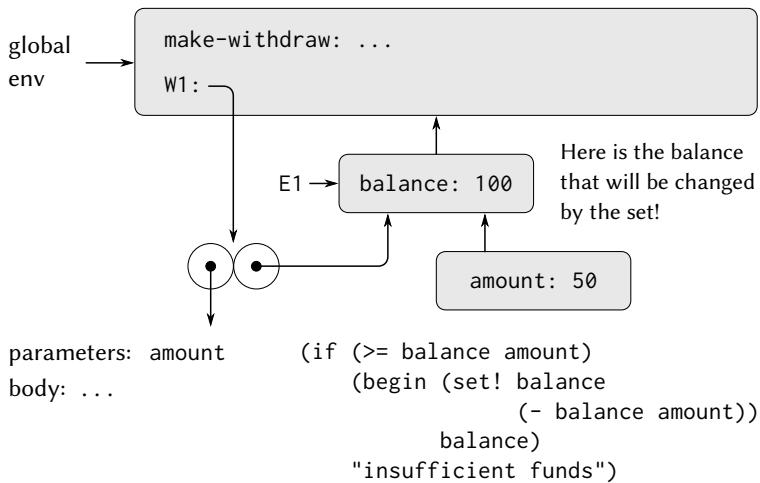
**Figure 3.7:** (`define W1 (make-withdraw 100)`) を評価した結果

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。

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)`)

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` のボディ内の中身によりコードは指定されて



**Figure 3.8:** 手続オブジェクト W1 を適用したことにより作成された環境

います。<sup>15</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:** `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)
```

---

<sup>15</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ではコードは実際に共有されます。

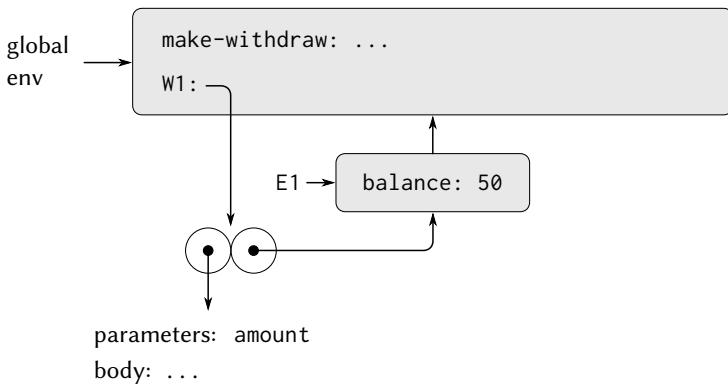


Figure 3.9: W1 呼出後の環境

```

(begin (set! balance (- balance amount))
       balance)
      "Insufficient funds"))

```

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>)
```

上記は代替的な文法として以下に翻訳される。

```
((lambda (<var>) <body>) <exp>)
```

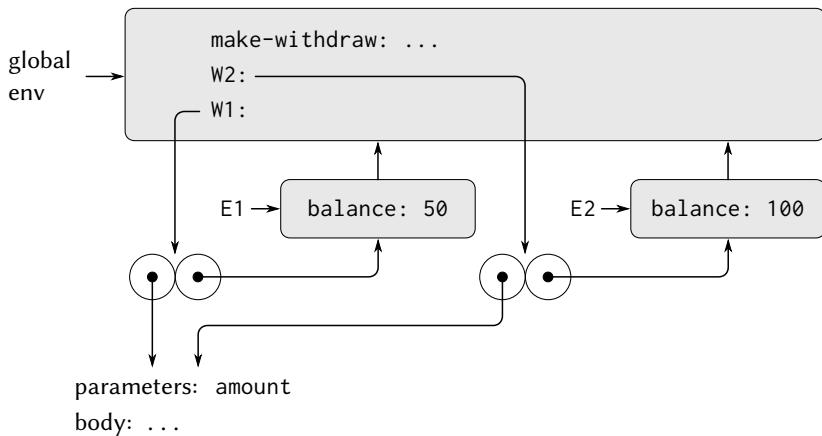
環境モデルを用いてこの `make-withdraw` の代替版を解析し、先に記述したような図を描き相互作用を説明せよ。

```

(define W1 (make-withdraw 100))
(W1 50)
(define W2 (make-withdraw 100))

```

`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?



**Figure 3.10:** (`define W2 (make-withdraw 100)`) を用いて  
2つ目のオブジェクトを作成

### 3.2.4 内部定義

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))
```

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。

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、

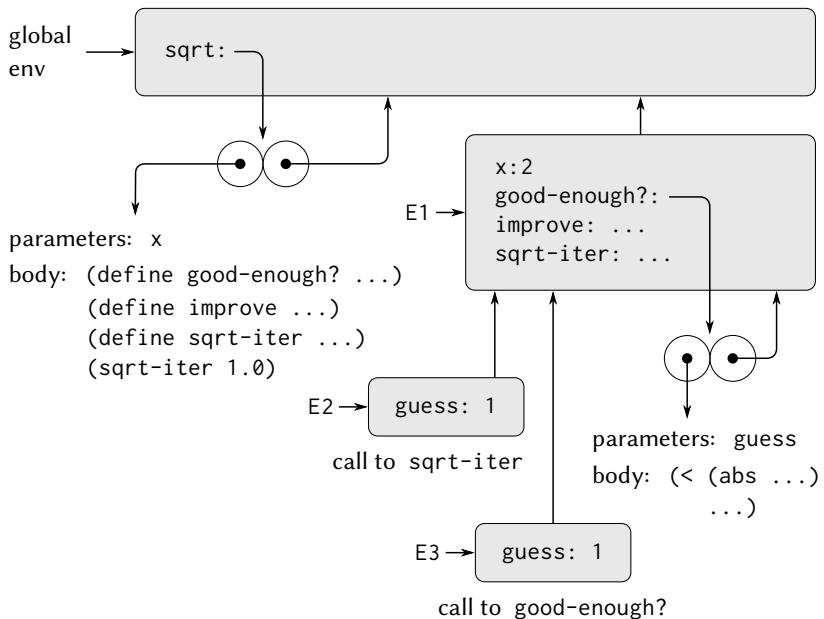
```
(define (good-enough? guess)
  (< (abs (- (square guess) x)) 0.001))
```

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。

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ります。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。これの結果の 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。

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。

- 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はありません。
- 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。



**Figure 3.11:** 内部定義を持つ sqrt 手続

**Exercise 3.11:** 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キ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)

```

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

```

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))
```

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 ミュータブルデータによるモーリング

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>)
```

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](#) 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 ミュータブルなリスト構造

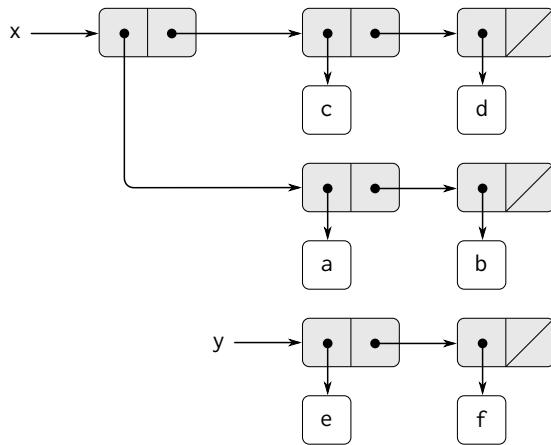
ペア上の基本的な命令—`cons`, `car`, `cdr`—はリスト構造の構築とリスト構造からの部品の選択に用いることができます。しかしそれらはリスト構造を変更する能力はありませんでした。同じことが今までに使用した `append` や `list` の様なリスト命令にも正しいと言えます。これらが `cons`, `car`, `cdr` を用いて定義できるためです。リスト構造を変更するため新しい命令が必要です。

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 `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>16</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

---

<sup>16</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。



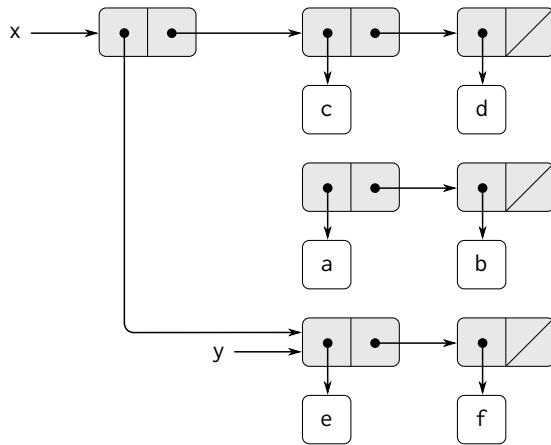
**Figure 3.12:** リスト  $x: ((a\ b)\ c\ d)$  と  $y: (e\ f)$

(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たが、元の構造から取り外されました。<sup>17</sup>

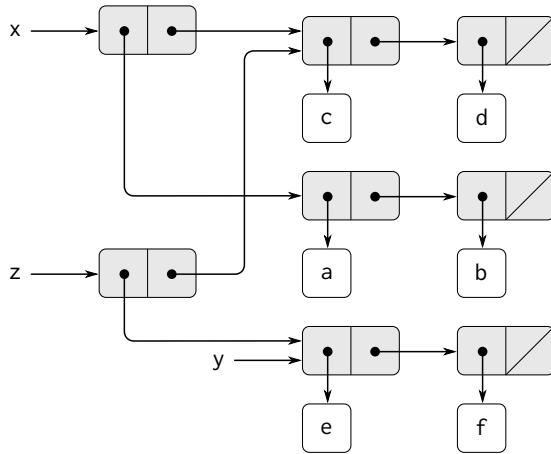
Figure 3.13 と Figure 3.14 を比べてください。これは  $x$  と  $y$  が Figure 3.12 の元のリストに束縛されている時に (`define z (cons y (cdr x)))` を実行した結果を図示しています。変数  $z$  はこれで `cons` 命令により作成された新しいペアに束縛されます。 $x$  が束縛されるリストは変更されません。

---

<sup>17</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、それにより必要の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 3.13:** Figure 3.12のリスト上で (set-car! x y) の効果



**Figure 3.14:** Figure 3.12のリスト上で (define z (cons y (cdr x))) の効果

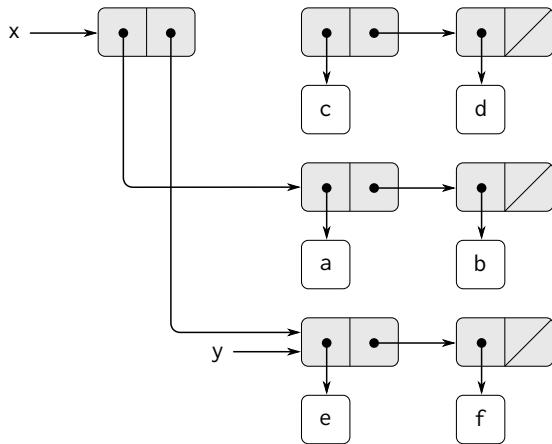
`set-cdr!` 命令は `set-car!` と同様です。違いは `car` ポインタでなく、`cdr` ポイントが置き換えられます。[Figure 3.12](#)のリスト上での `(set-cdr! x y)` の実行の結果は[Figure 3.15](#)に示されます。ここでは `x` の `cdr` ポインタは `(e f)` へのポインタにて置き換えられます。また `x` の `cdr` として用いられるリスト `(c d)` はこれで構造から取り外されます。

`cons` 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 つのミューテ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>18</sup>

```
(define (cons x y)
  (let ((new (get-new-pair)))
    (set-car! new x)
    (set-cdr! new y)
    new))
```

---

<sup>18</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。



**Figure 3.15:** Figure 3.12のリスト上で `(set-cdr! x y)` の効果

**Exercise 3.12:** 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))))
```

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テータである。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! 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)
```

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))))
```

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>
```

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:** 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))
```

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)))
```

(last-pair z) を演算すると何が置くるか？

**Exercise 3.14:** 以下の手続はとても便利であるが不明瞭である。

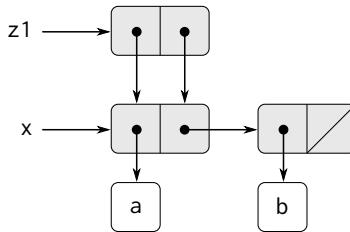
```
(define (mystery x)
  (define (loop x y)
    (if (null? x)
        y
        (let ((temp (cdr x)))
          (set-cdr! x y)
          (loop temp x))))
  (loop x '())))
```

loop 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？

## 共有と自己同一性

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))
```



**Figure 3.16:** (`cons x x`). で形成されたリスト `z1`

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)))
```

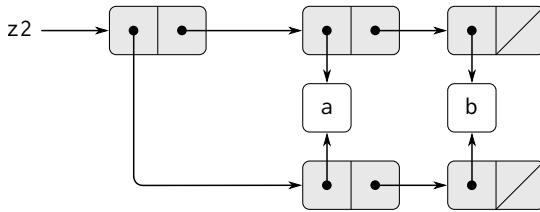
この構造においては、2つの(`a b`)リスト内のペアは実際のシンボルが共有されていても区別可能です。<sup>19</sup>

リストとして考えた時、`z1` と `z2` 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)
```

例え `z1` と `z2` 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。`z1` では `car` の変更は `cdr` も変更します。`z1` では `car`

<sup>19</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:** (`cons (list 'a 'b) (list 'a 'b)`) により形成されたリスト `z2`

と `cdr` が同じペアであるためです。`z2` 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)

```

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`

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。ミュータブルなデータオブジェクトである `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>20</sup>

---

<sup>20</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

**Exercise 3.15:** 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 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)))
```

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。

**Exercise 3.17:** 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:** 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。

---

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“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 3.19:** 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)。

## 変更とは代入のこと

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))
```

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)
```

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>21</sup>

**Exercise 3.20:** 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
```

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 キューの表現

ミューデータの `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。

`queue(キュー)` はアイテムが一方の端(キューの `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。

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。

---

<sup>21</sup>一方で、実装上の視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:** キュー命令

- 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
- 2 つのセレクタ :

(empty-queue? <queue>)

キューが空であるかテストする

(front-queue <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。

- 2 つのミューテータ :

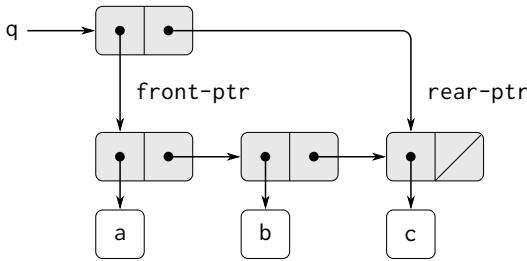
(insert-queue! <queue> <item>)

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>)

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。

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



**Figure 3.19:** 先端と終端のポインタを持つリストとしてのキューの実装

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。

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。

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。

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))
(define (set-front-ptr! queue item) (set-car! queue item))
(define (set-rear-ptr! queue item) (set-cdr! queue item))
```

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)))
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。
```

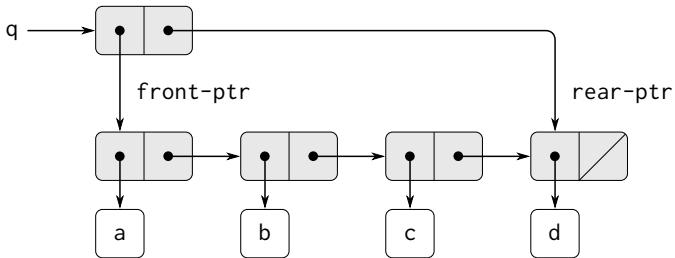
```
(define (make-queue) (cons '() '()))
```

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ペアの 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))))
```

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))))
```



**Figure 3.20:** Figure 3.19のキューに (insert-queue! q 'd) を用いた結果

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>22</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)))
```

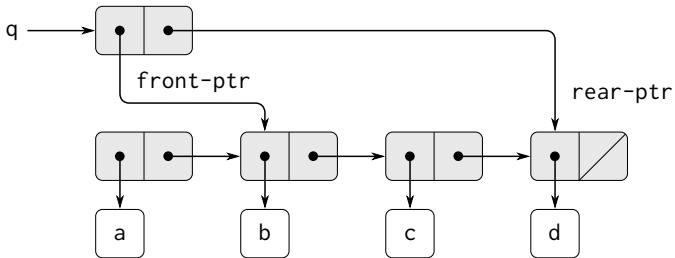
**Exercise 3.21:** Ben Bitdiddle は上で説明されたキューの実装をテストすることに決めた。彼は Lisp インタプリタに対し手続を入力し、続いて以下のように試行を行った。

```
(define q1 (make-queue))

(insert-queue! q1 'a)
((a) a)
```

---

<sup>22</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。



**Figure 3.21:** Figure 3.20のキューに (delete-queue! q) を用いた結果

```
(insert-queue! q1 'b)
((a b) b)
```

```
(delete-queue! q1)
((b) b)
```

```
(delete-queue! q1)
(() b)
```

“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 が何が起っ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:** 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))
```

`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:** `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>23</sup> 全ての命令は  $\Theta(1)$  ステップで達成すること。

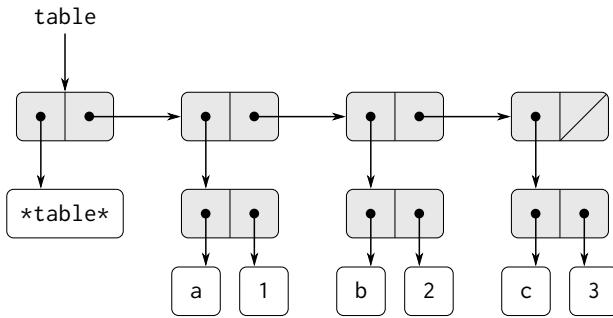
### 3.3.3 テーブルの表現

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。

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

---

<sup>23</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:** 頭出しリストとして表現されたテーブル

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
```

