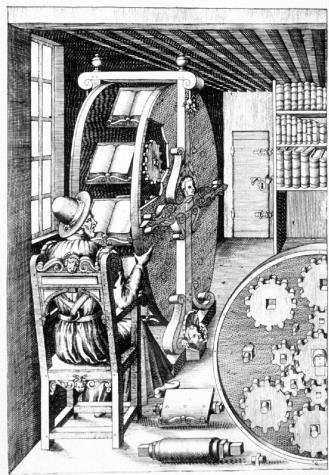


Structure and Interpretation of Computer Programs



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.

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
非公式日本語版翻訳改訂版	xiii
献辞	xiv
前書き	xv
第二版 序文	xx
第一版 序文	xxii
謝辞	xxv
1 手続きを用いた抽象化の構築	1
1.1 プログラミングの要素	5
1.1.1 式	6
1.1.2 命名と環境	8
1.1.3 組み合わせの評価	9
1.1.4 複合手続き	12
1.1.5 手続き適用の置換モデル	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	22
1.1.8	ブラックボックス抽象化としての手続き	26
1.2	手続きとそれが生成するプロセス	32
1.2.1	線形再帰と反復	33
1.2.2	木の再帰	38
1.2.3	増加オーダー	44
1.2.4	指数計算	46
1.2.5	最大公約数	50
1.2.6	例: 素数判定	53
1.3	高階手続きによる抽象の定式化	60
1.3.1	引数としての手続き	61
1.3.2	<code>lambda</code> を使って手続きを構築する	66
1.3.3	汎用手法としての手続き	72
1.3.4	返り値としての手続き	77
2	データを用いた抽象化の構築	85
2.1	データ抽象化入門	89
2.1.1	例: 有理数の数値演算	90
2.1.2	抽象化の壁	94
2.1.3	データとは何か	97
2.1.4	発展問題: 区間演算	100
2.2	階層データと閉包性	104
2.2.1	列の表現	106
2.2.2	階層構造	116
2.2.3	標準インターフェイスとしての列	122
2.2.4	例: 図形言語	137
2.3	記号データ	154
2.3.1	クオート	154
2.3.2	例: 記号微分	158
2.3.3	例: 集合を表現する	163
2.3.4	例: ハフマン符号化木	175
2.4	抽象データの多重表現	184
2.4.1	複素数の表現	186

2.4.2	タグつきデータ	190
2.4.3	データ主導プログラミングと加法性	194
2.5	ジェネリック演算によるシステム	204
2.5.1	ジェネリック算術演算	205
2.5.2	異なる型のデータを組み合わせる	210
2.5.3	例: 記号代数	220
3	モジュール性、オブジェクト、状態	235
3.1	代入と局所状態	236
3.1.1	局所状態変数	237
3.1.2	代入を導入する利点	244
3.1.3	代入を導入することのコスト	249
3.2	評価の環境モデル	256
3.2.1	評価規則	258
3.2.2	単純な手続きの適用	261
3.2.3	局所状態の保管場所としてのフレーム	264
3.2.4	内部定義	270
3.3	可変データによるモデル化	273
3.3.1	可変リスト構造	274
3.3.2	キューの表現	285
3.3.3	テーブルの表現	291
3.3.4	デジタル回路シミュレータ	299
3.3.5	制約伝播	313
3.4	並行性: 期限厳守	325
3.4.1	並行システムにおける時間の性質	327
3.4.2	並行性制御メカニズム	333
3.5	ストリーム	348
3.5.1	ストリーム=遅延リスト	349
3.5.2	無限ストリーム	359
3.5.3	ストリームパラダイムの活用	368
3.5.4	ストリームと遅延評価	382
3.5.5	関数型プログラムのモジュール性とオブジェクト のモジュール性	389

4 メタ言語抽象化	396
4.1 メタ循環評価器	400
4.1.1 評価器のコア	402
4.1.2 式の表現	406
4.1.3 評価器のデータ構造	415
4.1.4 評価器をプログラムとして実行する	419
4.1.5 プログラムとしてのデータ	423
4.1.6 内部定義	427
4.1.7 構文解析を実行から分離する	433
4.2 Scheme の変異版 --- 遅延評価	439
4.2.1 正規順序と適用順序	439
4.2.2 遅延評価のインタプリタ	441
4.2.3 遅延化リストとしてのストリーム	450
4.3 Scheme の変異版 --- 非決定性計算	453
4.3.1 amb と探索	455
4.3.2 非決定性プログラムの例	460
4.3.3 amb 評価器の実装	469
4.4 論理プログラミング	482
4.4.1 推論的情報検索	486
4.4.2 クエリシステムの仕組み	499
4.4.3 論理プログラミングは数学論理学か	509
4.4.4 クエリシステムの実装	516
4.4.4.1 ドライバループと具体化	516
4.4.4.2 評価器	518
4.4.4.3 パターンマッチングによる表明の検索 .	521
4.4.4.4 規則とユニフィケーション	524
4.4.4.5 データベースの保守	529
4.4.4.6 ストリーム演算	533
4.4.4.7 クエリ構文手続き	534
4.4.4.8 フレームと束縛	537

5 レジスタマシンによる計算	542
5.1 レジスタマシンの設計	544
5.1.1 レジスタマシンの記述言語	547
5.1.2 マシン設計における抽象化	551
5.1.3 サブルーチン	554
5.1.4 再帰の実装にスタックを使う	558
5.1.5 命令まとめ	565
5.2 レジスタマシンシミュレータ	566
5.2.1 マシンモデル	568
5.2.2 アセンブラー	573
5.2.3 命令の実行手続きの生成	577
5.2.4 マシンパフォーマンスの監視	585
5.3 記憶領域の割り当てとガベージコレクション	588
5.3.1 ベクタとしてのメモリ	589
5.3.2 無限メモリ幻想の維持	596
5.4 明示制御評価器	604
5.4.1 明示制御評価器のコア	606
5.4.2 列の評価と末尾再帰	612
5.4.3 条件文、代入、定義	616
5.4.4 評価器の実行	618
5.5 コンパイル	625
5.5.1 コンパイラの構造	629
5.5.2 式のコンパイル	634
5.5.3 組み合わせのコンパイル	641
5.5.4 命令列のコンパイル	649
5.5.5 コンパイル済みコードの例	653
5.5.6 レキシカルアドレッシング	665
5.5.7 コンパイル済みコードと評価器の接続	670
参考文献	679
練習問題リスト	687
図一覧	689

索引	690
奥付	700

非公式 Texinfo フォーマット

これは SICP の非公式 Texinfo 版^{訳注 1}の第二版です。

あなたはきっと、これを Emacs の Info モードのような Info ハイパーテキストブラウザで読んでいるのでしょう。もしかすると、TEX で組版されたものを画面上で、または印刷して読んでいるのかもしれませんね。もっとも、それは間抜けなことだと思いますが。それに、印刷するにはだいぶお金がかかりますよね。

HTML-GIF 版は公式に無料で公開されているのですが、それを初めて Unofficial Texinfo Format(非公式 Texinfo 版、UTF) の初版として Lytha Ayth が個人的に変換したのは、2001 年 4 月の長い Emacs Lovefest Weekend の間のことでした。

UTF は HTML 版よりも検索が簡単です。また、あまりよくないコンピュータ(たとえば寄付された 386 マシンのようなもの)を使ってる人にとってもアクセスしやすいでしょう。386 マシンは、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 マシンでは、Netscape とそれに必要な X Window System の両方を動かそうとすると、まだ始めたてでお金もない若いハッカーに *thrashing*(スラッシング) という概念を早い段階から教えるはめになるのがオチです。それに、UTF 版なら圧縮なしで 1.44MB のフロッピーディスクに収まりますので、インターネットや LAN への接続環境のない PC にインストールするのにも便利です。

Texinfo への変換は、可能な範囲で直接的な翻字をしました。TEX から HTML への変換のように、これによってある程度内容が損なわれてしましました。非公式 TexInfo 形式では、図はいにしえの技術であるアスキーアートに

訳注 1 英語版には Texinfo 版がありますが、日本語版にはありません。LATEX ソースと PDF のみです。

よって不格好に復活させられることになりました。また、多量の上付き文字('^')と下付き文字('_')を変換するうちに、曖昧さによる変換ミスが含まれてしまつた可能性も大いにあります。どれが変換ミスかを当てるのは、読者への課題とします。しかし、少なくとも私たちは、“より大きい”という記号を<u>>と符号化して我らの勇敢な宇宙飛行士たちを危険にさらすようなことはしていません。

もしあなたが `sicp.texi` を変更して間違いを直したりアスキーアートを改善したりした場合、`@set utfversion utfversion` の行を更新し、あなたの差分をを反映してください。例えば、もしあなたが Lytha のバージョン 1 から始めていて、あなたの名前が Bob なら、改訂版には `1.bob1, 1.bob2, ..., 1.bobn` のように名前をつけるのがいいでしょう。また、`utfversiondate` も更新してください。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、Web 検索エンジンから見つけやすくなるでしょう。

非公式 Texinfo 形式は、寛大にもフリーに配布されている HTML 版の魂を引き継いでいると信じられています。しかし、いつどこかの法律家の艦隊が、何でもないような小さなことをめぐって大騒ぎをしないとも限りません。ですから、あなたのフルネームを使ったり、あなたのアカウント名やマシン名を含むかもしれない Info, DVI, PostScript, PDF 版を配布したりするのは慎重に考えてからにしてください。

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 CSAIR, MIT OCW.

付録 2: 上記は 2001 年の元の UTF の紹介です。10 年後のいま、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 形式に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時代になって、画面上で PDF を読むことは、もはやバカバカしいことではないとはつきり言えるようになりました。楽しんでください！

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が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

非公式日本語版翻訳改訂版

当面、非公式日本語版をもとに改訂作業を進めています。作業が終わり次第、翻訳改訂版についてのコメントを加える予定です。

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たちにとって、コンピューティングを楽しむことはとても大事だと私は思っています。コンピューティングの始まりの時期には、それはとても楽しいことでした。ご存知のとおり、お金を払うお客様たちが騙されるということがそこかしこで起こり、それからしばらくして、私たちは彼らの不満を真面目に受け取るようになりました。私たちは、あたかもコンピュータをきちんとエラーなしで完璧に使えるようにすることが私たちの責任であるかのように考え始めてしまいました。私はそうは思いません。私たちは、コンピュータを発展させ、新しい方向に向かわせ、家の中で楽しめるようにしておくことについては責任があると思います。コンピュータサイエンスのその分野が、ずっとその楽しさの感覚を失わないでくれればと願っています。特に、私たちが伝道師になつたりしないことを望んでいます。聖書のセールスマンのような感覚を持たないでください。そんな人たちは、もう世界に十分すぎるほどいます。あなたがコンピューティングについて知っていることは、ほかの人たちも学ぶことができます。コンピューティングをうまくやる鍵が、あなたの手の中にしかないとは思わないでください。あなたの手の中にあるものは知性だと、私は思い、またそうであることを願っています。それはつまり、あなたが出会ったマシンに対して、出会ったときよりも多くのものを見ることができる能力です。そして、それをそれ以上のものにすることができる能力です。”

—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、計画を立て (program) ます。軍隊、学生、そして一部の社会は、計画によって動いています。大きな問題を解決しようとするとときは、一連の計画によって行います。それらのほとんどは、途中でひょっこり現れてきたものです。これらの計画は、いま取り組んでいる問題に特化した問題であふれているように見えます。計画を立てることを、それ自身知的な活動として理解するためには、コンピュータプログラミングに目を向ける必要があります。つまり、コンピュータプログラムを読み書きしなければいけないということです。それも、大量に。どういうプログラムか、どういう実際の応用のためであるかはあまり関係ありません。重要なのは、それらがどのようにふるまい、またより大きなプログラムを作る中でそれらがどれだけスムーズにほかのプログラムと協調できるかということです。プログラマは、部品の完全性と、組み合わせたものの妥当性の、両方を追求しなければいけません。この本では、“プログラム”というのは、Lisp のある方言で書かれた、デジタルコンピュータ上で実行されることを前提としたプログラムの、生成、実行、そして研究に焦点を当てています。Lisp を使うことは、プログラムの表記法についての制約を与えるだけで、何をプログラムするかには制約を与えません。

この本の主題について考えるうえで、3つの事象を考えることになります。人間の心、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。コンピュータプログラムはすべて、実際のプロセスや精神的なプロセスのモデルで、人間の心の中で生まれるものです。これらのプロセスは、人間の経験と思考から浮かび上がってくるもので、数はとても多く、細かいところまで入り組んでいて、いつでも部分的に理解することしかできません。それらがコンピュータプログラムとして永遠に満足できるようにモデル化されるなんていうことは、めったにありません。ですから、私たちのプログラムは、注意深く工夫された

バラバラの記号の集合であり、絡まり合う関数の組み合わせでありながらも、絶えず進化していきます。私たちのモデルに対する理解が深まったり、広がったり、一般化したりするたびに、私たちはプログラムに手を加えていきます。最終的に、モデルは準安定状態に達し、それは私たちが取り組む別のモデルの中に位置を占めることになります。コンピュータプログラミングに関する興奮の源は、プログラムとして表現されたメカニズムが私たちの心の中とコンピュータ上で絶え間なく広がっていくことであり、またそれが生み出す知覚の爆発です。芸術が私たちの夢を解釈するものだとすると、コンピュータは夢を、プログラムという形で実行するものなのです！

コンピュータは強力ですが、厳しい親方でもあります。プログラムは正しくなければならず、伝えたいことは細かいところまで正確に伝えなければいけません。ほかのすべての記号的な活動と同じく、プログラムが真であることは論証によって知ることができます。Lisp はそれ自身、意味論（これもまたモデルです）を割り当てることができ、そしてプログラムの仕様が、例えば述語論理などによって決められるのであれば、論理学の証明方法によって、容認可能な正確性の論証をすることができます。残念ながら、プログラムが大きく複雑になるにつれ（いつもそうなるのですが）、仕様自体の妥当性、一貫性、正しさが疑わしいものになっていきます。そのため、完全な形式的な正しさの論証というものが大きなプログラムについてくるということはめったにありません。大きなプログラムは小さなものから成長していくものなので、正しさについて確信を持てるような標準的なプログラム構造（これをイディオムと呼びます）の武器庫を作り上げ、それらを定評のある組織化技術によってより大きな構造に組み立てていくことを学ぶことが、決定的に重要になります。それらの技術については、この本で詳細に扱いますが、それらを理解することは、プロメテウスのような創造的な営みであるプログラミングという分野に加わるために何よりも、不可欠なことです。何よりも、強力な組織化の技術を発見し身につけることは、私たちが大きな、重要なプログラムを書くにあたって必要な能力を育ててくれます。また、逆方向の流れとして、大きなプログラムを書くということはとても大変なことなので、新しい方法を発明して、機能や詳細事項の山を減らしてより大きなプログラムに組み込むことができるようになります。私たちは促されています。

プログラムとは違って、コンピュータは物理法則に従わなければなりません。もし素早く一状態遷移あたり数ナノ秒といったレベルで一動作したければ、電子をほんの短い距離（せいぜい $1\frac{1}{2}$ フィートだけ）しか運ぶことができません。空間的にぎっしり詰め込まれた、膨大な数のデバイスが発する熱は、取り除かなければいけません。高機能性とデバイス密度のバランスを取りながら、

精緻な工学的技法が発展してきました。いずれにせよ、ハードウェアは私たちがプログラムしようとしているレベルよりもプリミティブなレベルで動いています。私たちの Lisp のプログラムを “機械の” プログラムに変換する処理は、それ自身が抽象的なモデルで、それは私たちがプログラムするものです。その研究と作成は、どんなモデルをプログラムするときにも関わってくるような組織的プログラムについて、多くの洞察を与えてくれます。もちろん、コンピュータそれ自身もそのようにモデル化されています。考えてみてください。一番小さい物理的なスイッチング素子は量子力学によってモデル化され、量子力学は微分方程式によって記述され、微分方程式の詳細なふるまいは数値近似によって表され、数値近似はコンピュータプログラムによって表現され、そのコンピュータプログラムはコンピュータによって実行され、そのコンピュータを構成する素子は…！

3つの焦点を別々に設定することは、単なる方法上の便利さのためだけではありません。よく「すべては頭の中のことだ」と言われますが、このように論理的に分けることによって、これらの焦点の間の記号的通信を促進することができます。人間の経験の中で、これら3つの豊かさ、生命力、潜在力を超えるものは、人生そのものの発展ぐらいです。これらの焦点の間の関係は、せいぜい準安定状態です。コンピュータの容量の大きさや速さは、これで十分となることは決してありません。ハードウェア技術の進歩のたびに、より大きなプログラミングの取り組み、新しい組織化原則、そして抽象モデルの発展が引き起こされます。読者は、定期的に“どこに向かっているんだ？どこに向かっているんだ？”と自問しなければなりません。—でも、あまり問い合わせないようにしましょう。そうでないと、ほろ苦い哲学的な便秘のために、プログラミングの楽しさを見逃してしまうことになります。

私たちが書くプログラムの中で、一部のもの（しかし、決して十分にはならない）は、並べ替え、数列の最大値を見つける、素数性判定、平方根の計算といった、精密な数学的な関数を実行します。そのようなプログラムはアルゴリズムと呼ばれ、その最適なふるまいについては多くのこと（特に、実行時間とデータストレージの必要量という二つのパラメータについて）が知られています。プログラマは、よいアルゴリズムとイディオムを身につけなければいけません。プログラムの中には厳密な仕様が記述しにくいものもありますが、プログラマとしては責任を持ってパフォーマンスを推定し、また常に改善しようとしなければなりません。

Lisp は、“生き残ったもの”です。約四半世紀にわたって使われてきています。活発なプログラミング言語の中で、これより長い時間をしてきたものは Fortran だけです。どちらの言語も、重要な応用領域でのプログラミングの必

要性を支えてきています。Fortran は科学計算と工学計算を支え、Lisp は人工知能を支えてきました。この二つの分野は今でも重要であり続け、これらの分野のプログラマは Lisp と Fortran という二つの言語に力を注いでいるので、少なくとももうあと四半世紀の間活発に使われる言語であり続けるということは十分ありうることです。

Lisp は変化します。このテキストで使っている Scheme 方言は、オリジナルの Lisp から進化してきたもので、いくつかの重要な点で異なっています。例えば、変数束縛のための静的スコーピングや、関数が関数を値として返すことを許可することなどです。構文的な構造において、Scheme は Algol 60 に似ています。その近さは、Scheme と初期の Lisp との近さと同じ程度です。Algol 60 は、もう活発な言語になることはないでしょうが、Scheme と Pascal の遺伝子の中に生き続けています。これら二つの言語ほど、かけ離れた文化に取り囲まれ、その流通貨幣となっているような二つの言語というのもなかなかないでしょう。Pascal はピラミッドを建てるためのものです。雄大で、息をのむような、静的な構造物で、軍隊が重い石を運び、所定の位置に押しこむことによって作られています。Lisp は有機体を作るためのものです。雄大で、息をのむような、動的な構造物で、小さなチームが無数のより単純な揺れ動く有機物を所定の位置に入れることによって作られています。どちらの場合にも、組織化のための原則は同じですが、ひとつだけ非常に重要な違いがあります。個々の Lisp プログラムに委ねられたエクスポート可能な関数は、Pascal 企業の中で見つけられるものより桁違いに多いのです。Lisp のプログラムは、ライブラリを関数でいっぱいにします。それらの関数は、それらを生んだアプリケーションを超えて使えるものです。リストという Lisp のネイティブデータ構造は、そのような実用性の成長の主な要因です。リストの単純な構造と自然な適用可能性は、驚くほど普遍的な関数という形で表れてきます。Pascal では、宣言可能なデータ構造の多さが関数の専門化を引き起こし、気軽な連携は禁じられ、ペナルティを与えられるものとなります。ひとつのデータ構造を扱う 100 の関数があるほうが、10 のデータ構造を扱う 10 の関数があるのよりもよいのです。結果として、ピラミッドは千年もの間変化せずに存在しつづけなければならぬのに対し、有機体は存在する限り進化し続けなければなりません。進化を止めるということは滅びることを意味します。

この違いをはっきり知るためにには、この本の中での教材と練習問題の扱いを、何でもいいので Pascal を使った入門テキストと比べてみてください。このテキストは MIT でしか消化できない、MIT に固有の種族のためのものだとかいった妄想は抱かないでください。このテキストは、真面目な Lisp のプログラミングの本はどのようなものでなければいけないかというところを形にしたも

のであって、学生が誰であるかとかどこで使われるかとかは関係がありません。

ひとつ気をつけておいていただきたいのは、この本はプログラミングに関するテキストだということです。ほとんどの Lisp の本は人工知能を使うための準備となっていますが、この本は違います。いずれにしても、対象となるシステムが大きくなるにつれ、ソフトウェア工学と人工知能に関わる重要なプログラミング上の課題は融合していく傾向にあります。まさにそのために、人工知能以外の領域でも、Lisp に対してこれほどの関心が向けられているのです。

その目的とするところから予想できる通り、人工知能の研究は数多くの重要なプログラミングの問題を生み出しています。ほかのプログラミング分野であれば、このような大量の問題に対しては新しい言語が生み出されるところでです。実際に、非常に大きなプログラミングのタスクにおいて、組織化のために便利な方法のひとつは、新しい言語を発明して、タスクモジュールの中にやりとりを閉じ込め、コントロールするということです。これらの言語は、私たちがよく扱うシステムの辺縁に向かうにつれて、どんどんプリミティブでなくなっていく傾向があります。結果として、そのようなシステムでは、複雑な言語処理の機能が何回も繰り返されることになります。Lisp は、非常に単純なシンタックスとセマンティクスを持つため、パージングは初步的なタスクとして扱うことができます。ですので、パージング技術は Lisp のプログラムの中ではほとんど何の役割も果たしません。また、大きな Lisp のシステムが成長し変化していく中で、言語処理系の構築がその成長速度の妨げになることはめったにありません。最後に、まさにこのシンタックスとセマンティクスの単純さこそが、すべての Lisp プログラムが付き合うことになる重荷と自由の源泉なのです。数行以上の Lisp プログラムを書こうとすると、それは必ず自由に使える関数でいっぱいになります。発明し、組み込み (fit) ましょう。かんしゃくを起こし (have fits)、再発明しましょう！括弧のネストの中に自分の考えを書いていく 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、ほかの何にも似ていないということはあるでしょうか。それは捨てられるためのもので、常にシャボン玉のよう見ることがすべてだと。

—Alan J. Perlis

この本で扱われている題材は、1980 年から MIT の入門レベルの計算機科学科目の基礎となっていました。初版の発売時点では、もうこの教材で 4 年間教えていました。それから、この第二版が出てくるまでには、さらに 12 年が過ぎました。私たちの成果が広く受け入れられ、ほかのテキストにも取り込まれていて、私たちはうれしく思っています。私たちは、学生たちがこの本に出てくる考え方やプログラムを身につけ、それらを新しいコンピュータシステムやコンピュータ言語の核として組み込むのを見てきました。古代のタルヌードに出てくる言葉遊びを文字通り実現し、私たちの学生は建築者になったのです。このような有能な学生、功績ある建築者を得ることができたのは幸運なことでした。

この版の準備にあたって、何百ものぼるわかりやすい説明を取り込みました。それらは、私たち自身の教育経験や、MIT やほかの場所の同僚たちのコメントから得られたものです。この本に出てくる主なプログラミングシステムのほとんどを再設計しました。それには一般計算システム、インタプリタ、レジスタマシンシミュレータ、コンパイラが含まれます。さらに、プログラムの例をすべて書き換え、IEEE Scheme 標準 (IEEE 1990) に適合した Scheme 実装であればどんなもの上でもコードを実行できるようにしました。

この版では、いくつかの新しいテーマに重点を置いています。これらの中で最も重要なものは、複数の計算モデルの中で時間を扱う異なるアプローチの中心的な役割を果たしているものです。それらは状態つきのオブジェクト、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。並行性と非決定性については新たに節を設け、本全体を通してこのテーマを統合させるよう力を尽しました。

この本の初版は、MIT の半期の科目のシラバスをなぞったものでした。第二版では、多くの新しい題材を含めたため、半期すべてをカバーするのは無理でしょう。ですので、講師は教えるべき内容を選ぶ必要があります。私たち自身が講義をする際には、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わせますが、その実装 (Section 5.2) はやりません。そして、コンパイラ (Section 5.5) はざっと概要を見るだけです。それでもなお、これは中身の濃い授業です。講師によつては、最初の 3 章か 4 章だけをやって、残りの題材は次の科目に譲りたいという人もいるでしょう。

World-Wide-Web サイト <http://mitpress.mit.edu/sicp> は、この本の読者へのサポートを提供します。それにはこの本に出てくるプログラム、プログラミング練習問題のサンプル、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

コンピュータはバイオリンのようなものです。次のような場面を想像してみてください。ある初心者が人が、まず蓄音機を動かしてみて、それから次にバイオリンを触ってみます。その人は、バイオリンの音はひどいと言います。人間主義者たちや多くの計算機科学者から私たちが聞く話はそういったものです。コンピュータのプログラムは特定の目的にはよいが、柔軟ではないと。バイオリンやタイプライターだってそんなものです。使い方を学ばなければの話ですが。

—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で電気工学や計算機工学を専攻する学生はすべて、4つある“共通コアカリキュラム”的ひとつとして履修しなければなりません。残りは、回路と線形システムについての科目2つと、デジタルシステム設計の科目ひとつです。私たちは、この科目的開発に1978年から携わってきました。そして、1980年の秋からは、この教材の現在の形で毎年600人から700人の学生を教えてきています。ほとんどの学生はコンピュータについてそれまでまったく秩序立ったトレーニングを受けていないか、受けていてもわずかなものでした。とはいっても、コンピュータでちょっと遊んだことのある学生は多くいましたし、プログラミングやハードウェア設計に相当な経験のある学生も数人はいましたが。

私たちがこの計算機科学の入門科目の授業計画を策定するにあたって、二つの懸念点を内容に反映させました。一つ目は、コンピュータ言語はただコン

ピュータに操作を実行させる方法であるにとどまらず、方法論についての考え方を表現するための今までにない形式的な媒体であるという考え方を打ち立てるということでした。ですから、プログラムは人間が読めるように書かなくてはならず、それがたまたま機械によって実行できるというだけのことです。二つ目は、私たちがこのレベルの科目で扱われるべき中心的な題材について信じていることです。教えるべきはことは、特定のプログラミング言語の文法ではなく、また特定の関数を計算するための賢いアルゴリズムでもなく、さらに言うとアルゴリズムの数学的解析やコンピューティングの基礎ですらなく、重要なのは大きなソフトウェアシステムの知的複雑性をコントロールする技術であるということです。

私たちの目標は、この科目を修了した学生たちが、プログラミングのスタイルと美学について、基礎的な感覚を十分に身につけてくれることです。学生たちが、大きなシステムの複雑性をコントロールするための主要な技術を使いこなせるようになること。模範的なスタイルで書かれていれば、50ページにわたるプログラムでも読めるようになっていること。どんなときにも、何を読むべきでないか、何を理解しないでもいいかを判断できるようになること。また、プログラムを変更するにあたって、自信を持って、元の作者の意図とスタイルを保ったままで変更を加えることができるようになっていること。

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特有のものではありません。私たちが教え、また利用する技術は、すべての工学的設計に共通するものです。私たちは、詳細な部分を隠す抽象化を適宜加えることによって、複雑性をコントロールします。私たちは、共通のインターフェイスを確立し、標準的かつ中身のよくわかっている部品を「うまく組み合わせる」やり方で、複雑性をコントロールします。私たちは、設計を記述するためにさまざまな言語を作り出し、それぞれの言語が設計のある特定の側面を際立たせ、それ以外の側面を目立たなくさせることによって、複雑性をコントロールします。

私たちのこの科目に対するアプローチの底にあるものは、“計算機科学”は科学ではなく、その意義はコンピュータとほとんど関係がないという信念です。コンピュータ革命とは、私たちの考え方と、考えを表現するやり方の革命です。この変化を象徴する出来事は、*procedural epistemology*(手続き的認識論)—古典的な数学的テーマで取られるような宣言的な見方ではなく、命令的な見方による知識構造の研究—の出現でしょう。数学は、“何であるか”という考えについて正確に扱う枠組みを提供します。計算機科学は、“どうやるか”という考えについて正確に扱う枠組みを提供します。

この教材では、プログラミング言語 Lisp のひとつの方言を使います。言語を正式に教えることはしません。その必要がないからです。普通に使っていく

だけで、数日のうちに学生は身につけます。これは Lisp 系の言語の大きな利点です。複合式を作るやり方はごくわずかで、構文構造はあってないようなものです。形式的な性質はすべて、一時間もあればカバーできます。チェスのルールのようなものです。少ししたら、言語の構文的な詳細について忘れてしまい（元からそんなものはないので）、実際の問題 — 何を計算したいのか、どうやって問題を扱えるサイズの部品に分解するか、それらの部品をどのように作っていくか — に取り組むことになります。もうひとつの Lisp の利点は、私たちの知る限りほかのどんな言語よりも、プログラムをモジュールに分解する大規模な戦略をサポートしてくれる（しかし強制はしない）というところです。手続きとデータの抽象化、一般的な使い方のための高階関数の使用、代入とデータ変更による局所状態のモデル化、ストリームと遅延評価によるプログラム同士の連携。さらに、組み込み言語の実装も簡単にできます。また、これらすべては、インクリメンタルなプログラム設計、構築、テスト、デバッグに対する優れたサポートを提供する対話的環境に埋め込まれています。これまでにないパワーと優美さを備えたすばらしいツールを作り上げた John McCarthy をはじめ、これまでのすべての世代の Lisp ウィザードたちに感謝の気持ちを捧げます。

Scheme は私たちが使う Lisp の方言なのですが、これは Lisp と Algol のパワーと優美さをひとつにまとめようとしたものです。Lisp からは、シンプルな構文によるメタ言語的なパワー、プログラムをデータオブジェクトとして表す統一的なやり方、ガベージコレクションつきヒープへのデータ割り当てを取り込んでいます。Algol からは、レキシカルスコープとブロック構造を取り込みました。これらは、Algol 委員会のプログラミング言語設計の先駆者たちからの贈り物です。チャーチの入計算とプログラミング言語の構造との間の関係についての洞察は John Reynolds と Peter Landin によるものです。また、コンピュータが世界に現れる何十年も前からこの領域を探索してきた数学者たちに対する恩義も感じています。彼ら開拓者たちには、Alonzo Church, Barkley Rosser, Stephen Kleene, Haskell Curry などが含まれます。

謝辞

この本とこのカリキュラムの開発を手助けしてくださった多くの方々に感謝いたします。

私たちの科目は、明らかに“6.231”の知的な系譜を継ぐものです。それは、1960年代後半に MIT で Jack Wozencraft と Arthur Evans, Jr. が担当していた、プログラミング言語学と λ 計算についてのすばらしい科目でした。

Robert Fano が MIT の電気工学と計算機科学の入門カリキュラムを再編成し、工学設計の原理に重点を置くようにしてくれたことに非常に感謝しています。彼はこの取り組みを始めるよう私たちを導き、最初の科目ノートをまとめてくれました。この本は、それをもとに発展してきたものです。

私たちが教えようとしているプログラミングのスタイルと美学は、その多くが Guy Lewis Steele Jr. と連携して開発したものです。彼は、Gerald Jay Sussman とともに Scheme 言語の初期の開発に携わりました。さらに、David Turner, Peter Henderson, Dan Friedman, David Wise, Will Clinger は、この本に登場する関数プログラミングコミュニティのテクニックを数多く教えてくれました。

Joel Moses は、大きなシステムを構築する方法を教えてくれました。彼は、記号計算に使われる Macsyma システムでの開発経験を通して、制御の複雑性を回避して、モデル化しようとしている世界の現実の構造を反映するデータを体系化することに集中することが重要だということを学んでいました。

Marvin Minsky と Seymour Papert は、プログラミングと、知的生活の中でそれをどのように位置づけるかについて、私たちの姿勢の多くを形作ってくれました。コンピュータがなければ正確に扱うには複雑すぎるような考えを探究するにあたって、コンピュータが表現手段を与えてくれるということを理解できるようになったのは彼らのおかげです。彼らは、学生のプログラムを書き、変更する能力が、自然活動を探究するための強力な方法になると強調してい

ます。

私たちはまた、プログラミングはとても楽しく、プログラミングの楽しみをサポートするよう注意しなければいけないという Alan Perlis の考え方に対する同意します。この楽しみの一部分は、偉大な職人たちの働いているところからも得ることができます。Bill Gosper and Richard Greenblatt のもとで見習いプログラマとして働けたことは幸運なことでした。

私たちのカリキュラムの開発に貢献してくださったすべての方々について名前を挙げるのは困難です。過去 15 年間私たちと共に働き、私たちの科目に多くの時間を費していただいたすべての講師、口答の指導者、チューターたち、中でも Bill Siebert, Albert Meyer, Joe Stoy, Randy Davis, Louis Braida, Eric Grimson, Rod Brooks, Lynn Stein, Peter Szolovits に感謝します。現在はウェルズリーにいる Franklyn Turbak に、教育上の際立った貢献について特に感謝します。彼の学部生向けの授業は、私たち皆が目指す基準を打ち立てました。Jerry Saltzer と Jim Miller には、私たちが並行性のミステリーに取り組むのを手助けしてくださったことを感謝します。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。

多くの方々が、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。中でも私たちが緊密に連携していたのは、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。私たちは、他大学にこの科目を適合させることで主要な教育の賞を受けた同僚たち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 が含まれます。

Al Moyé は、ヒューレット・パッカードの技術者にこの教材を教え、その講義をビデオテープにすることを手配してくれました。これらのビデオを使った生涯教育のコースを計画し、世界中の大学や企業で教えた有能な講師たち、特に Jim Miller, Bill Siebert, Mike Eisenberg に感謝します。

第一版の翻訳には、ほかの国の多くの教育者が多大な労力をつぎ込んでくれました。Michel Briand, Pierre Chamard, and André Pic はフランス語版を、Susanne Daniels-Herold はドイツ語版を、元吉文男は日本語版を製作しました。中国語版は誰が製作したのかわかりませんが、“無許可”翻訳の題材として選ばれたことを光栄に思います。

私たちが教育目的で使用する Scheme システムの開発に技術的な貢献をしたすべての方々の名前を挙げることは困難です。Guy Steele のほかに、主なウイ

ザードには Chris Hanson, Joe Bowbeer, Jim Miller, Guillermo Rozas, Stephen Adams がいます。ほかに多大な時間を費やしてくれた方々は、Richard Stallman, Alan Bawden, Kent Pitman, Jon Taft, Neil Mayle, John Lamping, Gwyn Osnos, Tracy Larrabee, 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, Ruth Shyu です。

MIT の実装を越えて、私たちは IEEE の Scheme 標準仕様に取り組んでくれた多くの人々に感謝したいと思います。その中には R⁴RS 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 が含まれます。

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 をもとにした高校生向けカリキュラムや、Mike Eisenberg や Brian Harvey と Matthew Wright によるすばらしい本のような、意義深い教育上のイノベーションまで含みます。

私たちは、このコースをちゃんとした本にすることを助けてくれた方々、特に MIT 出版の Terry Ehling, Larry Cohen, Paul Bethge に感謝します。Ella Mazel はすばらしい表紙絵を見つけてくれました。第二版については、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 ワイザードである David Jones に感謝します。私たちはまた、新しいドラフトに対して洞察力のあるコメントをしてくださった読者の方々、Jacob Katzenelson, Hardy Mayer, Jim Miller に感謝します。そして特に Brian Harvey には、Julie が彼の本 *Simply Scheme* にしたのと同じような貢献を返してくれたことを感謝します。

最後に、この作品を何年にもわたって支持してくれた団体に感謝します。その中には、Ira Goldstein と Joel Birnbaum が実現してくれたヒューレット・パッカードからの支持と、Bob Kahn が実現してくれた DARPA からの支持を含みます。

1

手続きを用いた抽象化の構築

心が単純な考えについてその力を發揮する活動は、主に次の 3 つです。1. 単純な考えをひとつの合成物に組み合わせること。すべての複雑な考えはこれによってできています。2. 二番目は、単純なものでも複雑なものでも、二つの考えをひとつの場所に並べて置き、それらをひとつに結合することなしに同時に眺めることができます。これによって、心は関係性についての考えを得ることができます。3. 三番目は、その考えの実在に伴うほかのすべての考え方から、その考え方を切り離すことです。これは抽象化と呼ばれ、これによってすべての一般概念というものはできています。—John Locke,
An Essay Concerning Human Understanding (1690)

私たちは、*computational process*(演算プロセス)について学ぼうとしています。演算プロセスというのは抽象的な存在で、コンピュータの中に生息しています。それらが進化するにつれ、プロセスは *data*(データ) と呼ばれるまた別の抽象物を操作します。プロセスの進化は、*program*(プログラム) と呼ばれる規則のパターンによって指揮されています。プログラムを作るのは、プロセスを指揮するためです。実質的に、私たちは呪文を使ってコンピュータの妖精に魔法をかけるのです。

演算プロセスは、魔法使いが妖精について考えることと似ています。それは、見たり触れたりすることはできません。そもそも、物質によって作られて

いないです。それでも、それはとてもリアルな存在です。知的な作業を行うことができます。質問に答えることができます。銀行のお金を支出したり、工場でロボットの腕を操作したりすることで、世界に影響を与えることもできます。私たちがプロセスに魔法をかけるのに使うプログラムとは、魔法使いの呪文のようなものです。それは難解で深遠な*programming languages* (プログラミング言語) による記号的表現によって慎重に組み立てられ、プロセスに実行させたいタスクを指示します。

演算プロセスは、正しく動くコンピュータでは、プログラムを正確かつ精密に実行します。そのため、魔法使いの見習いのように、初心者プログラマは魔法のもたらす結果を理解し、予測することを学ばなければいけません。プログラムのほんの小さな間違い (普通、*bugs*(バグ) や *glitches*(グリッチ、誤作動) と呼ばれます) も、複雑な予期しない結果を引き起こすことがあります。

運のいいことに、プログラムを学ぶことは魔法を学ぶことに比べるとずっと安全です。私たちが扱う妖精は、都合のいいことに安全な方法で封じ込まれているからです。しかし、現実世界のプログラミングでは、注意力、専門知識、そして知恵が必要となります。例えば、CAD(コンピュータによる設計支援) プログラムの小さなバグが、飛行機やダムの破滅的な崩壊を引き起こしたり、産業ロボットの自己破壊を起こしたりするのです。

達人ソフトウェアエンジニアは、プログラムを構築する能力があり、結果となるプロセスが意図するタスクを実行してくれることにある程度自信を持つことができます。彼らは、システムの動作を前もって思い描くことができます。プログラムを構造化して、予期しない問題が破滅的な結果を引き起こさないようにする方法を知っています。そして、問題が起こってしまったときには、プログラムを *debug*(デバッグ、バグ取り) することができます。うまく設計されたコンピュータシステムは、うまく設計された自動車や原子炉のように、モジュール化して設計されていて、部品が別々に作成、置き換え、デバッグできるようになっています。

Lisp プログラミング

プロセスを記述するのには、適切な言語が必要です。この目的のために、私たちはプログラミング言語 Lisp を使うことになります。私たちの日々の考えが普通自然言語 (英語やフランス語や日本語など) によって表現されるように、そして定量的な現象の記述が数学的記法によって表現されるように、私たちの手続き的な思考は Lisp によって表現されます。Lisp は 1950 年代の後半に、*recursion equations*(再帰方程式) と呼ばれるある種の論理表現についての

推論を定式化するものとして発明されたものです。この言語は John McCarthy により考え出され、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) に基づいています。

数学的な定式化としての生まれにもかかわらず、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) は Lisp 言語によっ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、MIT 研究所の人工知能部門の同僚、学生に手伝ってもらいながら、McCarthy が実装したものです。¹ Lisp という名前は List Processing(リスト処理) の頭文字で、記号微分や代数式の積分のようなプログラミング状の問題に取り組む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導入しています。この特徴が、Lisp がその時代のほかの言語から際立ったものとなっている最大の特徴です。

Lisp は、一致協力した設計の努力によってできたものではありません。そうではなく、ユーザの必要とすることや実用的な実装の検討事項に応えるものとして、実験的なやり方で非公式に進化してきたものです。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、この言語の“公式な”定義を公布しようとすることに抵抗してきました。この進化が言語の柔軟性と元々のエレガントさと組み合わざって、広く使われている言語としては二番目に古い(これより古いのは Fortran だけです)言語である Lisp はずっと適応を続け、プログラム設計についての最新の考えを含むようになっています。このため、今では Lisp は方言の集まりで、元々の特徴の多くを共通して持ちながらも、それぞれ大きく違うこともあるものとなっています。この本で使う方言は Scheme と呼ばれるものです。²

¹ *Lisp 1 Programmer's Manual* は 1960 年に登場し、*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 で説明されています。

² 1970 年代に主要な Lisp プログラムの大部分に用いられた 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 研究センターで開発された Interlisp (Teitelman 1974) でした。Portable Standard Lisp (Hearn 1969; Griss 1981)、異なるマシンの間で簡単に移植できるように設計された Lisp 方言です。MacLisp は、カリフォルニア大学バークレー校で開発された Franz Lisp や、MIT 人工知能研究所が Lisp をとても効率良く実行できるよう設計した特定目的プロセッサ(処理機)に基づく 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 Sheme (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、のちに MIT で教育目的で再実装されたものです。Common Lisp 方言 (Steele 1982, Steele 1990) は、初期の Lisp 方言の機能を組み合わせ、Lisp の業界標準を作るために Lisp コミュニティにより開発

実験的な性質を持ち、また記号操作に重点を置いていたため、初期の Lisp は数値計算については、非常に非効率なものでした。少なくとも、Fortran と比べるとそうでした。しかし、年を経るにつれ、プログラムを機械語に変換して、十分効率的に数値計算を実行できるようにする Lisp コンパイラが開発されるようになりました。それに、特別な応用に対しては、Lisp は非常に有効に使われています。³ 今でも Lisp は、絶望的に非効率的だという昔からの評判を乗り越えられていませんが、今では Lisp は効率が最優先事項ではない数多くの応用に使われています。例えば、Lisp は OS のシェル言語や、エディタや CAD システムの拡張言語として選ばれるものになっています。

Lisp がメインストリームの言語でないとしたら、なぜそれをプログラミングの考察のための枠組みとして使うのでしょうか。それは、この言語が独特な特徴を持っているため、プログラミングの重要な構成とデータ構造について学び、それらを支える言語的特性と結びつけるのにとても便利な媒体だからです。これらの特性の中でも最も意義深いものは、プロセスの Lisp による記述（これは *procedure*（手続き）と呼ばれます）が、それ自身 Lisp のデータとして表され、操作できるということです。これが重要なのは、“受動的な” データと“能動的な” プロセスという伝統的な区別を曖昧にする能力を使った強力なプログラム設計のテクニックが存在するからです。これから見ていくように、Lisp は手続きをデータとして扱う柔軟性のおかげで、これらのテクニックを探求するのに最も便利な言語のひとつになっています。手続きをデータとして表現する能力は、コンピュータ言語を支えるインターフリタやコンパイラのような、ほか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のにも、Lisp をとても優れたものにしています。また、これらの考慮点を越えて、Lisp でのプログラミングはとても楽しいのです。