テーブルから情報を抽出するには鍵を取り相対する値(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 `cadr` として持つレコードを返します。<sup>24</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)))
```

---

<sup>24</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。

```

        false)))
(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records))))))

```

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>25</sup>

```

(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)

```

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*))

```

## 二次元テーブル

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。

---

<sup>25</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。

```

math:    +:  43          letters:    a:  97
         -:  45          b:  98
         *:  42

```

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)。

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)))

```

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)))))

        (set-cdr! table
                  (cons (list key-1

```

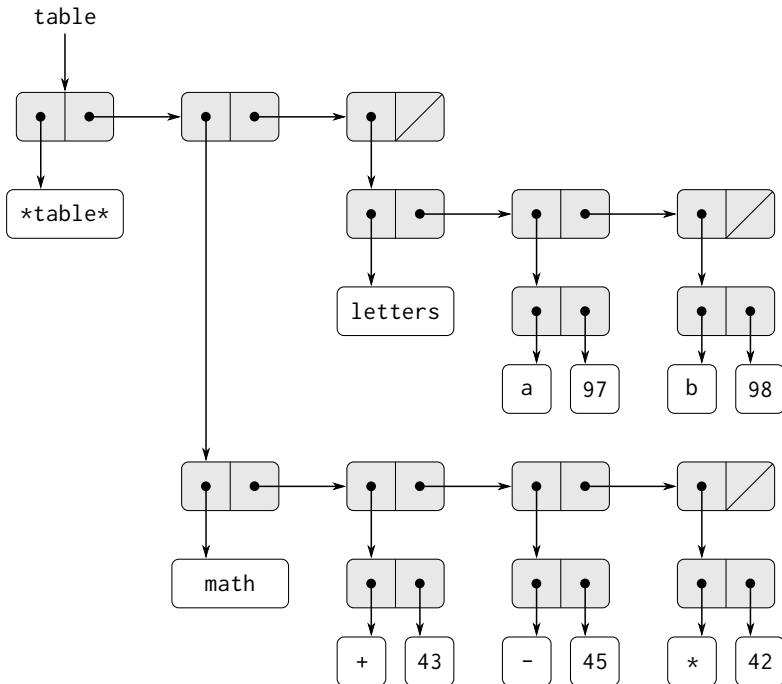


Figure 3.23: 二次元テーブル

```
(cons key-2 value))
(cdr table))))))
'ok)
```

## ローカルなテーブルの作成

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))
```

`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` 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:** 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:** 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:** 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(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と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))))))
```

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))))))))
```

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))))))
```

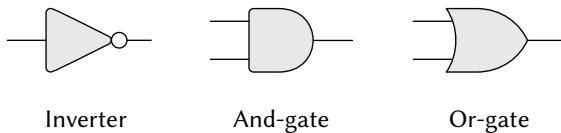
`(memo-fib 3)` の演算を分析するための環境図を描け。なぜ `memo-fib` が  $n$  番目のフィボナッチ数を  $n$  に比例するステップ数で演算するのか説明せよ。単に `memo-fib` を `(memoize fib)` と定義した場合にも Scheme は正しく処理できるだろうか？

### 3.3.4 デジタル回路のシミュレータ

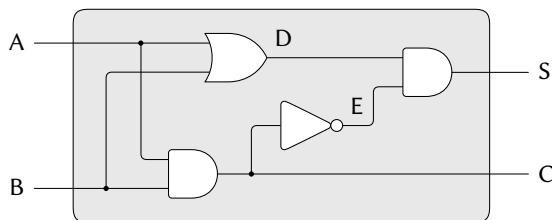
コンピュータのような複雑なデジタルシステムの設計は重要な工学の活動領域です。デジタルシステムは簡単な要素を相互接続することで構築されます。これらの個々の要素の振舞は単純ですが、それらのネットワークはとても複雑な振舞をします。提案された回路設計のコンピュータシミュレーションはデジタルシステムエンジニアにより使用される重要なツールです。この節ではデジタル論理シミュレーションを実行するためのシステムを設計します。このシステムは*event-driven simulation*(イベント駆動シミュレーション)と呼ばれる種類の典型であり、その行動(“イベント”)は後に起こるさらなるイベントを引き起こし、順により多くのイベントを引き起します。

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。

プリミティブな関数と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



**Figure 3.24:** デジタル論理回路シミュレータにおけるプリミティブな関数



**Figure 3.25:** 半加算器回路

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多くはこの事実から生じます。

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。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。

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))
```

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
```

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られた時、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。

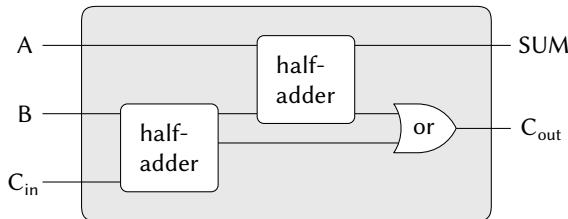
```
(define (half-adder a b s c)
  (let ((d (make-wire)) (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'ok))
```

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ることです。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>26</sup> 全加算器を以下のように構築できます。

```
(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))
```

---

<sup>26</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:** 全加算器回路

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できます。(例えばExercise 3.30を参照)。

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。

### プリミティブな関数箱

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>)`  
回線上的信号の現在地を返す
- `(set-signal! <wire> <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>)`  
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。

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。

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)

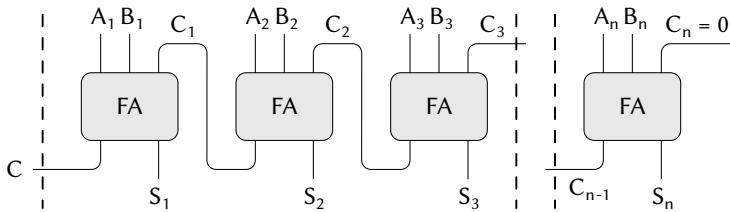
(define (logical-not s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else (error "Invalid signal" s))))
```

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:** 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



**Figure 3.27:**  $n$ -bit 数の桁上げ伝播加算器

ればならない。

**Exercise 3.29:** 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 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。

## 回路の表現

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)))
          '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))
```

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。もしそう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))))))
```

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)

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>27</sup>

---

<sup>27</sup>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

```
(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))
```

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`手続が構築され返され、新しい状態変数を持つ環境が確保されます。

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。

## 予定表

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。ここでのアイデアは`agenda(予定表)`と呼ばれるデータ構造を保持し、それに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`) は新しい空の予定表を返す。
- (`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>`) は予定表の最初のアイテムを返す。
- (`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>)`) は指定された時間後に実行される行動手続を追加する。

---

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`を手續`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續をオブジェクトとして扱う言語には“手續”と“データ”的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。

- (`(current-time <agenda>)`) は現在のシミュレーション時間を返す。

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))
```

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))))
```

## サンプルシミュレーション

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))))))
```

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)
```

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
```

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
```

`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。この時点では `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
```

`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:** `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)))
```

## 予定表の実装

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。

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))
```

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。

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>28</sup>

```
(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)))
```

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)))
```

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)
        (< 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))))
```

---

<sup>28</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さるので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)を必要としません。

```

(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))))

```

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>29</sup>

```

(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))))

```

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>30</sup>

---

<sup>29</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、*<alternative>* は有りません。

<sup>30</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。

```
(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:** 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 制約伝播

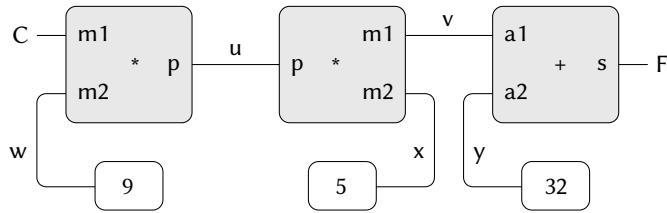
コンピュータプログラムは伝統的に一方向の演算として体系化されます。これは事前に指定した引数上で命令を実行し、望んだ出力を生成します。一方で私達は時折、量の間の関係を用いてシステムをモデル化します。例えば機械構造の数理的モデルは金属棒の偏差  $d$  が棒上の力  $F$ 、棒の長さ  $L$ 、断面積  $A$ 、弾性率  $E$  に方程式を通して関連するという情報を含むでしょう。

$$dAE = FL.$$

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  $d$  を求めるこには、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>31</sup>

---

<sup>31</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 パ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:** 制約ネットワークとして表した関係

$$9C = 5(F - 32)$$

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。例えば  $c$  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。

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。例えば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。

$$9C = 5(F - 32).$$

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。

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コ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。

## 制約システムの利用

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
```

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))
```

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。

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)
```

次に C の値を 25 に設定します。`(set-value! 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
```

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。

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)
```

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 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。

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
```

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。

## 制約システムの実装

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。コネクタ上の基本的な命令は次のとおりです。

- (`has-value? <connector>`) はコネクタが値を持つかどうか判断する
- (`get-value <connector>`) はコネクタの現在地を返す
- (`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>`) はコネクタに対し撤回を望む者が値を忘れる要求をしていると伝える
- (`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

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續 `inform-about-no-value` を用いて制約と通信を行います。

`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` 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))

```

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` を接続す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。

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)

```

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)

```

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)

```

## コネクタの表現

コネクタは局所状態変数を持つ手続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))

```

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>32</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))

```

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。

---

<sup>32</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。

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。次にもしこ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。

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:** 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 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 `square` 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 (square a b) (multiplier a a b))
```

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 は Louis に Exercise 3.34 の問題を避ける 1 つの方法として `square` 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 (square 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:** 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)

```

`set-value!` の評価の間のある時点では、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)

```

上の式が評価される環境を示す環境の図を描け。

**Exercise 3.37:** `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 5) 9) 32))
(define C (make-connector))
(define F (celsius-fahrenheit-converter C))

```

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))

```

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>33</sup>

## 3.4 並行性: 時間が本質

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。

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

<sup>33</sup>式指向形式は便利です。それは演算の中間式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)
```

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)))
```

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ります。式指向形式の利点を与え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。

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
```

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。

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）に一同時に一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。

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>34</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。

---

<sup>34</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。

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 並行システム内の時間の性質

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>35</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  $B$  が同時か、 $A$  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。

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。

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)
  (if (>= balance amount)
      (begin
        (set! balance (- balance amount)) balance)
      "Insufficient funds"))
```

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。

さらに悪くなりえます。以下の式について考えてみましょう。

---

<sup>35</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”

```
(set! balance (- balance amount))
```

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、二人の引き出しありは `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。

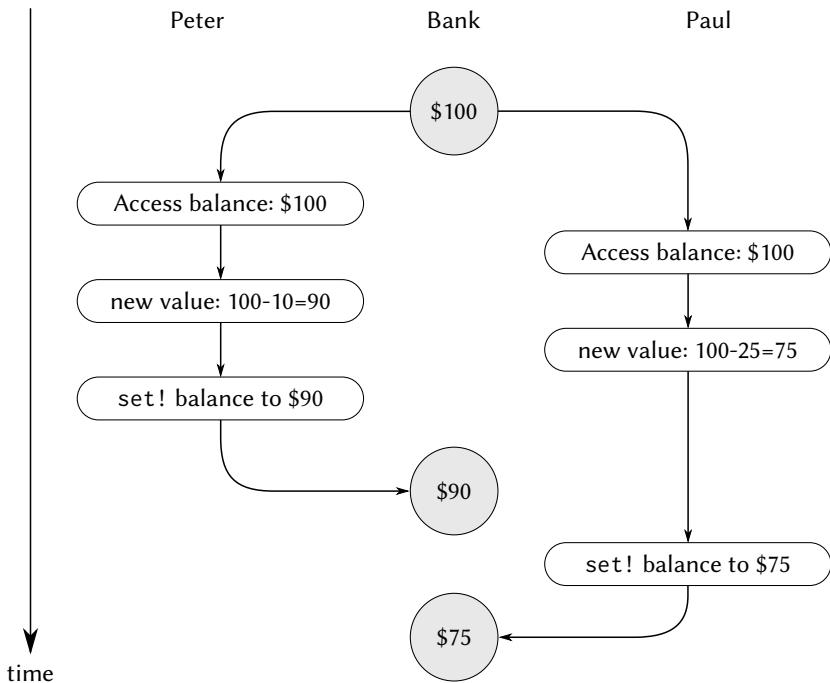
Figure 3.29 のタイミング図は `balance` が 100 で開始し、Peter が 10 を引き出し、Paul が 25 を引き出し、それでも `balance` の最終の値が 75 である場合のイベントの順を描写しています。図に示されるとおり、この異例の理由は Paul の `balance` への 75 の代入が減算されるべき `balance` の値が 100 であるという前提の下で行われているためです。しかしこの前提は Peter が `balance` を 90 に変更した時に無効になります。これは銀行システムにとって最悪な失敗です。なぜならシステム中のお金の総量が保存されていません。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>36</sup>

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。

## 並行プログラムの正しい振舞

先の例は並行プログラムに潜む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。

<sup>36</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トコル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（“キャッシングとして保存”）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*（キャッシング一貫性）プロトコルが必要となります。



**Figure 3.29:** 2回の引き出しのイベント順の相互配置が不正確な最終残高へどのように導くかを示したタイミング図

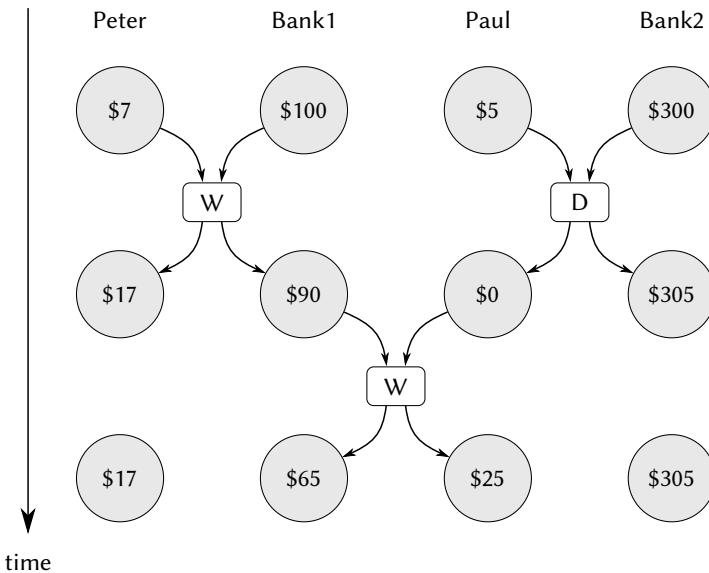
私達は既に `set!` を用いるプログラムを書く場合には気をつけねばならないことを知っています。計算の結果が代入の起こる順に依存するためです。<sup>37</sup>並行プロセスでは特に代入に気をつけねばなりません。異なるプロセスにより作られる代入の順をコントロールできないかもしれません。もしくは、そのような変更が(二人の預金者が連結口座にアクセスするように)並行に行われるのであれば私達のシステムの振舞が正しいことを確認するための何らかの方法を必要とします。例えば、連結口座からの引き出しの場合、お金が保管されていることを確認しなければなりません。並行プログラムの振舞を正しくするために、並行実行に何らかの制限を置かねばなりません。

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>38</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。

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

<sup>37</sup>Section 3.1.3における指紋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。

<sup>38</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:** 銀行 1 の連結口座と銀行 2 の個人口座への並行な預け入れと引き出し

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す。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>39</sup>

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

<sup>39</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、それらは単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。

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:** 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 これらの 3 つの取引が完了した後に、全ての異なる balance の起こり得る値を並べよ。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 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こり得るのか説明せよ。

### 3.4.2 並行性制御のための仕組み

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。

$(a, b, c, x, y, z)$	$(a, x, b, y, c, z)$	$(x, a, b, c, y, z)$	$(x, a, y, z, b, c)$
$(a, b, x, c, y, z)$	$(a, x, b, y, z, c)$	$(x, a, b, y, c, z)$	$(x, y, a, b, c, z)$
$(a, b, x, y, c, z)$	$(a, x, y, b, c, z)$	$(x, a, b, y, z, c)$	$(x, y, a, b, z, c)$
$(a, b, x, y, z, c)$	$(a, x, y, b, z, c)$	$(x, a, y, b, c, z)$	$(x, y, a, z, b, c)$
$(a, x, b, c, y, z)$	$(a, x, y, z, b, c)$	$(x, a, y, b, z, c)$	$(x, y, z, a, b, c)$

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。

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プ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*(シリアルライザ、並列直列変換器)について学びます。

## 共有状態へのアクセスの直列化

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。

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。

## Scheme のシリアルライザ

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>)`

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>40</sup>

---

<sup>40</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

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))))
```

これ2つの並行プロセス—`x`に`x`かける`x`を設定する`P1`と、`x`に1を足す`P2`を作成します。実行完了後に、`P1`と`P2`のイベントのインターリープに依存するため、`x`は5つの起こり得る値の内1つに成ります。

- 101: `P1`が`x`に100を設定し、次に`P2`が`x`を101に増やす
- 121: `P2`が`x`を11に増やし、次に`P1`が`x`を`x * x`に設定
- 110: `P2`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  
`P1`が`x`の値に`(* x x)`の評価の間にアクセスする
- 11: `P2`が`x`にアクセスし、次に`P1`が`x`に100を設定し、`P2`が`x`を設定
- 100: `P1`が`x`に(二度)アクセスし、次に`P2`が`x`を11に設定、次に`P1`が`x`を設定

並行性を`serializers`(シリアルライザ)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。シリアルライザは`make-serializer`により構築され、この実装は後程与えられます。シリアル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。与えられたシリアル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。

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)))))
```

`x`に対しだ2つの起こり得る値、101と121を返します。他の可能性は`P1`と`P2`の実行がインターリープ(相互配置)されないため排除されました。

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。

---

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させるために使用できます。

```

(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))

```

この実装により、2つのプロセスは単一の口座に並行に預け入れと引き出しを行な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なうことができます。

**Exercise 3.39:** 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:** 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))))
```

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 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))
```

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 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)))
```

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?

## 複数共有リソース使用の複雑さ

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(单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。

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。これを以下のように実装することができます。<sup>41</sup>

```
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                        (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))
```

この手続は単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>42</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)するよう準備をしなければなりません。

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

---

<sup>41</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>42</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、残高は何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。

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キ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)
            ((eq? m 'deposit) deposit)
            ((eq? m 'balance) balance)
            ((eq? m 'serializer) balance-serializer)
            (else
              (error "Unknown request: MAKE-ACCOUNT"
                     m))))
    dispatch)))
```

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の責任です。例えば以下の様にです。<sup>43</sup>

```
(define (deposit account amount)
  (let ((s (account 'serializer)))
    (d (account 'deposit)))
  ((s d) amount)))
```

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。

---

<sup>43</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。

```
(define (serialized-exchange account1 account2)
  (let ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer)))
    ((serializer1 (serializer2 exchange))
     account1
     account2)))
```

**Exercise 3.43:** 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:** 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に転送をしても、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、例え上テキストの 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 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 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は) 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)))
```

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))
```

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。

## シリアルライザの実装

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>44</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

---

<sup>44</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

続 p を与えられた場合、シリアルай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ай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證する必要のある、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)))
```

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。

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>45</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)
```

---

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）。これはアイ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。

<sup>45</sup>多くの時分割 OS では mutex でブ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にスケジュールし、ブ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。

```

(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!` 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)))

```

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)。

`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>46</sup>

---

<sup>46</sup> 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)

```

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>47</sup>

**Exercise 3.46:** `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:** (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 `mutex` を用いる
- b アトミックな `test-and-set!` 命令を用いる

## デッドロック

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

---

`false))))))`

`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>47</sup> そのような命令には `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令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ためです。

とを理解することができます。Peter が  $a_1$  と  $a_2$  を交換しようとした時、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。

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いますが、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>48</sup>

**Exercise 3.48:** 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:** 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。

## 並行性、時間と通信

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

---

<sup>48</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。

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。

`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>49</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>50</sup>

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

<sup>49</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。しかしそのプロセスも全てのプロセスがバリアに着くまでは前に進むことができな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>50</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。

りません。<sup>51</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 ストリーム

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。

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。

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>52</sup>

---

<sup>51</sup>分散システムに対するこの視点はLamport (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。

<sup>52</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

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。

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。

### 3.5.1 ストリームとは遅延化リスト

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。

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。

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>53</sup>

```
(define (sum-primes a b)
```

---

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。

<sup>53</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。

```
(define (iter count accum)
  (cond ((> count b) accum)
        ((prime? count)
         (iter (+ count 1) (+ count accum)))
        (else (iter (+ count 1) accum))))
  (iter a 0))
```

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))))
```

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。

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))))
```

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。

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。

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
```

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>54</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>55</sup>

```
(define (stream-ref s n)  
  (if (= n 0)  
      (stream-car s)  
      (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)
```

---

<sup>54</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>55</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。

```
(if (stream-null? s)
  'done
  (begin (proc (stream-car s))
    (stream-for-each proc (stream-cdr s)))))
```

`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))
```

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。

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 `<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` は特殊形式で、

```
(cons-stream ⟨a⟩ ⟨b⟩)
```

上が以下と同じになるよう設計されています。

```
(cons ⟨a⟩ (delay ⟨b⟩))
```

これの意味する所は、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` 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>56</sup>

## ストリーム実装の実践

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分析してみましょう。

```
(stream-car
  (stream-cdr
    (stream-filter prime?
      (stream-enumerate-interval
        10000 1000000))))
```

これが本当に効率的に働くことを見るでしょう。

`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))))
```

---

<sup>56</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。

従って `stream-enumerate-interval` で返される結果は、`cons-stream` で形成された<sup>57</sup>

```
(cons 10000
      (delay (stream-enumerate-interval 10001 1000000)))
```

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)
                      (stream-filter
                        pred
                        (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream)))))
```

`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` 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))))
```

これはこの場合以下のようになります。

---

<sup>57</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。

```
(cons 10007
      (delay (stream-filter
               prime?
               (cons 10008
                     (delay (stream-enumerate-interval
                             10009
                             1000000)))))))
```

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` 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。

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。

## delay と force の実装

`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>)
```

これは以下の構文糖になります。

```
(lambda () <exp>)
```

`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))
```

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。メモ化された手續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))))
```

`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>))
```

そして `force` は以前の定義と同じです。<sup>58</sup>

---

<sup>58</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

**Exercise 3.50:** 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とするSection 2.2.1の map, Footnote 12の同等品である

```
(define (stream-map proc . argstreams)
  (if (empty? argstreams)
      the-empty-stream
      (letrec ((proc (apply proc (map (lambda (x) x) argstreams)))
              (stream-map (lambda (proc)
                            (cons proc (map (lambda (x) (stream-map proc x)) argstreams)))))))
        (apply proc (cons (car argstreams) (stream-map proc (cdr argstreams)))))))
```

**Exercise 3.51:** 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)
```

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>59</sup>

```
(define x
  (stream-map show
    (stream-enumerate-interval 0 10)))
```

---

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(コールバイニード、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>59</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。

```
(stream-ref x 5)  
(stream-ref x 7)
```

**Exercise 3.52:** 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)
```

上記の各式が評価された後の `sum` の値はいくつ? 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 無限ストリーム

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列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))
```

`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

与えられ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。

`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))
```

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
```

`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` 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。

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>60</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

<sup>60</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。

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))))
      (stream-cdr stream)))))

(define primes (sieve (integers-starting-from 2)))
```

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
```

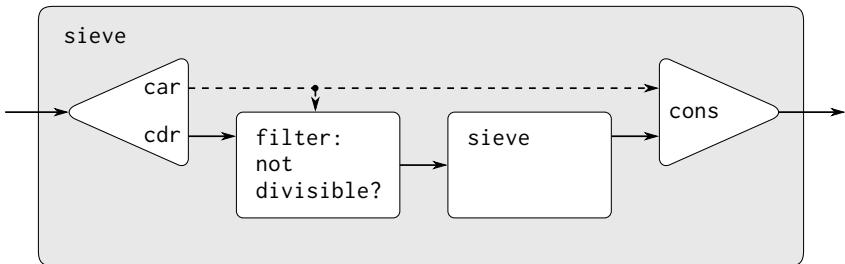
Figure 3.31の“ヘンダーソン図”に示されるように `sieve` により設定された信号処理システムを熟考することは面白いです。<sup>61</sup> 入力ストリームは“unconser”に流し込まれ、ストリームの最初の要素をストリームの残りから分離します。最初の要素は可分性フィルタを構築するのに用いられ、残りはそれに渡され通ります。フィルタの出力はもう1つの篩の箱に流し込まれます。次に元の最初の用途は内側の篩の出力上に `cons` され出力ストリームを形成します。従ってストリームのみが無限ではなく、信号処理器もまた無限です。なぜなら篩がその中に篩を持っているからです。

## 暗黙的ストリーム定義

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

---

<sup>61</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:** 信号処理システムとして見た素数の篩

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))
```

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、繰り返します。

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士の和を生成します。

```
(define (add-streams s1 s2) (stream-map + s1 s2))
```

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)))
```

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。

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))))
```

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	2	3	5	8	13	21	34	...	=	fibs

`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))
```

例として

```
(define double (cons-stream 1 (scale-stream double 2)))
```

は 2 の幂乗のストリームを生成します：1, 2, 4, 8, 16, 32, ...

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))))
```

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 (cdr ps))))))
```

```
(else (iter (stream-cdr ps)))))  
(iter primes))
```

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>62</sup>

**Exercise 3.53:** 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)))
```

**Exercise 3.54:** `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 ⟨?⟩ ⟨?⟩)))
```

**Exercise 3.55:** 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:** 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

---

<sup>62</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。ユークリッ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。

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を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。

- $S$  は 1 で始まる。
- (`(scale-stream S 2)`) の要素もまた  $S$  の要素である。
- 同じことが (`(scale-stream S 3)`) と (`(scale-stream 5 S)`) に対しても真である。
- これらは全て  $S$  の要素である。

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))))))))))
```

次に要求されたストリームが `merge` を用いて以下のよう構築されるだろう。

```
(define S (cons-stream 1 (merge ⟨??⟩ ⟨??⟩)))
```

上で ⟨??⟩ とマークされた箇所の欠けた式を埋めよ。

**Exercise 3.57:**  $n$  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數関数的に増加することを示せ。<sup>63</sup>

63

**Exercise 3.58:** 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よ。

```
(define (expand num den radix)
  (cons-stream
    (quotient (* num radix) den)
    (expand (remainder (* num radix) den) den radix)))
```

(`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:** 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。

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3 \cdot 2} + \frac{x^4}{4 \cdot 3 \cdot 2} + \dots,$$

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4 \cdot 3 \cdot 2} - \dots,$$

---

<sup>63</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。

$$\sin x = x - \frac{x^3}{3 \cdot 2} + \frac{x^5}{5 \cdot 4 \cdot 3 \cdot 2} - \dots$$

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にする。

a 級数  $a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$  の積分は次の級数になる。

$$c + a_0x + \frac{1}{2}a_1x^2 + \frac{1}{3}a_2x^3 + \frac{1}{4}a_3x^4 + \dots,$$

ここで  $c$  は任意の定数である。冪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、それは冪級数では無い。`integrate-series` を使う時、後で適切な定数を `cons` する。)

b 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)))
```

$\sin$  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:** 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 (??) (??))))
```

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:**  $S$  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。

$$\begin{aligned} S \cdot X &= 1, \\ (1 + S_R) \cdot X &= 1, \\ X + S_R \cdot X &= 1, \\ X &= 1 - S_R \cdot X. \end{aligned}$$

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:** Exercise 3.60 と Exercise 3.61 の結果を用いて 2 つの冪級数を割る手続 `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` の冪級数を生成するか示せ。

### 3.5.3 ストリームパラダイムの利用

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る、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。

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。

#### 反復をストリームプロセスとして定式化する

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

ではなく、“永遠”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)))
```

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>64</sup>

```
(define (sqrt-stream x)
  (define guesses
    (cons-stream
      1.0
      (stream-map (lambda (guess) (sqrt-improve guess x))
                  guesses)))
  guesses)

(display-stream (sqrt-stream 2))
1.
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
...
```

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)。

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できます。

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

---

<sup>64</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。

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
...
...
```

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。

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。しかしその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、ただしより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。

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。

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}.$$

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)))))
```

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))
3.1666666666666667
3.1333333333333337
3.1452380952380956
3.13968253968254
3.1427128427128435
3.1408813408813416
3.142071817071818
3.1412548236077655
...
```

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))))
```

タブローは以下の形を取ります。

$s_{00}$	$s_{01}$	$s_{02}$	$s_{03}$	$s_{04}$	...
$s_{10}$	$s_{11}$	$s_{12}$	$s_{13}$	...	
$s_{20}$	$s_{21}$	$s_{22}$	...		
...					

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)))
```

$\pi$  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.
3.1666666666666667
3.142105263157895
3.141599357319005
3.1415927140337785
3.1415926539752927
3.1415926535911765
3.141592653589778
...

```

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には必要です！

これらの加速テクニックをストリームを用いて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 はなぜ `sqrt-stream` 手続が以下のより簡単な方法で、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 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:** 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:** 以下の級数を用いて、

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots$$

$\pi$  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？

## ペアの無限ストリーム

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。

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>65</sup>

```
(stream-filter
  (lambda (pair) (prime? (+ (car pair) (cadr pair))))
  int-pairs)
```

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。

$(S_0, T_0)$	$(S_0, T_1)$	$(S_0, T_2)$	$\dots$
$(S_1, T_0)$	$(S_1, T_1)$	$(S_1, T_2)$	$\dots$
$(S_2, T_0)$	$(S_2, T_1)$	$(S_2, T_2)$	$\dots$
$\dots$			

---

<sup>65</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。

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。

$(S_0, T_0)$	$(S_0, T_1)$	$(S_0, T_2)$	$\dots$
	$(S_1, T_1)$	$(S_1, T_2)$	$\dots$
		$(S_2, T_2)$	$\dots$
			$\dots$

(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。)

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>66</sup>

$(S_0, T_0)$	$(S_0, T_1)$	$(S_0, T_2)$	$\dots$
	$(S_1, T_1)$	$(S_1, T_2)$	$\dots$
		$(S_2, T_2)$	$\dots$
			$\dots$

この分解の 3 つ目の断片(最初の行ないペア)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))
```

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)))))
```

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。

---

<sup>66</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。

```
(define (stream-append s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (stream-append (stream-cdr s1) s2))))
```

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)
```

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。

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>67</sup>

```
(define (interleave s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (interleave s2 (stream-cdr s1)))))
```

`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。

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)))
```

---

<sup>67</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)

```
(interleave
  (stream-map (lambda (x) (list (stream-car s) x))
              (stream-cdr t))
  (pairs (stream-cdr s) (stream-cdr t)))))
```

**Exercise 3.66:** 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。しかし行き詰ったと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:** `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) 生成するようにせよ。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 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提案した。

```
(define (pairs s t)
  (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                t)
    (pairs (stream-cdr s) (stream-cdr t)))))
```

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:** 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:** 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>68</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 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 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  $\text{math}j$  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:** 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>69</sup> ペアの順序有りストリームはこれらの数を計算する問題に対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

---

<sup>68</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。

<sup>69</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 のタクシーに乗っ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。

ペアを探すだけで良い。ラマヌジャン数を生成する手続を書け。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:** 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)。

## 信号としてのストリーム

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、

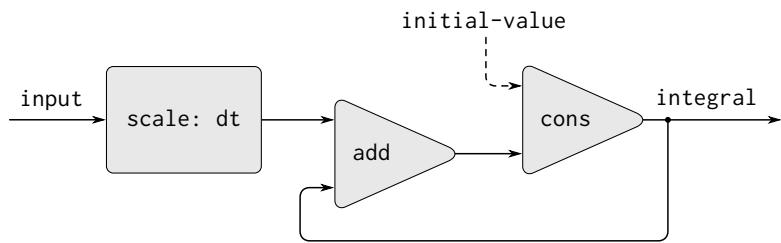
$$S_i = C + \sum_{j=1}^i x_j dt$$

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。

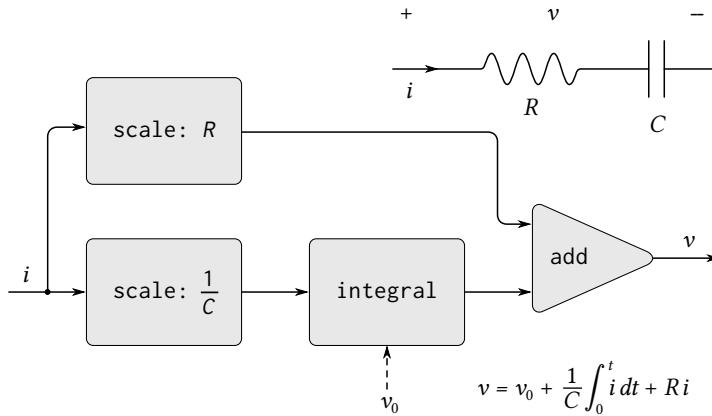
```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
      (add-streams (scale-stream integrand dt)
                   int))))
```

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:** 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。



**Figure 3.32:** 信号処理システムとして見た integral 手続



**Figure 3.33:** RC 回路と対応する信号処理図

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 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 0 -1 0 0 0 0 1 0 0 ...
```

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 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⟩))
```

⟨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:** 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ヤリスト、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))))
```

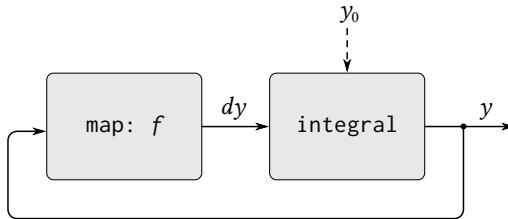
これは Alyssa の計画を正しく実装していない。Louis が入れてしまつたバグを見つけプログラムの構造を変更せずに直せ。(ヒント：`make-zero-crossings` の引数の数を増やす必要がある。)

**Exercise 3.76:** Eva Lu Ator 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 ストリームと遅延評価

先の節の終わりの `integral` 手続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
```



**Figure 3.34:** 方程式  $dy/dt = f(y)$  を解く“アナログ演算回路”

```
(add-streams (scale-stream integrand dt)
             int)))
```

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。

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。

$y$  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)
```

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` が定義されて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。

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。

この考え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)
```

これで `solve` 手続の実装が、 $y$  の定義内で  $dy$  を遅延させなければなります。

```
(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)
```

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  $y(0) = 1$  で求ることで `solve` 手続がうまく働くことを実演できます。

<sup>70</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:** 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))))
```

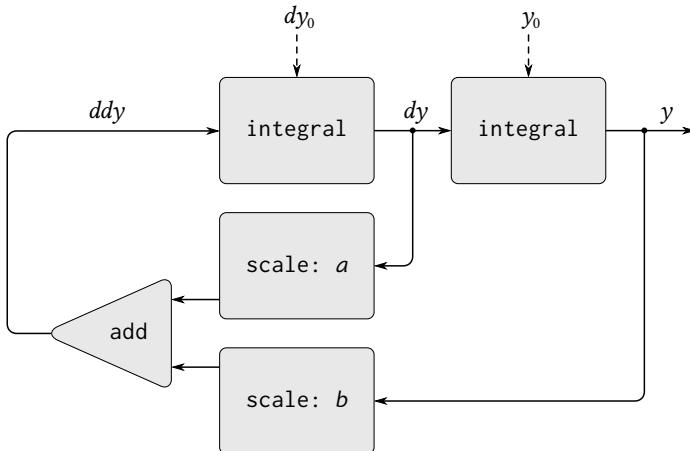
ループを持つシステム内で利用された場合、この手続は `integral` の元の版が抱えた問題と同じ問題を持つ。手続を変更して `integrand` に対し遅延された引数を要求するようにし、それ故に上で示されたように `solve` 手続で利用できるようにせよ。