されました。Common Lisp は 1994 年に ANSI 標準 ([ANSI 1994](#)) になりました。

³ そのような特別な応用のひとつは、自然科学的に重要な計算のブレークスルー——太陽系の運動についての積分の精度を二桁近く上げ、太陽系の力学はカオス的であることを示した——でした。この計算は、新しい積分アルゴリズム、専用コンパイラ、そして専用コンピュータによるものですが、これらはすべて Lisp によって書かれたソフトウェアツールの助けによって実装されたものです。（Abelson et al. 1992; Sussman and Wisdom 1992）

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道具というだけではありません。言語は、私たちがプロセスについて考えをまとめる枠組みとしても役に立ちます。そのため、言語を記述する際には、簡単な考え方を組み合わせてより複雑な考え方を作るためにその言語が提供する手段に対して特別な注意を払う必要があります。強力な言語は、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。

- 基本式は、言語に関わる最も単純な実体を表します。
- 組み合わせ方法は、複合要素をより単純なものから構築する方法です。
- 抽象化方法は、複合要素に名前をつけ、単体として扱うための方法です。

プログラミングでは、二つの要素を扱います。それは、手続きとデータです。(のちに、それらがそんなにはっきり分かれるものでないということがわかつてきます。) 大雑把に言うと、データは“物”で、私たちが操作したいものです。手続きは、データを操作するための規則を記述したものです。そのため、強力なプログラミング言語はすべて、基本的なデータや基本的な手続きを記述することができ、また手続きとデータを組み合わせ、抽象化する方法を備えていなければなりません。

この章では、手続きを構築する規則に焦点を当てるため、単純な数値データのみを扱うことになります。⁴ 後の章では、複合データを扱う手続きを構築するのにも、これらの同じ規則が使えるということを見ていきます。

⁴ 数値を“単純なデータ”と見なすのは白々しいごまかしです。実際のところは、数値の扱いは、どんなプログラミング言語でもかなりトリッキーで混乱を招く一面となっています。いくつかの典型的な問題は次のようなものです。コンピュータシステムによっては、2のような *integers*(整数) と 2.71のような *real numbers*(実数) を別物として扱います。実数 2.00 は整数 2 とは違うのでしょうか。整数に使われる算術演算は、実数に使われるものと同じものでしょうか。6 を 2 で割ると、答えは 3 でしょうか、それとも 3.0 でしょうか? どれだけ大きな数値まで表現できるのでしょうか。小数点以下は何桁まで表すのでしょうか。整数の範囲は実数と同じでしょうか。これらの問題の先には、もちろん、丸め誤差や切り捨て誤差に関する問題の山が横たわっています。それは、数値解析の科学そのものです。この本での焦点は大規模プログラム設計であり、数値計算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。この章での数値演算の例では、非整数の演算では有限的小数点以下の精度を保つ算術演算を使い、その場合に使われる通常の丸めの挙動が反映されています。

1.1.1 式

プログラミングを始める簡単な方法のひとつとして、Lisp の Scheme 方言のインタプリタと対話をしてみて、その結果を調べるというやり方があります。コンピューターミナルの前に座っているところを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によって応答します。

タイプできる基本的な式のひとつとして、数値があります。(より正確に言うと、あなたがタイプする式は、10進数の数値を表す数値からなります。) Lisp に次の数値を入力すると、

486

インタプリタは以下の内容を表示することで応答します。⁵

486

数値を表す式は、基本的な手続きを表す式(例えば + や *) と組み合わせることで複合式を作り、数値に対し手続きを適用することを表します。例えば:

(+ 137 349)

486

(- 1000 334)

666

(* 5 99)

495

(/ 10 5)

2

(+ 2.7 10)

12.7

このような式は、括弧の中の式のリストを区切ることにより作られ、手続きの適用を意味するもので、*combinations(組み合わせ)* と呼ばれます。リストの左

⁵ 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との違いを強調したい場合、傾いた文字で表します。

端の要素は *operator*(オペレータ、演算子) と呼ばれ、ほかの要素は *operand*(オペランド) と呼ばれます。組み合わせの値は、演算子によって指定された手続きを、オペランドの値である *arguments*(引数(ひきすう)) に適用することによって得られます。

演算子をオペランドの左に置くというやり方は、*prefix notation* (前置記法) として知られているものですが、数学で慣例となっているやり方とは大幅に違うので、最初は混乱するかもしれません。しかし、前置記法にはいくつかの利点があります。その中のひとつは、任意の数の引数を取る手続きにも対応できるということです。例を以下に示します。

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

```
75
```

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

```
1200
```

演算子はいつでも左端の要素で、組み合わせ全体は括弧で区切られているので、曖昧さの入り込む余地はありません。

前置記法の二つ目の利点は、単純に拡張して、組み合わせを *nested*(ネスト) させることができるということです。つまり、組み合わせの要素がそれ自身組み合わせであるようなものが作れるということです。

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

```
19
```

このようなネストや、Lisp インタプリタが評価できる式の全体としての複雑性には(原則的には)制限がありません。次のような、まだ比較的単純な式に出会った場合、混乱してしまうのは私たち人間のほうです。

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

インタプリタは、迷うことなく 57 だと評価するでしょう。このような式は、次のような形で書くことで、私たち自身にわかりやすいやうにできます。

```
(+ (* 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慣習は、オペランドが垂直に揃うようにそれぞれの長い組み合わせを書くというものなのですが、上の式はそれに従って書いたものです。結果として、字下げが明確に式の構造を示すことになります。⁶

複雑な式でも、インタプリタはいつでも同じ基本的なサイクルで動作します。式を端末から読み、式を評価し、結果を表示するというものです。この操作モードはよく、インタプリタの*read-eval-print loop*(REPL:レップル)モードと呼ばれます。特に、結果を表示することをインタプリタに明示的に指示しなくてもいいというところに注意してください。⁷

1.1.2 命名と環境

プログラミング言語の重要な特徴のひとつは、コンピュータ上のオブジェクトを指すために名前を利用する手段を提供してくれるということです。このことを、*value*(値)がそのオブジェクトである *variable*(変数)を名前によって特定すると言います。

Lisp の Scheme 方言では、`define`(定義)によって対象に名前をつけます。以下のように入力すると、

```
(define size 2)
```

インタプリタは `size` という名前と 2 という値を関連づけます。⁸ `size` という名前が一度 2 という数値に関連付づられると、2 という値を名前によって参照することができるようになります。

```
size  
2
```

⁶Lisp システムは、普通はユーザが式を整形しやすいようにする機構を持っています。その中でも特に役に立つものが二つあります。ひとつは、改行のタイミングで自動的に正しいプリティプリントの位置にインデントするというもので、もうひとつは右括弧が入力されるたびにマッチする左括弧をハイライトするというものです。

⁷Lisp は、すべての式は値を持つという慣習に従っています。この慣習と、Lisp は非効率的な言語であるという昔からの評判を組み合わせて、Alan Perlis は Oscar Wilde のセリフをもじった次のような皮肉を言っています。“Lisp プログラマはすべての値を知っているが、コストについては何も知らない”

⁸この本では、定義を評価した後にインタプリタが返す応答を示すことはしません。それは実装に強く依存するからです。

```
(* 5 size)
```

```
10
```

`define` の使い方をさらに見ていきましょう。

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

`define` は、この言語の持つ抽象化方法のうち、最も単純なものです。これによって、上で計算した `circumference`(円周) のような複合演算の結果を簡単な名前で参照することができます。一般的に、コンピュータ上のオブジェクトというものはとても複雑な構造を持っているので、使うたびにその細かいところを思い出して打ち直さないといけないとしたら、とんでもなく不便だったことでしょう。インタプリタでは、この名前とオブジェクトの関連づけを一連の対話を通して少しずつ作っていくことができるので、このような段階的なプログラムの構築に特に便利です。この特徴は、プログラムの開発・テストを少しずつ進めていくのに向いていて、Lisp プログラムが一般的に大量の比較的単純な手続きから構成されるということの大きな理由となっています。

当然のことですが、インタプリタが値と記号を関連づけ、後から取り出すことができるということは、名前とオブジェクトのペアを記録しておくために何らかのメモリを持っておかないといけません。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) です。後で学ぶように、演算には複数の異なる環境が使われることもあるからです。)⁹

1.1.3 組み合わせの評価

この章の目標のひとつは、手続き的に考えるうえでの問題点を分解することです。とりあえず、組み合わせを評価するにあたって、インタプリタはそれ自身、手続きに従っているということを考えてみましょう。

⁹ Chapter 3では、インタプリタがどのように動作しているかを理解するため、またインタプリタを実装するために、この環境という概念がキーポイントになることを示します。

組み合わせを評価するため、以下のことを行います。

1. 組み合わせの部分式を評価する
2. 部分式の左端(演算子)の値となっている手続きを、引数(オペランド)、つまり部分式の残り値に適用する

こんな単純な規則からも、手続き一般についてのいくつかの重要なポイントがわかります。一つ目のステップは、組み合わせに対する手続きを評価するには、組み合わせのそれぞれの要素に対する評価手続きを先にやらないといけないということを示しています。そのため、評価規則は本質的に *recursive(再帰)* 的なものになります。これは、ステップのひとつとして、その規則自身を呼び出さないといけないということです。¹⁰

再帰という考えが、深くネストした組み合わせ式をどれだけ簡潔に表現できるかというところに注目してください。再帰がなければ、かなり複雑な手続きになるところです。例えば、次の式の評価する場合について考えます。

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

この式を評価するには、4つの異なる組み合わせに対して評価規則を適用する必要があります。この手続きは、Figure 1.1 のように組み合わせを木の形で表すことによってイメージできます。それぞれの組み合わせはノードとして表され、そこから組み合わせの演算子とオペランドに対応する枝が生えています。終端ノード(そこから生えている枝のないノード)は、演算子か数字を表しています。評価を木という形で見ることで、オペランドの値が終端ノードから始まってそれぞれのレベルで組み合わさりながら上に向かって伝わっていく様子がイメージできます。一般的に、階層的な木のようなオブジェクトを扱うためには、再帰はとても強力なテクニックです。この“値を上に向かって伝える”という形の評価規則は、*tree accumulation*(木の集積)として知られています。

次に、一つ目のステップを繰り返して適用することで、ある点で評価する対象が、組み合わせではなく数値や組み込み演算子やその他の名前といった基本式になるというところに注目してください。このような基本的な場合については、以下のように規定することによって扱います。

¹⁰ 評価規則によると、一つ目のステップの一部として、組み合わせの左端の要素を評価しなければいけないことになります。このことを奇妙に思われるかもしれません。この時点では、左端の要素というと、+ や * といった、足し算やかけ算のような組み込みの基本的な手続きを表す演算子しかありえないからです。のちに、演算子がそれ自身組み合わせ式であるような組み合わせを使うことが便利だということを学びます。

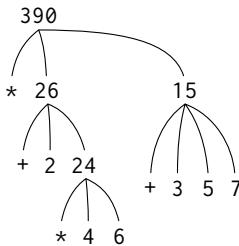


Figure 1.1: 部分組み合わせの値を示した木表現

- 数字の値は、それが示す値である
- 組み込み演算子の値は機械語の列で、それに対応する操作を実行する
- その他の名前の値は、現在の環境でその名前に関連づけられたオブジェクトである

二つ目の規則は、三つ目の規則の特殊なケースと考えることができます。このことは、+ と * といった記号もグローバル環境に含まれていて、一連の機械語命令がそれらの“値”として関係付けられると規定することによって可能になります。ここでキーになるのは、式に出てくる記号の意味を決めるうえで環境の果たす役割です。Lisp のような対話的な言語では、`(+ x 1)` という式の値について考えるのは、`x` という記号に意味を付与する環境についての情報がなければ意味がありません。`+` という記号に対しても、意味を付与するのは環境です。[Chapter 3](#)で学ぶように、評価を行う文脈を提供する存在としての環境という一般的概念は、プログラムの実行について理解するうえで重要な役割を果たしています。

上に書いた評価規則は、定義については扱っていないということに注意してください。例えば、`(define x 3)` を評価する際には、記号 `x` の値と `3` という二つの引数に `define` を適用するということはしません。`define` の目的は、まさに記号 `x` と値を関連づけるということだからです。(つまり、`(define x 3)` は複合式ではないということになります。)

このような、一般的評価規則に対する例外は、特殊形式と呼ばれます。ここまで範囲では、`define` は特殊形式の唯一の例ですが、ほかのものももうすぐ見えていくことになります。それぞれの特殊形式は、固有の評価規則を持っています。それらのいろいろな種類の式(それぞれ関連づけられた評価規則を持つ)は、プログラミング言語の構文を構成します。ほかの多くのプログラミ

ング言語と比べると、Lisp はとても単純な構文を持っています。それは、式の評価規則はひとつの単純な一般規則と少數の特殊な形に対する特殊規則からなるということです。¹¹

1.1.4 複合手続き

ここまでで、強力なプログラミング言語であれば必ず持っているようないくつかの要素について、それが Lisp にもあることを見てきました。

- 数値は基本データで、算術演算は基本手続きである。
- 組み合わせをネストすることで、演算を組み合わせができる。
- 定義は名前と値を関連づけ、抽象化のためにある程度役に立つ。

ここでは、*procedure definitions*(手続きの定義)について学びます。これははるかに強力な抽象化のテクニックで、複合演算に名前をつけ、それにひとつの単位として参照できるようにするというものです。

まずは、“二乗の計算”とはどのように表現できるか考えてみましょう。例えば、“何かを二乗するには、その何かにその何か自身をかける”的なようになるでしょう。これは、私たちの言語では次のように表すことができます。

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

これは、次のように理解できます。

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

¹¹統一された書き方でも書けるものに対して、便利さのために別の表面構造を持たせるような特殊な構文形式は、Peter Landin の造語を使って *syntactic sugar*(シンタックスシュガー)と呼ばれることがあります。ほかの言語の使用者と比べると、Lisp プログラマは一般に、構文の問題をあまり気にしていません。(これと対照的なのが Pascal です。何でもいいので Pascal のマニュアルを開いて、どれだけのページが構文の記述に充てられているか見てみてください。) このように構文を軽視しているのは、ひとつは表面的な構文を変えやすい Lisp の柔軟性のためです。もうひとつは、多くの“便利な”構文構造が言語の統一性を損ない、プログラムが大きく複雑になるにつれ利点よりも欠点のほうが多くなるという観察結果のためです。Alan Peril の言葉を借りると、“シンタックスシュガーはセミコロン(コロン(直腸)とかけて)の癌を引き起こす”ということです。

ここでは、*compound procedure*(複合手続き)を作り、それに `square` という名前をつけています。この手続きは、何かにそれ自身をかけるという演算を表しています。かける数には `x` という名前をつけていますが、これは自然言語で代名詞が果たすのと同じ役割を果たしています。この定義を評価すると、この複合手続きを作成し、それを `square` という名前と関連づけています。¹²

手続き定義の一般形式は以下の通りです。

```
(define (<名前> <仮引数>)
  <本体>)
```

`<名前>` は、環境の中で手続きに関連づける記号です。¹³ `<仮引数>` は、手続きの本体の中で対応する引数を参照するために使う名前です。`<本体>` は、その中に出てくる仮引数をその手続きが適用される実際の引数で置き換えた場合に、手続き適用後の値を返すような式です。¹⁴ `<名前>` と `<仮引数>` は、定義する手続きを実際に呼び出すときと同じように、括弧でくくります。

`square` を定義したので、もうそれを使うことができます。

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

`square` は、ほかの手続きを定義するための構成部品として使うこともできます。例えば、 $x^2 + y^2$ は次のように表現できます。

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

¹² ここで、二つの異なる操作が組み合わされていることに気をつけてください。まず手続きを作成して、それに `square` という名前をつけています。手続きを名前をつけずに作るということと、すでに作られている手続きに名前をつけることとの、二つの概念を区別することは可能であり、重要なことでもあります。そのやり方については [Section 1.3.2](#)で見ていきます。

¹³ この本全体を通して、式の一般的な構文について記述する際には、山括弧でくくったイタリックの記号—例えば、`<name>`—を使って、それらの式を実際に使うときに埋めなければならない“スロット”を表します。

¹⁴ もう少し一般化な言い方をすると、手続きの本体は式の列にもなりえます。その場合、インタプリタは列のそれぞれの式を順番に評価し、最後の式の値を手続き適用の値として返します。

二つの数値が引数として与えられたときにその二乗の和を求める `sum-of-squares` という手続きも、簡単に定義することができます。

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

25

これで、`sum-of-squares` をさらに別の手続きの構成部品として使うこともできるようになります。

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

136

複合手続きは、基本手続きとまったく同じように使うことができます。実際に、上に書いた `sum-of-squares` の定義を見ても、`square` が`+` や`*` のような組み込み手続きなのか、複合手続きとして定義されたものなのか、見分けることはできないでしょう。

1.1.5 手続き適用の置換モデル

演算子が複合手続きを指すような組み合わせを評価するときインタプリタは、[Section 1.1.3](#)で説明した、演算子が基本手続きを指す組み合わせを評価する場合とほぼ同じような手順をたどります。つまり、インタプリタは組み合わせの各要素を評価し、手続き(組合せの演算子の値)を引数(組合せのオペランドの値)に適用するということです。

基本手続きを引数に適用する仕組みは、インタプリタに組み込まれていると考えることができます。複合手続きについては、その適用手順は次のようになります。

複合手続きを引数に適用するには、手続きの本体に出てくる仮引数を対応する引数で置き換えて、それを評価する。

この手順の例として、次の組み合わせを評価してみましょう。

```
(f 5)
```

ここで、`f` は[Section 1.1.4](#)で定義した手続きです。まず、`f` の本体を取得することから始めます。

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

次に、仮引数である `a` を、引数 5 で置き換えます。

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

これによって、問題は二つのオペラントと `sum-of-squares` という演算子の組み合わせの評価ということになります。この組み合わせの評価は、三つの部分問題を持っています。適用する手続きを得るために演算子を評価する必要があり、引数を得るために二つのオペラントを評価しなければなりません。ここで、`(+ 5 1)` の結果は 6 で、`(* 5 2)` の結果は 10 なので、`sum-of-squares` という手続きを 6 と 10 に適用することになります。これらの値によって `sum-of-squares` の本体に出てくる仮引数 `x` と `y` を置き換えて、次の式を得ます。

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

`square` の定義を使うと、これは次の式になります。

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

かけ算によって、次のようにになります。

```
(+ 36 100)
```

そして、最終的には次のようにになります。

136

ここまでで説明した手順は、手続き適用の *substitution model*(置換モデル) と呼ばれます。これは、この章に出てくる手続きに限ると、手続き適用の“意味”を決めるモデルとして捉えることができます。しかし、二つ強調しておくことがあります。

- 置換の目的は、手続き適用について考えやすくするためのもので、インタプリタが実際にどのように動いているかについて記述したものではありません。インタプリタは普通、手続き適用の評価にあたって、手続きのテキストを操作して仮引数を値で置き換えるということはしません。実際には、この“置換”は仮引数のために局所環境を使うことによって実現します。このことについては、Chapter 3 と Chapter 4 で、インタプリタの実装について詳しく調べながら見ていきます。
- この本を通して、インタプリタの動作について、だんだん精巧になっていく一連のモデルを提示していきます。最終的には、Chapter 5 でインタ

プリタとコンパイラの完全な実装にまで到達します。置換モデルは、これらのモデルの最初のひとつ—評価手順について形式的に考えるための最初の一歩となるもの—に過ぎません。一般に、科学や工学で現象をモデル化するときには、単純化した不完全なモデルから始めます。物事を詳細に調べていくにつれ、これらの単純なモデルは不適切になり、より精密なモデルで置き換えなければならなくなります。置換モデルも例外ではありません。特に、Chapter 3で“可変データ”を持つ手続きについて考える場合に、置換モデルが破綻してもっと複雑な手続き適用モデルによって置き換えなければならないことがわかります。¹⁵

適用順序と正規順序

Section 1.1.3での評価の記述によると、インタプリタはまず演算子とオペランドを評価し、評価結果として返される手続きを評価結果として返される引数に適用します。これは、評価を行うための唯一のやり方ではありません。考えられるほかの評価モデルとしては、値が必要になるまでオペランドを評価しないというものがあります。その代わりに、まずオペランドの式を基本演算子しか出てこない式になるまで置き換えてから評価を行います。この方法を使うと、(f 5) の評価は次のような展開の連続によって進みます。

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

それから、次のように簡約されます。

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

136

この方法でも前の評価モデルと同じ値が得られますが、手順が違います。特に、こちらでは (+ 5 1) と (* 5 2) の評価がそれぞれ二回行われます。これは、式

¹⁵置換という考えは、置換手順の厳密な数学的定義をしようと驚くほど複雑になるということがわかっています。この問題は、手続きの仮引数に使われる名前と、手続きを適用する式で使われている（同じである可能性のある）名前とを混同する可能性から来ています。実際に、論理学とプログラミング意味論の文献においては、置換の間違った定義に関する長い歴史があります。置換に関する精密な考察については、Stoy 1977 を参照してください。

$(* x x)$ の x を、それぞれ $(+ 5 1)$ と $(* 5 2)$ で置き換えることに対応しています。

この“完全に展開してから簡約する”というもうひとつの評価方法は、正規順序評価として知られています。それに対して、インタプリタが実際に使っている“引数を評価してから適用する”という方法は、適用順序評価と呼ばれます。置換によってモデル化でき(この本の最初の二章の手続きはすべてそうです)、かつ正当な値を返す手続き適用については、正規順序評価と適用順序評価は同じ値になるということが証明できます。(正規順序評価と適用順序評価が同じ値にならない“不当な”値の例については、Exercise 1.5を参照してください。)

Lisp は適用順序評価を使っています。理由のひとつは、上で見た $(+ 5 1)$ と $(* 5 2)$ のような複数回の評価を避けることによる性能の向上です。そして、もっと大きな理由としては、置換によってモデル化できる手続きの範囲を超えると正規順序評価がとても複雑になることがあります。一方で、正規順序評価は非常に価値のあるツールにもなりえます。そのことについては、Chapter 3 と Chapter 4で一部見ていきます。¹⁶

1.1.6 条件式と述語

この時点で私たちが定義できる種類の手続きは、表現力がとても乏しいものです。というのは、検査を行う方法がなく、また検査の結果によって異なる演算を行うということもできないからです。例えば、次のような規則に従って、数値が正か負かゼロかを検査して絶対値を求めるという手続きを定義することはできません。

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

この構造は*case analysis*(場合分け)と呼ばれるもので、Lisp にはそのような場合分けを記述するための特殊な形式があります。それは `cond` (“conditional”(条件的)の略)と呼ばれ、次のように使います。

```
(define (abs x)
  (cond ((> x 0) x)
```

¹⁶Chapter 3ではストリーム処理を導入します。これは、限定された形の正規順序評価を組み入れることで、“無限”に見えるデータ構造を扱うという手法です。Section 4.2では、Scheme インタプリタに手を加え、Scheme の正規順序バージョンを作成します。

```
((= x 0) 0)
((< x 0) (- x))))
```

条件式の一般形式は以下の通りです。

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

これは、記号 `cond` に続く、 *clause(節)* と呼ばれる括弧でくくった式のペアの列によって構成されます。節の形式は次のようにになります。

```
(< p e)
```

それぞれのペアの最初の式は、 *predicate(述語)*—値が真か偽のどちらかとして解釈される式—です。¹⁷

条件式は、次のように評価されます。まず、述語_(p₁) が評価されます。もしその値が偽なら、_(p₂) が評価されます。もし_(p₂) の値も偽なら、次は_(p₃) が評価されます。この手順は、値が真である述語が見つかるまで続きます。見つかると、インタプリタはその節に対応する *consequent expression(結果式)* _(e) の値を返します。_(p) がどれも真でない場合、`cond` の値は未定義となります。

述語という単語は、真か偽のどちらかに評価される式だけでなく、真か偽のどちらかを返す手続きについても使われます。絶対値の手続き `abs` は、`>`, `<`, `=` という基本述語を使っています。¹⁸ これらは、二つの数値を引数として取り、最初の数が二番目の数に比べて大きいか、小さいか、同じかという検査を行い、その結果に従って真か偽かを返すというものです。

絶対値の手続きは、次のように書くこともできます。

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

¹⁷ “真か偽のどちらかとして解釈される”とは、以下のことを意味します。Scheme では、二つの特殊な値が存在し、`#t`、`#f` という定数として表されます。インタプリタが述語の値をチェックするとき、`#f` を偽として解釈します。ほかの値はすべて真として扱われます。(つまり、論理的に言うと `#t` を定義する必要はないところですが、あったほうが便利です。) この本では、`#t` と `#f` にそれぞれ関連づけた名前として、`true` と `false` を使います。

¹⁸ `abs` は、このほかに“マイナス”演算子 `-` も使っています。これは、`(- x)` のようにオペランドをひとつ取り、符号の反転を表します。

これは、日本語で言うと“もし x がゼロより小さければ $-x$ を返す。そうでなければ x を返す”となります。`else`は特殊な記号で、`cond`の最後の節の $\langle p \rangle$ の代わりに使うことができます。これを使うと、それまでのすべての節がスキップされたときに、これに対応する $\langle e \rangle$ を`cond`の値として返すようにできます。実は、値が常に真として評価される式であれば何でも、こここの $\langle p \rangle$ として使うことができます。

次は、また別の絶対値の手続きの書き方です。

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

ここでは、`if`という特殊形式を使っています。これは、場合分けがちょうど二つの場合からなる場合に使える、制約つきの条件式です。`if`式の一般形式は以下通りです。

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

`if`式を評価するにあたって、インタプリタは最初に式の $\langle predicate \rangle$ の部分を評価します。もし $\langle predicate \rangle$ の評価結果が真である場合、インタプリタは $\langle consequent \rangle$ を評価し、その値を返します。そうでなければ $\langle alternative \rangle$ を評価し、その値を返します。¹⁹

<や=,>のような基本述語に加えて、複合述語を構築するための論理複合演算というものがあります。次の三つは、最もよく使われるものです。

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

インタプリタは式 $\langle e \rangle$ を左から右にひとつずつ評価します。 $\langle e \rangle$ のどれかが偽と評価されると、`and`式の値は偽となり、残りの $\langle e \rangle$ は評価されません。すべての $\langle e \rangle$ が真と評価されると、`and`式の値は最後の式の値になります。

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

インタプリタは式 $\langle e \rangle$ を左から右にひとつずつ評価します。 $\langle e \rangle$ のどれかが真と評価されると、その値が`or`式の値として返され、残りの $\langle e \rangle$ は

¹⁹`if`と`cond`のちょっとした違いとして`cond`の各節の $\langle e \rangle$ は式の列であってもいいということがあります。対応する $\langle p \rangle$ が真になる場合、 $\langle e \rangle$ 内の式は順番に評価され、列の最後の式の値が`cond`の値として返されます。しかし、`if`式の中では $\langle consequent \rangle$ と $\langle alternative \rangle$ は单一の式でなければなりません。

評価されません。すべての $\langle e \rangle$ が偽と評価されると、`or`式の値は偽となります。

- `(not <e>)`

`not`式の値は、式 $\langle e \rangle$ が偽と評価される場合は真になり、そうでなければ偽となります。

`and`と`or`が特殊形式で、手続きではないということに注意してください。これは、部分式がすべて評価されるとは限らないからです。`not`は通常の手続きです。

これらをどのように使うかというひとつの例として、数値 x が $5 < x < 10$ という値域にあるかという条件を次のように表現することができます。

```
(and (> x 5) (< x 10))
```

もうひとつの例として、ある数値がもうひとつの数値と比べて等しいかより大きいかということを検査する述語を、以下のように定義することができます。

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

または、次のように書くこともできます。

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

Exercise 1.1: 以下の一連の式について、それぞれの式に対するインタプリタの応答として表示される結果はどうなるか。式の列は、下記に示した順に評価されるとする。

```
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: 以下の式を前置記法に書き換えよ。

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

Exercise 1.3: 三つの数値を引数として取り、そのうち大きいほうから二つの数値の二乗の和を返す手続きを定義せよ。

Exercise 1.4: 我々の評価モデルでは、演算子が複合式であるような組み合わせが作れるということを観察せよ。この観察結果を使って、次の手続きのふるまいを説明せよ。

```

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

```

Exercise 1.5: Ben Bitdiddle は、自分が対面しているインタプリタが適用順序評価と正規順序評価のどちらを使っているかを調べるテストを発明した。彼は、以下の 2 つの手続きを定義する。

```

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

```

それから、以下の式を評価する。

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

適用順序評価を使うインタプリタでは、Ben はどのようなふるまいを観察するだろうか。正規順序評価を使うインタプリタでは、どのようなふるまいを観察するだろうか。あなたの回答を説明せよ。(特殊形式である `if` の評価規則は、インタプリタが適用順序評価でも正規順序評価でも同じであると仮定せよ。述語式が最初に評価され、その結果によって結果式と代替式のどちらを評価するかが決まる。)

1.1.7 例: ニュートン法による平方根

ここまでで見てきた通り、手続きは普通の数学の関数にとても似ています。どちらも、ひとつ以上の引数によって決定される値を記述します。しかし、数学の関数とコンピュータの手続きの間には、ひとつの重要な違いがあります。手続きは、実効的なものでなければなりません。

一例として、平方根を計算問題について考えてみましょう。`square-root` 関数は、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、完全に正当な数学関数です。これを使って、ある数値がほかの数値の平方根かどうかわかりますし、平方根というものの全般についての事実を導出することもできます。一方で、この定義は手続きを記述するものではありません。ある数値の平方根を実際にどうやって求めるかということについては、ほとんど何も教えてくれません。この定義を Lisp っぽく書き換えてどうにもなりません。

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

これは、ただの循環論法です。

関数と手続きの違いは、物事の属性について説明することと、どうやって物事を行うかについて説明することの一般的な違いを反映したものです。これは、宣言的知識と命令的知識の違いと言われることもあります。数学では普通、宣言的な(何であるか)記述が関心の対象ですが、コンピュータサイエンスでは普通、命令的な(どうやるか)記述が関心の対象です。²⁰

²⁰宣言的記述と命令的記述は、数学とコンピュータサイエンスのように密接に関係し

平方根は、どうやって計算するのでしょうか。最も一般的な方法は、ニュートンの逐次近似法を使うというものです。それによると、数値 x の平方根の推定値として y があるとき、 y と x/y の平均を取るという単純な操作を行うことでより良い推定値(実際の平方根により近い値)が得られます。²¹ 例えば、2の平方根は以下のように計算できます。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

この手順を続けると、平方根の近似解をどんどん良くしていくことができます。

さて、この手順を手続きという形で定式化してみましょう。まず、被開平数(平方根を計算したい数値)と推定値から始めます。推定値が目的に十分な程度によければ、そこで終わりです。そうでなければ、改善した推定値を使って手順を繰り返す必要があります。この基本戦略を、次のような手続きとして書きます。

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

ています。例えば、プログラムによって計算された答えが“正しい”かどうかを決めるには、そのプログラムについての宣言的な文を作ります。これまでに、プログラムの正しさを証明する技術を確立することを目的とした膨大な量の研究がありますが、このテーマの技術的な難しさは、命令的な文(それによってプログラムを構築する)と宣言的な文(物事を推理する)との間の移行について、うまく折り合いをつけるということに関するものです。関連する話として、プログラミング言語設計の現在の重要な領域のひとつは、いわゆる超高校生言語についての調査です。そのような言語では、宣言的な文によって実際にプログラミングを行います。その発想となるものは、インタプリタを十分に高性能なものにして、“何であるか”という知識がプログラマによって与えられると、“どうやるか”についての知識を自動的に生成できるようにしようというものです。これは一般的には不可能ですが、進歩のある重要な領域もあります。この発想については、Chapter 4で再度触れることにします。

²¹この平方根アルゴリズムは、実際にはニュートン法の特殊なケースです。ニュートン法は、方程式の根を求めるための一般的な技法です。平方根アルゴリズム自体は、アレキサンドリアのヘロンにより A.D.1 世紀に開発されたものです。一般的なニュートン法をどのように Lisp の手続きとして表すかについては、Section 1.3.4 で見ていきます。

推定値を改善するには、古い推定値と、被開平数を古い推定値で割ったものとの平均を取ります。

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

average は以下のように定義します。

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

ここで、“十分によい”ということの定義も決める必要があります。次の方法は説明のためには十分ですが、本当はあまりよいテストではありません。
(Exercise 1.7を参照してください) その発想は、答えの二乗と被開平数の差が事前に決めておいた許容誤差(ここでは 0.001 とします)よりも小さくなるぐらいに答えが近くなるまで、答えの改善を続けるというものです。²²

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

最後に、始めるための方法が必要です。例えば、任意の数値の平方根について、1 という推定値を与えることもできます。²³

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

これらの定義をインタプリタに入力すると、`sqrt` をほかの手続きと同じように使えるようになります。

²²普通述語には、述語であるということがわかりやすいように、疑問符で終わる名前をつけます。これはただのスタイル上の慣例です。インタプリタにとって、疑問符はただの通常の文字です。

²³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ください。多くの Lisp の実装では、この二つの間に何の違いもありません。しかし、MIT Scheme は整数と小数を区別し、整数同士の割り算は小数ではなく有理数を返します。例えば、10 を 6 で割ると $5/3$ になりますが、10.0 を 6.0 で割ると 1.6666666666666667 になります。(有理数の演算をどのように実装するかについては、Section 2.1.1で学びます。) もし `square-root` プログラムの初期推定値を 1 として、 x が整数である場合、`square-root` の計算でその後生成される値はすべて、小数ではなく有理数になります。有理数と小数の混合演算は常に小数になるので、初期推定値を 1.0 として始めると、その後計算される値はすべて強制的に小数になります。

```
(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、ここまで紹介した単純な手続き型言語が、例えば C や Pascal で書けるようなものであれば、どんな純粋な数値計算のプログラムでも書けるということを示しています。これにはびっくりされるかもしれません。この言語には、コンピュータに何かを繰り返し実行させる繰り返し(ループ)の構造を含めていないのですから。それに対して `sqrt-iter` は、繰り返しを達成することは手続きを呼ぶという通常の能力だけができるということの実例となっています。²⁴

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。“`cond` を使って普通の手続きとして定義したらいいんじゃないの”と彼女は質問した。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` の新しい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 にプログラムのデモを行った。

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

²⁴繰り返しを実装するために手続き呼び出しを使うことの効率の問題について心配されている読者の方は、Section 1.2.1の“末尾再帰”についての説明を見てください。

喜びながら、Alyssa は new-if を使って 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 が平方根の計算するのにこれを使おうとすると、何が起こるだろうか。説明せよ。

Exercise 1.7: 平方根の計算を使った good-enough? テストは、とても小さい数値の平方根を求めるのにはあまり効率的ではないだろう。また、実際のコンピュータでは、数値演算はほとんど必ず有限の有効数字で行われることになる。この問題があるので、とても大きい数に対しては、我々のテストは不適切なものになる。これらの言明について、小さい数と大きい数でテストがうまくいかない例を使って説明せよ。good-enough? を実装するためのもうひとつ戦略としては、guess が繰り返しごとにどれだけ変化するかを確認して、差分が推定値に対してとても小さいものになったら止めるというものもある。これは、小さい数値や大きい数値に対して、よりうまくいくだろうか。

Exercise 1.8: 立方根に対するニュートン法は、 y が x の立方根の近似値である場合、次の値によってよりよい近似を求めることができるという事実に基づいています。

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

この式を使って、square-root と同じような仕組みの cube-root 手続きを実装せよ。(Section 1.3.4 では、これらの square-root と cube-root を抽象化したものとして、一般的なニュートン法をどのように実装するかを見ていきます。)

1.1.8 ブラックボックス抽象化としての手続き

sqrt は、相互に定義された手続きのセットとして定義されるプロセスの最初の例です。sqrt-iter の定義が recursive(再帰的) であることに注意してください

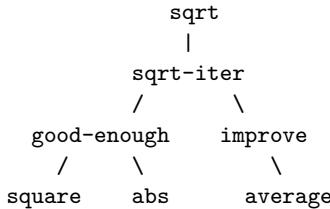


Figure 1.2: `sqrt` プログラムの手続き分解

さい。再帰的というのは、手続きがそれ自身を使って定義されているということです。手続きを定義するのにその手続き自身を使えるという考えには不安になるかもしれません。そんな“循環的な”定義が意味を持つのかわからないし、そんなものでコンピュータが実行できるような明確に定義されたプロセスを記述できるのかというところがはつきりしないように思えるかもしれません。このことについては、Section 1.2でもっと詳しく見ていくことにします。ここでは、`sqrt` の例からわかる、ほかのいくつかの重要な点について考えてみましょう。

平方根を計算するという問題が、自然にいくつかの部分問題に分解されているということに注意してください。推定値が十分によいかどうやって判断するか、どうやって推定値を改善するか、などです。これらのタスクは別々の手続きによって成し遂げられます。`sqrt` プログラム全体は、Figure 1.2に示すような手続きの塊として見ることができます。この構造は、問題を部分問題に分解する様子を反映しています。

この分解戦略の重要な点は、プログラムを部品に分けるということだけのことではありません。分けるだけなら、どんな大きなプログラムでも分解はできます—最初の 10 行、次の 10 行、その次の 10 行、といったやり方で。そうではなく、重要なのは、それぞれの手続きが特定のタスクを成し遂げていて、それがほかの手続きを定義する際にもモジュールとして使えるようになっているということなのです。例えば、`good-enough?` 手続きを `square` を使って定義するとき、`square` 手続きは“ブラックボックス”と見なせます。その時点では、その手続きがどうやって結果を計算するのかには関心がありません。関心があるのは、それが二乗を計算するという事実だけです。二乗がどのように計算されるかという詳細は隠しておいて、後で考えることにすることができます。実際、`good-enough?` 手続きを関しては、`square` は手続きというより手続きを抽象化

したもの、いわゆる抽象手続きと考えることができます。この抽象レベルでは、二乗を計算するものであればどんな手続きでも同じことです。

そのため、返り値だけを考えると、二乗を計算する以下の二つの手続きは区別不可能であるべきです。どちらもひとつの数値を引数として取り、その数値の二乗を返り値として生成します。²⁵

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

ですから、手続き定義は詳細を隠せるようになっていなければなりません。手続きを使う人は自分でその手続きを書いたのではなく、別のプログラマからブラックブックスとして取得したのかもしれません。手続きを使う人が、それがどうやって実装されているかについての知識を求められるべきではありません。

ローカル名

手続きを使う人が気にしないでいいようになっていなければならぬ実装上の詳細としては、手続きを実装した人が仮引数の名前に何を選んだかということがあります。そのため、次の二つの手続きは区別不可能であるべきです。

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

この原則—手続きの意味はその作者が使った仮引数の名前とは独立であるべきということ—は、ちょっと見ると自明なことのように思えますが、その影響範囲は広範にわたります。最も単純な影響としては、手続きの仮引数の名前は手続き本体にローカルなものであるべきだということです。例えば、平方根手続きの中で、`good-enough?` の定義に `square` を使いました。

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

²⁵どちらの手続きがより効率的な実装であるかということさえ、はっきりとはわかりません。それは利用できるハードウェアによります。機械によっては、“自明な”実装が非効率的なものになることもあります。対数と逆対数の巨大なテーブルを非常に効率的に格納している機械があると考えてみるのもよいでしょう。

`good-enough?` の作者の意図は、第一引数の二乗が第二引数と比べて許容差の範囲であるかを決定するということです。`good-enough?` の作者が、第一引数を参照するのに `guess` という名前を使い、第二引数を参照するのに `x` という名前を使っていることがわかります。`square` の引数は `guess` です。`square` の作者が(上の例のように)その引数を参照するのに `x` を使った場合、`good-enough?` の `x` は `square` の `x` とは別の `x` でなければならぬということがわかります。手続き `square` の実行は、`good-enough?` で使われている `x` の値に影響を与えてはいけません。その `x` の値は、`square` が計算を終えた後にも `good-enough?` に必要かもしだいからです。

仮引数がそれぞれの手続きの本体にローカルでなかったとすると、`square` の仮引数 `x` は、`good-enough?` の仮引数 `x` と混同される可能性があるので、`good-enough?` のふるまいが、どのバージョンの `square` を利用するかに依存することになります。つまり、`square` は私たちが望むようなブラックボックスではなくなるでしょう。

手続きの仮引数は、その名前が何であってもかまわないという点で、手続きの定義にとても特別な役割を果たしています。そのような名前は *bound variable* (束縛変数) と呼ばれ、手続き定義は仮引数を *bind*(束縛) するといった言い方をします。束縛変数が手続き定義の中で一貫性を持って改名された場合、手続きの意味は変わりません。²⁶ もし変数が束縛されていなければ、それは *free*(自由) であると言います。束縛によって名前が定義される式のセットは、その名前の *scope*(スコープ) と呼ばれます。手続き定義においては、その手続きの仮引数として宣言された束縛変数は、その手続きの本体をスコープとして持っています。

上記の `good-enough?` の定義の中で、`guess` と `x` は束縛変数ですが、`<`, `-`, `abs`, `square` は自由変数です。`good-enough?` の `guess` と `x` の代わりにどんな名前を選んだとしても、それらが `<`, `-`, `abs`, `square` と違うもので区別可能である限り、`good-enough?` の意味とは独立しています。(仮に、`guess` を `abs` と改名したとすると、変数 `abs` を *capturing*(キャプチャ) するというバグを入れてしまつたことになります。そうすると、`abs` が自由変数から束縛変数に変わってしまいます。) しかし、`good-enough?` の意味は、それが使う自由変数の名前と独立ではありません。その意味は、記号 `abs` が数値の絶対値を求める手続きを指しているという(この定義の外部の)事実に実際に依存しています。`good-enough?` の定義の中で、`abs` を `cos` によって置き換えたとすると、違う関数を計算することになります。

²⁶ 一貫性のある改名という概念は、実際には扱いにくく、形式的な定義が難しいものです。有名な論理学者たちも、この点について恥ずかしい間違いを犯してきました。

内部定義とブロック構造

ここまでで、名前を分離する方法を一種類手に入れました。手続きの仮引数はその手続きの本体にローカルになるというものです。square-root プログラムは、名前の使い方をコントロールするやり方をもうひとつ示してくれます。現在のプログラムは、次のような別々の手続きからなります。

```
(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`を使う人にとって重要な手続きは `sqrt`だけだということです。ほかの手続き (`sqrt-iter`, `good-enough?`, `improve`) は余計なもので、集中を妨げるだけです。また、平方根プログラムと組み合わせて動かす別のプログラムの部品として `good-enough?` という手続きを定義することもできなくなります。`sqrt` が使っているからです。この問題は、個別に働く多くのプログラマによって大きなシステムを構築する際に、特に深刻なものになります。例えば、数値演算手続きの大きなライブラリを作るときに、多くの数値関数は逐次近似法で計算され、そのために補助手続きとして `good-enough?` や `improve` といった手続きを持っているかもしれません。ここは、サブ手続きを局所化して `sqrt` の中に隠し、`sqrt` がほかの逐次近似法の手続き (それぞれ自分のプライベートな `good-enough?` 手続きを持っている) と共に存できるようにしたいところです。これを可能にするために、手続きがローカルな内部定義を持てるようにします。例えば、平方根プログラムは次のように書けます。

```
(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))