**Exercise 3.78:** 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。

$$\frac{d^2y}{dt^2} - a \frac{dy}{dt} - by = 0.$$

$y$  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:** 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。



**Figure 3.35:** 二次線形微分方程式の解のための信号フロー図

**Exercise 3.80:** 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。

$$v_R = i_R R, \quad v_L = L \frac{di_L}{dt}, \quad i_C = C \frac{dv_C}{dt},$$

そして回路の接続が以下の関係を決定する。

$$i_R = i_L = -i_C, \quad v_C = v_L + v_R.$$

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。

$$\frac{dv_C}{dt} = -\frac{i_L}{C}, \quad \frac{di_L}{dt} = \frac{1}{L}v_C - \frac{R}{L}i_L.$$

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。

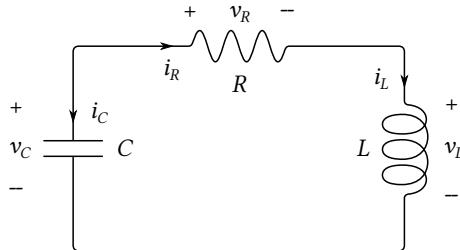


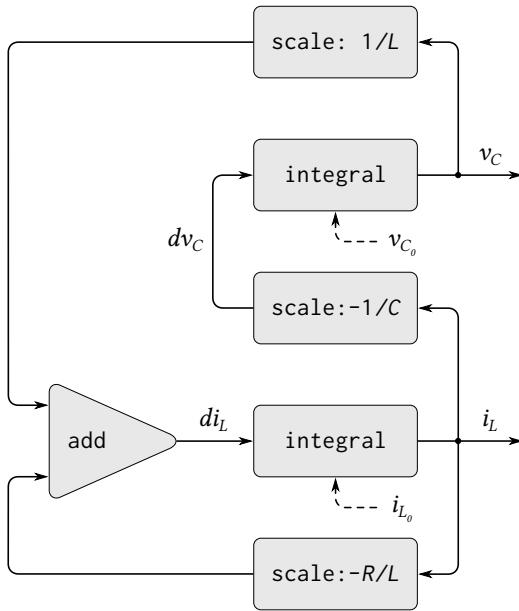
Figure 3.36: 連続 RLC 回路

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,  $dt = 0.1[s]$ , sorenri 初期値  $i_{L_0} = 0[A]$ ,  $v_{C_0} = 10[V]$  とする。

## 正規順評価

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。通常の手続と遅延化された引数を取る手続です。一般的に、分離された手続の組を作ることは私達に分離された高階手続の組もまた作ることを強います。<sup>71</sup>

<sup>71</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`



**Figure 3.37:** 連続 RLC 回路の解のための信号フロー図

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

---

のような単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。

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。残念なことに手続き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 関数型プログラムのモジュール化とオブジェクトのモジュール化

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。

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のに使用しました。

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

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)))
```

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))))))
```

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))))
```

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:** 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。

## 時間の関数型プログラミング的視点

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。

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。

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))
```

`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。

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` 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！

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ていると認識することです。もしユーザが相互作用から一步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>72</sup>

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

---

<sup>72</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。

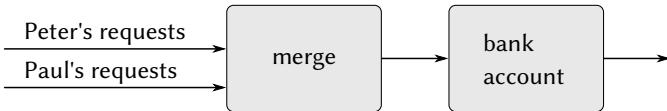
は時間と共に変化します。状態を局所状態変数でモデル化し、そして状態の変化をそれらの変数への代入を用いてモデル化します。これを行うことにより計算の実行時間を、私達がその一部である世界の時間のモデルとし、従って私達は“オブジェクト”を計算機の中に得ることになります。

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>73</sup>

一方で、もしきつ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。

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

<sup>73</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:** 取引要求リクエストの 2 つのストリームをマージすることでモデル化した連結銀行口座

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>74</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。

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>75</sup>

---

<sup>74</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。

<sup>75</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”的な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

## メタ言語抽象化

... 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。

... 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*

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>1</sup>

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(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。

<sup>1</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。

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。

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。

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。

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、抽象化の手段を持ちます。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。

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ます、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>2</sup>

---

<sup>2</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。

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ることです。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 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。

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 メタ循環評価機

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、従って評価過程を 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>3</sup>評価する対象と同じ言語で書かれ

<sup>3</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 で解説します。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

た評価機は *metacircular*(メタ循環) と呼ばれます。

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. 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. 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

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>4</sup>

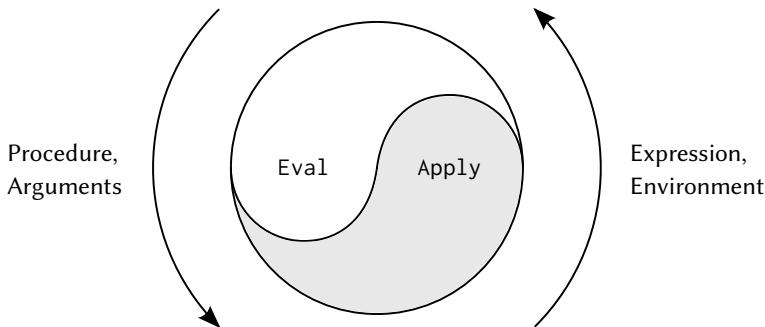
この評価サイクルは評価機内の 2 つの重大な手続、`eval` と `apply` の間の相互作用により具体化されます。これらの手続はSection 4.1.1にて説明されます。(Figure 4.1 参照)

---

により詳細に調べます。

<sup>4</sup>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、

- 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  $(+ 1 6)$  を評価するのには十分ですが、 $(+ 1 (* 2 3))$  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。
- 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。
- 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。
- 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:** eval-apply サイクルがコンピュータ言語の本質を顕在化させる

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続を与えられた引数のリストに対し適用します。

#### 4.1.1 評価機の核

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` 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。

手続の一般性を保つため、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。

### プリミティブな式

- 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` 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。

### 特殊形式

- 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。
- 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。
- `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。
- `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。
- `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。
- 事例分析(`cond`)は入れ子の `if` 式に変形し、それから評価する。

### 組み合わせ

- 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。結果となる手續と引数は `apply` に渡す。これは実際の手続適用を取り扱う。

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)))

```

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ないことです。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` 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)))

```

## 手続の引数

`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>5</sup>

```

(define (list-of-values exps env)
  (if (no-operands? exps)
    '()
    (cons (eval (first-operand exps) env)
          (list-of-values (rest-operands exps) env))))

```

## 条件文

`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)

```

---

<sup>5</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が)。

```
(eval (if-alternative exp) env)))
```

`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ありません。<sup>6</sup>

## 列

`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))))
```

## 代入と定義

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)
```

---

<sup>6</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。

変数の定義は同様の方法で扱われます。<sup>7</sup>

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (eval (definition-value exp) env)
    env)
  'ok)
```

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>8</sup>

**Exercise 4.1:** 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のなら、`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。

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 式の表現

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文法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。

以下に私達の言語の構文の仕様を示します。

---

<sup>7</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>8</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。

- 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)))
```

- 変数はシンボルにより表現されます。

```
(define (variable? exp) (symbol? exp))
```

- 引用は (quote <text-of-quotation>) の形式を持ちます。<sup>9</sup>

```
(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))
```

`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))
```

- 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))
```

- 定義は以下の形式を取ります。

```
(define <var> <value>)
```

または以下の形式になります。

```
(define (<var> <parameter1> ... <parametern>)
  <body>)
```

---

<sup>9</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。例え式がクォ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。

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)))
```

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
                   (caddr exp))))     ; body
```

- 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) (caddr exp))
```

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)))
```

- 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>10</sup>

---

<sup>10</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。

```
(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))
```

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 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>11</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))
```

また cond->if で用いるコンストラクタ sequence->exp も含めます。これは列を単一の式に、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))
        (else (make-begin seq))))
(define (make-begin seq) (cons 'begin seq))
```

- 手続き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。

---

<sup>11</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。

```
(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))
```

## 派生式

いくつかの私達の言語内で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)))
```

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)))
```

`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。

`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>12</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))
```

---

<sup>12</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。

```

(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))
                  (expand-clauses rest)))))))

```

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>13</sup>

**Exercise 4.2:** 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 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 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

---

<sup>13</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(マクロ) 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。

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:** `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:** [Chapter 1](#) の特殊形式 `and` と `or` の定義を思い出せ。

- `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`: 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。

`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 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ぬ。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))
```

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` 式は派生式である。なぜなら、

```
(let ((⟨var1⟩ ⟨exp1⟩) ... (⟨varn⟩ ⟨expn⟩))  
  ⟨body⟩)
```

は以下と等価である。

```
((lambda (⟨var1⟩ ... ⟨varn⟩)  
  ⟨body⟩)  
  ⟨exp1⟩  
  ...  
  ⟨expn⟩)
```

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\*** 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))
```

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)
```

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:** “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⟩)
```

⟨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  $\langle\text{body}\rangle$  を実行することができる。例えば、反復フィボナッチ手続 (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)))))
```

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:** 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:** 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 評価機のデータ構造

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。

#### 述語のテスト

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))
```

## 手続の表現

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>`)  
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>`)  
`<proc>` がプリミティブな手続であるか確認します。

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。

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))
```

## 環境上の命令

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>`)  
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>`)  
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 `<variables>` 中のシンボルがリスト中 `<values>` の対応する要素に束縛されます。取り囲む環境は環境 `<base-env>` です。

- `(define-variable! <var> <value> <env>)`  
環境 `<env>` の最初のフレームに変数 `<var>` を値 `<value>` に関連付ける新しい束縛を追加します。
- `(set-variable-value! <var> <value> <env>)`  
環境 `<env>` 中の変数 `<var>` の束縛を変更し、その変数が新しく値 `<value>` 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。

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 '())
```

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>14</sup>

```
(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))))
```

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))))
```

---

<sup>14</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。

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)))
```

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)))
```

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

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))))
```

ここで記述された手法は環境を表現する多くのもっともな方法の 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>15</sup>

**Exercise 4.11:** 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:** 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 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

---

<sup>15</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知れない点です。(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。

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 評価機をプログラムとして実行する

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。

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。

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))
```

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 はプリミティブの名前と実装手続をリストから得ます。16
```

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        (list 'more-primitives)))
(define (primitive-procedure-names)
  (map car primitive-procedures))
(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))
```

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>17</sup>

```
(define (apply-primitive-procedure proc args)
```

<sup>16</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。

<sup>17</sup> 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)
```

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。

```
(apply-in-underlying-scheme  
  (primitive-implementation proc) args))
```

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: レップル) をモデルにした *driver loop* (ドライバループ) 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>18</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))
```

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。(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>)))
```

---

<sup>18</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。

```
(display object)))
```

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。

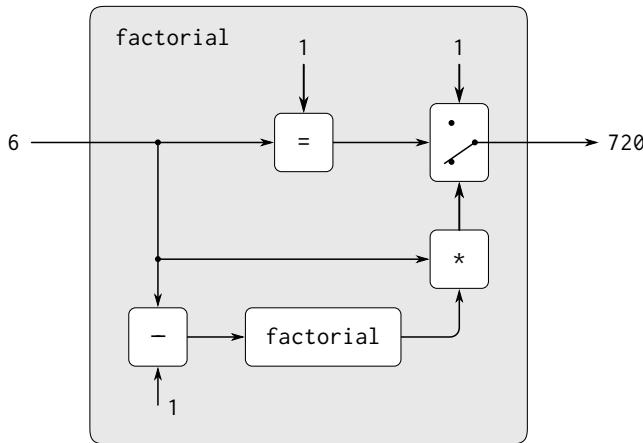
```
(define the-global-environment (setup-environment))
(driver-loop)
;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
;; M-Eval value:
ok
;; M-Eval input:
(append '(a b c) '(d e f))
;; M-Eval value:
(a b c d e f)
```

**Exercise 4.14:** Eva Lu Ator と Louis Reasoner はそれぞれ評価機を検証している。Eva は `map` の定義を入力しいく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 プログラムとしてのデータ

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)))
```



**Figure 4.2:** 抽象機械として見た階乗プログラム

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ためです)。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。

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述された機械を真似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。

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>19</sup> これは特筆

<sup>19</sup> 機械が Lisp で記述される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。ここでの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

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>20</sup>

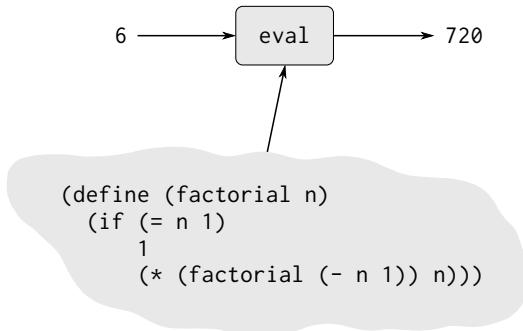
もう1つの特筆すべき評価機の側面はそれがプログラミング言語により操作されるデータオブジェクトとプログラミング言語それ自身との間のプリッジ(橋)として働くことです。(Lispで実装された)評価機プログラムが実行中であり、ユーザが式をその評価機に入力し結果を観察していると想像してみて下さい。ユーザの視点からは(\* x x)の様な入力式はプログラミング言語による式であり、評価機が実行すべき物です。しかし、評価機の視点からは式は単純なリスト(この場合ではシンボル\*, x, xのリスト)であり、これは明確なルール集合に従って操作されねばならぬ物です。

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

---

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は Hedges 1983 を参照して下さい。

<sup>20</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)



**Figure 4.3:** 階乗マシンを真似する評価機

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>21</sup>従って、

```
(eval '(* 5 5) user-initial-environment)
```

と

```
(eval (cons '* (list 5 5)) user-initial-environment)
```

の両方は 25 を返します。<sup>22</sup>

**Exercise 4.15:** 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 *halts?* を書くことは不可

<sup>21</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。

<sup>22</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。

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 `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))
```

ここで式 `(try try)` の評価について考え、どんな可能な結末（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>23</sup>

#### 4.1.6 内部定義

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は特に便利です。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。

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>)
```

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?` の定義が起こった箇所から始まる `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

<sup>23</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。

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。

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実際に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実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>24</sup>

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>)
```

以下の形式に変形されます。

```
(lambda (vars)
  (let ((u '*unassigned*)
        (v '*unassigned*))
    (set! u <e1>)
    (set! v <e2>))
```

<sup>24</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。

`(e3))`

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。

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>25</sup>

**Exercise 4.16:** 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 `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 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 `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:** この本の手続の式 `(e3)` 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:** 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。

---

<sup>25</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、プログラム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。

```
(lambda (vars)
  (let ((u '*unassigned*)
        (v '*unassigned*))
    (let ((a (e1)) (b (e2)))
      (set! u a)
      (set! v b))
    (e3)))
```

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)
```

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, 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 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。従つ

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  $b$  の定義が真に同時であ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 でなければならない。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>26</sup>

**Exercise 4.20:** 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)))
```

`letrec` 式は以下の形式を持つ。

```
(letrec ((<var1> <exp1>) ... (<varn> <expn>))
  <body>)
```

`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。

---

<sup>26</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は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。

```
(letrec
  ((fact (lambda (n)
    (if (= n 1) 1 (* n (fact (- n 1)))))))
  (fact 10))
```

a 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 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:** 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で求めている。<sup>27</sup>

```
((lambda (n)
  ((lambda (fact) (fact fact n))
   (lambda (ft k)
     (if (= k 1) 1 (* k (ft ft (- k 1)))))))
  10)
```

a (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>27</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 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))
```

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 構文分析を実行から分離する

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)))
```

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。

評価機を変形し構文上の分析をたった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>28</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 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。

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))
```

analyze の呼出の結果は環境に適用される実行手続です。analyze 手續は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續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))))
```

以下に最も簡単な構文分析手續があります。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續を返します。

```
(define (analyze-self-evaluating exp)
```

---

<sup>28</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。

```
(lambda (env) exp))
```

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度だけ行うことでの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)))
```

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>29</sup>

```
(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))
```

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)分析されることが出来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。

```
(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)))

```

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。

---

<sup>29</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可能です。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-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)))))
```

`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))))
```

(`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>30</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)))))
```

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

---

<sup>30</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。

`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せん。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))))
```

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:** この節の評価機を特殊形式 `let` をサポートするよう拡張せよ。(Exercise 4.6 参照)

**Exercise 4.23:** 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 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より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。

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ういうふうに書くか？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:** 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 Scheme 上でのバリエーション — 遅延評価

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。例えばもしわたしたちが 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>31</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 正規順と適用順

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>32</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))
```

`(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。

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))
```

これは以下のような式で使用できます。

<sup>31</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>32</sup>専門用語 “lazy” と “normal-order” の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。

```
(unless (= b 0)
  (/ a b)
  (begin (display "exception: returning 0") 0))
```

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。

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>33</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)

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:** (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))))
```

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?

---

<sup>33</sup> “厳密” 対 “非厳密” の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ンファ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。”

**Exercise 4.26:** 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よ。

## 4.2.2 遅延評価を持つインタプリタ

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。

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) と呼ばれるオブジェクトに変形されます。<sup>34</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。

サンク中の式の評価プロセスは *forcing*(強制) と呼ばれます。<sup>35</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

<sup>34</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>35</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。従つてこれを言語を通して同一化し、自動化しています。

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>36</sup>

## 評価機を変更する

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。

`eval` の `application?` 節は以下になります。

```
((application? exp)
 (apply (actual-value (operator exp) env)
        (operands exp)
        env))
```

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うためです。

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)))
```

これでもし式の値がサンクであれば強制されます。

---

<sup>36</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)が、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。

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ぎたことです。(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))))
```

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))))
```

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)))
```

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))
```

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
```

## サンクの表現

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))
```

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))
```

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>37</sup>

<sup>37</sup>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`への参照を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。

```

(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)
                     '()) ;不必要的 env を忘れる
           result)))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj)))

```

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:** 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)

```

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>38</sup>

```

(define w (id (id 10)))
;; L-Eval input:
count
;; L-Eval value:
<response>
;; L-Eval input:

```

---

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るでできました。

<sup>38</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。

```
w
;; L-Eval value:
⟨response⟩
;; L-Eval input:
count
;; L-Eval value:
⟨response⟩
```

**Exercise 4.28:** 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:** 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⟩
```

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 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 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` 手続を見せた。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)))))
```

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
```

なぜ Ben が `for-each` の振舞について正しいのか説明せよ。

b 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)))))
```

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 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 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:** 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))
  ...)
```

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 遅延化リストとしてのストリーム

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。

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>39</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)))
```

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)
```

---

<sup>39</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。

```

(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

```

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>40</sup> 実際に、遅延化ペアの `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。

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で元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))))

```

---

<sup>40</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。

```

    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:** 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 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。

```
(car '(a b c))
```

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:** 評価機のドライバ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 Scheme 上でのバリエーション — 非決定性演算

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。

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。

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)))
```

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は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>41</sup>

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、

---

<sup>41</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。

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マを選択がどのように行われるかの詳細から解放します。

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。

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)
```

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 と検索

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>42</sup>

```
(amb <e1> <e2> ... <en>)
```

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))
```

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)
```

单一の選択を行う `amb` は通常の(单一の)値を生成します。

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)))
```

`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` 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。

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))))
```

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

---

<sup>42</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)。

ての整数の列を表すオブジェクトを返します。一方、`amb` 手続は单一の整数を返します。<sup>43</sup>

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、さらなる分岐が起るか、完了するまで続けられます。<sup>44</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。この選択それが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

<sup>43</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>44</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。

が失敗となれば評価機は *automagically*<sup>45</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>46</sup>

## ドライバループ

`amb` 評価機のドライバ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`-

<sup>45</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、または恐らくさらはどうでも良すぎて）話者が説明する気にならない” (Steele et al. 1983, Raymond 1993)

<sup>46</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 de Kleer 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。

`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始し、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:** 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議論した。なぜ単純に `an-integer-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 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 非決定性プログラムの例

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。

## 論理パズル

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, 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?

誰がどの階に住んでいるかを、全ての可能性を列挙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>47</sup>

```
(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))))
```

---

<sup>47</sup>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 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。

```

(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)))

```

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))
```

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:** 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:** 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:** 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:** 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:** 次の“嘘つき”パズル (Phillips 1934から) を解け  
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 が試験では二位だった。私だけが 3 位だった。”
- Ethel: “喜んで、私がトップ。Joan が 2 位だった。”
- Joan: “私が 3 番。可哀想な Ethel は最下位だった。”
- Kitty: “私が 2 番。Mary が単独で 4 位。”
- Mary: “私が 4 位。トップは Betty が取ったわ。”

実際にはどの順に 5 人の女の子は並べられるか?

**Exercise 4.43:** amb 評価機を用いて以下のパズルを解け<sup>48</sup>

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?

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?

**Exercise 4.44:** 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。

## 自然言語の構文解析

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

---

<sup>48</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。

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と思います。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>49</sup>

```
(define nouns '(noun student professor cat class))
(define verbs '(verb studies lectures eats sleeps))
(define articles '(article the a))
```

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))
```

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)))
```

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)))
```

---

<sup>49</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。

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*` を持ります。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こを要求します。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>50</sup>

```
(define (parse-word word-list)
  (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)))
```

パ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、文のパースを試み、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))
```

これでパーザ(parser、パ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))
```

---

<sup>50</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できません。

`amb` 評価機はここでとても役立ちます。`require` の助けを用いてパースする上での制約を表現するにと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。

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))
```

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)))
```

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>51</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)))
```

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
```

---

<sup>51</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。

```

(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)))

```

私達の新しい文法はより複雑な文をパースできます。例えば、

```
(parse '(the student with the cat sleeps in the class))
```

(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))))

```

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))
```

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)))))
```

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:** 上で与えられた文法を用いて次の文は 5 通りにペースできる。“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:** 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。なぜ我々のパ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 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、手続き 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))))
```

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:** 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>52</sup>

**Exercise 4.49:** 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>53</sup>

### 4.3.3 Amb 評価機の実装

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。

---

<sup>52</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、例えおもちゃのザ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>53</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。

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>54</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 の違いは完全に実行手続の中に存在します。

## 手續と継続の実行

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*(継続手續) と呼ばれる 2 つの手續です。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パックトラックが実装される仕組みです。

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。

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呼び出し可能な失敗継続を構築し、一緒に渡します。

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

---

<sup>54</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パ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。

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。

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。

まとめとして、失敗継続は以下により構築されます。

- `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
- 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
- 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

失敗は行き止まりに遭遇したその時のみ起動されます。これは以下の場合一に起ります。

- ユーザプログラムが (`amb`) を実行した
- ユーザがトップレベルドライバにて `try-again` を入力した

失敗継続はまた失敗処理の間にも呼ばれます。

- 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。
- `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。

## 評価機の構造

`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續を `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>55</sup>

---

<sup>55</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。

```
(define (amb? exp)
  (tagged-list? exp 'amb))
(define (amb-choices exp) (cdr exp))
```

`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))
```

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))
```

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 () ...)
  ...)
```

例えば、以下を実行すると、

```
(ambeval <exp>
         the-global-environment
         (lambda (value fail) value)
         (lambda () 'failed))
```

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 `ambeval` 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。

`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。

## 単純な式

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))))
```

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。

## 条件文と列

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)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env succeed fail)
      (pproc env
             ;;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)
                   (aproc env succeed fail2)))
             ;;述語を評価するための失敗継続
             fail)))))


```

列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
         ;;aを呼ぶための成功継続
         (lambda (a-value fail2)
           (b env succeed fail2)))
         ;;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))))
```

## 定義と代入

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るためです。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))))
```

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。

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の一部分として取消す準備をしなければなりません。<sup>56</sup>

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

---

<sup>56</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)

`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))))
```

## 手続の適用

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)
fail2))
fail))))
```

`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
       ;;この aproc のための成功継続
       (lambda (arg fail2)
         (get-args
          (cdr aprocs)
          env
          ;;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))))
```

`execute-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))))
```

## amb 式の評価

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))))
```

## ドライバループ

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。

`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。

`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` 手続を伴ないます。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))))

```

`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:** `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:** 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)

```

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:** 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

取る。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:** 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:** 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))
```

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))
```

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 論理プログラミング

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。

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。

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>57</sup>

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单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))))
```

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。

---

<sup>57</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*(ユニファイ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。

- 任意のリスト  $y$  に対し、空リストと  $y$  の  $\text{append}$  は  $y$  を形成する。
- 任意の  $u, v, y, z$  に対し、 $(\text{cons } u \ v)$  と  $y$  の  $\text{append}$  はもし  $v$  と  $y$  の  $\text{append}$  が  $z$  を形成するならば  $(\text{cons } u \ z)$  を形成する。<sup>58</sup>

$\text{append}$  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。

$(a \ b)$  と  $(c \ d)$  の  $\text{append}$  を求めよ。

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。

$(a \ b)$  と  $\text{append}$  すると  $(a \ b \ c \ d)$  を生成するリスト  $y$  を求めよ。

$\text{append}$  すると  $(a \ b \ c \ d)$  を生成する全ての  $x$  と  $y$  を求めよ。

論理プログラミング言語ではプログラマは  $\text{append}$ “手続”を上で与えられた  $\text{append}$  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”的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  $\text{append}$  に関する質問に対して答えることを可能にします。<sup>59</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>60</sup>

---

<sup>58</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  $(\text{cons } u \ v)$  に対応させます。次にルールの  $z$  は  $(\text{cdr } x)$  と  $y$  の  $\text{append}$  に対応します。

<sup>59</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。 $\text{append}$  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得ます。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。

<sup>60</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し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

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” のパートが存在し、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” の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 演繹的情報検索

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。

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、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も得ることができます。

---

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。

## サンプルデータベース

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)
```

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。

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))
```

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))
```

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 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)
```

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))
```

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))
```

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ます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))
```

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))
```

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))
```

## 単純なクエリ

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))
```

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))
```

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。

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)
```

は全ての従業員の住所を並べます。

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。

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)
```

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はありません。)

以下のクエリは、

```
(job ?x (computer ?type))
```

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))
```

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))
```

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>61</sup>

```
(job ?x (computer . ?type))
```

例えば、以下のクエリは、

```
(computer . ?type)
```

次のデータに適合します。

```
(computer programmer trainee)
```

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。

```
(computer programmer)
```

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)
```

この時`?type` は空リスト () です。

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。

---

<sup>61</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。

- 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。
- 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。

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:** 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。

- Ben Bitdiddle により監督される (supervisor) 全ての人
- 経理部門に属す全ての人の名前 (name) と職種 (job)
- Slumerville に済む全ての人の名前と住所 (address)

## 複合クエリ

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。)

`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))
```

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)))
```

一般的に、

```
(and <query1> <query2> ... <queryn>)
```

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。

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。

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)))
```

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)))
```

一般的に、

```
(or <query1> <query2> ... <queryn>)
```

上の式はパターン変数に対する全ての値の集合が、<query<sub>1</sub>> ... <query<sub>n</sub>> の内、少くとも 1 つを満たす場合に満たされます。

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))))
```

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>)
```

上の式はパターン変数に対する全ての割り当てが <query<sub>1</sub>> を満たさない場合に満たされます。<sup>62</sup>

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>)
```

上の式は <predicate> がパターン変数に対してインスタンス化された <arg<sub>1</sub>> ... <arg<sub>n</sub>> 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:** 以下の情報を取得する複合クエリを定式化せよ。

- a Ben Bitdiddle に監督される全ての人の名前と住所を共に
- b 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 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

## ルール

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))))
```

---

<sup>62</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。

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>63</sup>

```
(rule (same ?x ?x))
```

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での“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)))
```

ルールの一般的な形式は以下となります。

```
(rule <conclusion> <body>)
```

`<conclusion>` がパターンであり `<body>` が任意のクエリです。<sup>64</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))
```

次の結果を生みます。

---

<sup>63</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成ります。)

<sup>64</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。

```
(lives-near (Reasoner Louis) (Bitdiddle Ben))  
(lives-near (Aull DeWitt) (Bitdiddle Ben))
```

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)))
```

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))))
```

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:** 人（`person`）その 1 が人その 2 を置き換えることは人その 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 Cy D. Fect を置き換える全ての人
- b 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:** ある人が自分が働いている同じ部署に上司（監督者）が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 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))
```

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 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 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 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:** 以下のクエリを与えることにより、

```
(lives-near ?person (Hacker Alyssa P))
```

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たい場合には以下のクエリを用いる。

```
(lives-near ?person-1 ?person-2)
```

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))
```

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。

## プログラムとしての論理

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。

- 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。
- 任意の `u, v, y, z` に対し、`(cons u v)` と `y` の `append` はもし `v` と `yappend` が `z` を形成する場合、`(cons u z)` を形成する。

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)
```

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))
```

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。

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))
```

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))
```

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))
```

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:** 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))
```

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:** 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:** 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)
```

“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 クエリシステムの働き方

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。

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。

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。

## パターンマッチング

*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。

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。

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ら

れ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。

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))
```

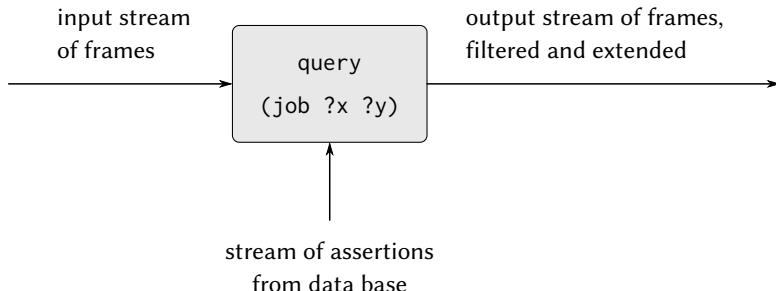
データベース内の全てのアサーションを探査し、最初は空のフレームを考慮してパターンに適合する物を選択します。探索を行った各適合に対して、適合により返されたフレームを用いてパターンを?x の値と共にインスタンス化します。

## フレームのストリーム

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、フィルタを通する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>65</sup>

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。言い替えれば、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

<sup>65</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:** フレームのストリームを処理するクエリ

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。

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。

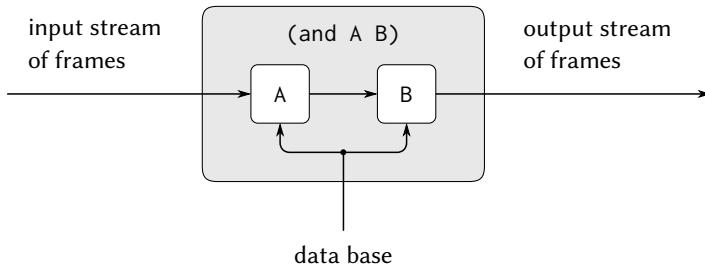
### 複合クエリ

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))
```

(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))
```



**Figure 4.5:** 2 つのクエリの `and` の組合せはフレームのストリーム上で連続した操作により生成される

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)
```

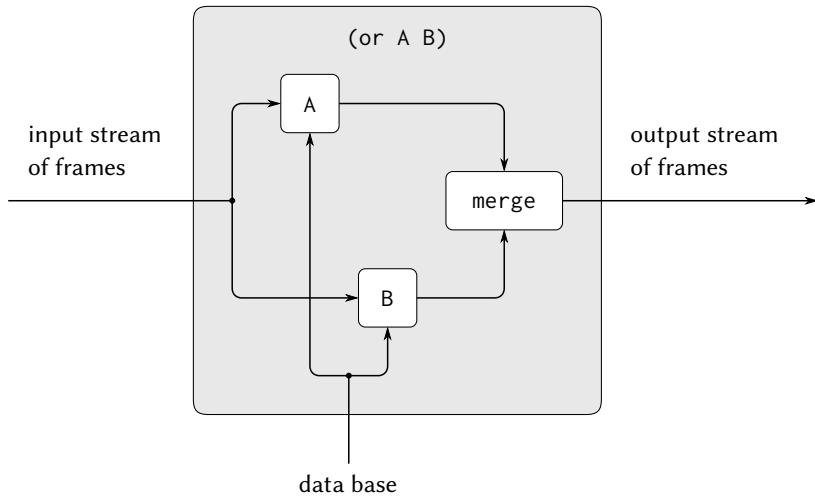
そのような適合のそれぞれは`?x`と`?person`に対する束縛を含むフレームを生成します。2つのクエリの `and` はFigure 4.5に示されるように、一連の2つのクエリのコンポーネントの組み合わせであると見做すことができます。最初のクエリフィルタを通過するフレームはフィルタをかけられ、2つ目のクエリにてさらに拡張されます。

Figure 4.6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。

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。最悪の場合にはクエリ数の指數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>66</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

---

<sup>66</sup>しかし、この種の指數関数爆発は `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。



**Figure 4.6:** 2つのクエリの codeor の組合せはフレームのストリームを並列に操作しその結果をマージすることで生成される

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>67</sup>

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れると、

```
(not (job ?x (computer programmer)))
```

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、

---

<sup>67</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。

```
(and (supervisor ?x ?y)
      (not (job ?x (computer programmer))))
```

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>68</sup>

`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。

## ユニフィケーション

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*(ユニフィケーション)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。

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)`のユニフィケーションは`?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)` のユニフィ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ィケーションを実行します。

ユニフィ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、ユニフィ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))` をユニフィ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

<sup>68</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。

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>69</sup> 例えば、 $(?x ?x)$  と  $((a ?y c) (a b ?z))$  のユニフィ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。

$$\begin{aligned} ?x &= (a ?y c) \\ ?x &= (a b ?z) \end{aligned}$$

これらの方程式は以下を暗示します。

$$(a ?y c) = (a b ?z)$$

これは順に次を暗示します。

$$\begin{aligned} a &= a, \\ ?y &= b, \\ c &= ?z, \end{aligned}$$

従って以下の通りです。

$$?x = (a b c)$$

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。

$(?x a)$  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

---

<sup>69</sup> 一方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。

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>70</sup>

## ルールの適用

ユニフィ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))
```

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ィ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ィケーションすると、

```
(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))))
```

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。

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。

---

<sup>70</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 ( $((b\ ?y)\ ?z)$ ) のユニフィケーションは ( $(b\ ?y)\ a$ ) であり、上で議論した ( $?x\ a\ ?y$ ) と ( $?y\ ?z\ a$ ) のユニフィケーションは ( $a\ a\ a$ ) です。私達の実装に対しては、ユニフィ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。

- 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
- 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

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。

- 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
- 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

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。

## 単純なクエリ

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。

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。

- (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
- (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>71</sup>

---

<sup>71</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。

## クエリ評価機とドライバループ

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。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。

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>72</sup>

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)
              (and (supervisor ?middle-manager ?person)
```

---

<sup>72</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。

```
(supervisor ?x ?middle-manager))))
```

#### 4.4.3 論理プログラミングは記号論理学なのか?

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>73</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))
```

または

```
(and (supervisor ?x ?y) (job ?x (computer programmer)))
```

もし会社に(通常の場合として) プログラマよりも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。

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、ここでの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。

---

<sup>73</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くことができます。

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>74</sup>

## 無限ループ

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))
```