```

このような定義のネストは *block structure*(ブロック構造) と呼ばれます。これは、基本的には最も単純な名前パッケージング問題の正しい解決方法です。しかし、ここにはもっといいアイデアが隠れています。補助手続きの定義を内部化することに加えて、それらを単純にすることもできます。`x` は `sqrt` に束縛されているので、`sqrt` の内部で定義された `good-enough?`, `improve`, `sqrt-iter` は `x` のスコープ内にあります。つまり、`x` をこれらの手続きに明示的に渡す必要はないということです。そうする代わりに、次に示すように内部定義では `x` を自由変数にします。そうすると、`x` の値は、それを囲む手続きである `sqrt` の引数から取ってくることができます。このような規定は *lexical scoping*(レキシカルスコピング) と呼ばれます。²⁷

```

(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))

```

この本では、大きなプログラムを扱いやすい部品に分割しやすいように、ブロック構造をどんどん使っていきます。²⁸ ブロック構造という考え方とは、プログラミング言語 Algol 60 に端を発します。ブロック構造はほとんどの先進的なプログラミング言語に存在し、大きなプログラムの構築を体系化するための重要なツールとなっています。

²⁷ レキシカルスコピングにおいては、手続きの自由変数はその手続きを包む手続き定義の束縛を参照すると規定します。つまり、手続きが定義された環境での探索が行われるということです。Chapter 3 で、環境とインタプリタの詳細なふるまいについて学ぶときに、この仕組みがどう働くのか詳細に見ていくことにします。

²⁸ 組み込み定義は、手続き本体の最初に来なければなりません。定義とその使用をごちやごちやに混ぜたようなプログラムの実行については責任を負いかねます。

1.2 手続きとそれが生成するプロセス

ここまでで、プログラミングの要素について考えてきました。基本的な算術演算を使い、その演算を組み合わせ、その組み合わせた演算を複合手続きと定義して抽象化を行いました。しかし、それだけではプログラミングのやり方を知っているとは言えません。私たちの状況は、チェスの駒がどう動くかというルールを覚えたものの、定跡や戦術、戦略については何も知らない人と同じです。チェスの初心者のように、そのドメイン内でのよくあるパターンについてまだ知らないということです。私たちは、どの手を指すのがいい手なのか(どの手続きを定義するのがいいのか)という知識を持っていません。指し手(手続きの実行)の結果を予想する経験も足りません。

考慮中のアクションについて結果を思い描く能力は、達人プログラマになるために決定的に重要なことです。統合的・創造的な活動であれば、何でもそうです。例えば、熟練した写真家になるためには、シーンの見方を学び、露出と現像の条件の選び方によってそれぞれ領域がどれだけの濃さで写真に表れるかを学ばなければなりません。そうすることによって初めて、望む効果を得るためにフレーム、光の当て方、現像などをどのように計画すればいいかを逆算することができるようになります。それはプログラミングでも同じことです。プログラミングにおいては、プロセスが取るべき処理の流れを計画し、プログラムという手段でプロセスをコントロールします。達人になるためには、さまざまなタイプの手続きが生み出すプロセスを思い描けるようになる必要があります。そのようなスキルを身につけて初めて、望むふるまいをするプログラムを確実に構築できるようになります。

手続きは、計算プロセスの *local evolution*(局所展開) のためのパターンです。それは、プロセスの各段階が、どのように前の段階の上に構築されるかを記述するものです。プロセスの局所展開は手続きによって記述されるのですが、できればプロセスの全体的なふるまい、別の言い方をすると *global* (グローバル) なふるまいについても何か言えるようになりたいところです。これは、一般的にはとても難しいことです。しかし、プロセスの展開のよくあるパターンについての説明を試みるぐらいはできるでしょう。

この節では、単純な手続きによって作られたプロセスの、いくつかの一般的な“形”について見てきます。また、これらのプロセスが時間や空間といった重要な計算リソースをどういった比率で消費していくのかについても調べていきます。ここで考える手続きはとても単純なものです。これは、写真撮影でテストパターンが果たす役割と同じ役割を果たしています。つまり、独立した実用的な例というよりは、極度に単純化したプロトタイプパターンです。

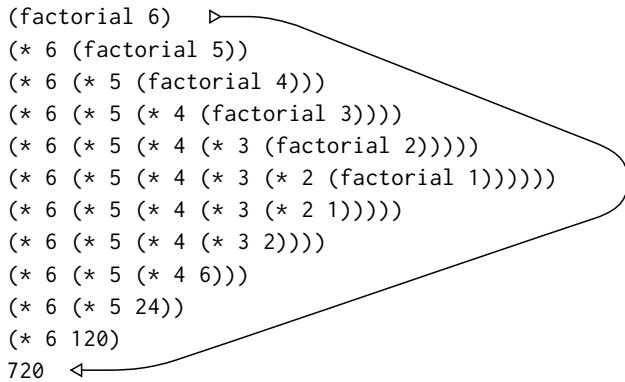


Figure 1.3: $6!$ を求めるための線形再帰プロセス

1.2.1 線形再帰と反復

始めに、階乗関数について考えることから始めましょう。定義は次の通りです。

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

階乗を計算する方法は数多くあります。ひとつの方法は、任意の正の整数 n に対して、 $n!$ は n と $(n - 1)!$ をかけたものと等しいということを利用したものです。

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

そのため、 $(n - 1)!$ を計算し、その答えに n をかけることで $n!$ を計算することができます。そこに $1!$ は 1 に等しいという規定を加えると、このことは直接手続きに変換できます。

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

この手続きが $6!$ の計算を実行する様子を観察するのに、Section 1.1.5 の置換モデルを使うことができます。そうすると、Figure 1.3 のようになります。

```

(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

ここで、階乗計算を別の角度から見てみましょう。 $n!$ の計算規則として、最初に 1×2 を行い、それから答えに 3 をかけ、それから 4 をかけ、 n まで続けるというものに決めることもできます。もう少し形式的に書くと、計算中の積と、1 から n まで数え上げていくカウンタとの両方を保存するということです。計算については、次の規則に従ってカウンタと積がステップごとに同時に変化するというように記述できます。

```

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。

ここでまた、問題の記述を階乗計算の手続きとして書き直します。²⁹

```
(define (factorial n)
  (fact-iter 1 1 n))
```

²⁹ 実際のプログラムでは、前の節で紹介したブロック構造を使って `fact-iter` の定義を隠すところです。

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

ここでは、一度に考えることを最小限にするためにそうしました。

```
(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、 $6!$ の演算プロセスを視覚化するために置換モデルを使い、Figure 1.4として示します。

2つのプロセスを比べてみましょう。ある観点からは、ほとんど違いがないように見えます。どちらも、同じ定義域の同じ数学関数を計算しています。 $n!$ を求めるのに n に比例したステップ数が必要だという点も同じです。それどころか、どちらのプロセスも同じ順番のかけ算を行い、同じ順番で部分積を得ます。一方で、プロセスの“形”について考えると、これらがまったく違う展開の仕方をしていることがわかります。

一つ目のプロセスについて考えてみましょう。置換モデルは、Figure 1.3の矢印で示されるような、展開に続く収縮という形を表しています。展開は、プロセスが *deferred operations*(遅延演算) の連鎖(ここではかけ算の連鎖)を構築する際に起こります。収縮は、演算が実際に行われる際に起こります。遅延演算の連鎖によって特徴付けら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。このプロセスを実行するには、インタプリタは後で実行する演算について記録しておく必要があります。 $n!$ の計算では、遅延乗算の連鎖の長さ、またそれを記録するために必要な情報の量は、 n に対して線形に(n に比例して)増加します。ステップ数も、同じように線形に増加します。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。

それとは対照的に、二つ目のプロセスは伸びたり縮んだりしません。各ステップで記録しておく必要があるのは、どんな n に対しても、変数 `product`, `counter`, `max-count` の現在の値だけです。これは *iterative process*(反復プロセス)と呼びます。一般的に、反復プロセスとは状態が限られた数の *state variables*(状態変数)に集約されるようなプロセスです。それに、プロセスの状態が変化するとともに状態変数がどのように更新されるかという固定した規則と、どのような条件でプロセスを停止するかを定めた(必須ではない)終了テストが加わります。 $n!$ の計算では、必要なステップ数は n に対して線形に増加します。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。

この二つのプロセスの違いは、別の見方もできます。反復プロセスの場合、プログラムの変数はどの時点でもプロセスの状態を完全に記述しています。ステップの間で計算を止めたとして、計算を再開するのに必要なのは、三つのプログラム変数の値をインタプリタに与えることだけです。再帰プロセスではそうはいきません。この場合、インタプリタに保持されていてプログラム変数には含まれていない、いくつかの“隠された”情報が追加で存在し、遅延演算の連鎖を扱ううえで“プロセスがどの地点にいるか”を指し示しています。連鎖が長くなると、保持すべき情報もそれにつれて大きくなります。³⁰

反復と再帰を比較する際に、再帰プロセスという概念と再帰手続き という概念を混同しないように気をつける必要があります。手続きについて再帰と言うときは、その手続きの定義が(直接または間接的に)その手続き自身を参照しているという構文的事実を指しています。しかし、プロセスが例えば線形再帰パターンになっていると言う場合は、どのようにプロセスが展開するかということであって、手続きがどのように書かれているかという構文の問題ではありません。`fact-iter`のような再帰手続きについて、それが反復プロセスを生成すると言うのは、ちょっと気になるかもしれません。しかし、このプロセスは実際に反復的です。状態は三つの状態変数によって完全に表され、プロセスを実行するためにインタプリタは三つの変数を記録しておくだけで十分です。

プロセスと手続きの区別が厄介なのは、一般的な言語(Ada や Pascal、C 言語を含む)のほとんどの実装では、再帰手続きの解釈には必ず手続き呼び出しの回数に比例してメモリを消費するように設計されているからです。その手続きが原理的に反復的である場合であっても、そうなってしまいます。その結果、そのような言語では、反復手続きを記述するには専用の“ループ構造”に頼らなければなりません。`do`, `repeat`, `until`, `for`, `while`などがその例です。[Chapter 5](#)で考える Scheme の実装には、この欠点がありません。反復プロセスは、再帰手続きとして記述されても、固定の空間で実行できます。この性質を持った実装は *tail-recursive*(末尾再帰)と呼ばれます。末尾再帰の実装では、反復は普通の手続き呼び出しメカニズムを使って表現できるので、特別な反復構造は必要なく、シンタックスシュガーとして役に立つ程度です。³¹

³⁰ [Chapter 5](#)でレジスタマシン上での手続きの実装について考えるとき、再帰プロセスはどんなものであっても、固定のレジスタセットを持ち補助メモリを持たない機械として“ハードウェアで”実現できるということを見ていきます。それに対し、再帰プロセスを実現するには *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。

³¹ 末尾再帰は、コンパイラの最適化トリックとして長い間知られてきました。Carl Hewitt (1977)は、末尾再帰に一貫した意味論上の基礎を与えました。その説明は、“メッセージパッシング”という計算モデルによるもので、それについては [Chapter 3](#)で考察します。これに影響を受けて、Gerald Jay Sussman と Guy Lewis Steele Jr. (*Steele and*

Exercise 1.9: 次の二つの手続きは、どちらも `inc`, `dec` という手続きによって二つの正の整数を加算する方法を定義している。手続き `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))))
```

置換モデルを使って、それぞれの手続きが $(+ 4 5)$ の際に生成するプロセスを図示せよ。これらのプロセスは反復だろうか、それとも再帰だろうか。

Exercise 1.10: 次の手続きは、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))))))
```

以下の式の値は何になるか。

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

`A` が上で定義された手続きであるとき、以下の手続きについて考えよ。

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

手続き `f`, `g`, `h` によって、正の整数 n に対して計算される関数に対し、簡潔な数学的定義を与えよ。例えば、`(k n)` は $5n^2$ を計算する。

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、末尾再帰が手続き呼び出しを自然にコンパイルすることの結果であることを示しました (Steele 1977)。Scheme の IEEE 規格は、Scheme の実装が末尾再帰であることを要求しています。

1.2.2 木の再帰

計算するうえでよく使われるパターンとして、ここでは新たに *tree recursion*(木の再帰) を紹介します。例として、フィボナッチ数列の計算について考えてみましょう。この数列では、それぞれの数値はその直前にある二つの数値の合計になっています。

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

一般に、フィボナッチ数は次のような規則で定義できます。

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

この定義は、フィボナッチ数列を計算する再帰手続きに直接変換できます。

```
(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す。`(fib 5)` を計算するには、`(fib 4)` と `(fib 3)` を計算します。`(fib 4)` を計算するには、`(fib 3)` と `(fib 2)` を計算します。一般に、展開されたプロセスは、Figure 1.5で示すように木のような形をしています。枝は、(底以外の) それぞれのレベルで二つに分かれています。これは、`fib` 手続きが呼ばれるたびに自身を二回呼び出すということを反映しています。

この手続きは、木の再帰のプロトタイプになっていますので、学習のためには役に立ちます。しかし、フィボナッチ数を計算する方法としてはひどいものです。余計な計算が多すぎるからです。Figure 1.5で、`(fib 3)` の計算が—それは全体の半分ぐらいの仕事ですが—完全に重複してしまっていることがわかるでしょうか。さらに言うと、この手続きが `(fib 1)` や `(fib 0)` を計算する回数(一般に、上記の木の中での葉の数)が正確に $\text{Fib}(n+1)$ になることは簡単に示せます。このやり方がどれだけひどいかということは、 $\text{Fib}(n)$ の値が n に対して指数的に増えるということを考えればわかるでしょう。より正確には、 $\text{Fib}(n)$ は $\varphi^n / \sqrt{5}$ に最も近い整数になります (Exercise 1.13参照)。 φ は次

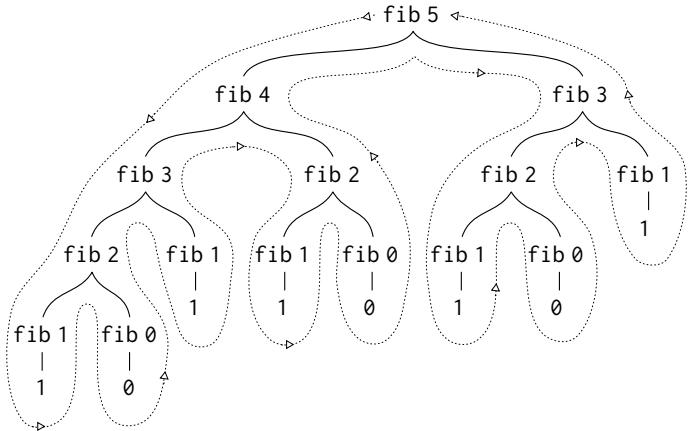


Figure 1.5: (fib 5) の計算時に生成される木の再帰プロセス

の値です。

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

φ は *golden ratio*(黄金比) で、次の等式を満たします。

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

そのため、このプロセスのステップ数は、入力に対して指数的に増加します。一方で、必要な空間は入力に対して線形にしか増加しません。計算のどの時点でも、木の中で上にあるノードだけを記録しておけばいいからです。一般に、木の再帰のプロセスで必要なステップ数は木のノード数に比例し、必要な空間は木の最大の深さに比例します。

フィボナッチ数の計算は、反復プロセスとして定式化することもできます。考え方には、 a と b という整数のペアを使い、 $\text{Fib}(1) = 1$, $\text{Fib}(0) = 0$ と初期化して、次の変換を同時に適用することを繰り返すというものです。

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

この変換を n 回適用した後、 a と b がそれぞれ $\text{Fib}(n+1)$ と $\text{Fib}(n)$ に等しくなるということは簡単に示せます。つまり、以下の手続きによって、フィボナッチ数を反復的に計算できるということです。

```
(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))))
```

こちらの、 $\text{Fib}(n)$ を計算する二つ目の方法は線形反復です。二つの方法で必要なステップ数の違いは —ひとつは n に対して線形、もうひとつは $\text{Fib}(n)$ そのものと同じ割合 — 小さな入力でも、とても大きなものになります。

のことから、木の再帰のプロセスが役に立たないと結論づけてはいけません。数値ではなく、階層的な構造を持つデータを操作するプロセスについて考える場合、木の再帰は自然で強力なツールだということがわかります。³² しかし、数値演算の場合であっても、木の再帰のプロセスはプログラムを理解し、設計するのに役に立ちます。例えば、一つ目の `fib` 手続きは二つ目よりずっと非効率的ですが、ずっと素直な書き方で、ほとんどフィボナッチ数列の定義を Lisp に変換しただけのようなものです。反復アルゴリズムを定式化するのは、計算が三つの状態変数を使った反復として書き直せることに気づく必要があります。

例: 両替パターンの計算

反復的フィボナッチアルゴリズムを思いつくには、ほんのちょっとの工夫で十分でした。それに対して、次の問題を考えてみてください。1 ドルを両替するやり方はいくつあるでしょうか。使うのは、50 セント、25 セント、10 セント、5 セント、1 セントのコインです。より一般的には、任意の金額に対して、両替のパターン数を計算する手続きを書くことはできるでしょうか。

この問題には、再帰手続きによる簡単な解き方があります。使えるコインの種類が、何らかの決まった順番を持つとしましょう。すると、次の関係が成り立ちます。

³²Section 1.1.3では、この例がちらっと出ていました。インタプリタ自身、式を評価するのに木の再帰のプロセスを使っています。

n 種類のコインを使って金額 a を両替するやり方のパターン数は、以下の合計となる。

- 一つ目の種類のコイン以外のすべての種類のコインを使って金額 a を両替するやり方のパターン数。
- n 種類の硬貨すべてを使って、金額 $a - d$ を両替するやり方のパターン数。 d は、一つ目の種類のコインの額面とする。

どうしてこれが言えるのかを考えるには、両替のやり方が二つのグループに分けられることに注目します。一つ目の種類のコインを使わないものと、使うものです。そのため、ある金額に対する両替のパターン数は、一つ目の種類のコインを使わないでその金額を両替するやり方の数と、一つ目の種類のコインを使って両替するやり方の数の合計になります。しかし、後者のパターン数は、最初の種類のコインを一枚使った後の、残りの金額に対する両替のパターン数と等しくなります。

このようにすると、ある金額に対する両替の問題を、より小さな金額をより少ない種類のコインで両替する問題に再帰的に小さくしていくことができます。この小さくする規則を注意深く考えてみて、次の単純な場合について決めておけば、それがアルゴリズムの記述に使えるということを確認してください。

³³

- もし a がちょうど 0 なら、両替パターンは 1 と数える。
- もし a が 0 未満なら、両替パターンは 0 と数える。
- もし n が 0 なら、両替パターンは 0 と数える。

この記述は、再帰手続きに簡単に翻訳できます。

```
(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
```

³³ 例えば、10 セントの両替を 5 セントと 1 セントを使って行う場合、小さくする規則がどのように適用されるか、詳細にやってみてください。

```

                                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)))

```

(`first-denomination` 手続きは、利用可能な硬貨の種類の数を入力として、一つ目の種類のコインの額面を返します。ここでは、硬貨は額面の大きなものから小さなものの順番で並んでいるとされていますが、どのような順番でも同じようにうまくいきます) これで、1 ドルの両替という元々の質問に答えることができます。

```
(count-change 100)
292
```

`count-change` は、`fib` の一つ目の実装同様、冗長な木の再帰のプロセスを生成します (292 の計算にはかなりの時間がかかるでしょう)。一方で、答えを計算するためのよりよいアルゴリズムをどう設計するかというの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。木の再帰のプロセスはとても非効率になりうるもの、記述と理解が簡単であることが多いので、“賢いコンパイラ”を設計して、木の再帰のプロセスと同じ結果を計算するより効率的な手続きに変換できるようにしたら両者のいいとこ取りができるんじゃないいかという提案がされています。³⁴

Exercise 1.11: 関数 f は次のように定義される。

$$f(n) = \begin{cases} n & \text{if } n < 3, \\ f(n-1) + 2f(n-2) + 3f(n-3) & \text{if } n \geq 3. \end{cases}$$

³⁴冗長な計算に対処するひとつのアプローチは、値が計算されるときに自動的にテーブルが構築されるようにするというものです。ある引数に手続きを適用するよう要求されるたびに、まずその値がすでにテーブルに格納されているか確認します。格納されていれば、冗長な計算を実行しないようにします。この戦略は、*tabulation*(テーブル化) や *memoization*(メモ化) として知られるもので、素直に実装することができます。テーブル化は、(`count-change` のような) 指数的なステップ数の必要なプロセスを、必要な時間と空間が入力に対して線形に増加するプロセスに変換するのにも時々使われます。Exercise 3.27 参照。

f を演算する手続きを再帰プロセスによって書け。また、 f を演算する手続きを反復プロセスによって書け。

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。

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

三角形の辺の数値は全て 1 で、三角形内部の数値は、それぞれその上にある 2 つの数値の合計となっている。³⁵ パスカルの三角形の要素を再帰プロセスによって求める手続きを書け。

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。帰納法とフィボナッチ数の定義 (Section 1.2.2 参照) を使って、 $\text{Fib}(n) = (\varphi^n - \psi^n)/\sqrt{5}$ であることを証明せよ。

1.2.3 増加オーダー

前節の例は、プロセスが計算リソースを消費する量の増え方に大幅に差が出ることがあるということを示しています。この違いを記述する便利な方法のひとつとして、*order of growth* (増加オーダー) という概念によって、入力が増えるに従ってプロセスが必要とするリソースがどの程度になるかについての大雑把な感覚をつかむことがあります。

問題の大きさを測るパラメータを n として、大きさ n の問題についてプロセスが必要とするリソースの量を $R(n)$ とします。前節の例では、ある関数が計算を行う対象とする数値を n としましたが、ほかの可能性もあります。例えば、もし目的がある数値の平方根の近似値を求めることであれば、求める精度の桁数を n とするのもいいでしょう。行列のかけ算であれば、 n を行列の行数

³⁵ パスカルの三角形の要素は *binomial coefficients*(二項係数) と呼ばれます。これは、 n 番目の行が $(x + y)^n$ を展開したときの各項の係数となっているからです。この係数計算のパターンは、Blaise Pascal が 1653 年に書いた確率理論の偉大な論文である *Traité du triangle arithmétique* に書かれたものです。Knuth (1973) によると、同様のパターンは 1303 年に中国の数学者朱世傑により出版された *Szu-yuen Yü-chien* (四元玉鑑) や、12 世紀のペルシャの詩人であり数学者でもあった Omar Khayyam の著作や、12 世紀インドの数学者 Bháskara Áchárya の著作にも現れるということです。

としてもいいかもしれません。一般に、問題というものは、プロセスについて分析するために都合のいい属性をいくつか持っているものです。同じように、 $R(n)$ は使用される内部ストレージのレジスタ数であったり、実行される基本的な機械語命令数であったりします。一度に決まった数の命令しか実行しないコンピュータでは、実行時間は実行する基本的な機械語命令数に比例します。

任意の十分に大きな n に対して、 n と独立な正の定数 k_1 と k_2 が存在し、 $k_1 f(n) \leq R(n) \leq k_2 f(n)$ を満たすとき、 $R(n)$ は増加オーダーが $\Theta(f(n))$ であると言い、 $R(n) = \Theta(f(n))$ (“シータ $f(n)$ ”と読む) と書きます。(別の言い方をすると、大きな n に対して、 $R(n)$ の値は $k_1 f(n)$ と $k_2 f(n)$ で挟まれるということです)

例えば、Section 1.2.1 で記述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。つまり、このプロセスに必要なステップ数は $\Theta(n)$ で増加します。必要な空間も $\Theta(n)$ で増加するということを確認しています。反復による階乗計算では、ステップ数は依然として $\Theta(n)$ ですが、空間は $\Theta(1)$ —つまり、定数です。³⁶ 木の再帰によるフィボナッチ数の計算は、 $\Theta(\varphi^n)$ ステップと空間 $\Theta(n)$ を必要とします。ここで、 φ は Section 1.2.2 で示した黄金比です。

増加オーダーは、プロセスのふるまいについて大雑把な記述をするだけです。例えば、 n^2 が必要なプロセス、 $1000n^2$ が必要なプロセス、 $3n^2 + 10n + 17$ ステップが必要なプロセスは、すべては $\Theta(n^2)$ という増加オーダーになります。一方で、問題の大きさを変えた場合のプロセスのふるまいをどう予測するかということについては、増加オーダーは役に立つ指標となります。 $\Theta(n)$ の(線形の)プロセスはで、問題の大きさを 2 倍にすると、使うリソースもだいたい 2 倍になります。指数的プロセスは、問題サイズが 1 増えるたびに、リソース使用量は定数倍になります。Section 1.2 の残りで、増加オーダーが対数的であり、問題の大きさが 2 倍になると必要リソースが定数量だけ増える 2 つのアルゴリズムを見ていきます。

Exercise 1.14: Section 1.2.2 の count-change 手続きによって 11 セントに対する両替のやり方を求める際に生成されるプロセスを図示する木を描け。両替する金額の増加に対して、このプロセスが

³⁶ この書き方の背後には、非常に多くの過度の単純化が隠れています。例えば、プロセスのステップ数を“機械語命令”で数えるとすると、例えばかけ算を実行するのに必要な機械語命令の数はかける数値の大きさと独立であると想定しています。これは、数値が十分に大きければ成り立ちません。空間の見積もりについても同じようなことが言えます。プロセスの設計や記述と同じように、プロセスの分析もさまざまな抽象化レベルで行うことができます。

使う空間とステップ数はどのような増加オーダーになるか。

Exercise 1.15: (ラジアンで指定される) 角度の正弦は、 x が十分に小さいとき $\sin x \approx x$ の近似式によって計算できる。また、次の三角法の恒等式によって、 \sin の引数の大きさを小さくすることができます。

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

(この練習問題では、角が“十分に小さい”とはその大きさが 0.1 ラジアン以下であることとする) これらの考えは、以下の手続きに組み込まれている。

```
(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 $(\text{sine } 12.15)$ を評価する際に、手続き p は何回適用されるか。

b $(\text{sine } a)$ が評価される際に sine 手続きによって生成されるプロセスが使用する空間とステップ数の増加オーダーを (a の関数として) 求めよ。

1.2.4 指数計算

与えられた数値の指数を求める問題について考えましょう。基底 b と正の整数の指数 n を引数に取り、 b^n を求める手続きということになります。ひとつのやり方として、これは次のように再帰的定義によって定義できます。

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

これは、すぐに次の手続きに変換できます。

```
(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、 $\Theta(n)$ のステップ数と $\Theta(n)$ の空間を必要とします。階乗と同じように、等価な線形反復としてすぐに定式化することができます。

```

(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)))))

```

こちらのバージョンは、 $\Theta(n)$ のステップ数と $\Theta(1)$ の空間を必要とします。

指数は、二乗を連続して使うことによって、より少ないステップ数で計算できます。例えば、 b^8 を次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できます。

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

この方法は、指数が2の幂乗である場合にうまくいきます。また、次の規則を使うことで、一般的な指数計算に二乗の連続を使うことができるようになります。

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

この方法は、手続きとして表現できます。

```

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

```

ここで、ある整数が偶数かどうかをテストする述語は、基本手続きの `remainder` (割り算の余り) を使って次のように定義します。

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

`fast-expt` により展開されるプロセスは、空間・ステップ数とともに、 n に対して対数的に増加します。このことは、`fast-expt` を使って b^{2^n} を計算するのは b^n の計算よりかけ算を 1 回多く実行するだけだということからわかります。このため、かけ算の回数が 1 回増えるたびに、計算できる指数の大きさは(ほぼ)倍になります。したがって、 n 乗の計算をするのに必要なかけ算の回数は、だいたい 2 を底とした n の対数と同じくらいの速さで増加することになります。このプロセスは $\Theta(\log n)$ の増加オーダーを持っています。³⁷

$\Theta(\log n)$ での増加と $\Theta(n)$ での増加の違いは、 n が大きくなるにつれて驚くようなものになります。例えば、`fast-expt` は $n = 1000$ に対して 14 回の乗算しか必要としません。³⁸ 二乗の連続という考え方を使って、対数的ステップ数で指数を計算する反復アルゴリズムを作ることもできます (Exercise 1.16 参照)。しかし、反復アルゴリズムにはよくあることですが、再帰アルゴリズムほど素直に書き下すことはできません。³⁹

Exercise 1.16: `fast-expt` と同じように二乗の連続を使って、指数計算を対数的ステップ数で実行する反復的プロセスを生成する手続きを設計せよ。(ヒント: $(b^{n/2})^2 = (b^2)^{n/2}$ を使い、指数 n 、基底 b のほかに追加の状態変数 a を持ち、積 ab^n が各状態を通して一定であるような状態変換を定義せよ。プロセスの最初では a は 1 とし、答えはプロセス終了時の a の値として得られる。一般的に、各状態を通して一定であるような *invariant quantity*(不変量) を定義する

³⁷ より正確には、必要なかけ算の回数は、底を 2 とした n の対数から 1 を引いたものに n の二進数表現での 1 の数を足したものになります。この合計は、底を 2 とした n の対数の 2 倍より常に小さくなります。オーダー記法の定義には k_1 と k_2 という任意の定数が出てきますが、このことは対数的なプロセスについて対数の底は問題にならないということを意味します。そのため、対数的プロセスはすべて $\Theta(\log n)$ として記述されます。

³⁸ 1000 乗の計算なんか誰もしないだろうとお考えかもしれません。Section 1.2.6 を参考してください。

³⁹ この反復アルゴリズムは古代から存在します。紀元前 200 年より前に Áchárya Pingala によって書かれた *Chandah-sutra* に記載されています。Knuth 1981 の 4.6.3 節には、この方法を含むさまざまな指数関数の計算方法についての詳細な考察と分析があります。

というテクニックは、反復アルゴリズムを設計するうえで強力な方法である。)

Exercise 1.17: この節の指数計算アルゴリズムは、指数をかけ算の繰り返しによって計算することを基本としている。同じような方法で、整数のかけ算を足し算の繰り返しによって実行することもできる。次のかけ算手続き(私たちの言語では足し算しかできず、かけ算ができないと仮定して)は `expt` 手続きと同じような仕組みになっている。

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

このアルゴリズムは、`b` に対して線形のステップ数を取る。ここで、足し算のほかに、整数を倍にする `double` という演算と、(偶数の) 整数を 2 で割る `halve` という演算があるとする。これらを使って、`fast-expt` と同じような仕組みで対数的ステップ数を取るかけ算手続きを設計せよ。

Exercise 1.18: Exercise 1.16 と Exercise 1.17 の結果を使って、足し算、`double`、`halve` によって、かけ算を対数的ステップ数で実行する反復的プロセスを生成する手続きを考えよ。⁴⁰

Exercise 1.19: フィボナッチ数を対数的ステップ数で計算する巧妙なアルゴリズムがある。Section 1.2.2 の `fib-iter` プロセスでの状態変数 `a` と `b` の変換、 $a \leftarrow a + b$, $b \leftarrow a$ を思い出そう。この変換を T と呼ぶことにする。すると、1 と 0 から始めて T を繰り返し n 回適用すると $\text{Fib}(n+1)$ と $\text{Fib}(n)$ というペアができることがある。言い換えれば、フィボナッチ数はペア $(1, 0)$ に対して T^n (変換 T の n 乗) を適用することで求められるということだ。ここで、 (a, b) というペアを $a \leftarrow bq + aq + ap$ と $b \leftarrow bp + aq$ に変換するような変換族 T_{pq} の特殊な場合 ($p = 0, q = 1$) として T を考えることにする。このような変換 T_{pq} を二回適用すると、その効果は同じ形

⁴⁰ このアルゴリズムは、“ロシア農民法”と呼ばれることがあるかけ算の方法で、古くから存在します。その使用例は、最も古い二つの数学文献のひとつ、リンド数学パピルスにも見られます。これは、紀元前 1700 年ごろにエジプトの書記官 A'h-mose によって書かれた(それも、より古い文献から写本された)ものです。

式を持つ $T_{p'q'}$ を一回適用するのと同じであることを示し、 p と q に対する p' と q' を求めよ。こうすると、この種の変換に対する二乗を明示的に行うことができるようになり、**fast-expt** 手続きと同じように、 T^n を二乗の連続によって求められるようになる。これらをすべて組み合わせて、対数的ステップ数で動く次の手続きを完成させよ。⁴¹

```
(define (fib n)
  (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   ; p' を計算する
                   ; q' を計算する
                   (/ 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) は a と b の両方をぴったり割り切れる最大の整数と定義されます。例えば、16 と 28 の GCD は 4 です。Chapter 2 で、有理数演算をどのように実装方法するかを検討する際に、有理数を既約にするために GCD の計算が必要になります。(有理数を既約にするためには、分母と分子をそれらの GCD で割ります。例えば、 $16/28$ は $4/7$ になります) 2つの整数の GCD を求める方法のひとつとしては、それらを素因数分解して共通の素因数を探すというものがありますが、それよりずっと効率的な有名なアルゴリズムが存在します。

⁴¹ この練習問題は Joe Stoy によって提案されたもので、Kaldewaij 1990 での例をもとにしています

そのアルゴリズムの考え方は、 r が a を b で割った余りであれば、 a と b の公約数は b と r の公約数とまさに同じものになるという観察に基づいています。このため、次の等式

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

によって、GCD を計算する問題を、より小さな整数ペアの GCD を計算する問題へと連続的に簡約していくことができます。例えば、

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

この例では $\text{GCD}(206, 40)$ を $\text{GCD}(2, 0)$ まで簡約しています。その答えは 2 です。任意の 2 つの正の整数から始めても、繰り返し簡約していくことで最終的には必ず 2 つ目の数値が 0 であるペアにすることができるということが証明可能です。そのとき、GCD はペアのもう 1 つの値です。GCD を求めるこの方法は、*Euclid's Algorithm*(ユークリッドの互除法)として知られています。⁴²

ユークリッドの互除法は、手続きとして簡単に表現できます。

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

これは反復プロセスを生成し、ステップ数は計算する数値に対して対数的に増加します。

ユークリッドの互除法が必要とするステップ数が対数的に増加するという事実は、フィボナッチ数と興味深い関係を持っています。

⁴² ユークリッドの互除法は、ユークリッドの *Elements*(紀元前 300 年ごろ、第 7 卷)に出てきたためにそう呼ばれます。Knuth (1973)によると、自明でないアルゴリズムとしては知られている限り最古のものと考えられるそうです。(Exercise 1.18) で出てきた古代エジプトのかけ算方法は確かにこれよりも古いのですが、Knuth が説明するように、ユークリッドのアルゴリズムは、具体例の集まりとしてではなく一般的なアルゴリズムとして示されたものとしては最古のものです。

Lamé の定理: ユークリッドの互除法によって、ある数値ペアの GCD を計算するのに k ステップを必要とする場合、ペアの小さいほうの数値は k 番目のフィボナッチ数以上である。⁴³

この定理を使って、ユークリッドの互除法の増加オーダーを得ることができます。手続きの n を二つの入力のうちの小さいほうだとします。プロセスが k ステップかかるとすると、 $n \geq \text{Fib}(k) \approx \varphi^k / \sqrt{5}$ が必ず成り立ちます。したがって、ステップ数 k は n に対して (φ を底とする) 対数的に増加します。つまり、増加オーダーは $\Theta(\log n)$ となります。

Exercise 1.20: ある手続きが生成するプロセスは、もちろん、インタプリタの規則によって異なる。例として、上で説明した反復 gcd 手続きについて考える。この手続きを、Section 1.1.5で考察した正規順序評価によって解釈するとする (if に対する正規順序評価規則は Exercise 1.5 に記述されている)。置換法を (正規順序に対して) 使って、(gcd 206 40) を評価する際に生成されるプロセスを図示し、実際に実行される remainder 演算を示せ。(gcd 206 40) の正規順序評価では、remainder 演算は何回実行されるだろうか。適用順序評価はどうだろうか。説明せよ。

⁴³ この定理は、1845 年に Gabriel Lamé によって証明されたものです。彼はフランスの数学者かつエンジニアで、主に数理物理学への貢献で知られています。この定理を証明するには、ユークリッドの互除法が k ステップで停止する、 $a_k \geq b_k$ であるペア (a_k, b_k) について考えます。証明は $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$ が簡約プロセスの中の連続する 3 つのペアである場合、必ず $b_{k+1} \geq b_k + b_{k-1}$ であるという主張に基づいています。この主張を証明するには、簡約ステップが $a_{k-1} = b_k, b_{k-1} = "a_k を b_k で割った余り"$ という変換を適用することとして定義されていることについて考えます。2 つ目の等式は、 $a_k = qb_k + b_{k-1}$ が成り立つある正の整数 q が存在することを意味しています。ここで、 q は最小でも 1 なので、 $a_k = qb_k + b_{k-1} \geq b_k + b_{k-1}$ が成り立ちます。しかし、一つ前の簡約ステップから、 $b_{k+1} = a_k$ です。そのため、 $b_{k+1} = a_k \geq b_k + b_{k-1}$ が成り立ちます。これで主張は証明できました。ここで、アルゴリズムが停止するのに必要なステップ数である k に対して数学的帰納法を使うことで定理が証明できます。 $k = 1$ のとき、これは b が $\text{Fib}(1) = 1$ 以上であればよいので、真です。次に、 k 以下のすべての整数について定理が真であると仮定し、 $k + 1$ の場合について導きます。 $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$ が簡約プロセスにおける連続するペアであるとします。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、先ほど証明した主張とフィボナッチ数の定義を適用すると、 $b_{k+1} \geq b_k + b_{k-1} \geq \text{Fib}(k) + \text{Fib}(k - 1) = \text{Fib}(k + 1)$ が導けます。これで Lamé の定理の証明が完成します。

1.2.6 例: 素数判定

この節では、整数 n が素数性をチェックする二つの方法について記述します。ひとつは増加オーダーが $\Theta(\sqrt{n})$ で、もうひとつは増加オーダーが $\Theta(\log n)$ の“確率的”なアルゴリズムです。節末の練習問題では、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

古代から、数学者は素数に関する問題に魅了されてきました。そして、多くの人々が数値が素数であるかテストする方法に取り組んできました。ある数値が素数であるかテストする方法のひとつとして、その数値の約数を探すというものがあります。次のプログラムは、ある数値 n をちょうど割り切る (1 以上の) 最も小さな数を見つけます。このプログラムは、2 から始まる一連の数値 n について割り切れるかどうかを調べるという素直な方法でそれを行っています。

```
(define (smallest-divisor n) (find-divisor n 2))
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b) (= (remainder b a) 0))
```

数値が素数であるかは次のようにテストできます: n は、 n 自身がその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、それは \sqrt{n} 以下の約数を持つという事実に基づいています。⁴⁴ つまり、このアルゴリズムは、1 から \sqrt{n} までの約数についてだけテストすればよいということです。そのため、 n が素数であるかを判定するのに必要なステップ数は、増加オーダーが $\Theta(\sqrt{n})$ となります。

⁴⁴もし d が n の約数であれば、 n/d も約数になります。しかし、 d と n/d の両方とも \sqrt{n} より大きいということはありません。

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整数論の結果に基づいています。⁴⁵

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

(二つの数値は、その両方を n で割ったときの余りが同じになる場合、*congruent modulo n*(法 n に関して合同) と言います。また、 a を n で割ったときの余りは法 n に関する a の剩余、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 $a < n$ であるほとんどの a は上記の関係を満たしません。このことから、次の素数判定アルゴリズムが導かれます。ある数値 n が与えられたとき、 $a < n$ である適当な数値 a を取り、法 n に関する a^n の剩余を求めます。もし結果が a と等しくなければ、 n は確実に素数ではありません。もし a と等しければ、 n は素数かもしれません。今度は、別の適当な数値 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たせば、 n が素数であることについてより確信が持てるようになります。たくさんの a について試験を行っていくことで、結果についての確信を増やしていくことができます。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、ある数値の幂乗の、別のある数値を法とした剩余を求める手続きが必要です。

```
(define (expmod base exp m)
  (cond ((= exp 0)
         1)
        ((even? exp)
         (remainder
```

⁴⁵Pierre de Fermat (1601-1665) は、近代整数論の創始者と考えられています。彼は多くの重要な整数論上の結果を発見しましたが、彼は普通その結果だけを公表し、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。最初の公開された証明は、オイラーによって 1736 年に与えられました(また、それより早い同じ証明がライブニッツの未発表の草稿に発見されています)。フェルマーの結果の中でも最も有名なものは—フェルマーの最終定理として知られているものですが—1637 年に、彼の持っていた(3 世紀のギリシャ人数学者 Diophantus による)Arithmetica という本に“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”という注記をつけてメモされていたものです。フェルマーの最終定理の証明を見つけることは、整数論の中で最も有名な課題のひとつとなりました。完全な解答は、1995 年にプリンストン大学の Andrew Wiles によってついに与えされました。

```

(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` 手続きにそっくりです。二乗の連続を使っているため、ステップ数は指数に対して対数的に増加します。⁴⁶

フェルマーテストは、1 以上 $n - 1$ 以下のランダムな数値 a を選び、 a の n 乗の $\text{modulo } n$ が a に等しいかをチェックすることで行います。ランダムな数値 a は手続き `random` を使って選びます。`random` は Scheme の基本手続きに含まれているとします。`random` は、受け取った整数よりも小さい非負整数を返します。そのため、1 から $n - 1$ の乱数を得るために `random` に $n - 1$ を渡し、返り値に 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))))))

```

次の手続きは、引数で与えられた数値の回数だけテストを実行します。テストがすべて成功すれば `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)))

```

⁴⁶ 指数 e が 1 より大きい場合の簡約ステップは、任意の整数 x, y, m に対して、 $(xy \text{ modulo } m)$ を求めるためには $x \text{ modulo } m$ と $y \text{ modulo } m$ を別々に求め、これらをかけ算し、その結果の法 m に関する剰余を求めればいいという事実に基づいています。例えば、 e が偶数であれば $b^{e/2} \text{ modulo } m$ を求め、それを二乗し、法 m に関する剰余を取ります。このテクニックを使うと、計算する際に m よりずっと大きな数値を扱わないですませられるので便利です。(Exercise 1.25 と比較すること)

確率的手法

ほとんどの馴染みのあるアルゴリズムは正しいことが保証された答えを返すのですが、フェルマーテストはその点で性質が異なります。フェルマーテストでは、得られた答えは答えの正しさは確率的なものでしかありません。より正確には、 n がフェルマーテストに失敗すれば、 n が素数でないことは確実です。しかし、 n がテストをパスするということは、非常に強い指標ではありますが、それでも n が素数であるという保証にはなりません。ここで主張したいと思っていることは、任意の数値 n について、十分な回数のテストを行って n がすべてのテストをパスすることが確認できれば、この素数判定が間違う確率を好きなだけ小さくできるということです。

残念ながら、この主張はありません。フェルマーテストを騙してしまう数というものは存在します。素数でない数値 n で、 $a < n$ であるすべての整数 a について、 $a^n \equiv n \pmod{a}$ が成り立つことになります。そのような数は非常にまれなので、フェルマーテストは実用的にはかなり信頼できます。⁴⁷

騙されないフェルマーテストの変種もあります。そのテストでは、フェルマーテストと同様に、整数 n の素数判定を行うのに $a < n$ である整数 a を選び、 n と a に依存するある条件についてチェックします (Exercise 1.28 で、そのようなテストの例が出てきます)。一方、フェルマーテストと違って、任意の n に対し、 n が素数でない限り $a < n$ であるほとんどの整数 a についてその条件が成立しないことが証明できます。そのため、もし n がランダムに選んだ数 a についてテストを通過するなら、 n が素数である確率は半々より高くなります。もし n がランダムに選んだ a の値二つに対してそのテストを通過するなら、 n が素数である確率は 4 分の 3 より高くなります。このテストをランダムに選んだ数 a について繰り返していくと、間違いの確率を好きなだけ低くすることができます。

間違いの確率が好きなだけ小さくなることが証明できるテストの存在は、この種類のアルゴリズムへの関心を引き起きました。今では、そのようなア

⁴⁷ 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、非常にまれであるということ以外はあまりよくわかつていません。100,000,000 より小さい数には 255 のカーマイケル数が存在します。最小のものからいくつか挙げると、561, 1105, 1729, 2465, 2821, 6601 となっています。ランダムに選ばれた巨大数の素数性をテストする際にたまたまフェルマーテストを騙す数値にぶつかってしまう確率は、“正確な” アルゴリズムを実行する際に宇宙放射線がコンピュータのエラーを引き起す確率よりも低いのです。一つ目の理由からアルゴリズムを不適切だと考え、二つ目の理由からはそう考えないというのは、数学と工学の違いを示しています。

ルゴリズムは *probabilistic algorithms*(確率的アルゴリズム)と呼ばれるようになっています。この領域では多くの研究が行われていて、確率的アルゴリズムは多くの領域に適用され、成果をもたらしてきました。⁴⁸

Exercise 1.21:smallest-divisor 手手続きを使って次の数値の最小の約数を求めよ。199, 1999, 19999.