ここで以下を尋ねた場合、

```
(married Mickey ?who)
```

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)))
```

---

<sup>74</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、記号論理学とより一致する方法を見つけることです。

そして再び質問します。

(married Mickey ?who)

残念ながら、これはシステムを無限ループに追いります。以下のとおりです。

- 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。
- 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`)
- `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。

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。これは起こ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>75</sup>

## not の問題

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。

<sup>75</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る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。

```
(and (supervisor ?x ?y)
      (not (job ?x (computer programmer))))
(and (not (job ?x (computer programmer)))
      (supervisor ?x ?y))
```

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。

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。

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>76</sup>

**Exercise 4.64:** 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。

---

<sup>76</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。

```
(rule (outranked-by ?staff-person ?boss)
      (or (supervisor ?staff-person ?boss)
           (and (outranked-by ?middle-manager ?boss)
                 (supervisor ?staff-person
                             ?middle-manager))))
```

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。

```
(outranked-by (Bitdiddle Ben) ?who)
```

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:** 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)
```

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))
```

何故、Oliver Warbucks は 4 度表示されたのか？

**Exercise 4.66:** 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)))
```

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>)
```

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。

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:** 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:** 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:** 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 クエリシステムの実装

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 ドライバループとインスタンス化

クエリシステムのためのドライバループ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` に単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."))
          (else
           (newline)
           (display output-prompt)
           (display-stream
```

```
(stream-map
  (lambda (frame)
    (instantiate
      q
      frame
      (lambda (v f)
        (contract-question-mark v))))))
  (qeval q (singleton-stream '())))
(query-driver-loop))))
```

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。

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)。

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。値はそれら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)))
```

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 評価機

`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` 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。

#### 単純なクエリ

`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))
```

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することができる全ての手段により、1つの巨大なストリームが形成されます。

## 複合クエリ

`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))))
```

以下の式は

```
(put 'and 'qeval conjoin)
```

`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` 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)

```

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。

## フィルタ

`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` 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` 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)))

```

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)

```

`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 パターンマッチングによりアサーションを見つける

`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` 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))))
```

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)))
```

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))))
```

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できません。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。

`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。

## 末尾ドット付きパターン

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。

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。

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 ルールとユニフィケーション

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ら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。

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えるよう惑わされてしまいません。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))))))
```

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。

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)))
```

ユニフィ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))))
```

ユニフィ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` はユニフィ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。

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

に常に起こり得ます。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つのパターンを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>77</sup>

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

<sup>77</sup>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可能です。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。

$$2 = y = 1 + \frac{y}{2} = 1 + \frac{1}{2} \left( 1 + \frac{y}{2} \right) = 1 + \frac{1}{2} + \frac{y}{4} = \dots,$$

これは以下の式へと導きます。

$$2 = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

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、

$$-1 = y = 1 + 2y = 1 + 2(1 + 2y) = 1 + 2 + 4y = \dots,$$

これは以下の式へと導きます。

$$-1 = 1 + 2 + 4 + 8 + \dots$$

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、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。

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? 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 データベースの保守

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` 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)))
```

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!` 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))
```

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)))))))

```

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))))

```

パターンがその下に格納されるテーブル内のキーは(変数で始まる場合には)?、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)))

```

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:** 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 ストリーム命令

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 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` 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` 手續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))))))
```

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 クエリ構文手続

`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` 手續と同じですが、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)))
```

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)))
```

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))
```

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) は query-syntactic-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>78</sup>

---

<sup>78</sup>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

```

(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))))
        ((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)))

```

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))

```

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))))

```

---

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。

`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 フレームと束縛

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 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
        (qeval (first-disjunct disjuncts)
               frame-stream)
        (disjoin (rest-disjuncts disjuncts)
               frame-stream))))
```

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:** 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:** 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 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、单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 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 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:** 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)))
```

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)))
```

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)))
```

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)))
```

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。

`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)
```

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。

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。

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:** 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。

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:** 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”的な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:** 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つても気付くだろう。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:** 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)
```

`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。

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

## レジスタマシンによる演算

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)

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)を生成するのに対し、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。

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*(レジスター) 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。

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。

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。

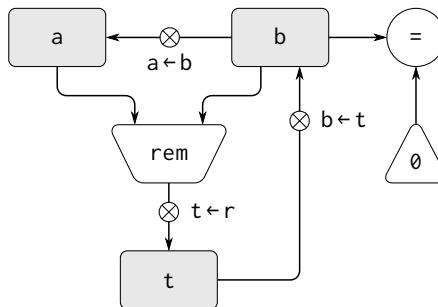
## 5.1 レジスタマシンの設計

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))))
```

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。そうすることでこ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実行さ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。)

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、 $a <- b$ はボタンを押すとレジスタ $b$ の中身を $a$ に割り当てるこを意味します)。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( $a <- b$ の代入の

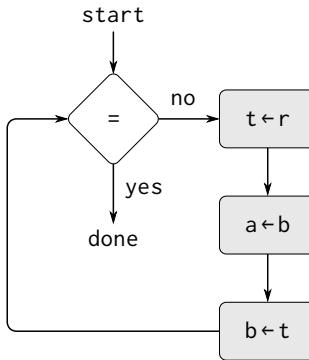


**Figure 5.1:** GCD マシンのデータパス

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。

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。

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



**Figure 5.2:** GCD マシンのコントローラ

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のです。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:** 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 レジスタマシンを記述するための言語

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。

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。

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。

- 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)
- **test(テスト)** 命令、特定のテストを実行する。
- 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。
- 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

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。

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。

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  $t <- r$  を押せ”と言い、別に(データパスの中で)“ボタン  $t <- r$  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(データパスの中で)“= 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。データパスの記述は省略し、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。

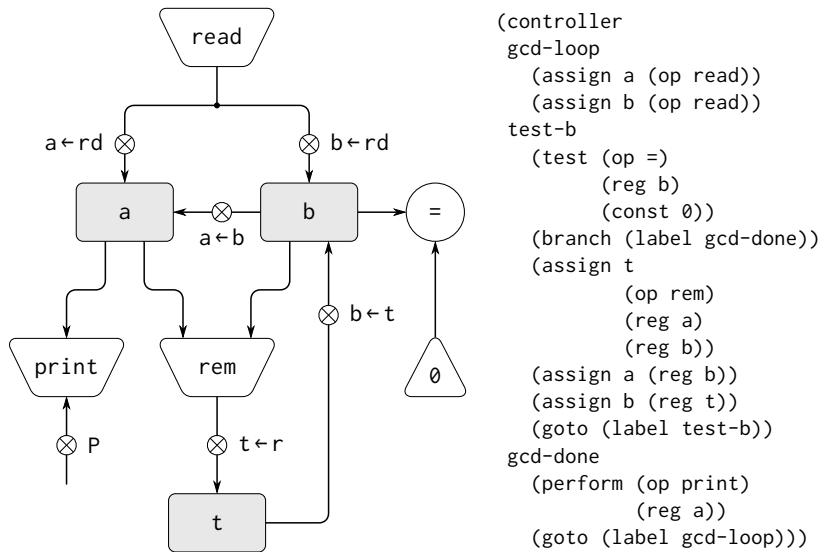
```
(controller
  test-b
    (test (op =) (reg b) (const 0))
    (branch (label gcd-done))
    (assign t (op rem) (reg a) (reg b))
    (assign a (reg b))
    (assign b (reg t))
    (goto (label test-b))
  gcd-done)
```

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。

- 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。
- 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。

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:** 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。



**Figure 5.4:** 入力を読み込み結果を表示する GCD マシン

## アクション

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>1</sup>

**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、記述すること他の任意の値を求める命令と

<sup>1</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。

全く同様に行います。

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 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 機械設計における抽象化

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。

GCD マシンについて考えます。マシンはレジスタ `a` と `b` の中身の剰余を求める、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剰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剰余を求めるのかを指定しなければなりません。実際に、この方法で剰余を見つける 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)))
```

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))
```

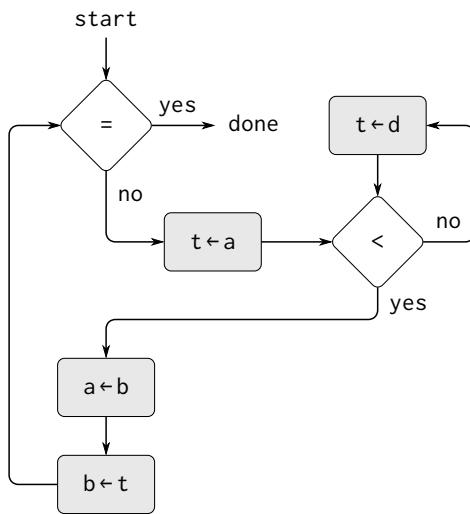
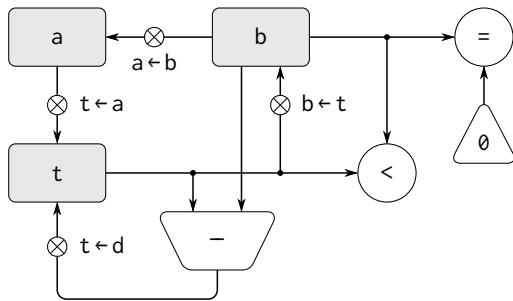
Figure 5.6に示されるように、ループを含む一連の命令により置き換えることができます。

**Figure 5.6:** ↓ Figure 5.5の GCD マシンのコントローラの命令列

```
(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:** 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))
```



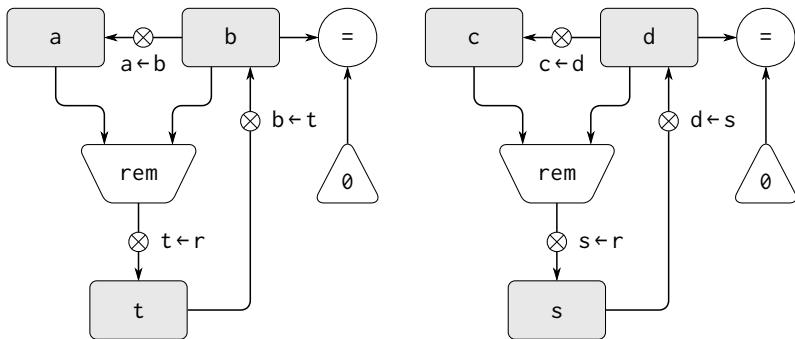
**Figure 5.5:** 精緻な GCD マシンのデータパスとコントローラ

`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 サブルーチン

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。

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。



```

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:** 2 つの GCD 演算を持つ機械のデータパスとコントローラシーケンスの一部

**Figure 5.8:** ↓ 2 つの異なる GCD 演算に対して同じデータパスコンポーネントを使用する機械のコントローラシーケンスの一部

```

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

```

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:** ↓ Figure 5.8 でコントローラシーケンスの重複を防ぐため `continue` レジスタを用いる

```

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))

...
;; 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
...
;; 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

```

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。しかもしも数多くの 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そうです。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ツ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ツ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。

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ツのみでなく、レジスタの中により表されたエントリポインツ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))

...
;; 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
...
;; 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

```

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` は完了した時点での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 再帰実装にスタックを使用する

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。

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)))
```

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、

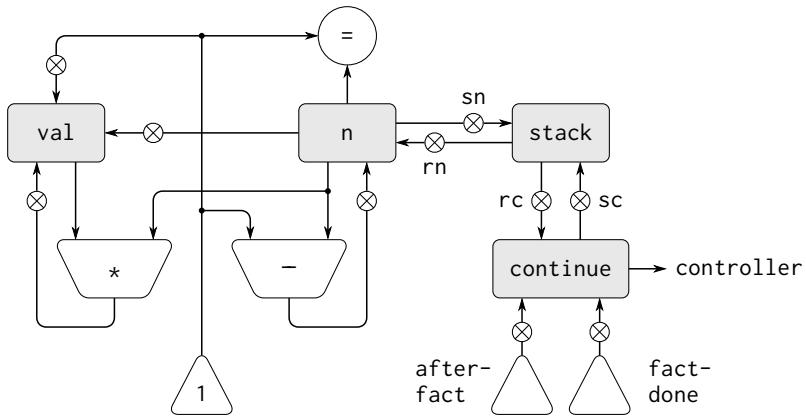
```
(define (gcd a b)
  (if (= b 0) a (gcd b (remainder a b))))
```

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 手続と、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、機械のデータパスを同じコントローラ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。

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に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。

それにもかかわらず、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を再利用し、そして中断した演算を続けることが可能です。

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



```

(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: 再帰階乗マシン

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>2</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>3</sup>

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

---

<sup>2</sup> 古い  $n$  を保存する必要は無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>3</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。

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。

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の機械は實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）。

## 二重再帰

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))))

```

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 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
)
(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:** 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 再帰指數計算

```

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

```

b 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:** 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 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 命令の要約

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  $\langle input_i \rangle$  は  $(reg \langle register-name \rangle)$  か  $(const \langle constant-value \rangle)$  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>))
```

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>))
```

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>)
```

ここまでで見た  $\langle constant-value \rangle$  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),  
(const ()) は空リスト
```

## 5.2 レジスタマシンシミュレータ

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>)
```

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>)
```

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>)
```

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>)
```

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。

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)))
```

`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 手續(つまり、同じ入力値を与えられ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。

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
```

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:** 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 マシンモデル

`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` 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)))
```

### レジスタ

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))

```

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))

```

## スタック

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))
```

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))
```

## 基本的な機械

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`(レジスターテーブル) 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。

`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は使用されます。`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。

`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:** ↓ 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)
            (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)))

```

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` 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。

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)

```

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 アセンブラー

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。

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。

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っ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。

`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 つ目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` 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` 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>4</sup>

---

<sup>4</sup>`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、`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))))
```

`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))
        (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)))
```

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 '()))
_____
(if (symbol? next-inst)
  (cons insts
        (cons (make-label-entry next-inst insts)
              labels))
  (cons (cons (make-instruction next-inst) insts)
        labels))))
```

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)))
```

`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。

```
(define (instruction-text inst) (car inst))
(define (instruction-execution-proc inst) (cdr inst))
(define (set-instruction-execution-proc! inst proc)
  (set-cdr! inst proc))
```

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。[\(Exercise 5.16参照\)](#)

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))
```

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:** 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
```

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 各命令に対する実行手続の生成

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)))))
```

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 命令

`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` 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))


```

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 (アセンブラー実行時) 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。

`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。その次に `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 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, goto 命令

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))
```

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))

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))

```

## 他の命令

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))

```

`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))

```

## 部分式の実行手続

`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)))

```

`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`, `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—に対する実行手續を生成します。

```

(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)))))


```

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))
```

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 手續を適用します。シミュレーション手續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:** 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:** 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:** 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)
```

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`) 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`) 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`) 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:** 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。アセンブリを拡張し以下の情報をマシンモデルに格納せよ。

- 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)
- 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)
- `save` または `restore` されたレジスタの (重複の無い) リスト
- 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))`).

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:** 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 機械のパフォーマンスの監視

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))))
```

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:** 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:** 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:** 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:** 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:** 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。レジスタがトレースの on, off を行うメッセージを受け入れなければならない。レジスタがトレースされている時、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 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>)
```

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)
```

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>)
```

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ない。

```
(cancel-breakpoint <machine> <label> <n>)
```

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 記憶域の割当とガベージコレクション

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。

リスト構造の実装には 2つの考慮点が存在します。1つは純粋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つ目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ょうが、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 ベクタとしてのメモリ

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) または *location*(位置) と呼ばれる個別の名前を持ちます。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。

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>5</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。

---

<sup>5</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。

- (`(vector-ref <vector> <n>)`) はベクタの  $n$  番目の要素を返す。
- (`(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。

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>6</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)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。

## Lisp データの表現

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”的概念を拡張しデータの型の情報を含めることです。<sup>7</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>8</sup> Figure 5.14 はこの手法を用いてリスト((1 2) 3

---

<sup>6</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。

<sup>7</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>8</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

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。

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>9</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>10</sup>

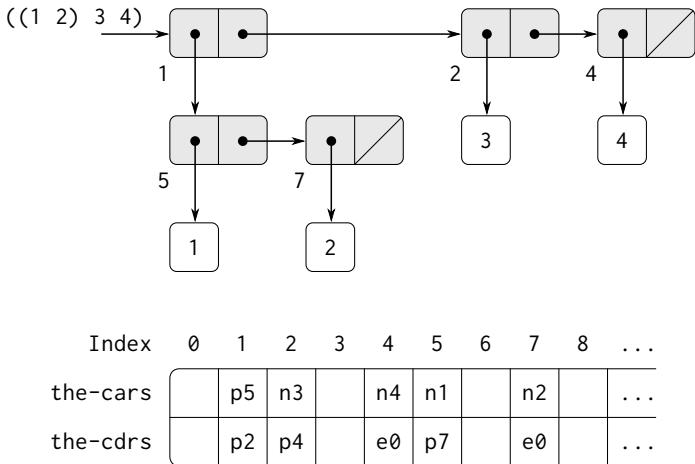
シンボルはその表示内容を形成する文字の列を指定する型付きポインタとして表現されることができるでしょう。この列は Lisp の reader により、最初に入力の中の文字列に出くわした時に構築されます。2つのシンボルのインスタンスが eq? により“同じ”シンボルであると認識されて欲しいことと、eq? にポインタの等価性のための簡単なテストになって欲しいことから、もし reader が同じ文字列を 2 回見た場合、(同じ文字列に対する) 同じポインタを両方の出現に対して表現するために利用することを保証しなければなりません。これを達成するためには、reader は伝統的に *obarray*(オブジェクト配列) 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

---

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。

<sup>9</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。しかもしもし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>10</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:** リスト  $((1\ 2)\ 3\ 4)$  の“箱とポインタ”とメモリベクタの表現

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。

### プリミティブなリスト命令の実装

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。

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>))
```

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>))
```

以下の命令は、

```
(perform (op set-car!) (reg <reg1>) (reg <reg2>))
(perform (op set-cdr!) (reg <reg1>) (reg <reg2>))
```

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` 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>11</sup> 例えれば、以下の命令は

```
(assign <reg1> (op cons) (reg <reg2>) (reg <reg3>))
```

次の一連のベクタ命令として実装されます。<sup>12</sup>

```
(perform
  (op vector-set!) (reg the-cars) (reg free) (reg <reg2>))
(perform
  (op vector-set!) (reg the-cdrs) (reg free) (reg <reg3>))
(assign <reg1> (reg free))
(assign free (op +) (reg free) (const 1))
```

---

<sup>11</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>12</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。

以下の `eq?` 命令は

```
(op eq?) (reg <reg1>) (reg <reg2>)
```

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。

## スタックの実装

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るためです。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。従つて (`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))
```

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))
```

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 ()))
```

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` を行な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:** 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))
```

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:** 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 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 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:** 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 無限のメモリの幻想を維持する

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

します。<sup>13</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)))
```

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、それで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。

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>14</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。

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

---

<sup>13</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。

<sup>14</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。

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>15</sup>

## stop-and-copy ガベージコレクタの実装

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

---

<sup>15</sup>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。

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。

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得ます。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。

データを指し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納し、`root`により指し示させることで準備が行えます。<sup>16</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。

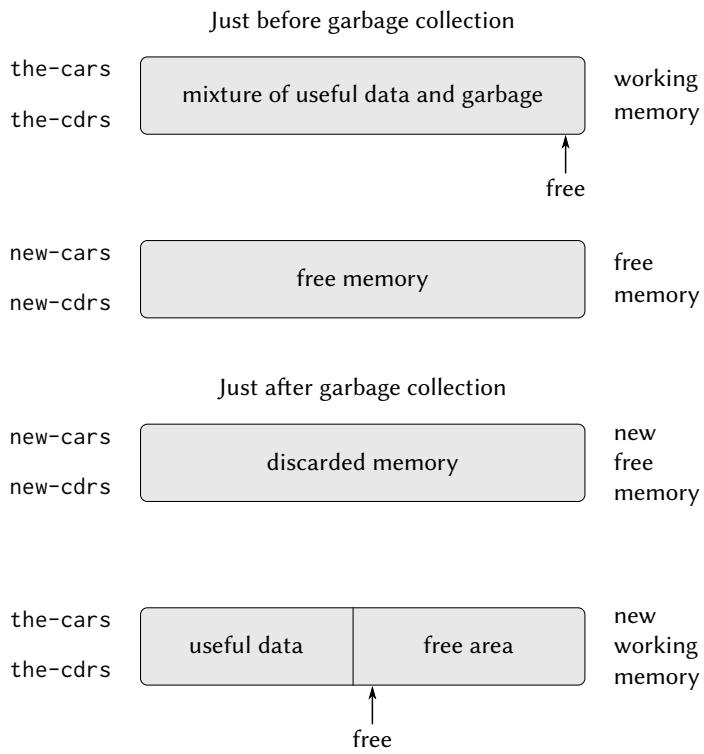
ガベージコレクションは現在のワーキングメモリ内の空きセルが枯渇した時に引き起こされます。それはつまり、`cons` 命令が `free` ポインタをメモリベクタの終端を越えてインクリメントしようとした時です。ガベージコレクションの処理が完了した時、`root` ポインタは新しいメモリの中を指し示し、`root` からアクセス可能な全てのオブジェクトは新しいメモリに移動されています。そして `free` ポインタは新しいメモリ内の新しいペアを割り当てる次の位置を示します。加えて、ワーキングメモリと新しいメモリの役割が交換されます。新しいペアは `free` により指し示される位置から始まる新しい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。

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>17</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。

`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 `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

<sup>16</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。

<sup>17</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。



**Figure 5.15:** ガベージコレクションプロセスによるメモリの再構成

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。

`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))
reassign-root
  (assign root (reg new))
  (goto (label gc-loop))
```

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))
```

`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))
```

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>18</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))
  (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))

```

---

<sup>18</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。

```

  (perform (op vector-set!)
            (reg new-cdrs) (reg new) (reg oldcr))
;; ブロークンハートの構築
  (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))

```

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。

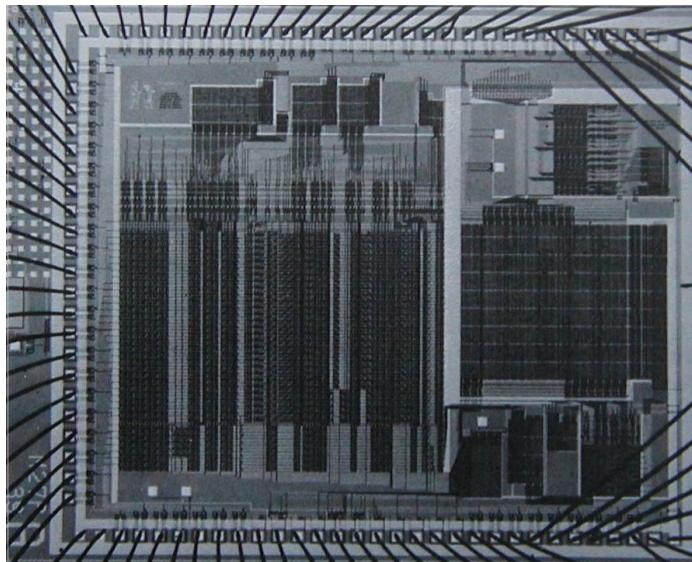
```

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 明示的制御評価機

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、従来の計算機の生来の機械語ととてもそっくりな言



**Figure 5.16:** Scheme 評価機のシリコンチップ実装

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>19</sup>

---

<sup>19</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。

## レジスタと命令

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。

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`, `argl`, `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。

私達はデータバ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ないません。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 明示制御評価機の核

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` 手續に対応します。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

れる式の構文型上の事例分析です。<sup>20</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))
```

## 単純な式の評価

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。

---

<sup>20</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。

```

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))

```

`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。

## 手続適用の評価

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>21</sup>

---

<sup>21</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

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))
```

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>22</sup>

---

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あります。これはそうでなければガバージコレクションされ、再使用するために領域が解法されたはずです。

<sup>22</sup>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)))
```

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)))
```

```

ev-appl-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)

```

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))

```

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。そのオペランドはその後 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))
```

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>23</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))
```

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>24</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。

## 手続適用

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

---

<sup>23</sup>最後のオペランドの処理の特別な最適化は *evlis 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>24</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)。

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))
```

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へ飛びます。

```
primitive-apply
  (assign val (op apply-primitive-procedure)
         (reg proc)
         (reg argl))
  (restore continue)
  (goto (reg continue))
```

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` 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 列の評価と末尾再帰

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。

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))
```

手続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。

`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。

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))
```

## 末尾再帰

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)))
```

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はありません。<sup>25</sup> `sqrt-iter` のような手続を、手続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。

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>26</sup>

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>27</sup>

```
ev-sequence
  (test (op no-more-exp?) (reg unev))
```

---

<sup>25</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。

<sup>26</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>27</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))
```

```

(branch (label ev-sequence-end))
(assign exp (op first-exp) (reg unev))
(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-end
  (restore continue)
  (goto (reg continue))

```

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、この変更は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。この違いは重大事に成り得ます。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)))

```

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 条件文、代入、定義

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。

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
```

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))
```

## 代入と定義

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)))

```

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:** 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>28</sup>

**Exercise 5.24:** `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 `eval-sequence` を使用してその節のアクションを評価する。

**Exercise 5.25:** 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 評価機を実行する

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。

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>29</sup>

---

<sup>28</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。

<sup>29</sup> 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。

`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))
```

```

read-eval-print-loop
  (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))

```

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>30</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))

```

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>31</sup>

---

```
(define (get-global-environment) the-global-environment)
```

<sup>30</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>31</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。

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) )))
```

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)))
```

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))
(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)
```

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。

## 評価機のパフォーマンスの監視

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
```

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
```

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:** 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))
```

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  $n!$  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 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:** 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)))
```

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push 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		

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:** 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:** 木再帰フィボナ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  $n \geq 2$  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  $n \geq 2$  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  $n$  の指數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:** 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>32</sup> 本物のエラーシステムを動かせることは大き

---

<sup>32</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、

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 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 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きっと酷い物だ。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 コンパイル

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

---

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。

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。

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>33</sup>

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。

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。

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

---

<sup>33</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に特に便利な、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。

令列へと翻訳します。<sup>34</sup>

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ます。

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。

## コンパイラの概要

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

<sup>34</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。

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。

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)とオペランド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 (arg1 内の) その引数に適用します。

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ためです。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。

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))
```

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。

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 コンパイラの構造

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。

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` 手續、Section 4.1.7の `analyze` 手續、そしてSection 5.4.1の明示的制御評価機のエントリポイント `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>35</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)
         (compile-variable exp target linkage))
        ((assignment? exp)
```

<sup>35</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 手續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 の手續を使用しました。)

```

  (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)))

```

## ターゲットとリンク記述子

`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内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。

- 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)
- 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)
- 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)

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))
```

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))
```

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。

## 命令列とスタックの使用法

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。

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⟩)
```

次の列を生成します。

```
⟨seq1⟩
⟨seq2⟩
```

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⟩) ⟨seg1⟩ ⟨seg2⟩)` は、 $\langle seq_1 \rangle$  と  $\langle seq_2 \rangle$  がどのように  $\langle reg_1 \rangle$  と  $\langle reg_2 \rangle$  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。

$\langle seq_1 \rangle$	( <b>save</b> $\langle reg_1 \rangle$ )	( <b>save</b> $\langle reg_2 \rangle$ )	( <b>save</b> $\langle reg_2 \rangle$ )
$\langle seq_2 \rangle$	$\langle seq_1 \rangle$	$\langle seq_1 \rangle$	( <b>save</b> $\langle reg_1 \rangle$ )
	( <b>restore</b> $\langle reg_1 \rangle$ )	( <b>restore</b> $\langle reg_2 \rangle$ )	$\langle seq_1 \rangle$
	$\langle seq_2 \rangle$	$\langle seq_2 \rangle$	( <b>restore</b> $\langle reg_1 \rangle$ )

`preserving` を用いて命令列を組み立てることにより、コンパイラは不必要的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。

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。

命令列は 3 つの情報を持ちます。

- 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）
- 列内の命令によりその値が変更されるレジスタ集合
- 列内の実際の命令 (*statements*（命令文）とも呼ばれる）

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))
```

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、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))))
```

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 '() '() '()))
```

命令列を組み立てる手続は [Section 5.5.4](#) に示します。

**Exercise 5.31:** 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:** `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 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 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と提案した。あなたはこの考えをどう思うか?

### 5.5.2 式のコンパイル

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。

#### リンクコードのコンパイル

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>36</sup>

<sup>36</sup> 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、リスト内のカンマで合図された物全てを評価することが異なります。

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)))
```

次のリストとして評価されます。

```
((goto (label branch25)))
```

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))
```

```
(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)))))))
```

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)
  (preserving '(continue)
    instruction-sequence
    (compile-linkage linkage)))
```

## 単純な式のコンパイル

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)))
```

---

次のリストとして評価されます。

(1 2 a)

```

`((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)))))))

```

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いう物は `env` レジスタを必要とします。

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られ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)))))))

```

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。)

## 条件式のコンパイル

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 nextafter-if⟩
false-branch
  ⟨代替部与えられたターゲットリンク記述子⟩
after-if

```

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>37</sup> このコードの準備で

---

<sup>37</sup> 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

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))))
```

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)))))
```

```

        (branch (label ,f-branch)))))
(parallel-instruction-sequences
  (append-instruction-sequences t-branch c-code)
  (append-instruction-sequences f-branch a-code))
after-if))))))

```

`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。

`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。

## 列のコンパイル

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))))))

```

## lambda 式のコンパイル

`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)

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>38</sup>

```
(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
```

---

<sup>38</sup> 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))
```

```

(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` 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、そこに置くことが便利だから、その列の中にあります。

`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ポイントに対するラベルで開始します。次に来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 組み合わせのコンパイル

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>
```

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。

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))))))

```

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` 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。

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 ()))
```

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)))))))

```

## 手続の適用

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続を取り扱うのかについて見てていきます。手続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 arg1))
  <リンクコード>
after-call
```

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 arg1)
        (list target)
        `((assign ,target
          (op apply-primitive-procedure)
          (reg proc)
          (reg arg1)))))))
  after-call)))))

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。

```

## コンパイル済み手続の適用

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ツを持ちます。このエントリポインツにてコードは `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-app)` 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
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

```

このコードは手続が `proc-return` に戻るように `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。）

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>39</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))

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に 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))

```

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`

---

<sup>39</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。

(`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))
```

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。

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 (必要の無い) 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>40</sup>

`codecompile-proc-appl` 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>41</sup> また

<sup>40</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な無い言語内で、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ー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。)

<sup>41</sup> 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 argl continue))
```

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)
           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 命令列のコンパイル

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)))
```

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)))
```

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。

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。

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))
```

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 s1) s2) (list-difference (cdr s1) s2))
        (else (cons (car s1)
          (list-difference (cdr s1) s2)))))
```

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除去して変更します。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>42</sup>

```
(define (preserving regs seq1 seq2)
  (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)))))
```

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。

---

<sup>42</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` 手続を提供します。

```
(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` 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 コンパイルされたコードの例

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)
```

`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` 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))
```

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。その後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 argl)
    (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))

```

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` 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

```

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 argl 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 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

```

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))
```

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:** 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)))))
```

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:** 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))
```

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-call6))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch8

```

```

(assign val
        (op apply-primitive-procedure)
        (reg proc)
        (reg argl))
after-call6 ; 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-call9))
(assign val (op compiled-procedure-entry) (reg proc))
(goto (reg val))
primitive-branch11
(assign val
        (op apply-primitive-procedure)
        (reg proc)
        (reg argl))
after-call9 ; 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:** 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-call123
after-lambda15
(perform (op define-variable!)
        (const f)
        (reg val)
        (reg env))
(assign val (const ok))

```

**Exercise 5.36:** 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:** 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方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、生成された不要な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:** 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 `argl` 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>43</sup>

```
(assign
  val (op lookup-variable-value) (const a) (reg env))
(assign val (op +) (reg val) (const 1))
```

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。

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ぬ。`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>44</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。

<sup>43</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>44</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、これらの名前で手続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。

- a 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` はオペラントのリストを取り、与えられたオペラント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。オペラントが 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ト評価の間に維持されなければならない。
- b 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 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 `+` 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 レキシカルアドレッシング

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)))
```

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)

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>45</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.

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。

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

---

<sup>45</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。

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  $\langle e1 \rangle$  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  $\langle e2 \rangle$  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)
```

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](#)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:** 新しい検索命令を実装する `lexical-address-lookup` 手続を書け。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>46</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:** 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` の中で拡張せよ。

**Exercise 5.41:** 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
```

**Exercise 5.42:** 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` を返す場合においては（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>47</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

<sup>46</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。

<sup>47</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の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

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)。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:** 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:** 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)))
```

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

---

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。

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 コンパイル済みコードと評価機の連結

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、評価機のドライバループへと入る手続 `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
```

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>48</sup>

---

<sup>48</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, 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。

```

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))

```

`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ツは継続が `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。

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ツへと飛ばします。<sup>49</sup>

```

(branch (label external-entry)) ; flag が立っていれば飛ぶ
read-eval-print-loop
  (perform (op initialize-stack))
  ...

```

---

<sup>49</sup>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` 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))

```

`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>50</sup>

```
external-entry
  (perform (op initialize-stack))
  (assign env (op get-global-environment))
  (assign continue (label print-result))
  (goto (reg val))
```

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つて欲しいため、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
```

---

<sup>50</sup> 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))))
```

```

(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)))

```

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

```

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。

## 逐次翻訳とコンパイル

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>51</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

---

<sup>51</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。

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>52</sup>

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可能になります。<sup>53</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。

---

<sup>52</sup> 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得ます。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。

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>53</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

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>54</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:** 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を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、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 は、評価機が上で与えられた再帰階乗手続きを用いて  $n!$  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。

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。

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

---

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>54</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。

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 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:** Exercise 5.45のような分析を木再帰フィボナッチ手続き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)))))
```