Exercise 1.22: ほとんどの Lisp 実装は `runtime` という基本手続きを持っていて、システムが動いた時間を整数(例えば、マイクロ秒で計測したもの)として返す。次の `timed-prime-test` 手手続きは、整数 n を引数として呼ばれると、 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、手続きは 3 つのアスタリスクと、テスト実行にかかった時間を表示する。

```
(define (timed-prime-test n)
  (newline) (display n) (start-prime-test n (runtime)))
(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime (- (runtime) start-time))))
(define (report-prime elapsed-time)
  (display " *** ") (display elapsed-time))
```

この手続きを使って、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き `search-for-primes` を書け。その手続きを使って、1000, 10,000, 100,000 より大きな素数をそれぞれ 3 つ見つけよ。判定アルゴリズムは $\Theta(\sqrt{n})$ の増加オーダーを持っているので、10,000あたりの判定には 1000あたりの $\sqrt{10}$ 倍程度の時間がかかるはずである。あなたの計測データはこれを裏付けているだろうか。100,000 や 1,000,000 のデータは、 $\Theta(\sqrt{n})$ という予想はどれだけ当たっているだろうか。あなたの結果は、演算に必要なステップ

⁴⁸ 確率的素数判定のもっとも著しい応用は暗号分野です。現在、適当な 200 衡の数値を因数分解するというのは計算量的にできそうにありませんが、そのような数値の素数性チェックはフェルマーテストによって数秒で行えます。この事実が、Rivest et al. (1977) によって提案された“解読不能な符号”を構築する技術の基礎となっています。その結果として生まれた RSA アルゴリズムは、電子通信のセキュリティを強化する技術として広く使われるようになっています。一時は、素数の研究というものは“純粹”数学トピックの典型例で、研究のための研究のように思われていましたが、RSA アルゴリズムとその関連研究の進展によって、暗号学、電子送金、情報検索において重要な実用的応用があることがわかりました。

倍数に比例して実行時間が増えるという概念に矛盾していないだろうか。

Exercise 1.23: この節の最初で示した `smallest-divisor` 手続きは、必要のないテストを数多く行っている。数値が 2 で割り切れるかチェックした後に、それより大きな偶数で割り切れるかをチェックすることに意味はない。このことは、`test-divisor` に使われる値は 2, 3, 4, 5, 6, ... ではなく、2, 3, 5, 7, 9, ... であるべきだということを示している。この変更を実装するために、入力が 2 であれば 3 を返し、そうでなければ入力に 2 を足したものを探す手続き `next` を定義せよ。`smallest-divisor` 手続きを、`(+ test-divisor 1)` の代わりに `(next test-divisor)` を使うように修正せよ。この変更したバージョンの `smallest-divisor` 手続きを組み込んだ `timed-prime-test` を使い、Exercise 1.22 で見つけた 12 個の素数についてテストを実行せよ。この変更はテストのステップ数を半分にするため、2 倍の速度で実行できると予想できる。この予想は確認できただろうか。そうでなければ、二つのアルゴリズムの観測された速度の比率はどうなっていて、それが 2 でないという事実をどのように説明できるだろうか。

Exercise 1.24: Exercise 1.22 の `timed-prime-test` 手続きを `fast-prime?` (フェルマー法) を使うように修正し、Exercise 1.22 で見つけた 12 個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、1,000,000 に近い素数をテストするのにかかる時間は 1000 に近い素数をテストするのに必要な時間と比べてどの程度になると予想できるか。実際とのずれがあれば、それを説明できるだろうか。

Exercise 1.25: Alyssa P. Hacker は、`expmod` を書くのにだいぶ余計な手間がかかったと愚痴をこぼした。どうせ指数の計算のやり方はわかっているんだから、単純にこんなふうに書いたらよかつたんじゃないの、と。

```
(define (expmod base exp m)
  (remainder (fast-expt base exp) m))
```

彼女は正しいだろうか。高速素数判定にこの手続きを使っても同じようにうまく行くだろうか。

Exercise 1.26: Louis Reasoner は Exercise 1.24 を解くのにだいぶ苦労している。彼の `fast-prime?` テストは、彼の `prime?` テストよりだいぶ遅いようだ。Louis は友達の Eva Lu Ator に見てもらうことにした。二人で Louis のコードをよく見てみると、`expmod` 手続きで `square` 手続きを呼ぶのでなく、明示的にかけ算を使うように書き換えていることがわかった。

```
(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))))
```

Louis は、「何も違わないように見えるけど」と言っている。Eva は、「いや、違うって。手続きの書き方のせいで、 $\Theta(\log n)$ のプロセスが $\Theta(n)$ のプロセスになっちゃったんだよ。」と言う。

説明せよ。

Exercise 1.27: Footnote 1.47 で挙げたカーマイケル数が実際にフェルマーテストを騙すということを実験せよ。それはつまり、整数 n を取り、すべての $a < n$ について $a^n \equiv a \pmod{n}$ となるかテストする手続きを書き、与えられたカーマイケル数に対してその手続きを試すということになる。

Exercise 1.28: 騙すことができないフェルマーテストの変種のひとつに *Miller-Rabin test* (Miller 1976; Rabin 1980) がある。これは、次のようなフェルマーの小定理の別の形から始める。 n が素数で、かつ a が n 以下の任意の正の整数であれば、 a の $(n-1)$ 乗は n に関する 1 と合同であるというのだ。Miller-Rabin テストで数値 n の素数性をテストするには、 $a < n$ である適当な a を選び、`expmod` 手続きを使って a の $(n-1)$ 乗の n を法とする剰余を求める。しかし、`expmod` 手続きをの中で二乗を行うステップで、毎回 “法 n に関する自明でない 1 の平方根” を見つけたかチェックする。これは、

1 とも $n - 1$ とも等しくない数値で、その二乗が法 n に関して 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 は素数ではないということは証明できる。また、もし n が素数でない奇数であれば、少なくとも $a < n$ である a のうち半分は、このように a^{n-1} を演算すると、法 n に関する自明でない 1 の平方根が現れるということも証明できる（このため、Miller-Rabin テストは騙すことができない）。`expmod` 手続きを変更し、自明でない 1 の平方根を見つけるとシグナルを送るようにして、それを使って `fermat-test` と同じような手続きとして Miller-Rabin テストを実装せよ。既知の素数、非素数を使ってその手続きをチェックせよ。

ヒント：`expmod` にシグナルを送らせる簡単な方法のひとつとして、0 を返すようにするというものがある。

1.3 高階手続きによる抽象の定式化

ここまでで、手続きは事実上、数値に対する複合演算を特定の数値から独立した形で記述する抽象化であることを見てきました。例えば、次の手続きを見てみましょう。

```
(define (cube x) (* x x x))
```

ここで扱っているのは、特定の値の三乗ではなく、任意の数値の三乗を得るやり方です。もちろん、この手続きを定義しないでも、毎回次のような式を書くこともできます。

```
(* 3 3 3)  
(* x x x)  
(* y y y)
```

こうすると、明示的に `cube` について書く必要はありません。しかし、このやり方ではとても不利なことになります。たまたま言語が基本演算として持っていた特定の演算（この場合、かけ算）のレベルずっと作業しなければならず、高レベルの演算が使えないからです。プログラムは三乗を計算することができても、三乗という概念を表現する能力を言語内で持てないことになります。強力なプログラミング言語であれば、よくあるパターンに名前をつけて抽象化し、その抽象化したものを直接使って作業ができる能力が求められるところです。

手続きは、この能力を提供します。このため、極度に原始的なもの以外のすべてのプログラミング言語は、手続きを定義する仕組みを持っています。

しかし、数値計算であっても、もし引数が数値であるような手続きしか作れないとしたら、抽象化の能力は非常に限定されたものになるでしょう。同じプログラミングパターンがいくつもの異なる手続きに利用されるということはよくあります。そのようなパターンを概念として表現するには、手続きを引数として受け取ったり、引数を返り値として返したりできる手続きを作れる必要があります。手続きを操作する手続きは、*higher-order procedures*(高階手続き)と呼ばれます。この節では、高階手続きが言語の表現力を大幅に高める強力な抽象化メカニズムであることを示します。

1.3.1 引数としての手続き

次の3つの手続きについて考えてみましょう。一つ目は、 a から b まで整数の和を計算するものです。

```
(define (sum-integers a b)
  (if (> a b)
    0
    (+ a (sum-integers (+ a 1) b))))
```

二つ目は、範囲内の整数について、三乗の和を計算します。

```
(define (sum-cubes a b)
  (if (> a b)
    0
    (+ (cube a)
        (sum-cubes (+ a 1) b))))
```

三つ目は、次のような級数の和を計算します。

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

これは、 $\pi/8$ に(非常にゆっくり)収束します。⁴⁹

⁴⁹ この級数は Leibniz によるもので、普通は $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$ という等価な形で書かれます。Section 3.5.3では、これを基礎とした面白い数学上のトリックを紹介します。

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

これらの三つの手続きの背後には、明らかに共通のパターンがあります。それらはほとんどの部分が同じもので、違うのは手続き名や、足し合わせる各項を計算するために使われる `a` の関数、`a` の次の値を求める関数ぐらいです。それぞれの手続きは、同じひな形の穴を埋めることで生成できそうです。

```
(define (<name> a b)
  (if (> a b)
    0
    (+ (<term> a)
        (<name> (next a) b))))
```

こういった共通パターンがあるということは、便利な抽象化が潜んでいて、見つけ出されるのを待っているということの強力な証拠です。実際に、大昔の数学者たちは *summation of a series*(級数の総和) という抽象化を発見し、“シグマ記法”を発明しました。例えば、次のような書き方でこの概念を表現します。

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

シグマ記法の力によって、数学者は特定の値の総和についてだけでなく、総和という概念そのものを扱えるようになりました。例えば、特定の級数から独立して、総和についての一般的な結果について定式化できるようになったのです。

同様に、プログラムの設計者である私たちとしては、私たちの使う言語が特定の総和を計算する手続きだけでなく、総和という概念そのものを表現する手続きを書けるぐらいには強力であってほしいと思うところです。私たちの手続き言語を使うと、上に書いた共通のひな形を持ってきて、その“穴”を仮引数に変えるだけですぐにできます。

```
(define (sum term a next b)
  (if (> a b)
    0
    (+ (term a)
        (sum term (next a) next b))))
```

`sum` が、下限と上限である `a`, `b` のほかに、手続き `term(項)`, `next` を取ることに注意してください。`sum` は、ほかの手続きとまったく同じように使えます。例えば、それを使って(引数に 1 を足す手続き `inc` も加えて)`sum-cubes` を定義することができます。

```
(define (inc n) (+ n 1))
(define (sum-cubes a b)
  (sum cube a inc b))
```

これを使うと、1 から 10 までの整数の三乗の和を計算できます。

```
(sum-cubes 1 10)
3025
```

項を計算するのに `identity(恒等)` 手続きを使うと、`sum` によって `sum-integers` を定義できます。

```
(define (identity x) x)
(define (sum-integers a b)
  (sum identity a inc b))
```

こうすると、1 から 10 までの整数の合計が求められます。

```
(sum-integers 1 10)
55
```

`pi-sum` も、同じようにして定義できます。⁵⁰

```
(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))
```

これらの手続きを使うと、 π の近似値が計算できます。

⁵⁰ ブロック構造 (Section 1.1.8) を使って `pi-sum` の中に `pi-next` と `pi-term` の定義を埋め込んでいることに注意してください。これらの手続きは、ほかの手続きの役には立ちそうにないからです。この二つを厄介払いするやり方については、Section 1.3.2で見ていきましょう。

```
(* 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使って、また別の概念を定式化することができます。例えば、関数 f の範囲 a から b の定積分は、小さな dx の値を使い、次の式によって数値的に近似できます。

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

これは、手続きとしてそのまま表現できます。

```
(define (integral f a b dx)
  (define (add-dx x)
    (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
     dx))

(integral cube 0 1 0.01)
.24998750000000042

(integral cube 0 1 0.001)
.24999875000001
```

(`cube` を 0 から 1 の範囲で定積分したとき、その正確な値は $1/4$ です。)

Exercise 1.29: シンプソンの公式は、上で示したものより正確な数値積分の方法である。シンプソンの公式では、関数 f の範囲 a から b 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、 n を適当な偶整数として $h = (b - a)/n$ で、 $y_k = f(a + kh)$ である。 $(n$ を大きくすると近似的精度が上がる)。 f, a, b, n を引数に取り、シンプソンの公式によって計算した定積分の値を返す手続きを定義せよ。その手続きを使い、($n = 100$ と $n = 1000$ で) `cube` の範囲 0 から 1 の定積分を求め、上で示した `integral` 手続きと結果を比較せよ。

Exercise 1.30: 上で挙げた `sum` 手続きは線形再帰を生成する。この手続きは、総和計算が反復的に行われるよう書き直すことができる。次の定義の中で欠けている式を埋め、そのやり方を示せ。

```
(define (sum term a next b)
  (define (iter a result)
    (if (??)
        (??)
        (iter (??) (??))))
  (iter (??) (??)))
```

Exercise 1.31:

a `sum` 手続きを、高階手続きとして表せる数多くの似たような抽象化の中で、最も単純なものにすぎない。⁵¹特定範囲の点における関数の値の積を返す似たような仕組みの手続きを、`product` という名前で書け。`product` を使って `factorial` を定義するやり方を示せ。また、次の公式⁵²を使って、`product` によって π の近似値を計算せよ。

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

b あなたが書いた `product` 手続きを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。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:

⁵¹Exercise 1.31からExercise 1.33は、たくさんの一見ばらばらの演算をまとめるために適切な抽象化を使うことで、どれだけの表現力が得られるかということを示すためのものです。しかしながら、集積やフィルタという言葉は洗練された考え方ではあるのですが、この時点でそれらを使うのは、両手を縛られているようなものです。これらの抽象化を組み合わせる適切な手段となるデータ構造をまだ手に入れていないからです。Section 2.2.3で、フィルタと集積を組み合わせるインターフェイスとして *sequences* (列) を使うやり方を学んだのち、この考え方に戻ってきます。そこでは、プログラムを設計する強力で洗練されたアプローチとしてこれらの手法が本領を発揮することになります。

⁵²この公式は、17世紀のイギリス人數学者 John Wallis によって発見されたものです。

- a. `sum` と `product` (Exercise 1.31) は、どちらも `accumulate`(集積) というより一般的な概念の特殊なケースであることを示せ。集積とは、次のような一般的な集積関数を使って複数の項を組み合わせるものである。

```
(accumulate combiner null-value term a next b)
```

`accumulate` は、`sum` や `product` と同じく項と範囲指定の引数を取り、それに加えて、それまでの項の集積と現在の項をどうやって結合するかを指定する `combiner` 手続き(2引数)と、項がなくなったときにどのような基本値を使うかを指定する `null-value` を引数に取る。`accumulate` を書き、`sum` と `product` の両方を `accumulate` への単純な呼び出しとして定義できることを示せ。

- b. あなたが書いた `accumulate` 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。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 1.33: 結合する項に対する `filter`(フィルタ) という概念を導入することで、さらに一般的なバージョンの `accumulate` (Exercise 1.32) を手に入れることができる。指定された条件を満たす範囲内の値から導出される項だけを結合するというものだ。結果となる `filtered-accumulate` 抽象は、`accumulate` と同じ引数に加えて、フィルタを指定する 1引数の述語を取る。`filtered-accumulate` を手続きとして書け。次のものを `filtered-accumulate` を使ってどのように表現するかを示せ。

a a から b の区間の素数の二乗の和 (すでに `prime?` 述語を書いているとする)

b n と互いに素である n 未満のすべての正の整数 (つまり、 $\text{GCD}(i, n) = 1$ となるすべての整数 $i < n$) の積

1.3.2 lambda を使って手続きを構築する

`sum` を使用する際に、Section 1.3.1でやったように、高階手続きの引数として使うためだけに `pi-term` や `pi-next` のような些細な手続きを定義しないといけないというのは、とんでもなく不格好に思えます。`pi-next` や `pi-term` を

定義するより、“入力に 4 を足したもの返す手続き”や“入力と入力に 2 を足したもの積の逆数を返す手続き”を直接指定する方法があったほうが便利ではないでしょうか。これは、手続きを作る `lambda` という特殊形式を導入することでできるようになります。`lambda` を使うと、やりたいことはそれぞれ次のように記述できます。

```
(lambda (x) (+ x 4))  
  
(lambda (x) (/ 1.0 (* x (+ x 2))))
```

こうすると、`pi-sum` 手続きは次のように補助手手続きをまったく定義しないで表現できます。

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

同様に、`lambda` を使うことで、`integral` 手続きも補助手手続き `add-dx` を定義しないで書けます。

```
(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` 同様、手続きを作るために使われますが、その手続きに名前を指定しないという点が違います。

```
(lambda (<formal-parameters>) <body>)
```

作られる手続きは、`define` によって作られるものと同じく、普通の手続きです。唯一の違いは、環境の中で名前と関連づけられていないということだけです。

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

これは、以下と同値です。

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

lambda 式は、次のように読むことができます。^{訳注 2}

```
(lambda (x) (+ x 4))  
| | | |  
the procedure of an argument x that adds x and 4
```

値として手続きを持つほかの式と同じように、lambda 式は次のように複合式の中で演算子として使うことができます。

```
((lambda (x y z) (+ x y (square z)))  
 1 2 3)  
12
```

というよりも、さらに一般的に言うと、通常手続き名を使うようなどんな文脈でも使えます。⁵³

let を使ってローカル変数を作る

lambda の別の使い方として、ローカル変数を作るというものがあります。手続きの中で、仮引数として束縛されたもの以外のローカル変数を使いたいということはよくあることです。たとえば、次の関数を計算したいとします。

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

これは、次のように表現することもできます。

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

訳注 2 日本語では語順が異なるため、前から読み下すことはできません。意味は、「引数 x を取り、x と 4 を足す手続き」です。

⁵³lambda という名前ではなく、make-procedure(手続き作成)のような当たり前の名前をつけたほうが、Lisp を勉強する人にわかりやすく、怖そうな感じを与えないかもしれません。しかし、この慣習は深く根を下ろしたものです。この記法は、数理論理学者の Alonzo Church (1941)によって導入された λ 計算という数学的形式主義から取られています。Church は、関数と関数適用の概念を研究する厳密な基礎として λ 計算を開発しました。λ 計算は、プログラミング言語の意味論を数学的に研究するための基礎的な道具となっています。

f を計算する手続きを書く際には、 x と y だけでなく、 a や b といった計算過程の値の名前もローカル変数として持つておきたいところです。これを実現するひとつの方法は、ローカル変数を束縛するために補助手続きを使うというものです。

```
(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` 式を使って、ローカル変数を束縛するための無名手続きを指定することもできます。そうすると、`f` の本体はその手続きの呼び出しひとつだけになります。

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

この概念は便利なので、`let` という特殊形式でさらに手軽に使えるようになっています。`let` を使うと、手続き `f` は次のように書けます。

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

`let` 式の一般形式は次のようになります。

```
(let ((<var1> <exp1>)
      (<var2> <exp2>)
      ...)
```

```
((⟨varn⟩ ⟨expn⟩))  
⟨body⟩)
```

これは、次のように考えることができます。訳注3

```
let ⟨var1⟩ have the value ⟨exp1⟩ and  
⟨var2⟩ have the value ⟨exp2⟩ and  
...  
⟨varn⟩ have the value ⟨expn⟩  
in ⟨body⟩
```

let式の最初の部分は、名前・式というペアのリストです。letが評価されるとき、それぞれの名前は対応する式の値と関連づけられます。letの本体は、これらの名前がローカル変数として束縛された状態で評価されます。これは、let式が以下のものの別の文法として評価されているということです。

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

ローカル変数を提供するのに、インタプリタの新しい仕組みは必要ありません。let式は、その裏にあるlambda適用に対するシンタクスシュガーにすぎません。

この等価関係から、let式で指定された変数のスコープがletの本体であるということがわかります。ここから、次のようなことを導くことができます。

- letを使うと、変数を可能な限りローカルに使用箇所に束縛することができます。例えば、もし x の値が 5 であれば、次の式の値は 38 になります。

```
(+ (let ((x 3))  
      (+ x (* x 10)))  
    x)
```

ここで、letの本体の中では x は 3 なので let式の値は 33 になります。それに対して、一番外側の + の第二引数である x は 5 のままでです。

訳注3 ここでも、日本語では語順が異なります。「⟨body⟩の中で、⟨var₁⟩には⟨exp₁⟩、⟨var₂⟩には⟨exp₂⟩…という値を持たせる」という意味です。

- 変数の値は、`let` の外側で計算されます。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た変数に依存している場合には、このことが関係してきます。例えば、`x` の値が 2 であれば、次の式の値は 12 になります。

```
(let ((x 3)
      (y (+ x 2)))
  (* x y))
```

これは、`let` の本体の中では `x` は 3 になり、`y` は 4(つまり、外側の `x` に 2 を足したもの) になっているからです。

`let` と同じ結果を得るために、内部定義が使えることもあります。例えば、上の手続き `f` は次のように定義することもできました。

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

しかし、私たちはこのような状況では `let` を使い、`define` は内部手続きの定義に限定して使うようにしています。⁵⁴

Exercise 1.34: 次の手続きを定義するとする。

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

すると、次のような結果になる。

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

6

⁵⁴ 内部定義を十分に理解し、望むとおりの意味をプログラムに確實に持たせられるようになるには、この章で紹介したものよりも精巧な評価プロセスのモデルが必要です。しかし、手続きの内部定義ではこの微妙な問題は起こりません。Section 4.1.6で評価についてより深く学んだのちに、この問題に戻ってきます。

もしここで、インタプリタに $(f\ f)$ という組み合わせを評価させると、どのような結果が起るだろうか。

1.3.3 汎用手法としての手続き

Section 1.1.4では、数値計算のパターンを抽象化するメカニズムとしての複合手続きを導入し、パターンを特定の数値から独立したものにしました。Section 1.3.1の `integral` 手続きのような高階手続きでは、さらに強力な抽象化に足を踏み入れ、特定の関数から独立した計算の汎用手法を表現するために使われる手続きというものを見てきました。この節では、もう二つの精巧な例、関数の零点と不動点を見つける汎用手法について検討し、これらが手続きとして直接表現できるということを示します。

区間二分法によって方程式の根を求める

half-interval method(区間二分法)とは、 f という連続関数について方程式 $f(x) = 0$ の根を求めるための単純ながら強力なテクニックです。考え方は、 $f(a) < 0 < f(b)$ となる点 a と b があるとき、 f は a と b の間に少なくともひとつの零点を持つというものです。零点を特定するためには、 a と b の平均を x として $f(x)$ を計算します。もし $f(x) > 0$ なら、 f は a と x の間に零点を持つことになります。もし $f(x) < 0$ なら、 f は x と b の間に零点を持ちます。この方法を続けると、 f が零点を持つ区間をどんどん狭めていくことができます。区間が十分狭くなったら、処理は停止します。不確定区間は処理の各ステップごとに半分に縮まっていくので、必要なステップ数の増加オーダーは、元の区間の長さを L 、許容誤差(“十分に小さい”と見なす区間の大きさ)を T として、 $\Theta(\log(L/T))$ になります。この戦略を実装した手続きを次に示します。

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

関数 f と、それの値が負と正になる二つの点は、最初に与えられるものとします。まず、二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をチェックし、小さければ中間点を答えとして返します。そうでない場合、中間点での f の値を計算し、それをテスト値とします。テスト値が正であれば、元々の負の点から中間点までを新しい区間として、処理を続けます。もしテスト値が負であれば、中間点から正の点を区間として続けます。最後に、テスト値が0である可能性がありますが、その場合は中間点そのものが探索対象の根ということになります。

両端が“十分に近い”かどうかをテストするには、Section 1.1.7で平方根を求めるために使ったのと同じような手続きが使えます。⁵⁵

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

`search`を直接使うと、ちょっと面倒なことがあります。うっかり f の値が符号の条件を満たさないような点を与えてしまうかもしれないということです。その場合、間違った答えになってしまいます。そうする代わりに、以下に示す手続きを経由して`search`を使うことにしましょう。これは、どちらの端点が負の関数値を持ち、どちらが正の関数値を持つかを調べ、それに応じて`search`手続きを呼ぶというものです。関数が二つの与えられた点で同じ符号を持つ場合は、区間二分法は使えませんので、手続きはエラーのシグナルを送ります。

⁵⁶

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

⁵⁵ ここでは、0.001を代表的な“小さな”値とし、計算の許容誤差を示すものとして使っています。実際の計算における適切な許容誤差は、解くべき問題や、コンピュータやアルゴリズムの制約によって変わります。これは非常に繊細な考慮が必要な問題となることが多く、数値解析の専門家その他の魔法使いの助けが必要になることもあります。

⁵⁶ これは、ひとつの引数を取り、それをエラーメッセージとして表示する`error`を使って達成できます。

次の例では、 π を近似するのに、区間二分法を使って $\sin x = 0$ の 2 と 4 の間にある根として求めています。

```
(half-interval-method sin 2.0 4.0)
```

```
3.14111328125
```

もうひとつ例を挙げます。区間二分法を使って方程式 $x^3 - 2x - 3 = 0$ の 1 から 2 の間の根を探すというものです。

```
(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 が方程式 $f(x) = x$ を満たすとき、 x は関数 f の *fixed point* (不動点) と呼ばれます。関数 f によっては、最初の推定値から始めて、値があまり変わらなくなるまで f を繰り返し適用していくというやり方で不動点を求めることができます。

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

この考え方を使って、関数と初期推定値を入力として、その関数の不動点の近似値を生成する手続きを作ることができます。前もって決めておいた許容範囲に収まるような二つの値が得られるまで、関数を繰り返し適用していきます。

```
(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、初期近似値の 1 から始めてコサイン関数の不動点を近似するのにこの方法が使えます。⁵⁷

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

同様に、次の方程式の解を見つけることもできます。 $y = \sin y + \cos y$:

```
(fixed-point (lambda (y) (+ (sin y) (cos y))) 1.0)
1.2587315962971173
```

不動点処理は、Section 1.1.7で平方根を求めるのに使った処理と似たところがあります。どちらも、解が何らかの基準を満たすようになるまで繰り返し改善していくという考えに基づいています。実際に、平方根の計算は不動点探索としてすぐに形式化できます。ある数値 x の平方根を求めるには、 $y^2 = x$ となるような y を探す必要があります。この方程式を等価な形である $y = x/y$ に変えると、⁵⁸ 求めるものが $y \mapsto x/y$ の不動点であることがわかります。そのため、平方根は次のような計算で求められないか試してみることができます。

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

残念ながら、この不動点探索は収束しません。最初の推測値を y_1 とします。次の推測値は $y_2 = x/y_1$ となり、その次は $y_3 = x/y_2 = x/(x/y_1) = y_1$ です。結果として、 y_1 と y_2 という二つの推測値をずっと繰り返し、答えの両側で振動を続ける無限ループになります。

このような振動をコントロールする方法のひとつとして、推測値の大きな変化を防ぐことがあります。答えは常に推測値 y と x/y の間にあるはずなので、 y と x/y の平均を取ることで、新しい推測値を y からも x/y からもあまり遠くないものにすることができます。つまり、 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))
```

⁵⁷ 退屈な授業の間に、計算機をラジアンモードに設定し、不動点が得られるまで `cos` ボタンを連打するということをやってみてください。

⁵⁸ \mapsto (“maps to”(写像する)と読む)は、`lambda` の数学者式の書き方です。 $y \mapsto x/y$ は `(lambda (y) (/ x y))` という意味で、つまり y での値が x/y となるような関数を表します。

($y = \frac{1}{2}(y + x/y)$ が方程式 $y = x/y$ を単純に変形したものであることに注意してください。方程式の両辺に y を足し、2で割ることで得られます。)

この修正を加えると、平方根の手続きがうまくいくようになります。実は、定義を外して中を見していくと、ここで生成されている平方根の近似列が、Section 1.1.7 の元の平方根手続きによって生成されるものと完全に同じであることがわかります。解に対する連続した近似値の平均を取るというこのアプローチは、average damping(平均緩和法)と呼ばれるテクニックで、不動点探索の収束に役に立つことがあります。

Exercise 1.35: 黄金比 φ (Section 1.2.2) が $x \mapsto 1 + 1/x$ という変形の不動点であることを示し、そのことを利用して φ を fixed-point 手続きによって求めよ。

Exercise 1.36: fixed-point を修正し、Exercise 1.22 で示した newline と display という基本手続きを使って、生成する近似値列を表示するようにせよ。次に、 $x \mapsto \log(1000)/\log(x)$ の不動点を求めるこことによって $x^x = 1000$ の解を求めよ(自然対数を計算する Scheme の基本手続き log を利用せよ)。平均緩和法を使う場合と使わない場合のステップ数を比較せよ。(fixed-point を開始するのに、1 という推定値が使えないことに注意。1を使うと、 $\log(1) = 0$ による割り算が発生してしまう。)

Exercise 1.37:

- a 無限 continued fraction(連分数) とは、以下のような形の式である。

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

ひとつの例として、 N_i と D_i がすべて 1 の無限連分数は $1/\varphi$ になるということが示せる。ここで、 φ は (Section 1.2.2 で説明した) 黄金比である。無限連分数の近似値を求める方法のひとつは、与えられた項数で展開を打ち切るというものである。このように展開を打ち切ったもの—いわゆる *k-term finite continued fraction* (*k* 項有限連分数)—は、以下のようになる。

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

`n` と `d` を 1 引数 (項の添字 i) の手続きとし、それぞれ連分数の項 $N_i D_i$ を返すとする。`(cont-frac n d k)` を評価すると k 項有限連分数の値を計算するような手続き `cont-frac` を定義せよ。一連の `k` の値に対し、以下によって $1/\varphi$ の近似を求め、その手続きをチェックせよ。

```
(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` をどれだけ大きくしなければならないだろうか。

- b あなたが書いた `cont-frac` 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。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: スイスの数学者レオンハルト・オイラーは、1737 年に *De Fractionibus Continuis* という学術報告を発表した。それには、 $e - 2$ (e は自然対数の底) の連分数展開が含まれていた。この分数では、 N_i はすべて 1 で、 D_i は $1, 2, 1, 1, 4, 1, 1, 6, 1, 1, \dots$ という数列である。オイラーの連分数展開とともに、**Exercise 1.37** の `cont-frac` 手続きを使って e を近似するプログラムを書け。

Exercise 1.39: 正接 (タンジェント) 関数の連分数表現は、1770 年にドイツの数学者ヨハン・ハインリッヒ・ランベルトによって発表された。

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

ここで、 x はラジアンで表したものである。ランベルトの式をもとに、正接関数の近似値を求める手続き `(tan-cf x k)` を定義せよ。`k` は、**Exercise 1.37** 同様、計算する項の数を指定する。

1.3.4 返り値としての手続き

ここまでの一例では、手続きを引数として渡す能力が私たちのプログラミング言語の表現力を大幅に拡張するということを見てきました。この表現力は、

返り値自身が手続きであるような手続きを作成することによって、さらに強力なものにすることができます。

この考え方を理解するために、Section 1.3.3の最後で説明した不動点の例をもう一度見てみましょう。この例では、 \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スタートして、不動点探索という形で square-root 手続きの新しいバージョンを定式化しました。次に、平均緩和法を使って近似値が収束するようにしました。平均緩和法は、それ自体便利な汎用テクニックです。平均緩和法では、関数 f が与えられたとき、それに対応する別の関数を考えます。その関数は、 x における値が x と $f(x)$ の平均となるようなものです。

平均緩和法の考え方は、次の手続きによって表現することができます。

```
(define (average-damp f) (lambda (x) (average x (f x))))
```

average-damp は、引数として手続き f を取り、(lambda で生成された) 手続きを返り値とする手続きです。返り値となる手続きは、数値 x に適用すると x と $(f x)$ の平均を返すというものです。例えば、average-damp を square 手続きに適用すると、 x での値が x と x^2 の平均であるような手続きを返します。この返された手続きに 10 を適用すると、10 と 100 の平均、つまり 55 を返します。⁵⁹

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

average-damp を使うと、square-root 手続きは次のように定式化し直すことができます。

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

この定式化は、この手法の三つの考え方を明確に示しています。不動点探索、平均緩和法、そして関数 $y \mapsto x/y$ です。この square-root の手法の定式化を Section 1.1.7 の元のバージョンと比べると学ぶところがあるでしょう。これらの手続きは同じプロセスを表現しているということに気をつけて、これらの抽象化を使って表現することでどれだけ考え方が明確になっているかを見てください。一般的に、プロセスを手続きとして定式化するやり方はいくつもあり

⁵⁹ これは、演算子自身が複合式であるような複合式です。そのような複合式が作れるということは、Exercise 1.4 すでに見ていますが、そこでの例はおもちゃのようなものでした。ここからは、そういう複合式が実際に必要になるような場合—高階手続きの返り値となる手続きを適用するとき—について見ていくことになります。

ます。経験豊富なプログラマであれば、手続きによる定式化の方法から、特に明快で、プロセスの要素が独立した実態としてほかのアプリケーションで再利用可能であるようなものを選ぶことができます。再利用の単純な例として、 x の三乗根が $y \mapsto x/y^2$ の不動点であり、そのため私たちの square-root 手続きはすぐに立方根を求めるものに拡張可能だということがあります。⁶⁰

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

ニュートン法

Section 1.1.7で初めて square-root 手続きを導入したとき、それが *Newton's method*(ニュートン法) の特別な場合だということに触れました。 $x \mapsto g(x)$ が微分可能な関数であるとき、方程式 $g(x) = 0$ の解は、以下に示す関数 $x \mapsto f(x)$ の不動点となります。

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

ここで、 $Dg(x)$ は g の導関数を x で評価したものです。ニュートン法は、上で見た不動点探索法を利用して、関数 f の不動点を探すことによって方程式の解を近似するというものです。⁶¹

多くの関数 g と、十分によい初期推測値 x に対して、ニュートン法は $g(x) = 0$ の解に急速に収束します。⁶²

ニュートン法を手続きとして実装するには、まず微分という考え方を表現する必要があります。“微分”というものは、平均緩和法と同様に、ある関数を別の関数に変形するものだということに注目してください。例えば、関数 $x \mapsto x^3$ を微分したものは $x \mapsto 3x^2$ です。

一般的に、 g が関数で dx が小さな値であるとき、 g を微分した Dg は、任

⁶⁰さらなる一般化については、Exercise 1.45を参照してください。

⁶¹微積分の入門書では、ニュートン法は普通、 $x_{n+1} = x_n - g(x_n)/Dg(x_n)$ という近似列として記述されています。私たちはプロセスに関して語る言語を持っているため、不動点の考え方を使って手法の記述を簡単にすることができます。

⁶²ニュートン法は必ず解に収束するわけではありませんが、うまくいく場合は繰り返しごとに解の近似値の精度の桁数が倍になるということが示せます。そのような場合は、ニュートン法は区間二分法よりずっと速く収束します。

意の数値 x における値が(dx を極小にしたときに)次のようになる関数です。

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

このため、微分という考え方(例えば dx を0.00001として)次のような手続きとして表現できます。

```
(define (deriv g)
  (lambda (x) (/ (- (g (+ x dx)) (g x)) dx)))
```

次の定義も加えます。

```
(define dx 0.00001)
```

average-damp 同様、`deriv`は手続きを引数に取り手続きを返り値とする手続きです。例えば、 $x \mapsto x^3$ を微分したものの5での値(正確な値は75です)の近似値を求めるには、次の式を評価します。

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

`deriv`の助けを借りると、ニュートン法は不動点探索プロセスとして表現できます。

```
(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` 手続きはこの節の最初の式を表現しており、`newtons-method`はそれを使ってすぐに定義できます。引数として、零点を見つけたい関数を計算する手続きと初期推測値を取ります。例えば、 x の平方根は、初期推測値1から初めて、 $y \mapsto y^2 - x$ 関数の零点をニュートン法によって求めることによって求められます。⁶³

のことから、平方根手続きをまた別の形で書くことができます。

```
(define (sqrt x)
  (newtons-method
    (lambda (y) (- (square y) x)) 1.0))
```

⁶³ 平方根を求める場合、どんな初期値から初めても、ニュートン法は急速に正しい解に収束します。

抽象化とファーストクラス手続き

ここまで、平方根計算をより一般的な手法の具体例として表現する二つの方法を見てきました。ひとつは不動点探索で、もうひとつはニュートン法です。ニュートン法はそれ自身不動点プロセスとして表現されているので、実際には平方根を不動点として計算する二つの方法を見てきたことになります。どちらもある関数から始め、その関数を変形したものの不動点を求めます。この一般的な考え方自体、手続きとして表現できます。

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))
```

このとても一般的な手続きは、引数として手続き g (何らかの関数を計算する)、 g を変形する手続き、初期推測値を取ります。返り値は変形された関数の不動点です。

この抽象化によって、この節の最初の ($y \mapsto x/y$ の平均緩和バージョンに対して不動点を求める) 平方根計算を、この一般的な方法の一例として書き直すことができます。

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

同じようにして、この節の二つ目の ($y \mapsto y^2 - x$ のニュートン変形の不動点を求める) 平方根計算は、次のように表現できます。

```
(define (sqrt x)
  (fixed-point-of-transform
    (lambda (y) (- (square y) x)) newton-transform 1.0))
```

Section 1.3は、複合手続きが非常に重要な抽象化メカニズムであるという観察結果から始めました。複合手続きは、私たちのプログラミング言語の中で、一般的な計算手法を明示的な要素として表現することを可能にするものだからです。ここでは、高階手続きを使うことによって、これらの一般的な手法を組み合わせてさらなる抽象化を生み出していけるということを見てきました。

プログラマである私たちは、プログラムに潜んでいる抽象化を見つけ、それらの抽象化を使ってプログラムを構築し、さらに強力な抽象化を生み出すためにそれらを一般化する機会を見逃さないようにしなければいけません。これは、いつでも可能な限り抽象的にプログラムを書かなければいけないということではありません。熟練プログラマであれば、タスクに対して適切な抽象レベル

ルを選べるもので。しかし、新しい状況でも適用できるように、このような抽象化によって考えられるようになっておくことは重要です。高階手続きが重要なのは、これらの抽象化を明示的に私たちのプログラミング言語で表現することを可能にし、ほかの計算の要素と同じように扱えるようにしてくれるからです。

一般的に、プログラミング言語は計算要素の操作方法に制約を課すもので。制約が最も少ない要素は、*first-class*(ファーストクラス)の地位を持つと言います。ファーストクラス要素の“権利と特権”的いくつかを次に示します。⁶⁴

- 変数によって名前をつけることができる。
- 手続きに引数として渡すことができる。
- 手続きの返り値になることができる。
- データ構造に組み込むことができる。⁶⁵

Lisp は、ほかの一般的なプログラミング言語と違い、手続きに完全なファーストクラスの地位を与えていました。このことは効率的な実装をするうえでの課題となります。そのおかげで非常に強い表現力を手にすることができます。⁶⁶

Exercise 1.40: 次のような形の式で `newtons-method` 手続きと組み合わせて使えるような手続き `cubic`(三次方程式) を定義せよ。

```
(newtons-method (cubic a b c) 1)
```

これを使って、三次方程式 $x^3 + ax^2 + bx + c = 0$ の零点の近似値を求めよ。

Exercise 1.41: 引数がひとつの手続きを引数として取り、その手続きを二回適用する手続きを返す手続き `double` を定義せよ。例えば、`inc` が引数に 1 を足す手続きであれば、`(double inc)` は 2 を足す手続きになる。次の式はどんな値を返すだろうか。

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

⁶⁴ プログラミング言語の要素についてのファーストクラスの地位という概念は、イギリスの計算機科学者 Christopher Strachey (1916-1975) によるものです。

⁶⁵ この実例は、Chapter 2 でデータ構造を紹介した後に見ていきます。

⁶⁶ ファーストクラス手続きの実装コストのうち主要なものとして、手続きを返り値とすることができるようにするには手続きの自由変数の領域を手続きが実行中でないときにも予約しておかなければいけないということがあります。Section 4.1 で学ぶ Scheme の実装では、これらの変数は手続きの環境に保存されます。

Exercise 1.42: f と g を二つの 1 引数関数とする。 g に f を *composition*(合成) するということを、関数 $x \mapsto f(g(x))$ として定義する。合成を実装する手続き `compose` を定義せよ。例えば `inc` が引数に 1 を足す手続きであれば、以下のようにになる。