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:** 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きがプリミティブとコンパイルされた手続きのみでなく、逐次翻訳された手続き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。実際に手続き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 が立っていれば  
飛ぶ  
read-eval-print-loop ...
```

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:** 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 - インタプリタ間の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:** 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。言い換えれば、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:** 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:** 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:** [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なせよ。[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。

## 参考文献

- 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.

# 課題リスト

## 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								

# 図一覧

## 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		

# 索引

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*)

- |  |                                   |
|--|-----------------------------------|
| <i>k</i> -term finite continued fraction, 73 | assertions, 472                   |
| <i>n</i> -fold smoothed function, 80         | assignment operator, 230          |
| <i>n</i> 次畳み込み補間関数, 80                       | atomically, 333                   |
| abstract models, 94                          | automatic storage allocation, 573 |
| abstract syntax, 389                         | average damping, 72               |
| abstraction barriers, 84, 91                 | B-trees, 165                      |
| accumulator, 120, 235                        | backbone, 282                     |
| acquired, 331                                | backquote, 617                    |
| action, 535                                  | backtracks, 443                   |
| additive, 189                                | balanced, 116                     |
| additively, 85, 179                          | barrier synchronization, 336      |
| address, 573                                 | base address, 574                 |
| address arithmetic, 573                      | Bertrand's hypothesis, 352        |
| agenda, 297                                  | bignum, 575                       |
| aliasing, 245                                | bindings, 248                     |
| and-gate, 290                                | binds, 28                         |
| AND ゲート, 290                                 | binomial coefficients, 42         |
| applicative-order, 425                       | block structure, 30               |
| arbiter, 334                                 | bound variable, 28                |
| arguments, 6                                 | box-and-pointer notation, 101     |
| assembler, 552                               | breakpoint, 571                   |

broken heart, 582  
bugs, 2  
B 木, 165

cache-coherence, 318  
call-by-name, 428  
call-by-need, 428  
call-by-name, 345  
call-by-name thunks, 346  
call-by-need, 346  
call-by-need thunks, 346  
capturing, 28  
Carmichael numbers, 53  
case analysis, 17  
cell, 332  
chronological backtracking, 443  
Church numerals, 96  
Church-Turing thesis, 411  
clauses, 17  
closure, 85  
code generator, 612  
coerce, 211  
coercion, 205  
combinations, 6  
comments, 129  
compacting, 581  
compilation, 609  
compile-time environment, 650  
composition, 79  
compound data, 83  
compound data object, 83  
compound procedure, 12  
computability, 411  
computational process, 1  
concurrently, 316  
congruent modulo, 52  
connectors, 304  
consequent expression, 18  
constraint networks, 304  
constructors, 86  
continuation procedures, 455  
continued fraction, 73

control structure, 495  
controller, 528  
conventional interfaces, 85, 118  
current time, 300

data, 1, 93  
data abstraction, 83, 86  
data paths, 528  
data-directed, 179  
data-directed programming, 85, 189  
deadlock, 335  
deadlock-recovery, 335  
debug, 2  
deep binding, 405  
deferred operations, 34  
delayed argument, 372  
delayed evaluation, 229, 338  
delayed object, 341  
dense, 219  
dependency-directed backtracking, 443  
depth-first search, 443  
deque, 282  
derived expressions, 398  
digital signals, 290  
dispatching on type, 188  
displacement number, 649  
dotted-tail notation, 108  
driver loop, 408

empty list, 105  
enclosing environment, 248  
entry points, 531  
enumerator, 120  
environment, 9  
environment model, 229  
environments, 248  
Euclid's Algorithm, 49  
Euclidean ring, 224  
evaluation, 5  
evaluator, 384  
event-driven simulation, 290

evlvis tail recursion, 594  
execution procedure, 420  
explicit-control evaluator, 587  
expression, 5

failure continuation, 455  
FIFO, 276  
filter, 63, 120  
first-class, 78  
fixed point, 71  
fixed-length, 169  
forcing, 427  
forwarding address, 582  
frame, 485  
frame coordinate map, 141  
frame number, 649  
framed-stack, 591  
frames, 248  
free, 28  
free list, 577  
front, 276  
full-adder, 292  
function boxes, 290  
functional programming, 241  
functional programming languages, 381

garbage, 580  
garbage collection, 573, 580  
garbage collector, 266  
garbage-collected, 431  
generic operations, 85  
generic procedures, 174, 179  
glitches, 2  
global, 31, 248  
global environment, 9  
golden ratio, 38  
grammar, 449

half-interval method, 69  
half-adder, 290  
Halting Theorem, 413

headed list, 282  
hiding principle, 233  
hierarchical, 101  
hierarchy of types, 208  
higher-order procedures, 58  
Horner's rule, 124

imperative programming, 246  
indeterminates, 214  
index, 574  
instruction counting, 571  
instruction execution procedure, 554  
instruction sequence, 614  
instruction tracing, 571  
instructions, 527, 531  
integerizing factor, 225  
integers, 5  
integrator, 366  
interning, 576  
interpreter, 3, 384  
invariant quantity, 46  
inverter, 290  
iterative improvement, 80  
iterative process, 34

key, 168  
k 項有限連分數, 73

labels, 531  
lazy evaluation, 425  
lexical address, 649  
lexical addressing, 405  
lexical scoping, 30  
linear iterative process, 34  
linear recursive process, 34  
linkage descriptor, 613  
list, 103  
list structure, 103  
list-structured, 89  
list-structured memory, 572  
local evolution, 31  
local state variables, 230

location, 573  
logic-programming, 386  
logical and, 290  
logical deductions, 482  
logical or, 290  
  
machine language, 609  
macro, 398  
map, 120  
mark-sweep, 581  
memoization, 41, 288  
memoize, 428  
merge, 381  
message passing, 95, 197  
message-passing, 235  
metacircular, 387  
Metalinguistic abstraction, 384  
Miller-Rabin test, 57  
modular, 228  
modulo, 52  
modus ponens, 495  
moments in time, 316  
Monte Carlo integration, 240  
Monte Carlo simulation, 238  
mutable data objects, 265  
mutators, 265  
mutex, 331  
mutual exclusion, 331  
  
native language, 609  
needed, 615  
Newton's method, 76  
nil, 105  
non-computable, 413  
non-strict, 426  
nondeterministic, 321  
nondeterministic choice point, 442  
nondeterministic computing, 386, 438  
normal-order, 425  
normal-order evaluation, 386  
  
obarray, 575  
  
object program, 609  
objects, 229  
open-code, 647  
operand, 6  
operator, 6, 418  
or-gate, 290  
order of growth, 42  
ordinary, 199  
OR ゲート, 290  
output prompt, 408  
  
package, 190  
painter, 133  
pair, 88  
parse, 448  
Pascal's triangle, 42  
pattern matcher, 485  
pattern matching, 485  
pattern variable, 474  
pipelining, 316  
pointer, 101  
poly, 215  
power series, 354  
predicate, 17  
prefix, 170  
prefix code, 170  
prefix notation, 6  
pretty-printing, 7  
primitive constraints, 304  
probabilistic algorithms, 54  
procedural epistemology, xxii  
procedure definitions, 12  
procedures, 4  
program, 1  
programming languages, 2  
prompt, 408  
pseudo-random, 237  
pseudodivision, 225  
pseudoremainder, 225  
  
quasiquote, 617  
queries, 471

query language, 471  
queue, 276  
quote, 149

Ramanujan numbers, 365  
rational functions, 223  
RC circuit, 366  
RC 回路, 366  
read-eval-print loop, 8  
reader macro characters, 519  
real numbers, 5  
rear, 276

recursion equations, 2  
Recursion theory, 411  
recursive, 9, 26  
recursive process, 34  
red-black trees, 165  
referentially transparent, 244  
register machine, 527  
register table, 554  
registers, 527  
released, 331  
remainder, 52  
resolution principle, 469  
ripple-carry adder, 295  
robust, 148  
RSA アルゴリズム, 54  
rules, 478

scope, 28  
selectors, 86  
semaphore, 332  
separator code, 170  
sequence, 103  
sequence accelerator, 358  
sequences, 62  
serializer, 323  
serializers, 324  
series RLC circuit, 374  
shadow, 249  
shared, 271  
side-effect bugs, 245

sieve of Eratosthenes, 348  
smoothing, 80  
source language, 609  
source program, 609  
sparse, 219  
stack, 35, 545  
state variables, 34, 229  
statements, 615  
stop-and-copy, 580  
stratified design, 147  
streams, 229, 337, 338  
strict, 426  
subroutine, 540  
substitution model, 15  
subtype, 208  
success continuation, 455  
summation of a series, 59  
summer, 366  
supertype, 208  
symbolic expressions, 85  
syntactic sugar, 11  
syntax, 388  
systematically search, 442

tableau, 359  
tabulation, 41, 288  
tagged architectures, 575  
tail-recursive, 35, 598  
target, 613  
thrashing, ix  
thunk, 427  
thunks, 427  
time, 315  
time segments, 300  
tower, 208  
tree accumulation, 10  
tree recursion, 37  
truth maintenance, 443  
Turing machine, 411  
type field, 575  
type tag, 184  
type tags, 179

type-inferencing, 376  
typed pointers, 574  
  unbound, 249  
unification, 468, 485, 490  
unification algorithm, 469  
univariate polynomials, 214  
universal machine, 410  
upward-compatible extension, 435  
value, 8  
value of a variable, 249  
variable, 8  
variable-length, 169  
vector, 573  
width, 98  
wires, 290  
wishful thinking, 87  
Y コンビネータ, 418  
zero crossings, 368  
べき級数, 354  
アクション, 535  
アサーション, 472  
アセンブラー, 552  
アトミック, 333  
アドレス, 573  
アドレス演算, 573  
アナログ加算器, 366  
アービタ, 334  
イベント駆動シミュレーション, 290  
インタプリタ, 3, 384  
インデックス, 574  
エイリアシング, 245  
エニュメレータ, 120  
エブリス末尾再帰, 594  
エラトステネスの篩, 348  
エントリポイント, 531  
オブジェクト, 229  
オブジェクトプログラム, 609  
オブジェクト配列, 575  
オペランド, 6  
オペレータ, 6  
カーマイケル数, 53  
ガベージコレクション, 431, 573, 580  
ガベージコレクタ, 266  
キヤツシュー貫性, 318  
キュー, 276  
キー, 168  
クエリ, 471  
クエリ言語, 471  
クロージャ, 85  
クローズ, 17  
グリッヂ, 2  
グローバル, 248  
グローバル環境, 9  
コネクタ, 304  
コメント, 129  
コントローラ, 528  
コンパイル, 609  
コンパイル時環境, 650  
コード生成器, 612  
コールバイニード, 346  
コールバイニードサンク, 346  
コールバイネーム, 345  
コールバイネームサンク, 346  
ゴミ, 580  
サブタイプ, 208  
サブルーチン, 540  
サンク, 427  
シリライザ, 323, 324  
ジェネリック手続, 174, 179  
スコープ, 28  
スタック, 35, 545  
スタックフレーム, 591  
ストリーム, 229, 337, 338  
スラッシング, ix  
スーパータイプ, 208  
セマフォ, 332  
セル, 332  
ゼロ交差, 368

ソースプログラム, 609  
ソース言語, 609  
タイプタグ, 179, 184  
タイプ別処理, 188  
タイムセグメント, 300  
タグアーキテクチャ, 575  
タブロー, 359  
タワー, 208  
ターゲット, 613  
チャーチ・チューリングの定義, 411  
チャーチ数, 96  
チューリングマシン, 411  
デジタル信号, 290  
デッドロック, 335  
デッドロックリカバリ, 335  
デバッグ, 2  
データ, 1, 93  
データパス, 528  
データ抽象化, 83, 86  
データ適従, 179  
データ適従プログラミング, 85, 189  
ドット付き末尾記法, 108  
ドライバープ, 408  
ニュートン法, 76  
ネイティブ言語, 609  
バグ, 2  
バッククオート, 617  
バックトラック, 443  
バックボーン, 282  
バランスが取れた状態, 116  
パリア同期, 336  
パイプライン, 316  
パスカルの三角形, 42  
パターンマッチャ, 485  
パターンマッチング, 485  
パターン変数, 474  
パッケージ, 190  
ベース, 448  
ビッグナンバー, 575  
フィルタ, 63, 120  
フレーム, 248, 485  
フレーム座標マップ, 141  
ブレイクポイント, 571  
ブロック構造, 30  
プリティープリント, 7  
プリミティブ制約, 304  
プログラミング言語, 2  
プログラム, 1  
プロセス, 35  
プロンプト, 408  
ベクタ, 573  
ベルトランの仮説, 352  
ベース(基底)アドレス, 574  
ペア, 88  
ペインタ, 133  
ホーナー法, 124  
ポインタ, 101  
マクロ, 398  
マークアンドスイープ, 581  
マージ, 381  
ミュータブルデータオブジェクト, 265  
ミューテックス, 331  
ミューテータ, 265  
メタ循環, 387  
メタ言語抽象化, 384  
メッセージパッシング, 95, 197, 235  
メモ化, 41, 288, 428  
モジュラ, 228  
モンテカルロシミュレーション, 238  
モンテカルロ積分, 240  
ユニフィケーション, 468, 485, 490  
ユニフィケーションアルゴリズム, 469  
ユークリッドの互除法, 49  
ユークリッド環, 224  
ラベル, 531  
ラマヌジャン数, 365  
リスト, 103  
リスト構造, 103  
リスト構造メモリ, 572  
リスト構造化, 89  
リンク記述子, 613  
リーダマクロキャラクタ, 519  
ルール, 478  
レキシカルアドレス, 649

- レキシカルアドレッシング, 405  
レキシカルスコープ, 30  
レジスタ, 527  
レジスタテーブル, 554  
レジスタマシン, 527  
レープル, 8  
ローカル状態変数, 230
- 一変数多項式, 214  
万能機械, 410  
上位互換性のある拡張, 435  
不動点, 71  
不变量, 46  
不定元, 214  
両頭キュー, 282  
並行, 316  
予定表, 297  
事例分析, 17  
付加的, 85, 179, 189  
代入演算子, 230  
位置, 573  
体系的探索, 442  
依存型バックトラック, 443
- 値, 8  
停止性問題, 413  
先入れ先出し, 276  
先端, 276  
全加算器, 292  
共有, 271  
再帰, 9  
再帰方程式, 2  
再帰理論, 411  
再帰的, 26  
出力プロンプト, 408  
分数関数, 223  
分離符号, 170  
列, 62, 103  
列アクセラレータ, 358  
制御構造, 495  
制約ネットワーク, 304  
前置表記法, 6
- 副作用バグ, 245  
半加算器, 290  
半区間手法, 69  
占領, 28  
厳密, 426  
参照透明, 244  
反復プロセス, 34  
反復改善法, 80
- 可変長, 169  
合成, 79  
名前呼出, 428  
命令, 527, 531  
命令トレーサ, 571  
命令列, 614  
命令型プログラミング, 246  
命令実行手続, 554  
命令数カウンタ, 571  
命令文, 615  
回路, 290  
固定長, 169  
圧縮, 581  
型の階層, 208  
型フィールド, 575  
型付きポインタ, 574  
型推論, 376  
堅牢, 148  
増加のオーダー, 42  
変数, 8  
変数の値, 249  
外部環境, 248  
大域的, 31  
失敗継続, 455  
実数, 5  
実行手続, 420  
密, 219  
導出原理, 469  
局所展開, 31  
希望的観測, 87  
幅, 98  
平均減衰, 72  
年代順バックトラック, 443  
式, 5

- 引数, 6  
 引用, 149  
 強制, 205, 211, 427  
 必要時呼出, 428
- 慣習的インターフェイス, 85, 118  
 成功継続, 455  
 手続, 4, 35  
 手続の定義, 12  
 手続の抽象化, 27  
 手続的認識論, xxii  
 抑留, 576  
 抽象モデル, 94  
 抽象化バリア, 84, 91  
 抽象構文, 389  
 接頭符号, 170  
 接頭辞, 170  
 擬似クオート, 617  
 擬似乱数, 237  
 擬剩余, 225  
 擬除算, 225  
 整数, 5  
 整数化因数, 225  
 文法, 449  
 明示的制御評価機, 587  
 時間, 315  
 時間の瞬間, 316  
 木再帰, 37  
 末尾再帰, 598  
 束縛, 28, 248  
 束縛されない, 249  
 束縛変数, 28  
 枝上げ伝播加算器, 295  
 構文, 388  
 構文糖, 11  
 構文解析, 448  
 機械語, 609  
 正規順序, 425  
 正規順序評価, 16  
 正規順評価, 386  
 法  $n$  に関して合同, 52  
 派生式, 398
- 深い束縛, 405  
 深さ優先探索, 443  
 演算プロセス, 1  
 演算子, 6
- 特殊形式, 11  
 状態変数, 34, 229  
 獲得, 331  
 現在時刻, 300  
 環境, 9, 248  
 環境モデル, 229  
 疎, 219  
 相互排除, 331  
 真理維持, 443  
 確率的アルゴリズム, 54  
 積分器, 366  
 空きリスト, 577  
 空リスト, 105  
 第一級, 78  
 箱と点表記法, 101  
 級数の和, 59  
 終端, 276  
 組み合わせ, 6  
 結果式, 18  
 繼続手続, 455  
 総称命令, 85  
 線形反復プロセス, 34  
 置換, 15  
 置換モデル, 15
- 肯定式, 495  
 自動記憶域割当, 573  
 自由, 28  
 表形式化, 41, 288  
 補間, 80  
 複合データオブジェクト, 83  
 複合手続, 12  
 解放, 331  
 計算不可能, 413  
 計算可能性, 411  
 記号表現, 85  
 評価, 5

- 評価機, 384  
論理プログラミング, 386  
論理和, 290  
論理的推理, 482  
論理積, 290  
赤黒木, 165  
転送先, 582  
述語, 17  
  
逆変換器, 290  
連分数, 73  
連続 RLC 回路, 374  
遅延オブジェクト, 341  
遅延引数, 372  
遅延評価, 229, 338, 425  
適用順序, 425  
適用順序評価, 16  
閉包性, 101  
  
関数型プログラミング, 241  
関数型プログラミング言語, 381  
関数箱, 290  
階層, 101  
階層化設計, 147  
隠蔽する, 249  
隠蔽原則, 233  
集積木, 10  
集積機, 120  
非厳密, 426  
非決定性演算, 438  
非決定性選択点, 442  
非決定的, 321  
非決定的演算, 386  
頭出しリスト, 282  
高階手続, 58  
黄金比, 38

## 奥付

表紙は 1588 年、Agostino Ramelli のブックホイールのメカニズムです。これは初期のハイパーテキストナビゲーション支援と見ることができるのではないかでしょうか。この版画のイメージは [New Gotland](#). の J. E. Johnson により提供されています。

タイプフェイスは本文は Linux Libertine で、見出しは Linux Biolinum です。両方とも Philipp H. Poll の手によります。タイプライターフェイスは Raph Levien による Inconsolata であり、Dimosthenis Kaponis と Takashi Tanigawa により補完された Inconsolata LGC の形式で利用しています。

(日本語版では漢字に IPA フォントを使用させて頂いてます。)

グラフィックデザインとタイポグラフィは Andres Raba により行われました。Texinfo のソースは Perl スクリプトにより LaTeX に変換され、XeLaTeX により PDF にコンパイルされています。図は Inkscape を用いて描かれました。