```
((compose square inc) 6)
```

49

Exercise 1.43: f が数値関数で n が正の整数であれば、 f の n 回適用というものを作ることができ、それは x での値が $f(f(\dots(f(x))\dots))$ である関数として定義できる。例えば、 f が関数 $x \mapsto x + 1$ であれば、 f の n 回適用は関数 $x \mapsto x + n$ となる。もし f が数値を二乗する演算であれば、 f の n 回適用は、引数を 2^n 乗する関数になる。入力として f を計算する手続きと正の整数 n を取り、 f の n 回適用を計算する手続きを返す手続きを書け。その手続きは、以下のように使えるはずである。

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

625

ヒント：[Exercise 1.42](#) の `compose` を使うと便利かもしれない。

Exercise 1.44: 関数の *smoothing*(平滑化) は、信号処理において重要な概念である。 f が関数で dx がある小さな値であるとき、 f を平滑化したものは x での値が $f(x - dx), f(x), f(x + dx)$ の平均となる関数である。入力として f を計算する手続きを取り、 f を平滑化したものを作成する手続きを返す手続き `smooth` を書け。時には、関数を繰り返し平滑化する(つまり、平滑化した関数をさらに平滑化することを続ける)ことによって *n-fold smoothed function*(n 重平滑化関数)を得ることが役に立つこともある。`smooth` と [Exercise 1.43](#) の `repeated` を使って任意の関数の n 重平滑化関数を生成するやり方を示せ。

Exercise 1.45: [Section 1.3.3](#) では、単純に $y \mapsto x/y$ の不動点を探すことによって平方根を計算しようとすると収束しないということと、それを平均緩和法によって解決できるということを見た。三乗根の計算にも同じ方法を使うことができ、 $y \mapsto x/y^2$ を平均緩和したもののが不動点として求めることができる。残念ながら、このプロセスは四乗根ではうまくいかない。 $y \mapsto x/y^3$ を収束させるの

に、平均緩和を一回するだけでは十分ではない。一方、平均緩和を二回行う(つまり、 $y \mapsto x/y^3$ の平均緩和の平均緩和を使う)と、不動点探索は収束するようになる。 $y \mapsto x/y^{n-1}$ を繰り返し平均緩和したものに対する不動点探索によって n^{th} を計算する場合、何回の平均緩和が必要となるか実験せよ。その結果を利用し、Exercise 1.43の `fixed-point`, `average-damp`, `repeated` 手続きをを使って n 乗根を計算する単独の手続きを実装せよ。必要な数値演算は基本手続きとして利用できるとする。

Exercise 1.46: この章で記述した数値計算法のいくつかは、*iterative improvement* (反復改良法) と呼ばれる非常に汎用的な計算戦略の実例である。反復改良法というのは、何かを計算するにあたって、答えに対する初期推測値から始めて、その推測値が十分によいかテストし、よくなければ推測値を改良し、改良した推測値を新しい推測値としてそのプロセスを続けるというものである。二つの手続きを引数として取る手続き `iterative-improve` を書け。引数はそれぞれ、推測値が十分によいか判断する手続きと、推測値を改良する手続きとする。`iterative-improve` の返り値は、引数として初期推測値を取り、それが十分によくなるまで改良を続ける手続きとする。Section 1.1.7の `sqrt` 手続きを Section 1.3.3の `fixed-point` 手続きを `iterative-improve` によって書き直せ。

2

データを用いた抽象化の構築

私たちはここで、数学的抽象化の決定的なステップに到達しました。記号の意味について忘れるのです。…[数学者]はだらだらしていなくても大丈夫です。これらの記号を使ってできる演算はいくらでもあります。何を意味しているのか調べる必要はありません。

—Hermann Weyl, *The Mathematical Way of Thinking*

Chapter 1では、計算プロセスと、プログラム設計における手続きの役割に焦点を合わせました。基本データ(数値)と基本演算(数値演算)の使い方、組み合わせや条件式や引数の使用によって手続きを組み合わせて複合手続きを作る方法、`define`を使って手続きを抽象化する方法について見てきました。また、プロセスの局所展開のパターンとして手続きを捉えることができるということも学びました。さらに、手続きという形で具体化されたプロセスのいくつかの共通パターンについて分類や論証を行い、また簡単なアルゴリズム的解析をしました。高階手続きによって計算上の一般的手法を扱うことができ、またそれを使って考えることができるようになるため、私たちの言語を強力なものにしてくれるということも見てきました。これは、プログラミングの本質の大きな部分を占めています。

この章では、さらに複雑なデータについて見ていきます。第一章での手続きはすべて単純な数値データを処理するものでしたが、私たちがコンピュータで解決したいと思っている問題の多くでは、単純なデータは十分ではありません。

ん。プログラムは普通、複雑な現象をモデル化するように設計されるものです。いくつもの側面を持つ実世界の現象をモデル化するために、いくつもの部品を持つコンピュータ上のオブジェクトを作る必要に迫られることはよくあります。そのため、第一章での焦点が手続きを組み合わせて複合手続きを作ることによって抽象化を構築することであったのに対して、この章では、どんなプログラミング言語についても言える、もうひとつの重要な側面に目を向けます。データオブジェクトを組み合わせて *compound data*(複合データ) を作るためにプログラミング言語が提供する手段です。

なぜプログラミング言語に複合データが必要なのでしょうか。複合手続きが必要なのと同じ理由です。プログラムを設計する概念レベルを引き上げ、設計のモジュール性を高め、言語の表現力を強くしたいからです。手続きを定義できることによって基本演算よりも高い概念レベルで手続きを扱えるようになるように、複合データオブジェクトを構築できることによって、言語の基本データオブジェクトよりも高い概念レベルでデータを扱えるようになります。

有理数を使って計算を実行するシステムを設計するというタスクについて考えてみましょう。例えば、二つの有理数を取って和を返す *add-rat* という演算を考えることができます。単純なデータを使うと、有理数は分母と分子という二つの整数として考えられます。そのため、有理数が二つの整数(分母と分子)として表され、*add-rat* が二つの手続き(ひとつは和の分子を返し、もうひとつは分母を返す)として実装されるようにプログラムを設計することができます。しかし、これは不格好です。こうすると、どの分子がどの分母に対応するのか明示的に記録しておかないといけないからです。たくさんの有理数演算を実行するようなシステムでは、そんな簿記みたいな細かい処理があるとプログラムがとても読みにくくなってしまいますし、人間の心に与える影響も言うまでもありません。分母と分子を“くっつけて”、ペア—*compound data object*(複合データオブジェクト)—にすれば、状況はずっとよくなります。そうすると、有理数をひとつの概念単位として捉えるという考え方方に近いやり方でプログラムから扱うことができます。

複合データを使うと、プログラムのモジュール性を上げることもできます。有理数を一人前のオブジェクトとして直接操作することができれば、プログラムの中で有理数を有理数として扱う部分を、有理数を例えれば整数のペアとしてどうやって表すかといった細部から分離することができます。プログラムの中のデータオブジェクトをどうやって表すかを扱う部分と、データオブジェクトをどうやって使うかを扱う部分とを分離するという汎用的なテクニックは、*data abstraction*(データ抽象化)と呼ばれる強力な設計手法です。ここでは、データ抽象化によってプログラムの設計・保守・修正がどれだけ簡単

になるかということを見ていきます。

複合データを使うことによって、私たちのプログラミング言語の表現力は実際に向上します。“線形結合” $ax + by$ を作ることを考えてみましょう。書き方はいろいろ考えられますが、例えば a, b, x, y を引数として取り、 $ax + by$ の値を返す手続きを書くとします。引数が数値であれば、何も難しいことはありません。すぐに次のような手続きを定義できます。

```
(define (linear-combination a b x y)
  (+ (* a x) (* b y)))
```

しかし、扱いたいものが数値だけでないとどうでしょう。加算と乗算さえ定義されていれば、有理数でも複素数でも多項式でも何でも線形結合が作れるという考え方を、手続きという形で表現したいという場合について考えてみます。これは、次のような形の手続きとして表現できます。

```
(define (linear-combination a b x y)
  (add (mul a x) (mul b y)))
```

ここで、`add` と `mul` は基本手続き `+` と `*` ではなく、 a, b, x, y という引数として渡すデータに対して、その種類に応じて適切な演算を行うという、より複雑な何かです。ここでキーとなるのは、`linear-combination` が a, b, x, y について、手続き `add` と `mul` が適切な演算を行ってくれるはずだということだけ知っておけばいいということです。手続き `linear-combination` から見ると、 a, b, x, y が何であるかはどうでもいいことで、それらが基本データを使ってどういう形で表現されることになっているかというのはもととどうでもいいことです。この同じ例は、なぜ私たちのプログラミング言語が複合データを直接扱う能力を提供してくれないといけないのかを教えてくれます。その能力がないと、`linear-combination` のような手続きが、内部構造を知らないまま、引数を `add` と `mul` に渡すということを行なう方法がありません。¹

¹ これは、手続きを直接操作する能力がプログラミング言語の表現力を向上させるというのと似たようなものです。例えば、Section 1.3.1では `sum` という手続きを導入しましたが、これは手続き `term` を引数として取り、ある指定した区間での `term` の値の和を求めるというものでした。`sum` を定義するためには、`term` のような手続きについて、それがより基本的な演算によってどのように表現されているかということと関係なく、独立した実体として考えることができるということが決定的に重要です。もし“手続き”という概念がなかったら、`sum` のような演算が定義できるという可能性を考えることができなかつたかもしれません。それに、総和計算の実行ということに関する限り、`term` がより基本的な演算によってどのように構築されているかという細かいところはどうでもいいことです。

この章は、上で述べた有理数計算システムの実装から始めます。これによって、複合データやデータ抽象化について考察する基礎ができます。複合手続きの場合と同じように、対象とする主な問題は、複雑性をうまく扱うテクニックとしての抽象化です。ここでは、データ抽象化によって、プログラムの部品間に適切な *abstraction barriers*(抽象化の壁) を建てられるようになるということを見ていきます。

複合データを形成するうえで鍵となることは、データオブジェクトを組み合わせて複雑なデータオブジェクトを形成できるように、プログラミング言語がある種の“糊”を提供しなければいけないということです。糊として使うものには、いろいろな種類のものがあります。実際に、特別な“データ”操作をまったく使わずに、手続きだけを使って複合データを作ることができるということを見ていきます。このことによって、第一章の末尾すでに曖昧になつてきていた“手続き”と“データ”との境界線は、さらに曖昧なものになつていきます。また、列や木を表現するための確立されたテクニックについて見ていきます。複合データを扱ううえで鍵となる考え方のひとつは、*closure*(クロージャ) という概念です。これは、私たちがデータオブジェクトを組み合わせるのに使う糊は、基本データオブジェクトだけでなく、複合データオブジェクトも組み合わせられるようになっていなければいけないということです。もうひとつの鍵となる考え方とは、部品を組み立てるようにプログラムのモジュールを組み立てるうえで、複合データが *conventional interfaces* (確立されたインターフェイス) として使えるということです。これらの考え方の一部については、クロージャを使う簡単なグラフィック言語によって図解することにします。

また、*symbolic expressions*(記号式)—要素が数値だけでなく任意の記号からなるデータ—を導入することで、私たちの言語の表現力を強化します。さらに、オブジェクトの集合を表現するうえでのいろいろなやり方について詳しく見ていきます。ある数値関数を計算するのにいろいろな計算プロセスが考えられるのと同じように、あるデータ構造をより単純なオブジェクトによって表すのにもいろいろなやり方があり、その表現をどう選ぶかということが、そのデータを扱うプロセスが必要とする時間や空間に大きな影響を与えるということがわかります。これらの考え方については、記号微分、集合の表現、情報のエンコードといった文脈で検討していきます。

その次に、プログラムのいろいろな部分で異なる形で表現されているデータを扱うという問題を取り上げます。このためには、異なるデータ型を処理する *generic operations*(総称演算) というものを実装しなければなりません。総称演算というものがある状況でモジュール性を維持するには、単純なデータ抽象化だけの場合に比べて、より強力な抽象化の壁が必要になります。具体的

には、*data-directed programming*(データ主導プログラミング) というテクニックを導入します。これは、それぞれのデータ表現を独立して設計し、それらを *additively*(加法的) に(つまり、修正なしに)組み合わせられるようにするものです。システム設計におけるこのアプローチの力を示すために、そこまでで学んだことを多項式上での記号演算を実行するパッケージの実装に適用することで、この章の締めくくりとします。ここでの多項式の係数は、整数、有理数、複素数、さらにはほかの多項式となることもあります。

2.1 データ抽象化入門

Section 1.1.8では、ある手続きをより複雑な手続きの要素として使うとき、その手続きは特定の演算の集合としてだけでなく、手続きの抽象化として見ることもできるということに触れました。つまり、手続きがどうやって実装されているかという細かいところは隠しておくことができ、その特定の手続き自身、同じような全体的なふるまいを持つ別の任意の手続きによって置き換えることができるということです。別の言い方をすると、手続きがどのように使われるかというところを、その手続きがより基本的な手続きによってどのように実装されているかといった細かいところから分離するという抽象化ができるということになります。複合データについての同じような概念は、*data abstraction*(データ抽象化)と呼ばれます。データ抽象化は、複合データオブジェクトがどう使われるかというところを、それがより基本的なデータオブジェクトによってどのように構築されているかといった細かいところから分離することを可能にする方法です。

データ抽象化の基本的な考え方は、複合データオブジェクトを使うようなプログラムを構築する際に、“抽象データ”を扱うようにするということです。これは、プログラムでデータを扱うとき、手元のタスクを実行するのに絶対に必要とは言えないような想定をしないようにするということです。また、“具体的な”データ表現は、そのデータを使うプログラムとは独立に定義されます。私たちのシステムのこれら二つの部品をつなぐインターフェイスの役割を担うのは一組の手続きで、それらは *selectors*(セレクタ) と *constructors*(コンストラクタ) と呼ばれます。これらが抽象的なデータを具体的な表現によって実装することになります。このテクニックの説明のために、有理数を扱う手続きの集合をどのように設計するかということを考えていきます。

2.1.1 例: 有理数の数値演算

有理数を使って数値演算を行いたいという場合について考えてみましょう。行いたい演算は、足し算、引き算、かけ算、割り算や、二つの有理数が等しいかどうかテストするといったものになるでしょう。

まずは、分子と分母から有理数を構築する方法はすでにあると仮定しましょう。また、有理数が与えられたときに、その分子と分母を抽出(セレクト)する方法もあるとします。さらに、コンストラクタとセレクタは次のように手続きとして使うことができるとなります。

- (`(make-rat <n> <d>)`) は、分子が整数 $\langle n \rangle$ で分母が整数 $\langle d \rangle$ である有理数を返す。
- (`(numer <x>)`) は、有理数 $\langle x \rangle$ の分子を返す。
- (`(denom <x>)`) は、有理数 $\langle x \rangle$ の分母を返す。

ここでは、プログラムを合成していくうえでの強力な戦略である *wishful thinking*(希望的思考) を使っています。有理数をどうやって表現するのか、`numer`, `denom`, `make-rat` という手続きをどう実装するのか、まだ何も言っていません。それでも、もしこれらの手続きを持っていたとすると、足し算、引き算、かけ算、割り算、等価性のテストは、次のような関係を使って実行できます。

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

これらの規則は手続き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、有理数の演算が `numer`, `denom`, `make-rat` というセレクタ手続きとコンストラクタ手続きによって定義できました。しかし、セレクタ手続きとコンストラクタ手続きは定義していません。ここで、分母と分子を組み合わせて有理数を作るために何らかの方法が必要になります。

ペア

データ抽象化の具体的なレベルを実装することを可能にするために、私たちの言語は `pair(ペア)` という複合構造を用意しています。これは、`cons` という基本手続きによって構築できます。この手続きは二つの引数を取り、それら二つの引数を部品として含む複合データオブジェクトを返します。ペアがあるとき、それらの部品は `car` と `cdr` という基本手続きを使って取り出すことができます。² そのため、`cons`, `car`, `cdr` は次のような使い方ができます。

```

(define x (cons 1 2))
(car x)
1
(cdr x)

```

²`cons` という名前は “construct” を表しています。`car` と `cdr` という名前は、IBM 704 上での最初の Lisp 実装に由来します。この機械はメモリ位置の “アドレス” と “デクリメント” という部分が参照できるようなアドレス方式を持っていました。`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) を表しています。

ペアは、基本データオブジェクトと同じように、名前をつけて操作することができるデータオブジェクトです。さらに、`cons` は要素がペアであるペアや、またそれが要素となるペアといったものを作ることもできます。

```
(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では、ペアを組み合わせるこの能力が、ペアというものをすべての複雑なデータ構造を作るための汎用ブロックとして使えるものにしているということを学びます。手続き `cons`, `car`, `cdr` によって実装される、唯一の基本複合データである `pair` ペアは、私たちが必要とするただひとつの糊です。ペアによって構築されるデータオブジェクトは *list-structured*(リスト構造) のデータと呼ばれます。

有理数を表現する

有理数システムを作るのに必要な残りの部分を補うために、ペアは自然なやり方を提供してくれます。単純に、分子と分母という二つの整数のペアとして有理数を表現すればいいのです。そうすると、`make-rat`, `numer`, `denom` はすぐに次のように実装することができます。

³

³セレクタとコンストラクタは次のように実装することもできます。

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

一つ目の定義は `make-rat` という名前を、式 `cons` の値である、ペアを構築する基本手続きに関連づけるというものです。こうすると、`make-rat` と `cons` は同じ基本コンストラクタに対する名前になります。

セレクタとコンストラクタをこのやり方で定義すると、効率はよくなります。`make-rat` が `cons` を呼び出すのではなく、`make-rat` は `cons` であるため、`make-rat` が呼ばれるときに呼ばれる手続きは、二つではなくひとつになります。一方で、このやり方をし

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

また、計算結果を表示するために、有理数を分子、スラッシュ、分母として表示することにします。⁴

```
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

これで、有理数手続きを試してみることができます。

```
(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
```

最後の例からわかるように、この有理数計算の実装は有理数を既約のものに簡約してくれません。これを直すには、`make-rat` を修正します。Section 1.2.5で扱ったような、二つの整数の最大公約数を返す `gcd` 手続きがあれば、ペアを構築する前に `gcd` を使って分子と分母を既約にすることができます。

てしまうと、手続き呼び出しを追跡したり手続き呼び出しにブレークポイントを置いたりといったデバッグ手段が使い物にならなくなってしまいます。`make-rat` 手続きの呼び出しを監視したいという場合はあるかもしれません、`cons` のすべての呼び出しを監視するなんてことはしたくないはずです。

この本では、この定義スタイルは使わないことにしました。

⁴`display` はデータを表示する Scheme の基本要素です。Scheme の基本要素 `newline` は改行を行います。これらの手続きは、どちらも意味のある値は返しません。そのため、以下の `print-rat` の使用例では、`print-rat` が表示するもののみを示し、インタプリタが `print-rat` の返り値として表示するものは示さないことにします。

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
```

こうすることで、次のように望む結果が得られます。

```
(print-rat (add-rat one-third one-third))
2/3
```

この修正はコンストラクタ `make-rat` の変更だけで完了し、実際の演算を実装する (`add-rat` や `mul-rat` といった) 手続きはどれも変更する必要がありませんでした。

Exercise 2.1: 正と負の両方の引数を扱うことができる改良版 `make-rat` を定義せよ。`make-rat` は符号を正規化し、正の有理数であれば分子と分母の両方が正となり、負の有理数であれば分子のみが負になるようにする。

2.1.2 抽象化の壁

複合データとデータ抽象化の例を続けて見ていく前に、有理数の例で出てきたいいくつかの問題について考えてみましょう。有理数演算を定義するのに、私たちはコンストラクタである `make-rat` とセレクタである `numer`, `denom` を使いました。一般的に、データ抽象化の底にある考え方は、それぞれのデータオブジェクトの型に対して、それさえあればその型に対するどんな演算も行えるような基本演算セットを特定し、その後はデータを操作するのにそれらの演算しか使わないようにするというものです。

有理数システムの構造は、Figure 2.1 のようにイメージすることができます。水平線は、システムの異なる “レベル” を分離する *abstraction barriers* (抽象化の壁) を表しています。有理数を使うプログラムは、有理数パッケージによって “公用” として提供された手続きである `add-rat`, `sub-rat`, `mul-rat`, `div-rat`, `equal-rat?` だけを使って有理数を操作します。これらの手続きはというと、コンストラクタとセレクタである `make-rat`, `numer`, `denom` だけを使って実装されていますが、コンストラクタとセレクタはというと、これらはペアによって実装されています。ペアは、`cons`, `car`, `cdr` によって操作することさえできれば、どのように実装されているかは有理数パッケージの残りの部分とは無関係です。実質的に、それぞれのレベルの手続きは、抽象化の壁を定義し、異なるレベルをつなぐインターフェイスとなっています。

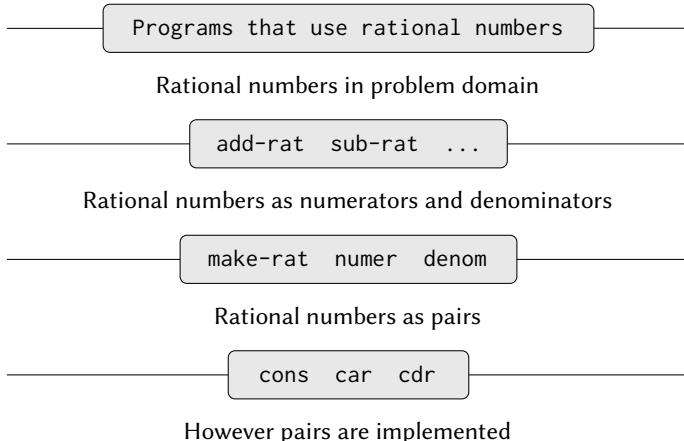


Figure 2.1: 有理数パッケージ内の抽象化の壁

この単純な考え方には、いろいろないいところがあります。そのひとつは、プログラムを保守したり修正したりするのが簡単になるということです。どんな複雑な構造でも、プログラミング言語によって提供される基本データ構造を使って、いろいろな方法で表すことができます。もちろん、どういう表し方を選ぶかということは、それを扱うプログラムに影響を与えます。そのため、後になってどこかの時点で表現方法を変えることになったら、それらのプログラムはすべてそれに合わせて修正しないといけなくなるかもしれません。もし表現方法への依存がごく少數のプログラムモジュールに制限されるように設計されていなければ、大きなプログラムではこの作業は時間とコストがかかるものになるでしょう。

例えば、有理数を既約に簡約する問題の別の解き方として、簡約を実行するタイミングを、有理数の構築時でなく、有理数の部品へのアクセス時にするというやり方があります。この場合、コンストラクタとセレクタは違ったものになります。

```
(define (make-rat n d) (cons n d))
(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
```

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

この実装と前の実装との違いは、いつ gcd を計算するかというところです。プログラムを普通に使う際に、同じ有理数の分子や分母に何回もアクセスするのであれば、gcd は有理数の構築時に計算したほうがいいでしょう。そうでなければ、gcd の計算はアクセス時まで遅らせたほうがいいかもしれません。どちらを選ぶにしても、実装を切り替える場合に、add-rat, sub-rat などはまったく修正する必要がありません。

表現方法への依存を少しだけのインターフェイス手続きに限定することは、プログラムの修正時だけでなく、プログラムの設計時にも役に立ちます。そうしておくと、別の実装を考慮するための柔軟性を維持することができるからです。上の単純な例を引き続き使って、有理数パッケージを設計するにあたって、gcd を実行するタイミングを構築時にするか選択時にするかを最初の段階で決められないとなります。データ抽象化という方法を使うことによって、その決定を後回しにしつつ、システムのほかの部分の開発を進めるということができるようになります。

Exercise 2.2: 平面上の線分を表現するという問題について考える。

それぞれの線分は、始点と終点という点のペアとして表す。コンストラクタ make-segment とセレクタ start-segment, end-segment を定義せよ。さらに、点は x 座標と y 座標という数値のペアとして表現できる。最後に、定義したセレクタとコンストラクタによって、線分を引数として取りその中点(両端点の座標を平均した座標を持つ点)を返す midpoint-segment 手続きを定義せよ。手続きをチェックするためには、例えば次のようにして点を表示することが必要となるだろう。

```
(define (print-point p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display ")"))
```

Exercise 2.3: 平面上の長方形の表現を実装せよ (ヒント: Exercise 2.2 を利用するといいかかもしれない)。そのコンストラクタとセレクタを使って、ある長方形の外周の長さと面積を計算する手続きを書け。次に、長方形の異なる表現を実装せよ。適切な抽象化の壁を使ってシステムを設計し、外周と面積を求める同じ手続きがどちらの表現によっても動くようにできるだろうか。

2.1.3 データとは何か

Section 2.1.1 での有理数の実装では、有理数演算 `add-rat`, `sub-rat` 等を、`make-rat`, `numer`, `denom` という仕様の決まっていない三つの手続きによって実装することから始めました。その時点では、それらの演算はデータオブジェクト—分子、分母、有理数—によって定義され、データオブジェクトのふるまいは後者の三つの手続きによって規定されると考えることができました。

しかし、`data(データ)` とは正確には何を意味しているのでしょうか。“あるセレクタとコンストラクタによって実装される何か”と言っただけでは十分ではありません。明らかに、任意の 3 つの手続きのセットがすべて有理数の実装に対する適切な基礎としての役割を果たせるわけではありません。有理数 `x` を `n` と `d` という整数のペアによって構築する場合、`x` の `numer` と `denom` を取り出して割り算を行ったものが、`n` を `d` で割ると同じ結果になるということを保証する必要があります。別の言い方をすると、`make-rat`, `numer`, `denom` は、任意の整数 `n` とゼロでない整数 `d` に対し、`x` が `(make-rat n d)` であるとき、次の条件を満たす必要があります。

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

実際のところ、これは `make-rat`, `numer`, `denom` が有理数表現の基礎となるために満たさないといけないただひとつの条件です。一般的に、データというものは、何らかのセレクタとコンストラクタの集合に加え、それらが有効な表現となるために満たさなければならないと規定された条件によって定義されるものと考えることができます。⁵

⁵ 意外なことに、この考え方は厳密に定式化することが非常に難しいものです。そのような定式化には二つのアプローチがあります。ひとつは C. A. R. Hoare (1972) が開拓したもので、*abstract models*(抽象モデル) という手法として知られています。この手法では、上の有理数の例で要点を述べた“手続きと条件”という仕様を定式化するものです。有理数表現での条件は整数に関する事実(等価性と割り算)によって記述されて

この見方は、有理数のような“高レベル”のデータオブジェクトを定義する場合だけでなく、低レベルなオブジェクトに対しても使えます。有理数の定義する際に使った、ペアという概念について考えてみましょう。ペアとは何かということについてはこれまで触れず、言語がペアを操作する手続きとして `cons`, `car`, `cdr` を用意してくれているということだけを述べてきました。しかし、これら三つの操作について知らなければいけないことは、二つのオブジェクトを `cons` によってくっつけたら、それらのオブジェクトを `car` と `cdr` によって取得できるということだけです。つまり、これらの操作は、任意のオブジェクト `x` と `y` に対して、もし `z` が `(cons x y)` であるなら `(car z)` は `x` であり、`(cdr z)` は `y` であるという条件を満たすということです。これら三つの手続きが私たちの言語に基本要素として含まれるということについてはすでに触れたとおりですが、上の条件を満たすものであれば、どんな三つの手続きであってもペアを実装する基礎として使うことができます。この点については、`cons`, `car`, `cdr` を実装するのに、何のデータ構造も使わずに、手続きだけによってそれができるということが鮮やかに示してくれます。以下はその定義です。

```
(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))
```

このような手続きの使い方は、データというものがどういうものかという私たちの直感的な概念とまったく対応していません。それでも、これらの手続きが上の条件を満たすということさえ確認できれば、これがペアを表現する有効な

います。一般的に、抽象モデルではそれまでに定義されたデータオブジェクトの型によって新しい種類のデータオブジェクトを定義します。そのため、データオブジェクトに関する表明は、それまでに定義済みのデータオブジェクトに関する表明に書き換えてチェックすることができます。もうひとつのアプローチは、MIT の Zilles と IBM の Goguen, Thatcher, Wagner, Wright (Thatcher et al. 1978 参照) や、トロント大学の Guttag (Guttag 1977 参照) によって導入されたもので、algebraic specification(代数的仕様) と呼ばれます。代数的仕様では、“手続き”を抽象代数システムの要素と見なし、そのふるまいは、ここでの“条件”に相当する公理によって規定されるとし、データオブジェクトに関する表明には抽象代数の技法を使います。どちらの技法も、Liskov と Zilles の論文 (1975) で調査されています。

やり方であるということが示せます。

ここで注意すべき微妙なところは、`(cons x y)` に返される値は手続き—具体的には、内部で定義された手続き `dispatch`—だということです。`dispatch` は、ひとつの引数を取り、その引数が 0 か 1 かによって `x` か `y` かを返します。それに対応して、`(car z)` は `z` に 0 を適用するよう¹に定義されています。よって、もし `z` が `(cons x y)` によって作られた手続きであれば、`z` を 0 に適用すると `x` を返します。こうして、`(car (cons x y))` が期待通り `x` を返すということが示せました。同様に、`(cdr (cons x y))` は `(cons x y)` の返り値となる手続きを 1 に適用し、`y` を返します。このため、この手続きによるペアの実装は有効な実装であり、`cons`, `car`, `cdr` だけを使ってペアにアクセスするのであれば、この実装を“本物の”データ構造を使う実装と区別することはできません。

手続きによるペアの実装の実例を示したことのポイントは、私たちの言語がこういうふうに動いているというわけではなく (Scheme を含め、Lisp システムは一般的に、効率上の理由からペアを直接的に実装しています)、こういうふうに動くこともできるということです。手続きによる表現はわかりにくいけれど、ペアが満たす必要のある条件を満たしているため、ペアを表現する完全に適切な方法と言えます。ここでの例は、手続きをオブジェクトとして操作する能力は自動的に複合データを表現する能力を提供するということも示しています。これは、今の段階ではただの面白い考え方のように思えるかもしれません、手続きによるデータの表現というものは、私たちのプログラミングのレパートリーの中で中心的な役割を果たしています。このプログラミングスタイルはよく *message passing* (メッセージパッシング) と呼ばれ、Chapter 3 でモデル化とシミュレーションの問題を扱う際には、これを基本的な道具として使うことになります。

Exercise 2.4: 次に示すものは、手続きによるペアの表現の別のやり方である。この表現で、任意のオブジェクト `x` と `y` に対して、`(car (cons x y))` が `x` を返すことを確認せよ。

```
(define (cons x y)
  (lambda (m) (m x y)))
(define (car z)
  (z (lambda (p q) p)))
```

対応する `cdr` の定義はどうなるだろうか。(ヒント：これが動作することを確認するにはSection 1.1.5の置換モデルを利用する)

Exercise 2.5: a と b のペアを積 $2^a 3^b$ の整数で表現することによつ

て、非負整数のペアを数値と数値演算だけを使って表現できるということを示せ。

それに対応する `cons`, `car`, `cdr` 手続きを定義せよ。

Exercise 2.6: ペアを手続きとして表現するという考え方で頭がごちやごちやにならないとしたら、次のようなことを考えてみよう。手続きを扱うことができる言語では、(少なくとも、非負整数に関する限りは) 数値なしでもやっていける。そのためには、0と、1を足すという演算を次のように実装する。

```
(define zero (lambda (f) (lambda (x) x)))
(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

この表現は、発明者の Alonzo Church にちなんで *Church numerals* (チャーチ数) として知られている。Alonzo Church は λ -演算を発明した論理学者である。

`one` と `two` を直接 (`zero` と `add-1` を使わずに) 定義せよ (ヒント：置換を使って (`add-1 zero`) を評価する)。加算手続きの直接的な定義 `+(add-1 の繰り返し適用は用いない)` を与えよ。

2.1.4 発展問題: 区間演算

Alyssa P. Hacker は工学上の問題の解決を助けるシステムを設計しています。システムで提供したい機能のひとつとして、精度のわかつている不正確な量 (例えば物理装置の測定値) を扱う能力があります。そのような近似値を使って計算を行った場合、結果が精度つきの数値となるようにしたいということです。

電気技師は Alyssa のシステムを使って電気に関するいろいろな値を計算します。その際に、二つの抵抗 R_1 , R_2 の並列に等価な抵抗値 R_p を次の式によって計算することが必要になるときがあります。

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

抵抗値は普通、抵抗の製造者によって保証された何らかの許容誤差の範囲内であることしかわかりません。例えば、“10% の許容誤差で 6.8Ω ” というラベルのついた抵抗を買うと、その抵抗の抵抗値は $6.8 - 0.68 = 6.12$ か

ら $6.8 + 0.68 = 7.48\Omega$ の間であるということしか確かに言えません。そのため、 6.8Ω 10% の抵抗と 4.7Ω 5% の抵抗を並列に接続すると、その組み合わせの抵抗は約 2.58Ω (二つの抵抗が下限値である場合) から約 2.97Ω (二つの抵抗が上限値である場合) までの範囲になります。

Alyssa のアイデアは、“区間”(不正確な量の取り得る値の範囲を表すオブジェクト)を組み合わせる数値演算のセットとして“区間演算”を実装するというものでした。二つの区間を足し算、引き算、かけ算、割り算した結果は、それ自身が区間であり、結果の範囲を表します。

Alyssa は、下限と上限という二つの端点を持つ“区間”という抽象オブジェクトがあると仮定しました。また、区間の両端点が与えられれば、データ構造 `make-interval` を使って区間を構築できるとしました。Alyssa はまず、二つの区間を足す手続きを書きます。彼女は、和が取り得る最小値は二つの下限の和であり、最大値は二つの上限の和だろうと推論します。

```
(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                 (+ (upper-bound x) (upper-bound y))))
```

また Alyssa は、両端点の積の最小値と最大値を求め、それらを答えとなる区間の下限と上限とすることによって、二つの区間の積を計算しました(`min` と `max` は、任意の数の引数の最小値と最大値を見つける基本要素です)。

```
(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)))
```

二つの区間の割り算は、一つ目に二つ目の逆数をかけることにします。区間の逆数の下限と上限は、上限の逆数と下限の逆数という順番になることに気をつけてください。

```
(define (div-interval x y)
  (mul-interval
   x
   (make-interval (/ 1.0 (upper-bound y))
                  (/ 1.0 (lower-bound y)))))
```

Exercise 2.7: 区間の抽象化の実装を規定していないため、Alyssa のプログラムは未完成である。区間のコンストラクタの定義は以下のようになる。

```
(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 と同じような推論を使って、二つの区間の差をどのように計算するかを記述せよ。それに対応する引き算の手続き `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分である。幅は、区間によって規定される数値の不確かさの程度である。いくつかの数値演算では、二つの区間を組み合わせた結果の幅は、引数の区間の幅のみによる関数となるが、それら以外の演算では、組み合わせたものの幅は、引数の幅の関数とはならない。二つの区間の和（または差）の幅は、足される（または引かれる）区間の幅の関数となることを示せ。これが乗算と除算については正しくないということを例を挙げて示せ。

Exercise 2.10: エキスパートシステムプログラマの Ben Bitdiddle は、Alyssa の肩越しにのぞいてきて、長さがゼロの区間で割るということの意味がはっきりしていないとコメントした。Alyssa のコードを修正し、この条件についてチェックして、もしこれが起こればエラーのシグナルを送るようにせよ。

Exercise 2.11: ついでに Ben はまた謎めいたコメントを残した。“区間の両端点の符号をテストすると `mul-interval` は 9 パターンに場合分けできて、2 回以上のかけ算が必要になるのはその中のひとつだけだよ。” Ben の提案に従って、この手続きを書き直せ。

プログラムをデバッグしてから、Alyssa はユーザ候補の一人にそれを見せた。彼は、このプログラムは解いている問題が違うと言う。ほしいのは中央値と許容誤差で表される数値を扱うプログラムだ。例えば 3.5 ± 0.15 のような区間を使って作業をしたいのであって、 $[3.35, 3.65]$ ではない。Alyssa は自分のデスクに戻って、別のコンストラクタと別のセレクタを提供することによってこの問題を解決した。

```
(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))
```

運の悪いことに、Alyssa のユーザのほとんどは技師です。実際の工学の現場では、測定値の誤差は小さなもので、区間の中央値に対する幅の割合として表されます。技師は普通、上で書いた抵抗のスペックのように、装置のパラメタに対するパーセント許容誤差を規定するものです。

Exercise 2.12: 中央値とパーセント許容誤差を取り、求める範囲を返すコンストラクタ `make-center-percent` を定義せよ。また、与えられた区間にに対してパーセント許容誤差を返すセレクタ `percent` も定義しなければならない。`center` セレクタは上に書いたものと同じである。

Exercise 2.13: パーセント許容誤差が小さいという前提のもとでは、二つの区間の積のパーセント許容誤差を因数の許容誤差の積によって近似する簡単な式が存在することを示せ。すべての数値は正であると仮定して問題を単純化してもよい。

かなりの作業のうちに、Alyssa P. Hacker は完成したシステムを納品しました。数年後、そのことをすっかり忘れていたころ、怒り狂ったユーザ Lem E. Tweakit から興奮した口調の電話がかかってきました。どうも Lem は、並列抵抗の式が代数的に等価な二つの方法で書けることに気づいたようです。

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

と、

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

彼は以下の二つのプログラムを書きました。それぞれ、並列抵抗の式を別々のやり方で計算します。

```
(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 のプログラムが二つの計算方法で違う値を返すと苦情を言います。これは深刻な苦情です。

Exercise 2.14: Lem が正しいことを示せ。いろいろな数値演算の式についてこのシステムの挙動を調査せよ。何らかの区間 A と B を作成し、それらを使って式 A/A と A/B を計算せよ。中央値に対して小さなパーセンテージの幅を持つ区間について調べると得られるところが大きいだろう。中央値 - パーセント形式 (Exercise 2.12 参照) の計算結果を調べよ。

Exercise 2.15: 別のユーザ Eva Lu Ator も、代数的に等価な異なる式によって計算すると異なる区間になることに気がついた。彼女は、不確定な数値を表す変数が繰り返し出てこないように書けば、より厳密な誤差限界を返すようにできると言う。つまり、並列抵抗を計算するのに `par2` は `par1` よりも “よりよい” プログラムだと言うのだ。彼女は正しいだろうか。また、それはなぜだろうか。

Exercise 2.16: 一般的に、なぜ代数的に等価な式が異なる答えになることがあるのかを説明せよ。この欠点のない区間演算パッケージを開発することはできるだろうか。または、このタスクは不可能なのだろうか。(警告：この問題はとても難しい)

2.2 階層データと閉包性

ここまで見てきたように、ペアは私たちが複合データオブジェクトを構築する基本的な“糊”となってくれます。Figure 2.2 は、ペア — この場合は `(cons 1 2)` で作られるペア — を図示する標準的なやり方を示しています。この表現は *box-and-pointer notation* (箱とポインタ記法) と呼ばれるもので、それぞれのオブジェクトは箱への *pointer* (ポインタ) として表わされます。基本オブジェクトの箱にはオブジェクトの表現が入っています。例えば、数値の箱に

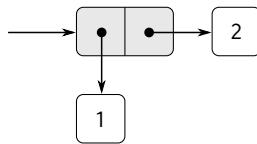


Figure 2.2: (cons 1 2) の箱とポインタ表現

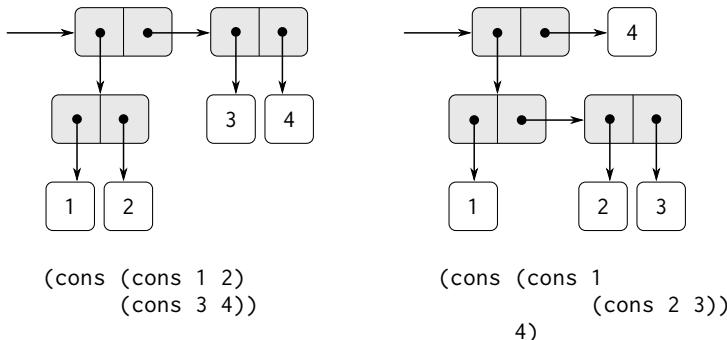


Figure 2.3: ペアを使って 1,2,3,4 を組み合わせる二つの方法

は数字が入っています。ペアの箱は実際には二つの箱で、左の箱にはペアの car(へのポインタ) が入り、右の箱には cdr が入ります。

cons は、数値だけでなくペアも結合できるということについてはすでに見ました (Exercise 2.2とExercise 2.3を解く際には、この事実を使ったか、使わざるを得なかつたでしょう)。結果として、ペアはどんな種類のデータ構造でも構築することができる汎用ブロックとなってくれます。Figure 2.3はペアを使って数値 1, 2, 3, 4 を組み合わせる二つの方法を示しています。

ペアを要素とするペアを作る能力は、表現ツールとしてのリスト構造の持つ重要性の本質となるものです。この能力のことを、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、その演算による組み合せの結果自身が同じ演算によって組み合わせること

ができるのであれば、閉包性を満たします。⁶ 閉包は、どんな組み合わせ手段であっても、表現力の鍵となるものです。それによって、*hierarchical(階層)* 構造—部品によって作られ、その部品もまた部品によって作られ…という構造—を作ることが可能になるからです。

Chapter 1の初めから、手続きを扱う際には本質的に閉包を使ってきました。とても単純なプログラム以外は、組み合わせの要素はそれ自身組み合わせとなりうるという事実に依存しているからです。この節では、複合データの閉包もたらす結果について扱います。ペアを使って列や木を表現するいくつかの確立されたテクニックについて説明し、閉包を鮮やかに図解してくれるグラフィック言語を示します。⁷

2.2.1 列の表現

ペアを使って構築できる便利な構造のひとつは、*sequence(列)*—データオブジェクトの順序つき集合—です。もちろん、ペアを使って列を表すやり方はいろいろあります。Figure 2.4には、特に素直な表現方法のひとつを示しています。このやり方では、1, 2, 3, 4という列はペアのチェーンとして表されます。それぞれのペアの *car* はチェーン内の対応する項で、ペアの *cdr* はチェーン内

⁶ ここでの“閉包”という言葉は抽象代数から来ています。抽象代数では、集合の要素に演算を適用して得られるものがまたその集合の要素であるなら、それらの要素の集合はその演算のもとで閉じていると言います。Lisp コミュニティでも（不幸なことに）“閉包”という言葉を使うのですが、まったく関係のない概念の記述に使っていて、その場合の閉包とは自由変数を持つ手続きを表現する実装上のテクニックを指しています。この本では、二番目の意味での“閉包”は使いません。

⁷ 組み合わせ手段は閉包性を満たすべきだという概念は素直な考え方です。残念なことに、一般的に使われる言語の多くでは、データ結合手段として提供されているものが閉包性を満たさなかつたり、閉包を使うのが面倒だったりします。Fortran や BASIC では、普通は配列にまとめて入れることでデータを組み合わせます。しかし、要素がそれ自身配列であるような配列は作れません。Pascal と C では、要素が構造体であるような構造体を認めています。しかし、これをするためにプログラマが明示的にポインタを扱わなければならず、またプログラマは構造体の各フィールドが前もって決められた形の要素しか持つことができないという制約に従わなければなりません。これらの言語は、Lisp がペアとして提供しているような、複合データを統一的な方法で簡単に操作できるビルトインの汎用の糊を持っていません。この制約が、Alan Perlis がこの本の前書きでコメントしたことの背景にあります。“Pascal では、宣言可能なデータ構造の多さが関数の専門化を引き起こし、気軽な連携は禁じられ、ペナルティを与えるものとなります。ひとつのデータ構造を扱う 100 の関数があるほうが、10 のデータ構造を扱う 10 の関数があるのよりもよいのです。”

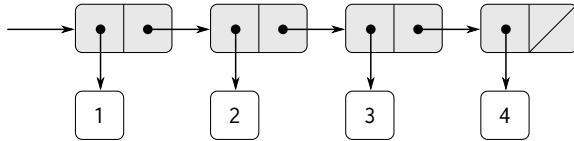


Figure 2.4: ペアの鎖として表現された列 1, 2, 3, 4

の次のペアです。最後のペアの `cdr` は、ペアでない特別な値を指すことによって、列の終端であることを伝えます。その値は、箱-ポインタ図では斜線として表され、プログラムでは変数 `nil` という値として表されます。列全体は、入れ子の `cons` 演算によって構築されます。

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

入れ子の `cons` によって作られるこのようなペアの列は *list* (リスト) と呼ばれます。Scheme は、リスト構築を便利にする `list` という基本手続きを提供しています。⁸ 上の列は、`(list 1 2 3 4)` によって生成することができます。一般的に、

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

は次のものと等価です。

```
(cons <a1>
      (cons <a2>
            (cons ...
                  (cons <an>
                        nil)...)))
```

Lisp システムは、慣例的にリストを括弧でくくられた要素の列として表示します。そのため、Figure 2.4 のデータオブジェクトは `(1 2 3 4)` と表示されます。

⁸ この本では、*list*(リスト) という言葉はリスト終端マーカーで終わるペアのチェーンという意味で使います。それに対して、*list structure*(リスト構造) という用語は、リストに限らず、ペアによって作られた任意のデータ構造を指します。

```
(define one-through-four (list 1 2 3 4))  
one-through-four  
(1 2 3 4)
```

式 (list 1 2 3 4) と、その式を評価した結果として得られるリスト (1 2 3 4) を混同しないように気をつけてください。式 (1 2 3 4) を評価しようとすると、インタプリタは手続き 1 を引数 2, 3, 4 に適用しようとして、エラーを起こします。

`car` はリストの最初の項を選択し、`cdr` は最初の項以外のすべてからなるサブリストを選択するものであると考えることができます。`car` と `cdr` を入れ子で適用していくと、リストの二つ目、三つ目、それからそれに続く項を取り出すことができます。⁹ コンストラクタ `cons` は、元のリストと同じリストに最初にひとつ項が加わったものを作ります。

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

`nil` という値はペアのチェーンを終了させるために使うものですが、これを要素のない列 *empty list* (空リスト) と考えることもできます。`nil` という単語は、“無”を意味するラテン語の単語 *nihil* を短くしたものです。¹⁰

⁹ `car` と `cdr` の入れ子適用は書くのが面倒なので、Lisp の諸方言はそれらの省略形を提供しています。例えば、

```
(cadr <arg>) = (car (cdr <arg>))
```

このような手続きの名前はすべて `c` で始まり `r` で終わります。間の `a` は `car` 演算を表し、`d` は `cdr` 演算を表し、名前の中での順番と同じ順番で適用されます。`cadr` のような単純な組み合わせが発音可能になるように、`car` と `cdr` の名前を残しています。

¹⁰ Lisp の諸方言の標準化にあたって、どれだけのエネルギーが文字通り無に消えたかというの特筆すべきことです。`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？ リストであるべきか？ ペアであるべきか？ Scheme では `nil` は普通の名前で、こ

リスト演算

ペアを使って要素の列をリストとして表現するやり方は、リストを“cdr ダウン”していくという確立されたプログラミングテクニックとセットになっています。例えば、`list-ref` 手続きは引数としてリストと数値 n を取り、リストの n 番目のものを返します。リストの要素の番号は 0 から始めるのが慣例となっています。`list-ref` を計算する方法は以下の通りです。

- $n = 0$ であれば、`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 $(n - 1)$ 番目のものを返す。

```
(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

リスト全体を `cdr` ダウンすることもよくあります。それをやりやすくするために Scheme は基本述語 `null?` を持っています。これは、引数が空リストかどうかをテストするものです。`length` というリストの要素数を返す手続きは、その典型的な使用パターンを示しています。

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

4

手続き `length` は、単純な再帰計画を実装しています。簡約ステップは以下の通りです。

この節では値がリスト終端マーカーであるような変数として使います (`true` が真の値を持つ普通の変数であるのと同じです)。Common Lisp などの Lisp 方言では、`nil` を特別な記号として扱います。この本の著者たちは言語標準化で多くの争いを経験してきたので、この問題そのものを避けたいと思っています。Section 2.3 で クオートを導入したら、空リストを '`()`' として示し、完全に変数 `nil` を使わないですむようにします。

- 任意のリストの `length` は、リストの `cdr` の `length` に 1 を足した値である。

これは、次の基本ケースに着くまで連続的に適用される。

- 空リストの `length` は 0 である。

`length` は、反復スタイルで計算することもできます。

```
(define (length items)
  (define (length-iter a count)
    (if (null? a)
        count
        (length-iter (cdr a) (+ 1 count))))
  (length-iter items 0))
```

もうひとつの確立されたプログラミングテクニックは、リストを `cdr` ダウンしながら答えのリストを “cons アップ” するというものです。これは、次の `append` 手続きで使われます。`append` 手手続きは、二つのリストを引数として取り、要素を連結して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` も再帰計画を使って実装されています。リスト `list1` と `list2` を `append` するには、次のようにします。

- もし `list1` が空リストであれば、結果は単に `list2` である。
- そうでなければ、`list1` の `cdr` と `list2` を `append` し、その結果に `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）リストの最後の要素のみを持つリストを返す手続き `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き `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の両替パターン数計算プログラムについて考える。プログラムで使う通貨を変えられるようにして、例えばイギリスポンドの両替パターン数を計算できるようになると便利だろう。現状のプログラムでは、通貨についての知識は一部は `first-denomination` に、一部は `count-change` (アメリカには5種類のコインがあるということを知っている) に分散されている。両替に使うコインのリストを与えられるようになればもっとくなるだろう。

手続き `cc` を書き直して、二つ目の引数として取るものと、どのコインを使うかを指定する整数ではなく、使うコインの値のリストとなるようにしたい。それから、それぞれの通貨の種類を定義するリストを次のように持つておく。

```
(define us-coins (list 50 25 10 5 1))  
(define uk-coins (list 100 50 20 10 5 2 1 0.5))
```

そして、`cc` は次のように呼び出すことにしよう。

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

このためには、プログラム `cc` に対して何らかの変更を加える必要がある。プログラムの形は同じだが、二つ目の引数に対するアクセスの仕方が次のように変わる。

```
(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基本演算によって、手続き `first-denomination`, `except-first-denomination`, `and no-more?` を定義せよ。リスト `coin-values` の順序は、`cc` によって返される解答に影響を与えるだろうか、それとも与えないだろうか。それはどうしてだろうか。

Exercise 2.20: 手続き `+`, `*`, `list` は任意の数の引数を取る。このような手続きを定義する方法のひとつとして、`define` で *dotted-tail notation* (ドット末尾記法) を使うというものがある。手続きの定義の中で、仮引数リストの最後の仮引数名の前にドットのあるものは、手続きが呼ばれるときに、前のほうの仮引数は(もしあれば)通常通り前のほうの引数の値を持つことになるが、最後の仮引数の値は残りの引数すべてのリストとなる。例えば、次のような定義の場合、

```
(define (f x y . z) (body))
```

手続き `f` は、二つ以上の引数で呼び出すことができる。次のものを評価すると、

```
(f 1 2 3 4 5 6)
```

`f` の本体では、`x` は 1 となり、`y` は 2 で、`z` はリスト (3 4 5 6) となる。次の定義の場合、

```
(define (g . w) (body))
```

手続き `g` は 0 個以上の引数で呼び出すことができる。次のものを評価すると、

```
(g 1 2 3 4 5 6)
```

`g` の本体では、`w` はリスト (1 2 3 4 5 6) となる。¹¹

¹¹ `lambda` を使って `f` と `g` を定義する場合、次のように書けます。

`same-parity` は、ひとつ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引数すべてのリストを返す手続きとする。例えば、次のようになる。

```
(same-parity 1 2 3 4 5 6 7)
(1 3 5 7)
(same-parity 2 3 4 5 6 7)
(2 4 6)
```

上記の記法を使ってこの手続きを書け。

リストに対するマップ

非常に便利な演算として、リストのそれぞれの要素に何らかの変換を適用し、結果のリストを返すというものがあります。例えば、次の手続きはリストのそれぞれの数値を与えられた係数倍します。

```
(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)
```

Section 1.3でやったように、この一般的な考え方を抽象化し、高階手続きによって表現される共通パターンとして捉えることができます。ここでの高階手続きは、`map` と呼ばれるものです。`map` は、1引数の手続きとリストを引数として取り、リストのそれぞれの要素に手続きを適用した結果のリストを返します。

¹²

```
(define f (lambda (x y . z) (body)))
(define g (lambda w (body)))
```

¹² Scheme は、ここで記述したものよりも汎用的な `map` 手続きを標準で提供しています。そのより汎用的な `map` は、引数として n 引数の手続きと n 個のリストを取り、手続きを、すべてのリストの一つ目の要素、すべてのリストの二つ目の要素…に対して適用

```
(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`を使って`scale-list`の新しい定義を書くことができます。

```
(define (scale-list items factor)
  (map (lambda (x) (* x factor))
       items))
```

`map`は重要な概念ですが、これが共通パターンを捉えているという理由からだけではありません。`map`はリストを扱ううえでのより高いレベルの抽象化を達成するからです。元の`scale-list`の定義では、プログラムは再帰構造になつていて、リストの要素ごとの処理に注意が向くようになっていました。`map`によって`scale-list`を定義すると、そのレベルの細かいことは隠されて、係数倍というものが要素のリストを結果のリストに変換するものであるというところが強調されます。二つの定義の違いは、コンピュータが異なるプロセスを実行しているというところではなく（プロセスは同じものです）、私たちがプロセスを異なるやり方で考えるというところにあります。実際に`map`は、リストを変形する手続きの実装を、リストの要素をどうやって取り出して結合していくかという細かいところから切り離す抽象化の壁を作ることを助けてくれます。`Figure 2.1`で示した壁のように、この抽象化を使うことで、列を列に変形するという概念的な演算フレームワークを保ったままで、列がどのように実装さ

していき、結果のリストを返します。例えば、次のようにになります。

```
(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)
```

れているかという低レベルな細かいところを変更できる柔軟性が得られます。Section 2.2.3では、列をプログラム組織化のフレームワークとするというこの使い方についてさらに詳しく説明します。

Exercise 2.21: 手続き square-list は、引数として数値のリストを取り、それらの数値の二乗の列を持つリストを返す。

```
(square-list (list 1 2 3 4))  
(1 4 9 16)
```

次に二つの異なる square-list の定義を示す。書けた部分を埋め、二つとも完成させよ。

```
(define (square-list items)  
  (if (null? items)  
      nil  
      (cons (??) (??))))  
(define (square-list items)  
  (map (??) (??)))
```

Exercise 2.22: Louis Reasoner はExercise 2.21の一つ目の square-list 手続きを書き直し、反復プロセスを展開するようにしようとしている。

```
(define (square-list items)  
  (define (iter things answer)  
    (if (null? things)  
        answer  
        (iter (cdr things)  
              (cons (square (car things))  
                    answer))))  
  (iter items nil))
```

残念ながら、このように square-list を定義すると、答えとなるリストは望むものの逆順になってしまう。なぜだろうか。

それから、Louis は cons の引数を逆順にしてバグを直そうとした。

```
(define (square-list items)  
  (define (iter things answer)
```

```

(if (null? things)
    answer
    (iter (cdr things)
          (cons answer
                (square (car things))))))
  (iter items nil))

```

これもうまくいかない。説明せよ。

Exercise 2.23: 手続き `for-each` は `map` に似ている。引数として手続きと要素のリストを取る。しかし、結果のリストを生成するのではなく、`for-each` は手続きをそれぞれの要素に左から右に順番に手続きを適用していくだけである。手続きを要素に適用して返される値はまったく使わない—`for-each` は、表示のような、何かのアクションを実行する手続きとともに使う。例えば次の例で、

```

(for-each (lambda (x)
  (newline)
  (display x))
  (list 57 321 88))

57
321
88

```

`for-each` への呼び出し (上では示していないが) に返される値は何でもよく、例えば真の値などでもよい。`for-each` を実装せよ。

2.2.2 階層構造

リストによる列の表現は、要素がそれ自身列であるような列を表現するように自然に一般化できます。例えば、次の式によって構築されるオブジェクト `((1 2) 3 4)` は、

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

三つの項目のリストとして見ることができます。最初の項目はそれ自身が `(1 2)` というリストとなります。実際、このことはインタプリタによって表示される結果の形からも見て取ることができます。Figure 2.5は、この構造のペアによる表現を示しています。

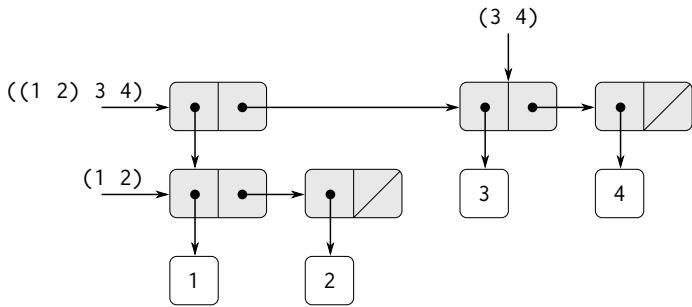


Figure 2.5: (`cons (list 1 2) (list 3 4)`) によって作られる構造

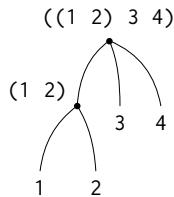


Figure 2.6: Figure 2.5のリスト構造を木として見たもの

要素が列であるような列は、*trees(木)* として考えることもできます。列の要素は木の枝で、それ自身が列である要素は部分木となります。Figure 2.6に、Figure 2.5を木として見た構造を示します。

再帰は木構造を扱う自然なツールです。再帰を使うことで、木への演算をその枝に対する演算に還元し、それを今度は枝の枝に対する演算に還元し…と続けていくことで木の葉にたどり着く、というようにできることができます。ひとつの例として、Section 2.2.1の `length` 手続きと、木の葉の総数を返す `count-leaves` 手続きを比べてみましょう。

```
(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` を実装するために、`length` の計算で使った再帰計画を思い出してみましょう。

- リスト `x` の `length` は、`x` の `cdr` の `length` に 1 を足した値である。
- 空リストの `length` は 0 である。

`count-leaves` も似たようなものになります。空リスト場合の値は同じです。

- 空リストの `count-leaves` は 0 である。

しかし、リストの `car` を取り外す還元ステップでは、その `car` 自身が木であって、その葉の数も数えなければいけないかもしれませんということを考える必要があります。そのため、適切な還元ステップは次のようになります。

- 木 `x` の `count-leaves` は、`x` の `car` の `count-leaves` と `x` の `cdr` の `count-leaves` を足した値である。

最後に、`car` を取っていくと実際の葉に到達するので、基本ケースがもうひとつ必要になります。

- 葉の `count-leaves` は 1 である。

木に対する再帰手続きが書きやすくなるように、Scheme は `pair?` という基本述語を提供しています。これは、引数がペアであるかどうかをテストするものです。以下に完成した手続きを書きます。¹³

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

¹³ `cond` の一番目と二番目の節は、この順番でなければいけません。空リストは `null?` を満たし、またペアでもないからです。

Exercise 2.24: 式 (list 1 (list 2 (list 3 4))) を評価するとする。インタプリタの表示する結果、それに対応する箱-点構造、木としての解釈 (Figure 2.6で示したようなもの) を示せ。

Exercise 2.25: 以下のそれぞれのリストから 7 を取り出す car と cdr の組み合わせを書け。

```
(1 3 (5 7) 9)
((7))
(1 (2 (3 (4 (5 (6 7))))))
```

Exercise 2.26: 二つのリスト x と y を定義するとする。

```
(define x (list 1 2 3))
(define y (list 4 5 6))
```

次のそれぞれの式を評価すると、インタプリタはどのような結果を表示するだろうか。

```
(append x y)
(cons x y)
(list x y)
```

Exercise 2.27: Exercise 2.18 の reverse 手続きを修正し、deep-reverse という手続きを書け。deep-reverse は、ひとつのリストを引数として取り、要素が逆順で、サブリストもすべて要素が逆順になっているリストを返す手続きである。例えば、次のようになる。

```
(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として取り、その木のすべての葉を左から右の順で要素として持つリストを返す手続き fringe を書け。例えば、次のようになる。

```
(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: 二枝モビールは、左の枝と右の枝のという二つの枝から構成されている。それぞれの枝はある長さを持つ棒で、おもりか別の二進モビールがぶら下がっている。二枝モビールは、二つの枝によって構成することで、(例えれば `list` を使って) 複合データによって表現できる。

```
(define (make-mobile left right)
  (list left right))
```

枝は、長さ `length`(必ず数値) と構造 `structure` からなり、`structure` は数値(単純なおもりを表す)か、または別のモビールである。

```
(define (make-branch length structure)
  (list length structure))
```

- a これに対応する、モビールの枝を返すセレクタ `left-branch` と `right-branch`、枝の構成要素を返す `branch-length` と `branch-structure` を書け。
- b これらのセレクタを使って、モビールの総重量を返す手続き `total-weight` を定義せよ。
- c モビールについて、*balanced*(バランスが取れている) というのは、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く(つまり、左の棒の長さとかかる重さの積が、右の棒についてのその積と等しい)、かつ両枝からぶら下がっている各部分モビールのバランスが取れている状態について言う。ある二枝モビールがバランスが取れているかどうかテストする述語を設計せよ。
- d モビールの表現を変更し、以下のようなコンストラクタにする。

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

あなたのプログラムを新しい表現に移行するにはどの程度の変更が必要だろうか。

木に対するマップ

`map` は列を扱うための強力な抽象化ですが、それと同じように、`map` に再帰を組み合わせたものは木を扱うための強力な抽象化です。例えば、`scale-tree` という手続きは、Section 2.2.1 の `scale-list` と似たようなものですが、引数としては、数値の係数と、葉が数値である木を取ります。返り値は同じ形の木で、それぞれの数値が計数倍されたものとなります。`scale-tree` の再帰計画は、`count-leaves` のものに似ています。

```
(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` の別の実装方法として、木を部分木の列と見なして `map` を使うというものがあります。列にマップを行い、それぞれの部分木を順番に係数倍し、結果のリストを返すということになります。木が葉である基本ケースでは、単純に係数倍します。

```
(define (scale-tree tree factor)
  (map (lambda (sub-tree)
          (if (pair? sub-tree)
              (scale-tree sub-tree factor)
              (* sub-tree factor)))
       tree))
```

木の操作の多くは、列の操作と再帰を組み合わせて同じように実装することができます。

Exercise 2.30: Exercise 2.21の square-list と似たような手続き square-tree を定義せよ。square-tree は以下のように動作する。

```
(square-tree
  (list 1
    (list 2 (list 3 4) 5)
    (list 6 7)))
(1 (4 (9 16) 25) (36 49))
```

直接的な(つまり、高階手続きをまったく使わない)やり方と、map と再帰を使うやり方の両方で、square-tree を定義せよ。

Exercise 2.31: Exercise 2.30の解答を抽象化し、手続き tree-map を作れ。square-tree が tree-map によって以下のように定義できるようにせよ。

```
(define (square-tree tree) (tree-map square tree))
```

Exercise 2.32: 集合は、それぞれ異なる要素を持つリストとして表現できる。また、集合のすべての部分集合の集合は、リストのリストとして表現できる。例えば、集合が(1 2 3)の場合、すべての部分集合の集合は((()) (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))となる。以下に、集合のすべての部分集合の集合を生成する手続きを示す。これを完成させ、なぜそれが動作するのかを明確に説明せよ。

```
(define (subsets s)
  (if (null? s)
    (list nil)
    (let ((rest (subsets (cdr s))))
      (append rest (map (lambda (rest) (cons (car s) rest)) rest)))))
```

2.2.3 標準インターフェイスとしての列

ここまで複合データについて見てきましたが、その中では、データ抽象化のおかげでデータ表現の細かいところに煩わされることなくプログラムが設計できること、また抽象化によってほかの表現を実験する柔軟性を確保できるということを強調してきました。この節では、データ構造を扱ううえでの

もうひとつの強力な設計原則を導入します。それは、*conventional interfaces*(標準インターフェイス)の使用です。

Section 1.3では、数値を扱うプログラムについて、プログラムの抽象化を高階手続きとして実装することによって共通のパターンを捉えることができるということを見てきました。複合データについて同じような演算を定式化する能力は、私たちがデータ構造を操作するスタイルに決定的に依存します。例えば、次の手続きについて考えてみます。これはSection 2.2.2のcount-leavesと同じような仕組みで、木を引数に取り、奇数の葉の二乗の合計を計算するというものです。

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

表面的には、この手続きは以下に示す手続きとまったく違うように見えます。以下の手続きは、与えられた整数 n 以下の k について、フィボナッチ数 $\text{Fib}(k)$ のうち偶数のもののリストを構築するというものです。

```
(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)))
```

これら二つの手続きは構造的には大きく違いますが、二つの計算をより抽象的に記述すると、とても似通っていることが明らかになります。最初のプログラムがしていることは、以下のように書くことができます。

- 木の葉を列举し、
- フィルタによって奇数を選び、
- 選ばれた数の二乗を求め、

- `+`を使って、0から始めて結果を集積する。

二つ目のプログラムは、以下の通りです。

- 0から n までの数値を列挙し、
- それぞれの整数に対するフィボナッチ数を求め、
- フィルタによって偶数を選び、
- `cons`を使って、空リストから始めて結果を集積する。

信号処理のエンジニアであれば、Figure 2.7に示したように、直列につながれたいくつかの段階を通る信号の流れとしてこれらのプロセスを概念化するのが自然だと思うでしょう。`sum-odd-squares`は *enumerator(列挙)* から始まります。この列挙は、与えられた木が持つ葉からなる“信号”を生成します。この信号は *filter(フィルタ)* に渡され、奇数要素以外がすべて取り除かれます。その結果となる信号は、今度は *map(マップ)* に渡されます。このマップは、それぞれの要素に `square` を適用する“変換器”です。マップの出力は、それから *accumulator(集積機)* に与えられ、集積機は 0から始めて要素を `+` によって組み合わせます。`even-fibs` の図式も、同じような仕組みです。

残念ながら、上に示した二つの手続きの定義は、この信号の流れという構造を見せることができていません。例えば、`sum-odd-squares` を詳しく見てみると、列挙の処理は、一部は `null?` と `pair?` というテストによって実装され、一部は木の再帰という手続きの構造によって実装されています。同じように、集積の処理の一部はテストのところにあり、一部は再帰の中で使われている足し算のところにあります。全体的に、どちらの手続きにも、信号の流れという描写の中の要素に対応するはっきりとした部分というものはありません。この二つの手続きでは、計算過程を違う方法で分解しています。列挙処理をプログラム全体に広げ、それをマップ、フィルタ、集積の処理と混ぜ合わせているのです。信号の流れという構造が手続きの中で明確に見て取れるようにプログラムを構成することができれば、結果となるコードの概念的な明確さを高めることができるでしょう。

列の演算

信号の流れという構造をより明確に反映するようにプログラムを構成するうえでのポイントは、処理の中で各段階の間を流れる“信号”に集中することです。これらの信号をリストとして表現するなら、各段階の処理を実装するのにリスト演算を使うことができます。例えば、信号の流れ図の中でのマップ段階は、Section 2.2.1の `map` 手続きを使って実装できます。

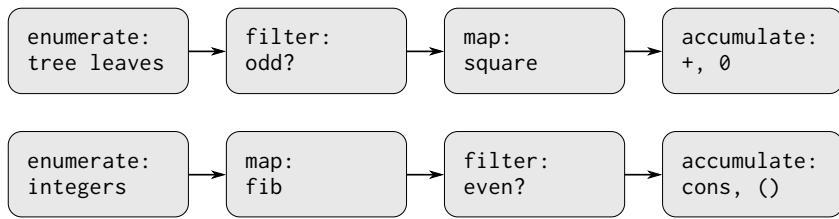


Figure 2.7: 手続き sum-odd-squares(上) と even-fibs(下)を信号の流れという図式によって表現すると、二つのプログラムの共通性が明らかになる。

```
(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だけを選択するのは、次のようにすればできます。

```
(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence)))))
        (else (filter predicate (cdr sequence))))))
```

例えば、次のようになります。

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

集積は、次のように実装できます。

```
(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)
```

後は、信号の流れ図を実装するのに足りないものは、処理する要素の列を列挙することだけです。`even-fibs` の場合、生成するものは与えられた範囲の整数列で、次のように実装できます。

```
(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)
```

木の葉の列挙は、次のようにします。¹⁴

```
(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、信号の流れ図として示したような形で定式化し直すことができます。`sum-odd-squares` のほうは、木の葉の列を列挙し、これをフィルタにかけて列の奇数だけが残るようにし、それぞれの要素を二乗し、結果を合計します。

```
(define (sum-odd-squares tree)  
  (accumulate  
    + 0 (map square (filter odd? (enumerate-tree tree)))))
```

¹⁴これは、実際はExercise 2.28の`fringe`手続きそのものです。ここでは、この手続きが汎用的な列操作手続きに属するものであることを強調するために名前をつけ替えていきます。

`even-fibs` のほうでは、0 から n までの整数を列挙し、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、結果の列をフィルタにかけて偶数の要素だけが残るようにして、結果をリストに集積します。

```
(define (even-fibs n)
  (accumulate
    cons
    nil
    (filter even? (map fib (enumerate-interval 0 n)))))
```

プログラムを列の演算として表すことの利点は、モジュール化された形でのプログラムの設計がやりやすくなるということにあります。モジュール化とは、比較的独立した部品を組み立ててプログラムを構築する設計のことです。標準コンポーネントのライブラリとともに、コンポーネントを柔軟に接続できる標準インターフェイスを提供することで、モジュール化された設計を促すことができます。

モジュールによる設計は、工学の設計において複雑性をコントロールする強力な戦略です。例えば、実際の信号処理の応用では、標準化されたフィルタや変換器から要素を選び、それを直列につなぐことによってシステムを構築するということを、設計者は日常的に行っています。同じように、列の演算は自由に組み合わせられる標準的なプログラムの要素のライブラリを提供します。例として、最初の $n+1$ 個のフィボナッチ数の二乗のリストを構築するプログラムを作るのに、`sum-odd-squares` 手続きと `even-fibs` 手続きの部品を再利用することができます。

```
(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)
```

部品を配置し直して、整数列の中の奇数ものの二乗の積を計算するために使うこともできます。

```
(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))
```

一般的なデータ処理アプリケーションを列の演算として定式化することもできます。人事記録の列があるとして、最も給料の高いプログラマの給料を見つけたいとします。ある人事記録に含まれる給料を返す `salary` というセレクタと、ある人事記録がプログラマのものであるかをチェックする `programmer?` というセレクタが用意されているとします。すると、次のように書くことができます。

```
(define (salary-of-highest-paid-programmer records)
  (accumulate
    max 0 (map salary (filter programmer? records))))
```

これらの例は、列の演算として表すことができる広範囲な演算のごく一部を垣間見たにすぎません。¹⁵

ここでリストとして実装した列というものは、処理モジュールを接続できるようにする標準インターフェイスとして使うことができます。また、構造を列として統一的に表現するときに、プログラム中のデータ構造の依存関係が少ない数の列の演算に局所化されるようにしています。これらを変えることで、プログラムの全体的な設計に手を加えずに、列の表現方法をいろいろ試してみることができます。Section 3.5で、無限列を扱えるように列処理のパラダイムを一般化する際に、この能力を利用することになります。

Exercise 2.33: 基本的なリスト操作のいくつかを集積として定義したものを作成せよ。

```
(define (map p sequence)
  (accumulate (lambda (x y) (??)) nil sequence))
(define (append seq1 seq2)
  (accumulate cons (??) (??)))
```

¹⁵Richard Waters (1979)は、伝統的な Fortran プログラムを自動的に解析し、プログラムをマップ、フィルタ、集積の組み合わせとして見ることができるようにするプログラムを開発しました。すると、Fortran 科学サブルーチンパッケージの実に 90% が、このパラダイムにうまくはまることがわかりました。Lisp がプログラミング言語として成功した理由のひとつには、順序つきのコレクションを表現する標準的な手段がリストとして提供され、高階演算によって操作ができるということがあります。プログラミング言語 APL も同じような選択をして、それが力と魅力の大きな源となっています。APL では、すべてのデータは配列として表され、配列のさまざまな演算ができる普遍的で便利な汎用演算子セットが提供されています。

```
(define (length sequence)
  (accumulate (??) 0 sequence))
```

Exercise 2.34: ある x の値によって x の多項式を評価することは集積として定式化できる。次の多項式について考える。

$$a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$$

この多項式を *Horner's rule*(ホーナー法) というアルゴリズムによって評価する。ホーナー法では、この計算を以下のような構造にする。

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

つまり、 a_n から始めて、それに x をかけ、 a_{n-1} を足し、 x をかけ…ということを、 a_0 まで繰り返す。¹⁶

以下のひな形を埋め、多項式をホーナー法によって評価する手続きを作れ。多項式の係数 $a_0 \dots a_n$ は列として並んでいるとする。

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms) (??))
              0
              coefficient-sequence))
```

例えれば、 $x = 2$ のときの $1 + 3x + 5x^3 + x^5$ を計算するには、次を評価する。

```
(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。

¹⁶Knuth 1981によると、この方法は 19 世紀始めに W. G. Horner によって定式化されたということですが、実際は 100 年以上前にニュートンによって使われていました。ホーナー法のやり方では、直接的なやり方（まず a_nx^n を計算し、 $a_{n-1}x^{n-1}$ を足し…）よりも少ない回数の和と積によって多項式を評価します。実際に、任意の多項式を評価するどんなアルゴリズムも、少なくともホーナー法以上の和と積を使う必要があり、ホーナー法が多項式評価のための最適なアルゴリズムであるということが証明可能です。このことは、(和の回数については)1954 年の A. M. Ostrowski の論文で証明され、それが実質的に最適アルゴリズムの現代的研究の始まりとなりました。積の回数についての同じような証明は、1966 年に V. Y. Pan によってなされました。Borodin and Munro (1975) の本では、これらを含めた最適アルゴリズムについて概観しています。

```
(define (count-leaves t)
  (accumulate ?? ?? (map ?? ??)))
```

Exercise 2.36: 手続き `accumulate-n` は `accumulate` に似ているが、三番目の引数として列の列を取る(要素となる列の長さは一定であるとする)という違いがある。この手続きは、指定された集積手続きを適用してそれぞれの列の最初の要素を結合したもの、それぞれの列の二番目の要素を結合したもの...を返り値とする。例えば、`s` が ((1 2 3) (4 5 6) (7 8 9) (10 11 12)) という四つの列を持つ列であるときに (`accumulate-n + 0 s`) の値が (`accumulate-n + 0 s`) になるようにする。次の `accumulate-n` の定義に欠けている式を埋めよ。

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (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。この表現を使うと、列操作によって基本的な行列とベクトルの演算を簡潔に表現することができる。これらの(どんな行列演算の本にも書いてある)演算は、次のようなものである。

<code>(dot-product v w)</code>	総和 $\sum_i v_i w_i$ を返す
<code>(matrix-*-vector m v)</code>	$t_i = \sum_j m_{ij} v_j$ であるような ベクトル \mathbf{t} を返す
<code>(matrix-*-matrix m n)</code>	$p_{ij} = \sum_k m_{ik} n_{kj}$ であるような 行列 \mathbf{p} を返す
<code>(transpose m)</code>	$n_{ij} = m_{ji}$ であるような 行列 \mathbf{n} を返す

内積は、次のように定義できる。¹⁷

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

ほかの行列演算を計算する以下の手続きについて、欠けた式を補え（手続き `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)))
```

Exercise 2.38: `accumulate` 手続きは、列の最初の要素と、右のすべての要素を組み合わせた結果とを組み合わせるため、`fold-right` としても知られている。`fold-left` というものもあり、これは `fold-right` に似ているが、要素の組み合わせを逆方向に行うという点が違う。

```
(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))
```

¹⁷この定義では、Footnote 12で説明した拡張バージョンの `map` を使っています。

`fold-right` と `fold-left` が任意の列に対して同じ値を返すことを保証するために、`op` はどのような性質を満たさなければならぬだろうか。

Exercise 2.39: 以下の `reverse`(Exercise 2.18) 手続きの定義を、Exercise 2.38 の `fold-right` と `fold-left` によって完成させよ。

```
(define (reverse sequence)
  (fold-right (lambda (x y) (??)) nil sequence))
(define (reverse sequence)
  (fold-left (lambda (x y) (??)) nil sequence))
```

マップのネスト

列というパラダイムを拡張して、普通はネストしたループによって表現されるような多くの計算に適用することができます。¹⁸次の問題について考えてみましょう。正の整数 n が与えられたとき、 $1 \leq j < i \leq n$ で、かつ $i + j$ が素数となるような異なる正の整数 i と j のすべての順序つきペアを見つけよ。例えば、 n が 6 のとき、ペアは以下のようになる。

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

この計算の自然な構築方法として、 n 以下の正の整数からな、大きい順に並んだすべてのペアの列を生成し、フィルタによって合計が素数となるペアを選択し、それからフィルタを通過したそれぞれの (i, j) のペアに対して $(i, j, i + j)$ という三つ組みを作るというものがあります。

ペアの列を生成する方法としては、すべての整数 $i \leq n$ に対して整数 $j < i$ を列挙し、そのような i と j すべてに対して (i, j) を生成するというものがあります。列の演算を使って、`(enumerate-interval 1 n)` という列にマップを行います。この列のそれぞれの i に対して、`(enumerate-interval 1 (- i 1))` という列にマップを行います。この後者の列のそれぞれの j に対して、`(list i`

¹⁸ このマップのネストというやり方は、David Turner によって示されたものです。彼が開発した言語の KRC と Miranda では、これらの概念を扱うエレガントな形式が用意されています。この節の例（と Exercise 2.42）は、Turner 1981 をアレンジしたものです。Section 3.5.3 では、このやり方が無限の長さの列に対して一般化できることを学びます。

j) というペアを生成します。これによって、それぞれの i に対するペアの列ができます。すべての i に対して、すべての列を (`append` で集積して) 組み合わせることで、求めるペアの列ができます。¹⁹

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

マップと集積を `append` によって組み合わせることは、この種のプログラムではとてもよくあることですので、独立した手続きとして分離することにします。

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

次に、このペアの列をフィルタして、和が素数となるものを探します。フィルタの述語は、列のそれぞれの要素に対して呼ばれます。述語の引数はペアで、述語はペアから整数を取り出す必要があります。そのため、列のそれぞれの要素に適用する述語は次のようになります。

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

最後に、フィルタを通ったペアの列に対して次の手続きでマップして、結果の列を生成します。手続きは、ペアの二つの要素とその合計からなる三つ組を構築するものです。

```
(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))
```

これらのステップすべてを組み合わせると、完全な手続きになります。

```
(define (prime-sum-pairs n)
  (map make-pair-sum
    (filter prime-sum?
      (flatmap
        (lambda (i)
          (map (lambda (j) (list i j))
```

¹⁹ ここではペアを、Lisp のペアとしてではなく、二つの要素のリストとして表現しています。このため、“ペア” (i, j) は `(list i j)` で、`(cons i j)` ではありません。

```
(enumerate-interval 1 (- i 1)))
(enumerate-interval 1 n))))
```

マップのネストは、区間の列挙以外の列に対しても役に立ちます。ある集合 S に対して、そのすべての順列を生成したいとします。つまり、集合の項目を並べる方法をすべて生成するということになります。例えば、 $\{1, 2, 3\}$ の順列は、 $\{1, 2, 3\}, \{1, 3, 2\}, \{2, 1, 3\}, \{2, 3, 1\}, \{3, 1, 2\}, \{3, 2, 1\}$ です。集合 S の順列を生成するための計画として、 S のすべての項目 x に対して、 $S - x$ の順列の列を生成し、²⁰ それぞれの先頭に x を置くというやり方が使えます。こうすると、 S の x それぞれに対して、 x から始まる S の順列の列が得られます。すべての x に対するこれらの列を組み合わせると、 S の順列すべてが得られます。²¹

```
(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)))
```

この戦略によって、 S の順列を求める問題を、 S よりも要素数の少ない集合の順列を求める問題に縮約していることに注意してください。最終的には、要素のない集合を表す空リストに到達します。この場合、(list nil) というひとつの項目からなる列、つまり空集合を生成します。permutations で使われる remove 手続きは、与えられた列から与えられた項目を除いたすべての項目を返します。これは単純なフィルタとして表現できます。

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item)))
          sequence))
```

Exercise 2.40: 整数 n に対し、 $1 \leq j < i \leq n$ となるペア (i, j) の列を生成する手続き unique-pairs を定義せよ。unique-pairs を使って上の prime-sum-pairs の定義を簡単にせよ。

²⁰集合 $S - x$ は、 S のすべての要素から x を除いた集合です。

²¹Scheme のコードでは、セミコロンは comments(コメント) を入れる場合に使います。セミコロンから行末までにあるものはすべてインタプリタに無視されます。この本ではありません。プログラムに対して説明的な名前を付けることで、プログラム自身がドキュメント化されたものとなるよう努力しています。

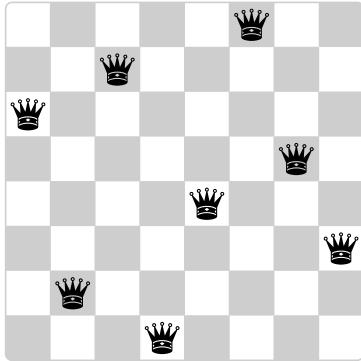


Figure 2.8: 8 クイーンパズルのひとつの解

Exercise 2.41: ある整数 n 以下の異なる正の整数が大小順に並んだ三つ組 i, j, k の中で、合計がある整数 s となるものすべてを見つける手続きを書け。

Exercise 2.42:

“8 クイーンパズル”とは、どのクイーンもほかのクイーンの利き筋に入らない（つまり、同じ行・列・対角線の上に二つのクイーンがあるということないようにする）ように 8 個のクイーンをチェス盤の上に置く方法を問うものである。Figure 2.8 に考えられる解のひとつを示す。このパズルを解くひとつのやり方として、各列にひとつのクイーンを置きながらチェス盤を横に移動していくというものがある。 $k - 1$ 個のクイーンを置いた状態では、 k 個目のクイーンは、すでに盤上にあるどのクイーンも利き筋に入らない位置に置く必要がある。この解き方は再帰的に定式化できる。最初の $k - 1$ 列に $k - 1$ 個のクイーンを置くパターンをすべて生成済みだと想定する。それぞれのパターンに対して、 k 列目のそれぞれの行にクイーンを置いていって、位置の集合を拡張したものを生成する。次にこれらをフィルタして、 k 列目のクイーンがほかのクイーンに利かないものだけを残す。こうすると、最初の k 列に k 個のクイーンを置くすべてのパターンの列ができる。このプロセスを続けると、パズルの解答のひとつだけではなく、すべての解

答が得られる。

この解き方を `queens` という手続きとして実装し、 $n \times n$ のチェス盤に n 個のクイーンを置くという問題に対するすべての答えの列を返すようにする。`queens` は、盤の最初の k 列にクイーンを置く全てのパターンの列を返す内部手続き `queen-cols` を持っている。

```
(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))
```

この手続きの中で、`rest-of-queens` は最初の $k - 1$ 列に $k - 1$ 個のクイーンを置くパターンのひとつで、`new-row` は k 列目のクイーンを置く候補となる行である。盤上の位置集合に対する表現方法と、位置集合に新しい行 - 列の位置を追加する `adjoin-position` 手続きと位置の空集合を表す `empty-board` を実装し、プログラムを完成させよ。また、位置集合に対して、 k 列目のクイーンがほかのクイーンに利いていないかを調べる `safe?` 手続きも書く必要がある(新しいクイーンの利きだけをチェックすればいいということに注意。ほかのクイーンは互いに利いていないことが保証済みである)。

Exercise 2.43: Louis Reasoner は Exercise 2.42 を解くのにだいぶ苦労している。彼が書いた `queens` 手続きは動いてはいるようだが、非常に遅いのだ (6×6 の場合でも、Louis は待ちきれなくなってしまう)。Louis が Eva Lu Ator に見てくれるよう頼むと、彼女は

Louis が `flatmap` 内のマップのネストの順番を次のように逆にしてしまっていることを指摘した。

```
(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))
```

逆にするとなぜプログラムの実行が遅くなるのか説明せよ。[Exercise 2.42](#) のプログラムが 8 クイーンパズルを解く時間を T として、Louis のプログラムがパズルを解くのにかかる時間を見積もれ。

2.2.4 例: 図形言語

この節では、図形を描く簡単な言語を紹介し、それによって抽象と閉包の持つ力を示し、また高階手続きの本質的な部分を利用します。この言語は、Figure 2.9 のような、要素をずらしたり拡大縮小したりしたものによって構成されるパターンについての実験がやりやすいように設計されています。²² この言語では、組み合わされるデータオブジェクトは、リスト構造ではなく手続きとして表現されます。閉包性を満たす `cons` によってどんな複雑なリスト構造でも簡単に構築できるのと同じように、この言語の演算も閉包性を満たしていて、どんな複雑なパターンでも簡単に構築できます。

図形言語

Section 1.1 でプログラミングの学習を始めたとき、言語を記述する際には言語の基本要素、組み合わせ方法、抽象化方法にフォーカスを当てることが重要だということを強調してきました。ここでもその枠組みに従います。

この図形言語のエレガントさの一部は、*painter*(ペインタ) という一種類の要素しかないというところにあります。ペインタは、指定された平行四辺形型の枠にフィットするように画像をずらしたり拡大縮小したりしたものを描画し

²² この図形言語は、M.C. Escher の “Square Limit” ([Henderson 1982 参照](#)) という木版画のような画像を構築するために Peter Henderson が作成した言語をもとにしています。この木版画は、この節の `square-limit` 手続きによって描かれるものと似た、拡大縮小した繰り返しパターンを含んでいます。

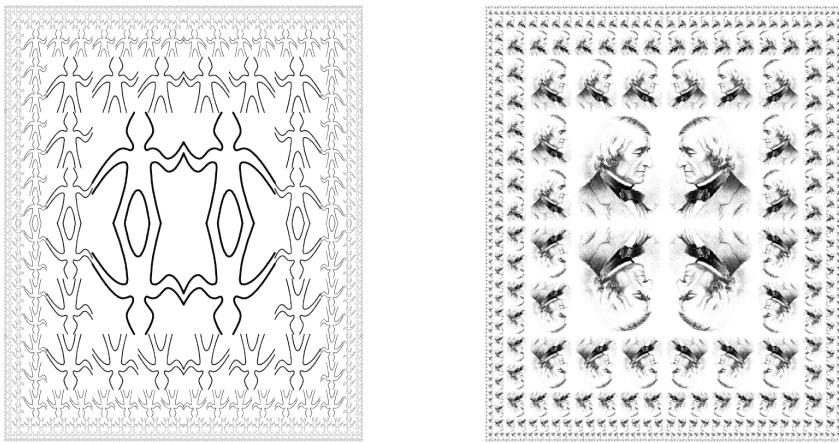


Figure 2.9: 図形言語によって生成されたデザイン

ます。例えば、`wave` という基本ペインタがあり、Figure 2.10のような粗い線画を描きます。実際の線画の形は枠によって変わります—Figure 2.10の四つの画像は同じ `wave` ペインタによって生成されたものですが、それぞれ違った枠に対して描かれています。もっと手の込んだペインタもあります。`rogers` という基本ペインタは、Figure 2.11に示すように、MIT 創設者の William Barton Rogers の絵を描きます。²³ Figure 2.11の 4 つの画像は、Figure 2.10の `wave` の

²³William Barton Rogers (1804-1882) は MIT の創設者で、また初代学長でもあります。地質学者であり、才能あふれる教師でもあった彼は、ウイリアム・アンド・メリーラー大学とバージニア大学で教鞭をとっていました。1859 年にはボストンに移り、それまでより研究の時間が持てるようになって、「工科大学」を設立する計画に取りかかりました。また、マサチューセッツ州で初めてのガスマータの州検査官も務めました。

1861 年に MIT が創設されたとき、Rogers は最初の学長に選ばれました。Rogers は「役に立つ学習」という理想を信奉していましたが、それは当時の大学教育とは異なるものでした。当時の大学教育は古典を過度に重視していて、彼の書くところによると「より幅広く高度で実用的な自然・社会科学の教育や訓練を行ううえでの障害となっている」ものでした。彼はまた、そこでの教育は職業専門学校の狭い教育とも異なるものにしようと思っていました。Roger の言葉によると、

世間が押しつけてくるような、実務者と科学者の区別というのはまったく不毛なものだ。現代の経験はすべて、そのような区別が完全に無益だとい

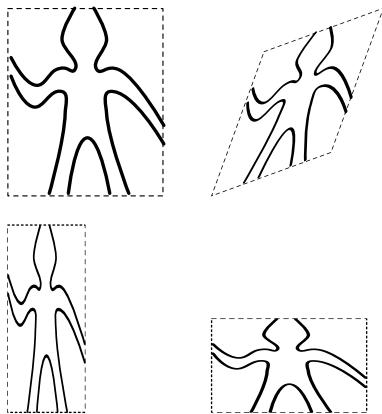


Figure 2.10: wave によって生成された画像

うことを示している。

Rogers は 1870 年に健康上の理由で辞任するまで MIT の学長を務めました。1873 年恐慌による財政危機とハーバード大学による併合の試みをはねのけるという重圧から二代目の MIT 学長 John Runkle が 1878 年に辞職すると、Roger は学長職に戻り、1881 年まで務めました。

Rogers は 1882 年の卒業式で MIT の卒業生に演説をしているときに倒れ、亡くなりました。Runkle は、同年の弔辞の中で、Roger の死に際の言葉を引用しています。

“本日ここに立ち、本校の現状を見ると… 科学の始まりのことが頭に浮かびます。記憶によると、150 年前に Stephen Hales が灯用ガスを主題にした小論文を発表しました。その中で彼は、彼の研究が示すところによると 128 グレインの瀝青炭 – ” “瀝青炭” というのが、この世界での彼の最後の言葉でした。ここで彼は前のめりになり、まるで目の前の台に置いてあるメモを確認するかのようでしたが、それからゆっくりと直立の姿勢に戻り、両手を上げ、そしてこの世の仕事と業績から “死の明日” へと送られたのです。そこでは人生の謎は解決され、肉体から解放された魂は、新しく未だ計り知れない永遠の未来の謎について考え続けることに終わりのない満足を得るのです。

Francis A. Walker(MIT の三代目の学長)によると、

彼は人生を通して誠実かつ勇敢に行動し、騎士であれば必ず望んだようなよい死の迎え方をした。職務中に、持ち場について、公務を果たすちょう

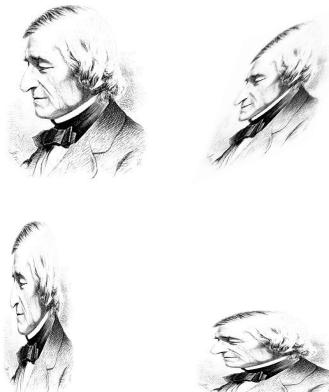


Figure 2.11: MIT 創設者・初代学長 William Barton Rogers の画像。Figure 2.10と同じ4つの枠に対して描いたもの。(元の画像はMIT博物館の許諾を得て複製した)

画像と同じ4つの枠に対して描いたものです。

画像を結合するには、与えられたペインタから新しいペインタを構築するいろいろな演算を使います。例えば、`beside` 演算は二つのペインタを引数に取り、枠の左半分に一つ目のペインタの画像を描き右半分に二つ目のペインタの画像を描く新しい複合ペインタを作ります。同じように、`below` は二つのペインタを引数に取り、二つ目のペインタの画像の下に一つ目のペインタの画像を描く複合ペインタを作ります。単独のペインタを変形して別のペインタを作る演算もあります。例えば、`flip-vert` はひとつのペインタを引数に取り、その画像を上下逆に描くプリンタを作ります。また、`flip-horiz` は元のペインタの画像を左右逆に描くプリンタを作ります。

`wave` から始めて、二段階を経て構築した `wave4` というペインタの描く画像を Figure 2.12 に示します。

どそのときに亡くなったのだ。

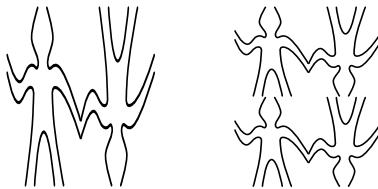


Figure 2.12: Figure 2.10の wave ペインタから始めて複雑な図を作る

```
(define wave2 (beside wave (flip-vert wave)))
(define wave4 (below wave2 wave2))
```

複雑な画像をこの方法で構築する際に、ペインタがこの言語の結合手段について閉じているということを利用しています。二つのペインタの `beside` や `below` は、それ自身ペインタになります。そのため、より複雑なペインタを作るのに、それを要素として使うことができます。リスト構造を `cons` を使って構築する場合と同じように、データが結合手段について閉じているということは、少しの演算によって複雑な構造を作る能力を実現するうえで決定的に重要なことです。

ペインタを結合することができるとなると、よくある結合パターンを抽象化したくなるところです。ここで、ペインタ演算を Scheme の手続きとして実装することにします。つまり、この図形言語のための特別な抽象化メカニズムというものは必要ないということです。結合手段は普通の Scheme 手続きなので、手続きについてできることは何でもペインタ演算についてもできる能力が自動的に得られることになります。例えば、`wave4` に出てくるパターンは次のように抽象化できます。

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

そして、`wave4` はこのパターンの具体例として定義します。

```
(define wave4 (flipped-pairs wave))
```

また、再帰演算を定義することもできます。以下に示すものは、Figure 2.13と Figure 2.14に示すような、右方向に枝分かれしていくペインタを作る再

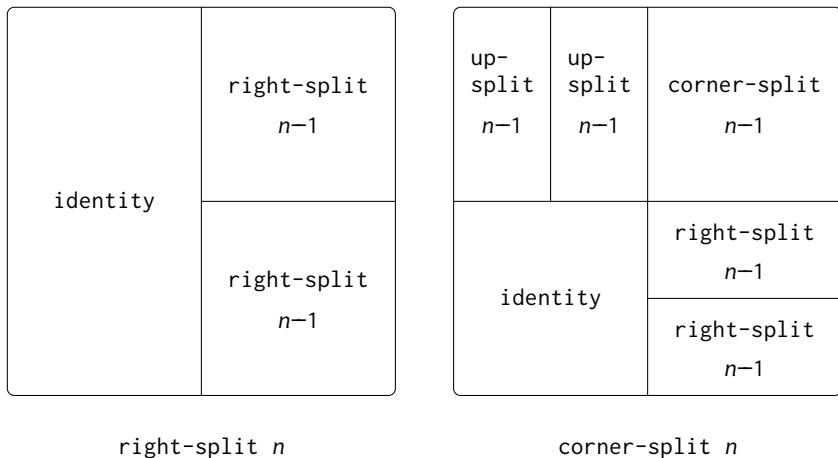


Figure 2.13: right-split と corner-split の再帰計画

帰演算です。

```
(define (right-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (right-split painter (- n 1))))
        (beside painter (below smaller))))))
```

右方向だけでなく上方向にも枝分かれさせることで、バランスの取れたパターンを作ることもできます(問題Exercise 2.44と、図Figure 2.13・Figure 2.14参照)。

```
(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)))
          (beside (below top-left bottom-right)))))))
```

```

        (corner (corner-split painter (- n 1)))
        (beside (below painter top-left)
            (below bottom-right corner))))))

```

`corner-split` の四つのコピーを適切に配置することによって、`square-limit` というパターンを得ることができます。それを `wave` と `rogers` に適用したものを見 [Figure 2.9](#) に示してあります。

```

(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` で使われている `up-split` 手続きを定義せよ。`right-split` に似ているが、`below` と `beside` の役割を入れ替えている。

高階演算

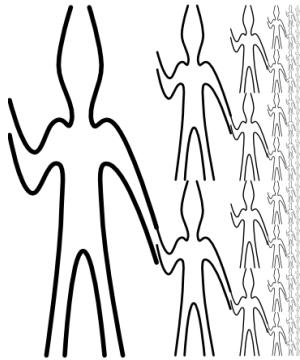
ペイントを組み合わせるパターンを抽象化するだけでなく、より高いレベルで、ペイント演算を組み合わせるパターンを抽象化することもできます。つまり、ペイント演算を操作対象の要素として扱い、それらの要素を組み合わせる手段 — ペイント演算を引数として取り、新しいペイント演算を作るような手続き — を書くことができるということです。

例えば、`flipped-pairs` と `square-limit` はどちらも、あるペイントの画像を四つ正方形のパターンに並べるもので、違うのはそれぞれの画像の向きだけです。このペイントの組み合わせパターンを抽象化する方法のひとつを次の手続きに示します。この手続きは、四つの一引数ペイント演算を引数として取り、ある与えられたペイントをこれらの四つの演算によって変形したものを正方形に配置するペイント演算を作るというものです。`tl`, `tr`, `bl`, `br` は、それぞれ左上、右上、左下、右下の画像に適用する変換です。

```

(define (square-of-four tl tr bl br)
  (lambda (painter)
    (let ((top (beside (tl painter) (tr painter)))
          (bottom (beside (bl painter) (br painter))))
      (below bottom top))))

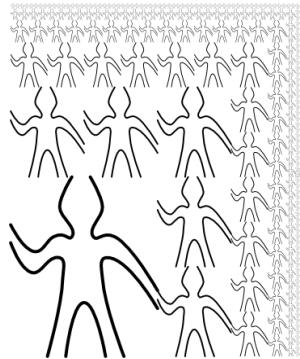
```



(right-split wave 4)



(right-split rogers 4)



(corner-split wave 4)



(corner-split rogers 4)

Figure 2.14: 再帰演算 right-split と corner-split をペインタ wave と rogers に適用したもの。corner-split した四つの図を組み合わせると、Figure 2.9に示すような対照的な square-limit のデザインになる。

こうすると、`flipped-pairs` は `square-of-four` によって以下のように定義で
きます。²⁴

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

また、`square-limit` は次のように表現できます。²⁵

```
(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、汎用の分割演算の具
体例として表現できる。以下の式を評価したとき、

```
(define right-split (split beside below))
(define up-split (split below beside))
```

すでに定義したのものと同じふるまいを持つ手続き `right-split`
と `up-split` が得られるように、`split` 手続きを定義せよ。

枠

ペインタやペインタの組み合わせ手段をどのように実装するかを示す前に、
まず枠について考える必要があります。枠は三つのベクトルとして記述するこ
とができます—原点ベクトルと、二つの辺ベクトルです。原点ベクトルは、平
面上のどこかにある絶対原点からの枠の原点の相対位置を指定します。辺ベク
トルは、枠の角について、枠の原点からの相対位置を指定します。もし二つの

²⁴等価な書き方として、次のように書くこともできます。

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

²⁵`Rotate180` はペインタを 180 度回転するものです (Exercise 2.50 参照)。`rotate180` の
代わりに、Exercise 1.42 の `compose` 手続きをを使って (`compose flip-vert flip-horiz`)
と書くこともできます。

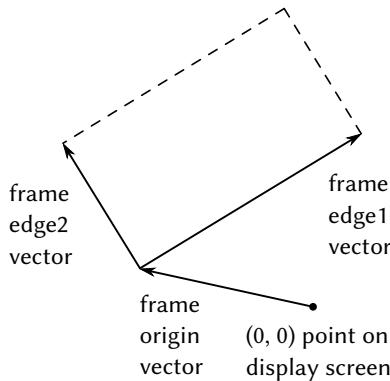


Figure 2.15: 枠は、原点と二つの辺という三つのベクトルによって記述される。

辺が垂直であれば、枠は長方形になります。そうでない場合、枠はより一般的な平行四辺形になります。

Figure 2.15は、枠とそれに関連づけられたベクトルを示しています。データ抽象化の原則通り、枠がどのように表現されているかについてここではまだ特定する必要がありません。三つのベクトルを引数に取り枠を作るコンストラクタ `make-frame` と、それに対応する三つのセレクタ `origin-frame`, `edge1-frame`, `edge2-frame` があるというだけで十分です (Exercise 2.47 参照)。

画像は、単位正方形内の座標($0 \leq x, y \leq 1$)を使って記述することにします。それぞれの枠には *frame coordinate map*(枠座標マップ) というものを関連づけ、画像が枠に合うようにずらしたり拡大縮小したりするのに使います。このマップは、ベクトル $\mathbf{v} = (x, y)$ を次のようなベクトルの和にマップすることによって単位正方形を枠内に変換します。

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

例えば、 $(0, 0)$ は枠の原点に、 $(1, 1)$ は原点と対角線上にある頂点に、 $(0.5, 0.5)$ は枠の中心にマップされます。枠座標マップは次の手続きによって作ることができます。²⁶

²⁶ `frame-coord-map` は、後で Exercise 2.46 で記述するベクトル演算を使っています。

```
(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` をある枠に適用すると、ベクトルが与えられるとベクトルを返す手続きを返します。引数ベクトルが単位正方形内にあれば、返り値のベクトルは枠内のものになります。例えば、次の式は

```
((frame-coord-map a-frame) (make-vect 0 0))
```

以下と同じベクトルを返します。

```
(origin-frame a-frame)
```

Exercise 2.46: 原点からある一点に張られる二次元ベクトルは、 x 座標と y 座標からなるペアとして表現できる。コンストラクタ `make-vect` と、それに対応するセレクタ `xcor-vect`, `ycor-vect` を与え、ベクトルに対するデータ抽象化を実装せよ。それらのセレクタとコンストラクタによって、ベクトルの足し算、引き算、スカラによるかけ算という演算を行う手続き `add-vect`, `sub-vect`, `scale-vect` を実装せよ。

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

Exercise 2.47: 次に、枠のコンストラクタとして使えるものを二つ挙げる。

```
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))
```

これらは何らかのベクトルの表現を使って実装されているとします。データ抽象化のおかげで、このベクトル表現がどのようなものであるかは、ふるまいさえ正しければ問題になりません。

```
(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
```

それぞれのコンストラクタに対し、適切なセレクタを加えて枠の実装を完成させよ。

ペインタ

ペインタは、枠を引数として取り、ある特定の画像をずらしたり拡大縮小したりして枠にフィットするように描画する手続きとして表現できます。つまり、`p` がペインタで `f` がフレームであれば、`f` を引数として `p` を呼び出すことで、`f` の中に `p` の画像を生成します。

基本ペインタがどのように実装されるかという具体的なところは、グラフィックシステムの特性と描画する画像のタイプに依存します。例えば、二つの指定された点の間にスクリーン上の線を引く `draw-line` という手続きを持っているとします。すると、Figure 2.10 の `wave` ペインタのような線画のペインタは、次のような線分のリストから作ることができます。²⁷

```
(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)))
```

線分は、単位正方形に対する座標を使って与えられます。リスト中のそれぞれの線分に対し、ペインタは線分の両端点を枠座標マップを使って変換し、変換した点の間に線を引きます。

ペインタを手続きとして表現することは、図形言語の中に強力な抽象化の壁を作ります。さまざまなグラフィック機能に基づくいろいろな基本ペインタを作り、また組み合わせることができます。枠を引数として取り、その枠に合

²⁷ `segments->painter` は、後で Exercise 2.48 で記述する線分の表現を使っています。また、Exercise 2.23 で記述した `for-each` 手続きも利用しています。

わせて何かを描くものであれば、どんな手続きでもペインタとして使うことができます。

²⁸

Exercise 2.48: 平面上の方向つきの線分は、ベクトルのペア—原点から線分の始点へと張られるベクトルと、原点から線分の終点へと張られるベクトル—として表現できる。Exercise 2.46で書いたベクトル表現を使って、`make-segment` というコンストラクタと `start-segment` と `end-segment` というセレクタによって線分の表現を定義せよ。

Exercise 2.49: `segments->painter` を使って、以下の基本ペインタを定義せよ。

- a 指定された枠の輪郭を描くペインタ。
- b 枠の対角線同士をつないで“X”を描くペインタ。
- c 枠の辺の中点をつないで菱形を描くペインタ。
- d `wave` ペインタ。

ペインタの変形と組み合わせ

ペインタに対する演算 (`flip-vert` や `beside` のようなもの) は、引数として取った枠から導出した枠に対して元のペインタを呼び出すようなペインタを作るという仕組みで動いています。ですので、例えば `flip-vert` が画像をひっくり返す際に、ペインタがどのように動作しているかを知る必要はありません—枠をひっくり返す方法さえわかっていれば大丈夫です。ひっくり返したペインタは元のペインタをそのまま使いますが、それをひっくり返した枠に対して使うというだけです。

²⁸ 例えば、Figure 2.11の `rogers` ペインタはグレースケール画像によって構築されています。枠内の各点に対し、`rogers` ペインタは枠座標マップによってそこにマップされる画像内の点を決定し、それに従って点の濃淡をつけます。Section 2.1.3では、有理数の実装はある適切な条件を満たすものであれば何でもいいということを述べましたが、ここでさまざまな種類のペインタを使えるようにしているのは、そこで述べた抽象データという考え方を利用したものです。ここでは、指定された枠の中に何かを描画するものであれば、ペインタはどのような実装でもいいということを利用しています。Section 2.1.3ではペアを手続きとして実装する方法を示しましたが、ペインタはデータを手続きによって表現する二つ目の例となっています。

ペインタ操作は、`transform-painter` という手続きに基づいています。この手続きは引数として、ペインタと、枠をどのように変形するかという情報を取り、新しいペインタを作ります。変形したペインタは、ある枠に対して呼ばれたとき、その枠を変形して、変形した枠に対して元のペインタを呼び出します。`transform-painter` の引数は(ベクトルとして表現した)複数の点で、新しい枠の頂点を指定するものです。これらが枠に対してマップされるとき、一つ目の点は新しい枠の原点を指定し、残り二つの点は枠の辺ベクトルの終点を指定します。つまり、単位正方形内の引数は、元の枠内に含まれる枠を指定することになります。

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

ペインタの画像を上下逆にするには次のようにします。

```
(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)))
```

画像を反時計回りに 90 度回転させるという変形もあります。²⁹

²⁹`rotate90` は回転した枠にフィットするように画像を伸縮させるということも行うため、純粋な回転になるのは枠が正方形のときだけです。

```
(define (rotate90 painter)
  (transform-painter painter
    (make-vect 1.0 0.0)
    (make-vect 1.0 1.0)
    (make-vect 0.0 0.0)))
```

また、画像を枠の中心方向につぶすという変形もあります。³⁰

```
(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)))
```

枠の変換は、二つ以上のペインタを組み合わせる手段を定義するためにキーとなるものであります。例えば、`beside` 手続きは二つのペインタを引数に取り、それらを引数の枠の左半分と右半分を描画するように変形し、新しい複合ペインタを作ります。ある枠を引数として複合ペインタを呼び出すと、一つ目の変形ペインタを呼んで枠の左半分を描画し、二つ目の変形ペインタを呼んで枠の右半分を描画します。

```
(define (beside painter1 painter2)
  (let ((split-point (make-vect 0.5 0.0)))
    (let ((paint-left
           (transform-painter
             painter1
             (make-vect 0.0 0.0)
             split-point
             (make-vect 0.0 1.0)))
          (paint-right
            (transform-painter
              painter2
              split-point
              (make-vect 1.0 0.0)
              (make-vect 0.5 1.0))))
      (lambda (frame)
```

³⁰Figure 2.10とFigure 2.11に出てきた菱形の画像は、`squash-inwards` を `wave` と `rogers` に適用して作ったものです。

```
(paint-left frame)
(paint-right frame))))
```

ペインタのデータ抽象化と、特にペインタを手続きとして実装したことによって、`beside` が実装しやすくなっているということがわかるでしょうか。部品となるペインタについて `beside` が知つておくべきことは、それぞれのペインタが指定した枠に何かを描画するということだけで、そのほかには何も知る必要がありません。

Exercise 2.50: ペインタを左右逆にする変換 `flip-horiz`、ペインタを反時計回りに 180 度、270 度回転させる変換を定義せよ。

Exercise 2.51: ペインタに対する `below` 演算を定義せよ。`below` は二つのペインタを引数に取る。返り値となるペインタは、枠が与えられたとき、一つ目のペインタで枠の下部を描画し、二つ目のペインタで上部を描画する。二つのやり方で `below` を定義せよ。一つ目は、上に示した `beside` 手続きと似たような手続きを書くというもので、二つ目は、`beside` と適切な回転演算 (Exercise 2.50 のもの) によって定義するというものである。

頑健な設計のための言語レベル

図形言語は、手続きとデータによる抽象化についてこれまで紹介してきた重要な考え方の練習となっています。基本的なデータ抽象化であるペインタは手続きによる表現を使って実装され、異なる基礎的な描画機能を言語が統一したやり方で扱うことができるようになっています。結合手段は閉包性を満たしているため、複雑な設計も簡単に組み立てることができます。最後に、手続きを抽象化するためのツールはすべて、ペインタの結合手段を抽象化するのに利用できます。

また、言語とプログラム設計についての、もうひとつの決定的な考え方を垣間見ることができました。それは *stratified design* (階層化設計) というアプローチで、複雑なシステムは一連の言語によって記述される一連のレベルとして構造化されるべきだという概念です。それぞれのレベルは、そのレベルで基本とされる部品を組み合わせて構築され、それぞれのレベルで構築された部品は次のレベルで基本部品として使われます。階層化設計のそれぞれのレベルで使われる言語は、その詳細さのレベルに合った基本要素、結合手段、抽象化の手段を備えています。

階層化設計は、複雑なシステムのエンジニアリングのあらゆるところに出てきます。例えば計算機工学では、抵抗やトランジスタを組み合わせて(またアナログ回路の言語を使って記述して)AND ゲートや OR ゲートのような部品を作り、それらはデジタル回路設計の言語では基本要素となります。³¹ これらの部品を組み合わせてプロセッサ、バス構造、メモリシステムを構築し、今度はコンピューターアーキテクチャに適した言語を使ってそれらを組み合わせてコンピュータを作ります。コンピュータは、ネットワーク相互接続を記述するのに適した言語を使って組み合わされて分散システムを構成します。こういった繰り返しです。

階層化のごく小さな例として、ここでの図形言語は、点や線を指定する言語を使って作られた基本要素(基本ペインタ)によって `segments->painter` の線分のリストや `rogers` のようなペインタの濃淡の詳細を提供します。図形言語についての説明の大半は、これらの基本要素を組み合わせることや、`beside` や `below` のような幾何学的な結合器について焦点を当てていました。さらに、`beside` や `below` を言語の基本要素と見なすことで、さらに高いレベルでの作業も行いました。その言語の `square-of-four` のような演算は、幾何学的な結合器を組み合わせる際のよくあるパターンを捉えています。

階層化設計は、*robust*(頑健)なプログラムを作りやすくなります。頑健というのは、仕様の小さな変更が、プログラム上でも相応に小さな変更ですむということです。例えば、Figure 2.9に示した、`wave` をもとにした画像を変更したいとします。このとき、最も下のレベルに手を加え、`wave` の細かい見た目を変えることができます。また、中間レベルで、`corner-split` が `wave` を複製するやり方を変えることもできます。最も上のレベルで、`square-limit` が四隅の画像を配置するやり方を変えることもできます。一般的に、階層化設計のそれぞれのレベルは、システムの特性を表現するうえでの異なる語彙を提供し、システムを変更するための異なる能力を提供します。

Exercise 2.52: 上で述べた各レベルに手を加え、Figure 2.9に示した `wave` による `square-limit` を変更せよ。具体的には、次のようなものである。

- Exercise 2.49の `wave` 基本ペインタに線分をいくつか加えよ(例えば、笑顔を追加するなど)。
- `corner-split` によって構築されるパターンを変更せよ(例えば、`up-split` や `right-split` の画像を二つではなく一つだけ使うようにする)

³¹Section 3.3.4でそのような言語のひとつを記述します。

c `square-of-four` を使うほうの `square-limit` を修正し、四隅を違うパターンで組み立てるようにせよ(例えば、正方形の四隅で大きなロジャーズ氏が外を向くようにするなど)

2.3 記号データ

ここまで使ってきた複合データオブジェクトはすべて、究極的には数値から構成されていました。この節では、任意の記号をデータとして扱う能力を導入し、私たちの言語の表現能力を拡張します。

2.3.1 クオート

記号を作って複合データを作れるとしたら、次のようなリストが使えることになります。

```
(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、私たちの言語の次のような式と同じように見えるかもしれません。

```
(* (+ 23 45)
  (+ x 9))
(define (fact n)
  (if (= n 1) 1 (* n (fact (- n 1)))))
```

記号を操作するためには、私たちの言語に新しい要素を入れる必要があります。それは、データオブジェクトを `quote`(クオート) する能力です。`(a b)` というリストを構築するという場合について考えてみましょう。これは、`(list a b)` と書いてもうまくいきません。この式は、`a` や `b` といった記号そのものではなく、その値を要素としたリストを作ってしまうからです。この問題は、自然言語についてよく知られています。自然言語では、単語や文が意味的な実体となることもあります、文字の列(統語的な実体)となることもあります。自然言語では、単語や文をそのまま文字の列として扱うということを示すのに、引用符を使うのが一般的です。例えば、“John”の最初の文字というと、それは当然 “J” です。誰かに“あなたの名前を声に出して言ってください”と頼んだら、その人

の名前を言ってくれるでしょう。しかし、“‘あなたの名前’を声に出して言ってください”と頼んだら、“あなたの名前”という単語列を言ってもらえるはずです。ほかの人の発言を伝えるために、引用符をネストさせないといけなくなっていることに気をつけてください。³²

リストや記号について、式として評価の対象にするのではなく、データオブジェクトとして扱いたいというときにも、これと同じような慣例に従うことができます。しかし、ここでのクオートの形式は、自然言語の場合と違って、クオートするオブジェクトの前に引用符（伝統的にシングルクオートの記号 ‘ を使います）をひとつだけ置くというものです。Scheme の文法では、オブジェクトを区切るのに空白や括弧を使っているため、クオートはひとつだけで問題ありません。このようなわけで、シングルクオートは次のオブジェクトをクオートするという意味になっています。³³

これで、記号とその値を区別することができるようになりました。

```
(define a 1)
(define b 2)
(list a b)
(1 2)
(list 'a 'b)
(a b)
(list 'a b)
(a 2)
```

³² 言語でクオートを使えるようにすると、等しいものは等しいもので置き換えられるという概念を破壊するため、言語を単純に推論する能力を台無しにしてしまいます。例えば、三は一足す二ですが、「三」という単語は「一足す二」という句ではありません。クオートを使うと、ほかの式を操作する式を作れるようになるので、そういう意味でクオートは強力です（これについては、Chapter 4でインタプリタを書く際に見ることになります）。しかし、ある言語の中で、その言語で書かれた文について表現する文というものを許すと、「等しいものの同士は交換できる」ということが何を意味するのかについて、一貫性のある原則を維持することがとても難しくなります。例えば、もし宵の明星は明けの明星であると知っていれば、「宵の明星は金星である」という文から、「明けの明星は金星である」と推論できます。しかし、「ジョンは宵の明星は金星であると知っている」ということからは、「ジョンは明けの明星は金星であると知っている」を導くことはできません。

³³ 表示する文字列を囲むのにはダブルクオートを使ってきましたが、シングルクオートはそれとは違うものです。シングルクオートはリストや記号を表すのに使われるのに対しても、ダブルクオートは文字列にしか使われません。この本では、文字列を使うのは項目を表示する場合だけです。

リストを表示するときに使う慣習的な表現を使って複合オブジェクトを入力することも、クオートによってできるようになります。³⁴

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

この原則通りに、'()を評価すると空リストが得られます。これによって、変数 nil は必要なくなります。

記号の操作に使われるもうひとつの基本要素は eq? です。これは二つの記号を引数として取り、それらが同じであるかどうかをテストするものです。³⁵ eq? を使うと、memq という便利な手続きを実装できるようになります。これは記号とリストという二つの引数を取り、記号がリストに含まれていなければ(つまり、リストのどの項目とも eq? でなければ)、memq は偽を返します。そうでなければ、そのリストの中で記号が最初に見つかったところから始まる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、次の式の値は偽になります。

³⁴厳密に言うと、この引用符の使い方は、言語内のすべての複合式は括弧で囲まれていてリストのように見えるという一般的な規則を破っています。ここで、quote という特殊形式を導入することによって、この一貫性を取り戻すことができます。quote の役割はクオートと同じです。つまり、「a」の代わりに (quote a) と入力したり、「(a b c)」の代わりに (quote (a b c)) と入力したりすることができます。これはインタプリタの動作方式そのままです。引用符は、次に来る完全な式を quote を使って包んで、(quote <expression>) を作るということを、一文字に省略したものにすぎません。これは重要なポイントです。こうすることによって、インタプリタから見える式はすべてデータオブジェクトとして操作できるという原則を維持できるからです。例えば、(car '(a b c)) という式は (car (quote (a b c))) と同じものなので、(list 'car (list 'quote '(a b c))) を評価することによって構築できます。

³⁵二つの記号が同じ順番の同じ文字で構成されているとき、それらは“同じ”であると考えることができます。このような定義は、まだここでは取り組むことが難しい、プログラミング言語における“同一性”的意味とは何かという深い問題を避けて通っています。Chapter 3 (Section 3.1.3) では、この問題に戻ってくることになります。

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

一方、次の式の値は (apple pear) になります。

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

Exercise 2.53: 以下の式をそれぞれ評価したとき、それらに対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: 二つのリストは、同じ要素を同じ順番で持つときに equal? であると言える。例えば、

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

は真である。しかし、

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

は偽である。より正確に言うと、記号が等しいかどうかという基本的な eq? を使って、再帰的に equal? を定義することができる。a と b は、どちらも記号であってそれらが eq? であるか、どちらもリストであって、(car a) と (car b) が equal? であり、かつ (cdr a) と (cdr b) が equal? である場合、equal? である。この考え方を使って、equal? を手続きとして実装せよ。³⁶

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

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

³⁶ 実際は、プログラマは記号だけでなく数値を含むリストに対しても使います。数値は記号とは見なされません。数として等しい二つの数値 (=でテストできます) が eq? でもあるかという問題は、高度に実装依存です。equal? の定義としては、(Scheme で基本要素となっているもののように)a と b が両方とも数値である場合、a が b が数として等しければそれらは equal? であるということも規定しておくほうがよいでしょう。

予想外なことに、インタプリタは `quote` という表示を返してきた。
このことを説明せよ。

2.3.2 例: 記号微分

記号操作の例として、またデータ抽象化の別の例として、代数式の記号微分を行う手続きの設計について考えてみましょう。手続きは、引数として代数式と変数を取り、その変数に関するその式の導関数を返すようにします。例えば、手続きに対する引数が $ax^2 + bx + c$ と x であれば、手続きは $2ax + b$ を返すというものです。記号微分は、Lisp にとって特別な歴史的意義があります。記号微分は、記号を操作するコンピュータ言語を開発するうえでの動機となる実例のひとつでした。さらに、記号微分の研究は、現在応用数学者や物理学者に使われるようになってきている強力な記号数式処理システムの開発につながる一連の研究のスタート地点となるものです。

記号微分プログラムを開発するにあたっては、Section 2.1.1 の有理数システムを開発したときと同じデータ抽象化戦略に従います。つまり、まず“和”、“積”、“変数”といった抽象オブジェクトの上で動く微分アルゴリズムを定義することにして、それらの抽象オブジェクトがどのように表現されるべきかということは考えないことにします。表現の問題については後ほど取り組みます。

抽象データによる微分プログラム

問題を簡単にするために、二引数の足し算・かけ算という演算だけから構成される式だけを扱う、とても単純な記号微分プログラムについて考えることにします。任意のそのような式の微分は、以下の簡約規則を適用することによって行うことができます。

$$\frac{dc}{dx} = 0, \quad c \text{ が定数または } x \text{ と異なる変数のとき,}$$

$$\begin{aligned}\frac{dx}{dx} &= 1, \\ \frac{d(u+v)}{dx} &= \frac{du}{dx} + \frac{dv}{dx}, \\ \frac{d(uv)}{dx} &= u \frac{dv}{dx} + v \frac{du}{dx}.\end{aligned}$$

後ろの二つの規則は本質的に再帰的であるということがわかるでしょうか。というるのは、和の導関数を求めるには、まずそれぞれの項の導関数を求め、それ

らを足すことになるからです。それぞれの項は、さらに分解が必要な式であるかもしれません。項をどんどん小さなものに分解していくと、最終的には定数か変数のものになり、その導関数は 0 または 1 になります。

これらの規則を手続きという形にするために、有理数の実装を設計したときと同じように、ちょっとした希望的思考をすることにします。もし代数式の表現手段があるとするなら、ある式が和なのか、積なのか、定数なのか、変数なのか識別できるはずです。また、式から構成要素を抽出することもできるでしょう。和については、例えば加数(第一項)と被加数(第二項)を抽出できるようになっていてほしいところです。それに、構成要素から式を構築することもできるはずです。ここでは、次のようなセレクタ、コンストラクタ、述語を実装する手続きがもうできているとします。

(variable? e)	e は変数か?
(same-variable? v1 v2)	v1 と v2 は同じ変数か?
(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

これらの手続きと、数値かどうかを判断する基本述語の number? を使って、微分の規則を次の手続きのように表現することができます。

```
(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 手続きには、完全な微分アルゴリズムが組み込まれています。抽象データによって表現されているため、ちゃんとしたセレクタとコンストラクタのセットを設計する限り、代数式をどう表現したとしても動くはずです。この問題については次で見ていきます。

代数式を表現する

リスト構造を使って代数式を表現するやり方はいろいろ考えられます。例えば、通常の代数記法を反映した記号のリストを使って、 $ax + b$ を (`a * x + b`) というリストとして表現するということも可能です。ですが、とりわけ素直なやり方は、Lisp が複合式に使うのと同じ、括弧でくくった前置記法を使うというものです。つまり、 $ax + b$ は (`(+ (* a x) b)`) と表現することになります。そうすると、微分問題に対するデータ表現は次のようになります。

- 変数は記号である。基本述語 `symbol?` で識別する。

```
(define (variable? x) (symbol? x))
```

- 二つの変数は、それらを表現する記号が `eq?` であれば等しい。

```
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
```

- 和と積は、リストとして構築する。

```
(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))
```

- 和は、最初の要素が記号 `+` であるリストである。

```
(define (sum? x) (and (pair? x) (eq? (car x) '+)))
```

- 加数は、和のリストの二つ目の項である。

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

- 被加数は、和のリストの三つめの項である。

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

- 積は、最初の要素が記号`*`であるリストである。

```
(define (product? x) (and (pair? x) (eq? (car x) '*)))
```

- 乗数は、積のリストの二つ目の項である。

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

- 被乗数は、積のリストの三つめの項である。

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

こうすると、`deriv`に具体的に書いたようなアルゴリズムによってこれらの要素を組み合わせるだけで、ちゃんと動く記号微分プログラムができます。そのふるまいについて、いくつかの例を見てみましょう。

```
(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答えを返します。しかし、答えは簡約されていません。次の結果は、確かに正しいものです。

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

しかし、プログラムには $x \cdot 0 = 0$, $1 \cdot y = y$, $0 + y = y$ だということをわかってほしいものです。二つ目の例の答えは、単純に y であるべきです。三つ目の例でわかるように、式が複雑であるとこれは深刻な問題になってきます。

ここで問題になっていることは、有理数の実装の際に問題になったこととても似ています。答えが最も単純な形に簡約されていないという問題です。有理数の簡約は、実装の中でコンストラクタとセレクタだけを変更するだけできました。ここでも、同じような戦略が使えます。`deriv`にはまったく手を加えないことにします。その代わり、`make-sum`に変更を加え、もし両方の加数が数値であれば、それらを足し合わせて和を返すようにします。また、加数のひとつが`0`であれば、もうひとつの加数のみを返すようにします。

```
(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))))
```

ここでは、ある式が与えられた数値と等しいかどうかをチェックする `=number?` という手続きを使っています。

```
(define (=number? exp num) (and (number? exp) (= exp num)))
```

同じように、`make-product` に変更を加えて、0 には何をかけても 0 で、1 に何かをかけるとその何か自身になるという規則を組み込みます。

```
(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))))
```

今回のものが、先ほどの三つの例に対してどう動作するかを次に示します。

```
(deriv '(+ x 3) 'x)
1
(deriv '(* x y) 'x)
y
(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* x y) (* y (+ x 3)))
```

これでだいぶよくなりましたが、三つ目の例を見ると、式を誰もが“最も単純”だといえるような形にするプログラムができるまでには、まだだいぶ距離がありそうです。代数的簡約化の問題は、何よりも、ある目的のためには最も単純である形式がほかの目的のためにはそうではないということがありうるため、複雑なものになっています。

Exercise 2.56: より多くの種類の式を扱えるようにこの基本的な微分プログラムを拡張するにはどうすればよいかを示せ。例えば、次の微分規則を実装せよ。

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

実装にあたっては、`deriv` プログラムに新しい節を追加し、また `exponentiation?`, `base`, `exponent`, `make-exponentiation` を適切に定義せよ(幕乗の表現には `**` という記号を使っててもよい)。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規則を組み込め。

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)
```

`deriv` 手続きにはまったく手を加えず、和と積の表現のみを変更することによって解け。例えば、和の `addend`(加数) は最初の項で、`augend`(被加数) は残りの項の和というようにする。

Exercise 2.58: 微分プログラムを修正して、`+` と `*` が前置演算子ではなく中置演算子となるような、通常の数学の記法に対して動作させたいとする。微分プログラムは抽象データによって定義されているので、微分プログラムの動作基盤となる代数式の表現を定義する述語、セレクタ、コンストラクタを変更するだけで、異なる表現を扱うように修正できる。

- a. $(x + (3 * (x + (y + 2))))$ のような中置記法で表された代数式を微分するにはどのようにするかを示せ。問題を簡単にするために、`+` と `*` は常に二つの引数を取り、式は完全に括弧でくくられていると仮定せよ。
- b. $(x + 3 * (x + y + 2))$ のような標準的な代数記法を認めるとき、問題はずつと難しくなる。この記法では、必要のない括弧は省略し、乗算は加算より先に行われると仮定している。ここでの微分プログラムがそのような記法に対してもうまく動くように、適切な述語、セレクタ、コンストラクタを設計することができるだろうか。

2.3.3 例: 集合を表現する

ここまでで例では、有理数と代数式という、二種類の複合データオブジェクトの表現を構築しました。この例のうちのひとつでは、式の単純化(簡約化)

をコンストラクタで行うかセレクタで行うかという選択肢がありましたが、そのほかの点では、これらの構造のリストによる表現の選択肢は素直なものでした。しかし、集合の表現ということになると、表現をどう決めるかということはあまり明らかではありません。実際に、可能な表現は数多くあり、それぞれいくつかの点でお互いに大きく異なっています。

ざっくりした言い方をすると、集合とはただの異なるオブジェクトの集まりです。より正確な定義をするには、データ抽象化の手法を使うことができます。つまり、集合に使われる演算を規定することによって“集合”を定義するということです。それらの演算は、`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構成要素であるかどう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に取り、元の集合の要素に加えて追加された要素を含む集合を返します。`union-set`は、二つの集合の和集合、つまりどちらかの引数に含まれる要素をすべて含む集合を計算します。`intersection-set`は、二つの集合の共通部分、つまり引数のどちらにも含まれる要素のみを含む集合を計算します。データ抽象という視点から見ると、上の解釈と整合性があるものでさえあれば、これらの演算を実装するのにどのような表現でも自由に設計できます。³⁷

順序なしリストとしての集合

リストを表現するやり方のひとつは、要素が二回以上現れることがないリストというものです。空集合は空リストとして表現します。この表現では、`element-of-set?`は Section 2.3.1 の `memq` 手続きと似たものになります。ここでは `eq?` ではなく `equal?` を使っているため、集合の要素は記号である必要はありません。

```
(define (element-of-set? x set)
  (cond ((null? set) false)
```

³⁷さらによく書くなら、“上の解釈と整合性がある”というところを、演算が次のような一連の規則を満たすというように規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))` は真である（ざっくり言うと、“集合にオブジェクトを追加すると、そのオブジェクトを含む集合ができる”）。

任意の集合 `S, 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。（ざっくり言うと、“`(union S T)` の要素は、`S` に含まれる要素か、`T` に含まれる要素である”）。

任意のオブジェクト `x` に対し、`(element-of-set? x '())` は偽である（ざっくり言うと、“空集合の要素となるオブジェクトは存在しない”）。

```
((equal? x (car set)) true)
(else (element-of-set? x (cdr set))))
```

これを使って、`adjoin-set` を書くことができます。追加するオブジェクトがすでに集合の中にあるなら、単にその集合を返します。そうでなければ、`cons` を使ってその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` には、再帰的な戦略が使えます。もし、`set2` と、`set1` の `cdr` との共通部分を作ることができるなら、`set1` の `car` をそれに含めるかどうかを決めるだけで十分です。しかし、これは (`car set1`) が `set2` にも含まれているかどうかによって決まります。結果として、手続きは次のようにになります。

```
(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))))
```

表現を設計する際に考慮するべき問題のひとつは効率です。この集合演算に必要となるステップ数を考えてみましょう。どの演算も `element-of-set?` を使っているので、この演算の速度は集合の実装全体の効率に大きな影響を与えます。ここで、あるオブジェクトが集合の要素かどうかチェックするために、`element-of-set?` は集合全体を走査する必要があるかもしれません（最悪の場合、オブジェクトは集合の中になかったという結果になります）。そのため、集合が n 個の要素を持っているとすると、`element-of-set?` には最大 n ステップかかるかもしれません。つまり、必要なステップ数は $\Theta(n)$ で増加するということです。`adjoin-set` はこの演算を使うので、やはり $\Theta(n)$ で増加します。`intersection-set` は、`set1` の要素それぞれに対して `element-of-set?` のチェックをするため、必要なステップ数は関係する集合の大きさの積に比例して増加、つまり大きさ n の二つの集合に対して $\Theta(n^2)$ で増加するということになります。`union-set` についても同じことが言えます。

Exercise 2.59: 順序なしリストとして表現した集合に対する `union-set` 演算を実装せよ。

Exercise 2.60: 上の例では、集合は重複のないリストとして表現するよう規定した。ここで、重複を許す場合について考えてみよう。その場合、例えば $\{1, 2, 3\}$ という集合は $(2 \ 3 \ 2 \ 1 \ 3 \ 2 \ 2)$ というリストとして表現することもできる。この表現に対して演算を行う手続き `element-of-set?`, `adjoin-set`, `union-set`, `intersection-set` を設計せよ。それぞれの効率は、重複なし表現に対する手続きとそれに対応するものと比べてどうだろうか。重複なしの表現よりもこの表現のほうが向いているような応用はあるだろうか。

順序つきリストとしての集合

この集合演算の速くする方法のひとつとして、集合の要素が昇順に並ぶように表現を変えるというものがあります。これをするためには、二つのオブジェクトを比較してどちらが大きいか判断するための何らかの方法が必要です。例えば、記号を辞書順に比較したり、オブジェクトに一意の数値を割り振ることに決めて対応する数値によって要素を比較したりといったことが考えられます。ここでは話を簡単にするため、集合の要素が数値である場合についてのみ考えることにします。この場合、要素は $>$ と $<$ によって比較できます。数値の集合を表現するのには、要素を昇順に並べたものを使います。上に挙げた最初の表現では、 $\{1, 3, 6, 10\}$ という集合を表すのに要素をどのように並べてもかまいませんでしたが、新しい表現では $(1 \ 3 \ 6 \ 10)$ というリストしか許可しません。

順序に従って並べることのひとつの利点は `element-of-set?` で出てきます。項目があるかどうかを調べるのに、もう集合全体を走査する必要はありません。探している項目よりも大きい要素に出会ったら、その項目が集合内にないということがわかります。

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

これはどれだけのステップ数の節約になるでしょうか。最悪の場合、探している項目は集合内の一一番大きなものかもしれない、その場合ステップ数は順序なし表現の場合と同じです。しかし、いろいろな大きさの項目を検索するときのことを考えると、場合によってはリストの始点のあたりで検索を終了することが

できたり、場合によってはやっぱりリストのほとんどを調べないといけなかつたり、となることが期待できます。平均的には、集合の約半分の項目を検査しなければいけないということになるでしょう。このため、必要な平均ステップ数はだいたい $n/2$ になります。これは依然として $\Theta(n)$ という増加オーダーですが、それでもやはり、平均的には、前の実装に比べてステップ数を半分に節約できます。

`intersection-set` では、さらに目覚ましい高速化が達成できます。順序なし表現では、この演算には $\Theta(n^2)$ というステップ数が必要でした。それは、`set1` の要素それぞれに対して `set2` の完全な走査を行っていたからでした。しかし、順序つきの表現では、より賢い方法が使えます。まず、二つの集合の最初の要素 `x1` と `x2` を比較します。もし `x1` と `x2` が等しければ、それは共通部分の要素になり、共通部分の残りは二つの集合の `cdr` の共通部分ということになります。では、`x1` が `x2` より小さい場合はどうなるか考えてみましょう。`x2` は `set2` の最小要素なので、`x1` は `set2` のどこにも現れないと結論づけることができます。つまり、共通部分にも含まれないということになります。このため共通部分は、`set2` と、`set1` の `cdr` との共通部分に等しくなります。同じように、もし `x2` が `x1` よりも小さければ、求める共通部分は、`set1` と、`set2` の `cdr` との共通部分となります。手続きは次のようになります。

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

このプロセスで必要となるステップ数を見積もるには、各ステップで、共通部分問題をより小さな集合の共通部分集合の計算に縮小していることに注意してください。縮小は、`set1` と `set2` のどちらかまたは両方から最初の要素を削除することによって行われます。このため、必要なステップ数は、最大でも `set1` と `set2` の大きさの合計で、順序なし表現のように大きさの積になることはありません。これは、 $\Theta(n^2)$ というオーダーに対して $\Theta(n)$ というオーダーになる

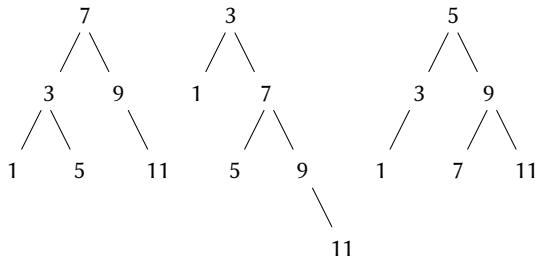


Figure 2.16: 集合 $\{1, 3, 5, 7, 9, 11\}$ を表現するさまざまな二分木

ということで、そこそこの大きさの集合に対してもかなりの高速化になります。

Exercise 2.61: 順序つき表現を使った `adjoin-set` を実装せよ。`element-of-set?` から類推して、順序つきであることの利点を生かして、順序なしの表現に比べて平均的に半分のステップを必要とする手続きを作るやり方を示せ。

Exercise 2.62: 順序つきリストとして表現された集合に対して、`union-set` を $\Theta(n)$ で実装せよ。

二分木としての集合

集合の要素を木という形式で配置することで、順序つきリスト表現をさらに改善することができます。木の各ノードは集合の要素をひとつ持ち、それはノードの“エントリ”と呼ばれます。また、ほかの二つのノード（ノードは空であることもあります）へのリンクも持っています。“左”的リンクは、そのノードより小さな要素を指していて、“右”的リンクはそのノードより大きな要素を指しています。[Figure 2.16](#)に、集合 $\{1, 3, 5, 7, 9, 11\}$ を表現するいくつかの木を示します。ある同じ集合を木という形で表現するのには、数多くの方法があります。ここで、有効な木の表現の条件として求めるものは、あるノードの左部分木のすべての要素がそのノードの要素よりも小さく、右部分木のすべての要素がそれよりも大きいということだけです。

木による表現の利点は次の通りです。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します。まず、 x とトップノードのエントリを

比較します。もし x がそれより小さければ、左部分木だけを探索すればいいということがわかります。もし x がそれより大きければ、右部分木だけを探索すればいいということです。ここで、もし木が“バランスが取れた”状態であれば、各部分木の大きさは元の木の約半分です。そのため、一回のステップで、大きさ n の木を探索するという問題を、大きさ $n/2$ の木を探索するという問題に縮小したことになります。木の大きさは各ステップで半分になるので、大きさ n の木の探索に必要なステップ数は $\Theta(\log n)$ で増加すると予測できます。³⁸大きな集合に対しては、これは以前の表現に比べてかなりの高速化になるでしょう。

木はリストを使って表現できます。各ノードは、ノードのエントリ、左部分木、右部分木という三つの項目のリストになります。左または右の部分木が空リストである場合は、そこにつながる部分木がないということを示します。この表現は、次のような手続きで表すことができます。³⁹

```
(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、上で述べたような戦略を使って `element-of-set?` を書くことができるようになります。

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

³⁸各ステップで問題の大きさが半分になるということは、対数増加の特徴的な性質です。このことについては、Section 1.2.4の高速累乗アルゴリズムや、Section 1.3.3の区間二分探索法見てきました。

³⁹ここでは、集合は木によって表現し、木はリストによって表現しています。つまり、データ抽象化の上にデータ抽象化が構築されているということです。手続き `entry`, `left-branch`, `right-branch`, `make-tree` は、リストによる表現といった特定のやり方から“二分木”という抽象化を分離する方法と見なすことができます。

集合に項目を追加するというのも同じように実装でき、必要ステップ数はやはり $\Theta(\log n)$ になります。項目 x を追加するには、 x をノードのエントリと比較し、右と左のどちらの枝に追加するべきかを決めます。そして、 x を適切な枝に追加したら、この新しく作った枝を元のエントリやほかの枝とくっつけ合わせます。もし x がエントリに等しければ、ノードをそのまま返します。もし空の木に x を追加するよう求められたら、エントリが x で、右と左の枝が空である木を生成します。

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。この手続きを以下に示します。

```
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
                    (right-branch set)))
        ((> x (entry set))
         (make-tree (entry set) (left-branch set)
                    (adjoin-set x (right-branch set)))))))
```

木の探索が対数的ステップで実行できるという上の主張は、木が“バランスが取れている”^{訳注4} という前提に依存しています。つまり、すべての木の左右の部分木はだいたい同じ数の要素を持っていて、各部分木は親の約半分の要素を持っているということです。しかし、ここで構築する木がバランスが取れないと当然にできるでしょうか。バランスの取れた木から始めたとしても、`adjoin-set` で要素を追加すると、バランスの取れていない結果になってしまうかもしれません。新たに追加する要素の位置は、その要素と集合にすでに存在する項目の大小を比べた結果によるので、要素を“ランダムに”追加すれば、平均的には木はバランスの取れたものになりやすくなります。しかし、このことは保証されていません。例えば、もし空集合から始めて、1から7の数値を

訳注4「バランスの取れた木」に対しては“平衡木”、“バランス木”などの訳語がありますが、ここでは*balanced(バランスの取れた)*との兼ね合いから「バランスの取れた木」としています。

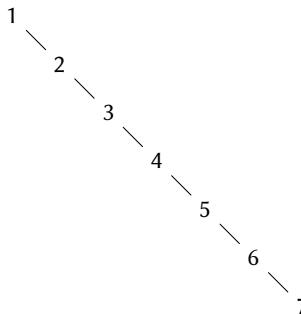


Figure 2.17: 1 から 7 まで順番に追加するとできるバランスの取れていない木

順番に追加していくと、Figure 2.17に示すようなとてもバランスの悪い木ができるあがってしまいます。この木では、すべての左部分木は空なので、単純な順序つきリストに対する優位点がありません。この問題を解決する方法のひとつとして、任意の木を同じ要素を持つバランスの取れた木に変換する演算を定義するというものがあります。こうすると、何回か `adjoin-set` 演算を行なうたびにこの変換を実行して、集合をバランスの取れた状態にしておくことができます。この問題を解く方法はほかにもありますが、だいたいは検索や挿入がどちらも $\Theta(\log n)$ ステップで行えるような新しいデータ構造を設計することをしています。⁴⁰

Exercise 2.63: 次の二つの手続きは、どちらも二分木をリストに変換する。以下の 2 つの手続き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))
              (cons (entry tree)
                    (tree->list-1 (right-branch tree))))))
```

⁴⁰このようなデータ構造の例としては、*B-trees*(B木) や *red-black trees*(赤黒木)などがあります。この問題については多くの文献があります。Cormen et al. 1990 参照。

```

        (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 二つの手続きは、すべての木に対して同じ結果を生成するだろうか。そうでないとしたら、結果はどのように異なるだろうか。二つの手続きは、Figure 2.16の木に対してどのようなリストを生成するだろうか。
- b 二つの手続きは、 n 要素のバランスの取れた木を変換する際に、同じステップ数の増加オーダーを持っているだろうか。そうでないなら、どちらが遅いだろうか。

Exercise 2.64: 次の手続き `list->tree` は、順序つきリストをバランスの取れた木に変換する。補助手続き `partial-tree` は、整数 n と、少なくとも n 個の要素を持つリストを引数に取り、リストの最初の n 個の要素を含むバランスの取れた木を生成する。`partial-tree` の返り値はペア (`cons` で構築される) で、`car` には構築された木を持ち、`cdr` には木に含まれなかった要素のリストを持つ。

```

(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 `partial-tree` がどのように動くのか、できるだけ明確に文
章で答えよ。リスト (1 3 5 7 9 11) に対して `list->tree`
が生成する木を描け。

b `list->tree` が n 要素のリストを変換するのに必要なステッ
プ数の増加オーダはどうなるだろうか。

Exercise 2.65: Exercise 2.63とExercise 2.64の結果を使って、集合
を (バランスの取れた) 二分木として $\Theta(n)$ で実装せよ。⁴¹

集合と情報検索

ここまで、集合をリストで表現するのにどのような方法があるかを検討し、
データオブジェクトに対してどのような表現を選ぶかということが、そのデータ
を使うプログラムのパフォーマンスに大きな影響を及ぼすということを見
てきました。集合に対して詳しく検討するもうひとつの理由は、ここで考察す
るテクニックは情報検索に関わる応用の中によく出てくるものだということ
です。

個別のレコードを大量に含むデータベースについて考えてみましょう。例
えば、企業の人事ファイルや、会計システムの取引などです。データ管理シス
テムは普通、そのレコードにアクセスしたり更新したりするのに多くの時間を

⁴¹Exercise 2.63からExercise 2.65までは Paul Hilfinger によるものである。

消費するので、レコードにアクセスする効率的な手法が必要となります。これは、各レコードの中で、識別key(キー)として使える部分を特定することによって行うことができます。キーは、レコードを一意に識別するものであれば何でもかまいません。人事ファイルであれば、従業員番号などがよいでしょう。会計システムであれば、取引番号になるかもしれません。キーが何であっても、レコードをデータ構造として定義する際には、あるレコードに対して関連づけられたキーを取得するkeyセレクタ手続きを持つ必要があります。

ここで、データベースをレコードの集合として表現することにします。あるキーに対するレコードの場所を調べるには、lookupという手続きを使います。この手続きは、キーとデータベースを引数に取り、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なければ偽を返します。lookupは、element-of-set?とほぼ同じように実装されます。例えば、レコードの集合が順序なしリストとして表現されれば、次のようにすることができます。

```
(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、順序なしリストよりもいい方法があります。レコードを“ランダムアクセス”する必要がある情報検索システムでは、普通は木構造による手法によって表現されます。例えば、前に検討した二分木のようなものです。そのようなシステムを設計するにあたっては、データ抽象化の方法論はとても役に立ちます。設計する人は、順序なしリストのような単純で素直な表現を使って、最初の実装を作ることができます。これは最終的なシステムのためには不適切なものでしょうが、システムの残りの部分をテストするための“quick and dirty”(手早く汚い)データベースを提供してくれるという意味では役に立ちます。データ表現を修正してより洗練されたものにすることは後になってからでもできます。もしデータベースが抽象セレクタとコンストラクタによってアクセスされるようになっていれば、そうやって表現を変更しても、システムの残りの部分には修正の必要がないはずです。

Exercise 2.66:

レコードの集合が、キーの数値の大小によって順序づけられた二分木という構造になっている場合について、lookup手続きを実装せよ。

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い方を練習します。ここでは、データを 1 と 0 の列(ビット列)として表現する手法を適用対象とします。例えば、コンピュータ内でテキストを表現するのに使われる ASCII 標準符号は、各文字を 7 ビットの列として表現します。7 ビットを使うことで、 2^7 、つまり 128 通りの異なる文字を区別することができます。一般的に、 n 個の異なる記号を区別しようと思うと、記号ごとに $\log_2 n$ ビットを使う必要があります。もしメッセージすべてが A, B, C, D, E, F, G, H という 8 個の記号からなるとすると、一文字あたり 3 ビットの符号を選ぶことができます。例えば、次のようにになります。

A 000	C 010	E 100	G 110
B 001	D 011	F 101	H 111

この符号を使って次のメッセージ

BACADAEAFABBAAGAH

を符号化すると、54 ビットの列になります。

001000010000011000100000101000001001000000000110000111

ASCII 符号や、上に挙げた A から H までの符号は、メッセージ内のそれぞれの記号を同じビット数で表すので、*fixed-length(固定長)* 符号といいます。しかし、時には異なる記号が異なるビット数で表されることもある *variable-length(可変長)* 符号を使ったほうがいい場合もあります。例えばモールス符号では、アルファベットの各文字に対して使う短点と長点の数が異なります。特に、最もよく出てくる文字の E は、短点ひとつで表されます。一般的に、もしメッセージの中でとてもよく出てくる記号とほとんど出てこない記号があるのなら、よく出てくる記号に短い符号を割り当てるデータをより効率的に(つまり、メッセージに対して少ないビット数で) 符号化することができます。次に挙げる、A から H までに対する別のコードについて考えてみましょう。

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111

この符号を使って符号化すると、上と同じメッセージが次のようにになります。

100010100101101100011010100100000111001111

この列は 42 ビットなので、上に示した固定長符号に比べて空間を 20% 以上節約できていることになります。

可変長符号を使ううえで難しいことのひとつに、0 と 1 の列を読んでいく際に、いつ記号の終わりに達したかをどうやって知るかということがあります。モールス符号では、各文字に対応する短点と長点の列の後に、特別な *separator code*(分離符号)(この場合は休止) を置くことで、この問題を解決しています。ほかの解決法として、どの記号についても、その符号全体がほかの記号に対する符号の前の部分 (*prefix(接頭辞)*) になることがないように符号を設計するというものがあります。そのような符号は *prefix code* (接頭符号) と呼ばれます。上の例では、A を 0、B を 100 として符号化しているので、ほかのどの記号も、0 や 100 で始まる符号を持つことはできません。

一般的に、符号化するメッセージの相対頻度を利用した可変長接頭符号を使えば、かなりの節約をすることができます。これを行う戦略のひとつにハフマン符号化法というものがあります。これは発見者の David Huffman にちなんでつけられた名前です。ハフマン符号は二分木として表現でき、葉は符号化対象の記号となります。木の中で、葉でないノードは、そのノード以下の葉に対応する記号をすべて含んだ集合を持っています。それに加えて、葉の記号それぞれには重み(相対頻度)が割り当てられていて、葉でないノードはその下にある葉の重みをすべて合計した重みをそれぞれ持っています。重みは符号化・復号化の際には使いません。以下では、木を構築する際に重みをどう利用するのか見ていきます。

Figure 2.18 に、上に書いた A から H までの符号に対するハフマン木を示しています。葉の重みは、A が相対頻度 8、B が相対頻度 3、ほかの文字は相対頻度 1 で現れるようなメッセージに対してこの木が設計されているということを示しています。

ハフマン木があれば、どの記号に対する符号も、根から始めてその記号を持つ葉にたどり着くまで下りていくことで知ることができます。左の枝を下りるたびに符号に 0 を追加し、右の枝を下りるたびに 1 を追加します(どちらの枝を下りるかについては、どちらの枝にその記号の葉があるか、またはその記号を集合に含むかを見て決めます)。例えば、Figure 2.18 の根から始めて D の葉に着くまでには、右枝、左枝、右枝、右枝とたどっていきます。そのため、D に対する符号は 1011 となります。

ハフマン木を使ってビット列を復号するには、根から始めて、左と右のどちらの枝を下りるのかをビット列の 0 と 1 の列を使って決めます。葉に着くたびに、メッセージ中の記号を新たにひとつ生成し、そこでまた根に戻って次の

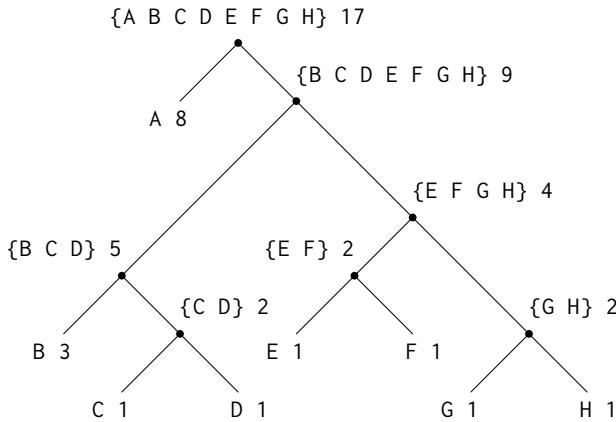


Figure 2.18: ハフマン符号化木

記号を探します。例えば、上記の木と 10001010 というビット列があるとします。根から始めて、右の枝を下り (ビット列の最初のビットは 1 ので)、それから左の枝を下り (二番目のビットは 0 ので)、次に左の枝を下ります (三番目のビットも 0 ので)。こうすると B の葉にたどり着くので、復号したメッセージの最初の記号は B です。ここでまた根から始めて、ビット列中の次のビットは 0 ので、左に動きます。こうすると、A の葉に着きます。それからまた根から始めると、残りのビット列は 1010 ので、右、左、右、左と動いて C に着きます。つまり、メッセージ全体は BAC ということになります。

ハフマン木の生成

記号の“アルファベット”と相対頻度がわかっているとき、“最善の”符号を作るにはどうしたらいいでしょうか (言い換えると、メッセージを最小のビット数で符号化するのはどのような木でしょうか)。ハフマンはこれを行うアルゴリズムを与え、結果となる符号が実際に、符号構築時に使った頻度と記号の相対頻度が一致するようなメッセージを符号化するのに最適な可変長符号であるということを示しました。ここではハフマン符号の最適性の証明はせずに、

ハフマン木の構築の仕方のみを示します。⁴²

ハフマン木を生成するアルゴリズムはとても単純です。考え方としては、出現頻度が最低の記号が根から一番遠いところに出てくるように木を配置するというものです。まず、符号構築対象の初期データによって決まる、記号と頻度を持つ葉ノードの集合から始めます。ここで、重みが小さいほうから二つの葉を選び、二つをくっつけて新しいノードを作り、新しいノードの左と右の枝がその二つのノードになるようにします。新しいノードの重みは二つの重みの合計とします。元の集合からその二つの葉を取り除き、代わりに新しいノードを入れます。それからこのプロセスを続けていきます。各段階で、重みが小さいほうから二つのノードを集合から取り除き、それらを左と右の枝として持つノードを代わりに入れることによって、その二つのノードをくっつけていきます。ノードがひとつだけになったら、それが木全体の根となり、プロセスは停止します。以下に、Figure 2.18のハフマン木が生成される過程を示します。

```
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になるとは限りません。各ステップで、最小の重みとなるノードのペアが一意であるとは限らないからです。また、二つのノードをくっつける順番をどうするか(つまり、どちらを右枝としてどちらを左枝とするか)も任意です。

ハフマン木の表現

以下の練習では、ハフマン木を使ってメッセージの暗号化・復号化を行ったり、上で概要を示したアルゴリズムに従ってハフマン木を作ったりするシステムを扱います。まず、木をどのように表現するかを検討しましょう。

⁴²ハフマン符号の数学的性質についての議論はHamming 1980を参照。

木の葉は、`leaf` という記号、その葉の記号、重みからなるリストによって表現します。

```
(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))
```

一般の木は、左枝、右枝、記号の集合、重みからなります。記号の集合は、単純に記号のリストにして、より複雑な集合の表現は使わないことにします。二つのノードをくっつけて木を作るとき、その木の重みは二つのノードの重みの合計となり、記号の集合は、二つのノードが持つ記号の集合の和集合となります。記号の集合はリストとして表されているため、和集合はSection 2.2.1で定義した `append` 手手続きによって作ることができます。

```
(define (make-code-tree left right)
  (list left
        right
        (append (symbols left)
                (symbols right)))
        (+ (weight left)
           (weight right))))
```

木をこうやって作ると、セレクタは次のようにになります。

```
(define (left-branch tree) (car tree))
(define (right-branch tree) (cadr tree))

(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))
(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (caddr tree)))
```

手続き `symbols` と `weight` は、渡されたものが葉である場合と一般の木である場合とで、少し違うことをしなければいけません。これらは、*generic procedures* (ジェネリック手続き)(二種類以上のデータを扱う手続き) の簡単な例になっています。これについては、Section 2.4 と Section 2.5 で詳しく説明します。

復号化手続き

以下の手続きは、復号化アルゴリズムを実装したものです。0 と 1 のリストに加え、ハフマン木を引数に取ります。

```
(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))))
```

手続き `decode-1` は、残りのビットのリストと、木の中での現在位置という、二つの引数を取ります。リストの次のビットが 0 か 1 かによって、左か右の枝を“下りて”いきます(これは手続き `choose-branch` で行っています)。葉にたどり着くと、木の根から始めてメッセージの残りを復号した結果に葉の記号を `cons` することで、メッセージの次の記号として返します。`choose-branch` の最終節のエラーチェックに注意してください。ここでは、入力データの中の 0 または 1 以外のものを手続きが見つけた場合にエラーを出します。

重みつき要素の集合

この木の表現では、葉でないノードはそれぞれ記号の集合を持つようになっていて、その集合は単純なリストとして表現してきました。しかし、上で検討した木の生成アルゴリズムでは、葉と木の集合も扱って、小さいほうから二つの項目を順にくっつけていくという必要がありました。集合の最小項目を何度も求めることになるので、この種の集合は順序つきの表現を使うほうが便利です。

葉と木の集合は、重みの昇順に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 `adjoin-set` 手続きは集合を構築するためのものですが、これは [Exercise 2.61](#) で説明したものと似ています。しかし、項目の比較は重みによって行い、また追加する要素がすでに集合にあるということはありません。

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

次の手続きは、((A 4) (B 2) (C 1) (D 1)) のような記号と頻度のペアのリストを取り、初期状態での葉の順序つき集合を構築し、ハフマンアルゴリズムに従ってすぐ連結できるようにします。

```
(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メッセージを次のように定義する。

```
(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` 手続きを使ってメッセージを復号し、結果を示せ。

Exercise 2.68: `encode` 手手続きは、引数としてメッセージと木を取り、メッセージを符号化したビットのリストを返す。

```

(define (encode message tree)
  (if (null? message)
      '()
      (append (encode-symbol (car message) tree)
              (encode (cdr message) tree))))

```

`encode-symbol` は、与えられた木に従って与えられた記号を符号化したビット列を返す。この手続きを書け。記号が木に含まれていなければエラーを出すように `encode-symbol` を設計せよ。[Exercise 2.67](#)で得た結果をサンプルの木を使って符号化し、それが元のサンプルメッセージと同じになるかどうか確認して、書いた手続きをテストせよ。

Exercise 2.69: 以下の手続きは、引数として記号・頻度ペアのリスト(同じ記号が二つ以上のペアに出てくることはない)を取り、ハフマンアルゴリズムに従って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` は上で与えた手続きで、ペアのリストを葉の順序つき集合に変換する。`successive-merge` は、集合の中で重みが最小の要素を `make-code-tree` を使って順番にくっつけていき、最後に要素がひとつだけ残るようにするというものである。その要素が求めるハフマン木となる。この手続きを書け(この手続きはちょっとトリッキーだが、そこまで複雑ではない。もし手続きの設計が複雑になったとしたら、ほぼ確実に何かを間違えている。順序つきの集合表現を使っているということが大きな助けになる)。

Exercise 2.70: 以下に示す 8 記号の相対頻度つきアルファベットは、1950 年代のロックの歌詞を効率よく符号化できるよう設計さ

れたものである (“アルファベット” の “記号” が単独の文字であるとは限らないことに注意)。

```
A    2    GET 2    SHA 3    WAH 1  
BOOM 1    JOB 2    NA 16    YIP 9
```

`generate-huffman-tree`(Exercise 2.69) を使って対応するハフマン木を生成し、`encode` (Exercise 2.68) を使って次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必要だろうか。もし固定長符号をこの 8 記号アルファベットに使ったとしたら、最低でどれだけのビット数が必要になるだろうか。

Exercise 2.71: n 記号のアルファベットに対するハフマン木があり、記号の相対頻度は $1, 2, 4, \dots, 2^{n-1}$ であるとする。 $n = 5$ 、 $n = 10$ の場合の木をスケッチせよ。そのような木では、(一般の n について) 最も頻度の高い記号を符号化するのに何ビット必要になるだろうか。最も頻度の低い記号はどうだろうか。

Exercise 2.72: Exercise 2.68で設計した符号化手続きについて考える。ひとつの記号を符号化するのに必要なステップ数の増加オーダーはどのようになるだろうか。各ノードに着くたびに記号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忘れないように。この問題の一般的の場合について答えることは難しい。ここでは、 n 記号の相対頻度が Exercise 2.71 のようになっている特別な場合について考えよう。アルファベット中で頻度が最大の記号と最小の記号を符号化するのにかかるステップ数の増加オーダーを (n の関数として) 答えよ。

2.4 抽象データの多重表現

ここまで、データ抽象化というものを紹介してきました。これはシステムを構築するための方法論で、プログラムの操作対象となるデータオブジェクトに対してどのような実装を選ぶかということと独立に、プログラムのほとんどの部分を記述できるようにするというものでした。例えばSection 2.1.1では、有理数を使うプログラムを設計するというタスクを、複合データを構築するためにコンピュータ言語が持っている基本メカニズムによって有理数を実装するというタスクから切り離すやり方について見てきました。ポイントとなる考え方は、抽象化の壁—この場合は、有理数のセレクタとコンストラクタ (`make-rat`, `numer`, `denom`)—を建てて、有理数をどう使うかというところと、有理数のリスト構造による表現という内部の構造とを分離するということでした。結果として、プログラムはFigure 2.1に示したような構造になりました。

これらのデータ抽象化の壁は、複雑性をコントロールする強力な道具です。データオブジェクトの内部表現を分離することで、大きなプログラムの設計というタスクを、別々に行うことのできる小さなタスクの集まりに分割することができます。しかし、この手のデータ抽象化はまだ十分に強力とは言えません。というのは、データ構造の“内部表現”というものがいつも意味があるとは限らないからです。

一つ目には、データオブジェクトの便利な表現方法が二つ以上あって、複数の表現を扱えるようなシステムを設計したいという場合があります。単純な例について考えると、複素数は二種類のほぼ等価な方法、直交形式（実部と虚部）と極形式（絶対値と偏角）によって表すことができます。直交形式のほうが適切な場合もあれば、極形式のほうが適切な場合もあります。実際に、複素数が両方の形式で表され、複素数を操作する手続きはどちらか一方の表現を扱うといったシステムも、十分に考えられます。

さらに重要なことは、プログラミングシステムは仕様変更を繰り返しながら長期間にわたって多くの人々によって設計されるということがよくあるということです。そのような環境では、データ表現として何を選ぶかについてみんなが前もって合意するということはまずありません。つまり、データの表現をそれを使うところから分離するデータ抽象化の壁に加えて、複数のデータ表現をそれぞれ分離し、ひとつのプラットフォーム上でそれらを共存させる抽象化の壁が必要だということです。また、大きなプログラムというものは、別々に設計されたすでに存在するモジュールを組み合わせて作られることが多いので、プログラマがモジュールを *additively* (加法的に)、つまり、これらのモジュールを設計し直したり実装し直したりすることなく、より大きなシステムに組

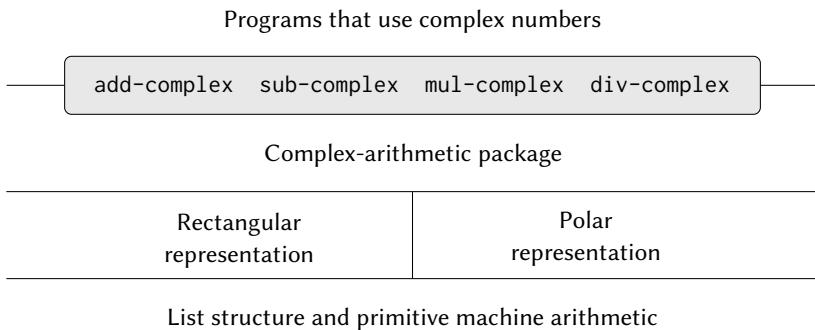


Figure 2.19: 複素数システムのデータ抽象化の壁

み込んでいくための規約が必要になります。

この節では、プログラムの部品によって違う方法で表現されるデータを扱うやり方について学びます。そのためには、*generic procedures*(ジェネリック手続き)——二種類以上の方法で表現されるデータを扱える手続き——を構築する必要があります。ジェネリック手続きを構築するためにここで使う主なテクニックは、*type tags*(タイプタグ)を持つデータオブジェクト、つまりどのように処理されるべきかという情報を明示的に持っているデータオブジェクトを使うということです。また、*data-directed*(データ主導)プログラミングについても検討します。これは、ジェネリック演算によって加法的にシステムを組み立てていくための強力で便利な実装戦略です。

最初に、単純な複素数の例を扱います。タイプタグとデータ主導スタイルを使うことで、“複素数”データオブジェクトという抽象概念を維持したまま、複素数の表現として直交形式と極形式を別々に設計できるということを見ていきます。これは、複素数がどのように表現されているかということと独立に複素数の各部分にアクセスするジェネリックセレクタによって複素数の算術演算(`add-complex`, `sub-complex`, `mul-complex`, `div-complex`)を定義することで達成できます。これによってできる複素数システムでは、Figure 2.19に示すように、二種類の異なる抽象化の壁を持つことになります。“水平な”抽象化の壁は、Figure 2.1に出てきたものと同じ役割を果たします。それらは、“高レベル”演算を“低レベル”演算から分離するものです。それに加えて、複数の表現を別々に設計して組み込めるようにするための“垂直な”壁があります。

Section 2.5では、タイプタグとデータ主導スタイルを使ってジェネリック算術演算パッケージを開発するやり方を示します。このシステムは、すべての種類の“数値”を操作するのに使える手続き (`add`, `mul` など) を提供し、新たな種類の数値が必要になったら簡単に拡張できるようになっています。Section 2.5.3では、記号代数を実行するシステムの中でジェネリック数値演算を使う方法を示します。

2.4.1 複素数の表現

ここでは、ジェネリック演算を使うプログラムの単純で非現実的な例として、複素数に対する算術演算を行うシステムを開発していきます。まず、複素数を順序つきペアとして表すときに考えられる二つの表現方法、直交形式(実部と虚部)と極形式(絶対値と偏角)について検討します。⁴³ Section 2.4.2では、タイプタグとジェネリック演算を使うことによって両方の表現を共存させるやり方を示します。

有理数と同じように、複素数は順序つきペアとして自然に表現できます。複素数の集合は、“実”軸と“虚”軸という二つの直交する軸を持つ二次元空間を考えることができます (Figure 2.20参照)。この見方をすると、 $z = x + iy$ ($i^2 = -1$) という複素数は、の実座標が x で虚座標が y の平面上の点として考えることができます。複素数の和は、この表現では、座標の和になります。

$$\begin{aligned}\text{Real-part}(z_1 + z_2) &= \text{Real-part}(z_1) + \text{Real-part}(z_2), \\ \text{Imaginary-part}(z_1 + z_2) &= \text{Imaginary-part}(z_1) + \text{Imaginary-part}(z_2).\end{aligned}$$

複素数をかけ算するときには、複素数を極形式で、つまり大きさと角度 (Figure 2.20内の r と A) を使って考えるほうが自然です。二つの複素数の積は、ひとつの複素数をもう一方の長さをかけて伸縮させ、もう一方の角度だけ回転させて得られるベクトルになります。

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

⁴³ 実際の計算システムでは、ほとんどの場合、直行形式が選ばれます。これは、直交形式と極形式の変換時に発生する丸め誤差のためです。このため、この複素数の例は現実的ではありません。しかし、ジェネリック演算を使ったシステムの設計を説明する例としては明快ですし、この章の後のほうで開発する、より実質的なシステムへの導入としても役に立ちます。

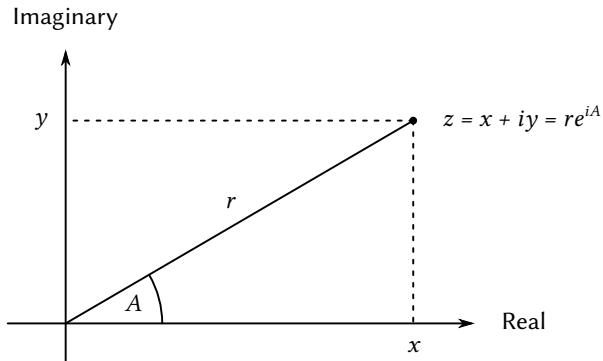


Figure 2.20: 平面上の点としての複素数

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

つまり、複素数には二つの異なる表現があり、それぞれ異なる演算に適しているということです。しかし、複素数を使うプログラムを書いている人の視点からは、データ抽象化の原則により、コンピュータがどちらの表現を使っていても、複素数の演算すべてが使えるようになっていなければいけません。例えば、直交形式によって指定された複素数の絶対値が求められると便利なことがあります。同じように、極形式によって指定された複素数の実部を求められると便利なことがあります。

そのようなシステムを設計するにあたっては、Section 2.1.1で有理数パッケージを設計する際に使ったのと同じデータ抽象化戦略が使えます。複素数演算は、`real-part`, `imag-part`, `magnitude`, `angle`という四つのセレクタによって実装されているとします。また、複素数の構築には二つの手続きがあるとします。`make-from-real-imag`は、指定された実部と虚部を持つ複素数を返し、`make-from-mag-ang`は、指定された絶対値と偏角を持つ複素数を返します。これらの手続きは、任意の複素数に対して

```
(make-from-real-imag (real-part z) (imag-part z))  
と  
(make-from-mag-ang (magnitude z) (angle z))
```

の両方とも z に等しい複素数を返すという性質を持っています。

これらのコンストラクタとセレクタを使うことで、Section 2.1.1で有理数について行ったように、コンストラクタとセレクタによって規定される“抽象データ”を使って複素数の算術演算を実装することができます。上の式に示したように、足し算と引き算は実部と虚部を使って、またかけ算と割り算は絶対値と偏角を使って行えます。

```
(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成させるには、複素数の表現を選び、コンストラクタとセレクタを基本数値と基本リストを使って実装しなければいけません。すぐわかるように、この方法は二通りあります。“直交形式”的ペア(実部、虚部)として表現するか、“極形式”的ペア(絶対値、偏角)として表現するかです。どちらを選ぶべきでしょうか。

この二つの選択の具体的な例として、Ben Bitdiddle と Alyssa P. Hacker という、独立に複素数システムの表現を設計している二人のプログラマがいると想像してください。Ben は複素数を直交形式で表現することにします。この場合、複素数の実部と虚部を選択することや、複素数を与えられた実部と虚部から構築することは、素直に実現できます。絶対値と偏角を求めたり、複素数を与えられた絶対値と偏角から構築するのは、次のような三角法の関係を使います。

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

これは、実部と虚部(x, y)と絶対値と偏角(r, A)の関係を表しています。⁴⁴ このため、Ben の表現は以下のようなセレクタとコンストラクタによって書くことができます。

```
(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 は複素数を極形式で表現することにしました。彼女の方では、絶対値と偏角のセレクトは素直にできます。しかし、実部と虚部を求めるには、三角法の関係を使う必要があります。Alyssa の表現は次のようにになります。

```
(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))
```

データ抽象化の規律に従っているため、`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動くということが保証されます。

⁴⁴ ここで出てきた逆正接関数は Scheme の `atan` 手続きで計算できるもので、引数 y, x を取り、正接が y/x となる角度を返すよう定義されています。引数の符号によって角度の象限が決まります。

2.4.2 タグつきデータ

データ抽象化は、“最小責任の原則”の応用として見ることもできます。Section 2.4.1で複素数システムを実装したとき、Ben の直行形式による表現と Alyssa の極形式による表現のどちらを使うこともできました。セレクタとコンストラクタが抽象化の壁を作っているため、データオブジェクトの具体的な表現に何を選ぶかということを最後の瞬間まで遅らせることができ、そのためシステム設計の柔軟性を最大限にできます。

最小責任の原則は、さらに極限まで進めることができます。もし望むなら、セレクタとコンストラクタを設計した“後”になっても表現を曖昧にします。Ben の表現と Alyssa の表現の“両方”を使うようにすることもできます。しかし、両方の実装が単独のシステムに含まれている場合、極形式のデータと直交形式のデータを区別するための何らかの方法が必要になります。そうしないと、例えば(3, 4)というペアの絶対値を求めるよう言われた場合に、答えを(数値を直行形式と解釈して)5とするべきか(数値を極形式と解釈して)3とするべきかわかりません。これを区別できるようにする素直なやり方は、*type tag*(タイプタグ)—rectangular または polar という記号—to、それぞれの複素数の一部として含めるというものです。こうすると、複素数を操作することが必要になったときに、どちらのセレクタを適用するべきかをこのタグを使って決めることができます。

タグつきデータを操作するために、*type-tag* と *contents* という手続きを持っていると想定します。*type-tag* はデータオブジェクトからタグを抽出するもので、*contents* は実際の中身(複素数の場合、極形式または直行形式の座標)を抽出するものです。また、*attach-tag* という手続きも持っていると仮定します。これは、タグと中身を取り、タグつきデータオブジェクトを返すというものです。これを実装する素直なやり方は、普通のリスト構造を使うというものです。

```
(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)))
```

これらの手続きを使って、直交形式と極形式をそれぞれ識別する述語 `rectangular?`, `polar?` を定義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))
```

タイプタグを使うことにしたので、Ben と Alyssa はコードを修正して、異なる表現が同じシステムの中で共存できるようにしました。Ben が複素数を構築するたびに、それに直交形式というタグをつけます。Alyssa が複素数を構築するたびに、それに極形式というタグをつけます。それに加えて、Ben と Alyssa はそれぞれの手続きの名前が衝突しないようにしなければいけません。これを行なう方法としては、Ben が自分の表現の手続きに接尾辞 `rectangular` を追加し、Alyssa は自分の表現手続きに `polar` を追加するというものがあります。以下は、Section 2.4.1 の Ben の直行形式表現を修正したものです。

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

Alyssa の極形式表現の修正版のは次のようになりました。

```
(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)))

```

どちらのジェネリックセレクタも、引数のタグをチェックして、そのタイプのデータを扱う適切な手続きを呼ぶ手続きとして実装されています。例えば、複素数の実部を得る場合、`real-part` はタグを調べ、Ben の `real-part-rectangular` と 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` を使って生のタグなしデータを取り出し、それを必要に応じて直交形式または極形式の手続きに渡します。

```

(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))))
```

複素数の算術演算を実装するのには、Section 2.4.1の手続き `add-complex`, `sub-complex`, `mul-complex`, `div-complex` と同じものが使えます。それらが呼び出すセレクタはジェネリックなので、どちらの表現を使っても動くからです。例えば、手続き `add-complex` は相変わらずそのままで、以下のようになります。

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

最後に、複素数を構築するのに Ben の表現と Alyssa の表現のどちらを使うかを決める必要があります。実部と虚部が渡されたときには直行形式を使い、大きさと角度を渡されたときには極形式を使うというのが妥当でしょう。

```
(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))
```

このようにして作った複素数システムは、Figure 2.21に示すような構造を持ちます。システムは、比較的独立した三つの部分に分解されています。複素数算術演算、Alyssa の極形式による実装、Ben の直交形式による実装です。極形式と直交形式の実装は、Ben と Alyssa が別々に書いていたとしても問題ありません。第三者のプログラマが抽象コンストラクタ、セレクタによって複素数算術演算手続きを実装する際に、内部表現として両方とも使うことができます。

それぞれのデータオブジェクトには型のタグがついているので、セレクタはジェネリックにデータを操作します。つまり、それぞれのセレクタは適用されるデータの個々の型によって異なるふるまいをするように定義されているということです。別々の表現をインターフェイスで接続する際の一般的なメカニズムに注目してください。ある表現の実装(例えば、Alyssa の極形式パッケージ)の中で極形式型の複素数を操作するときには、タグを取り除き、中身を Alyssa のコードに渡します。逆に、Alyssa が一般使用のための数値を構築する際には、それに型のタグをつけて、上のレベルの手続きが適切に認識できるようにします。データオブジェクトをレベル間でやりとりする際にタグをつけ

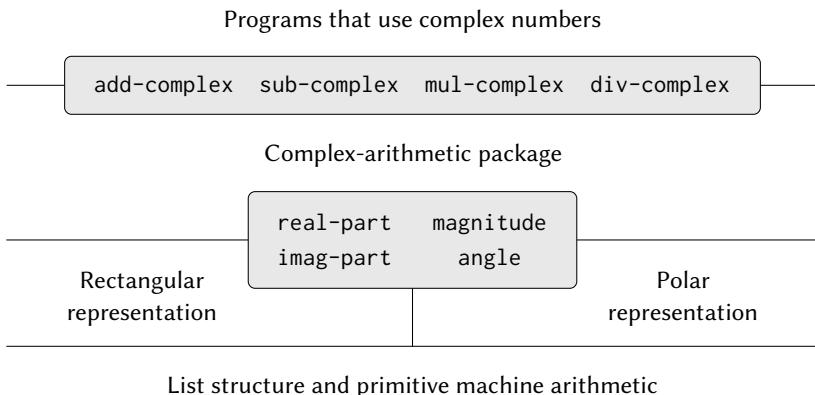


Figure 2.21: ジェネリックな複素数算術演算システムの構造

たり外したりするというこの規律は、重要な組織化戦略になります。Section 2.5では、このことについて見てきます。

2.4.3 データ主導プログラミングと加法性

データの型をチェックして適切な手続きを呼ぶという一般的な戦略は、データの型をチェックし適切な手続きを呼ぶ一般的な戦略は *dispatching on type* (型によるディスパッチ) と呼ばれるものです。これは、システム設計でモジュール性を達成するための強力な戦略です。一方で、Section 2.4.2のようにディスパッチを実装することには二つの大きな弱点があります。弱点のひとつは、ジェネリックインターフェイス手続き (real-part, imag-part, magnitude, angle) がすべての異なる表現について知っていなければいけないということです。例えば、複素数の新しい表現をこの複素数システムに組み込みたいとします。そのためには、この新しい表現を型で識別できるようにして、それから各ジェネリックインターフェイス手続きにその新しい型をチェックする節を追加し、そしてその表現に対する適切なセレクタを適用することが必要になるでしょう。

このテクニックのもうひとつの弱点は、個々の表現は別々に設計できるものの、システム全体で二つの手続きが同じ名前を持つということがないように

保証しなければならないということです。このために、Ben と Alyssa はSection 2.4.1の元の手続きの名前を変える必要があったのでした。

この二つの弱点の元となっているのは、ジェネリックインターフェイスを実装するテクニックが *additive*(加法的) でないということです。ジェネリックセレクタ手続きを実装する人は、新しい表現が組み込まれたびにそれらの手続きを変更しなければならず、また個々の表現のインターフェイスを作る人は、名前の衝突を避けるためにコードを変更しなければなりません。どちらの場合でも、コードに加えるべき変更は単純な物ですが、それでもやはり行わなければならず、このことが不便さとエラーの原因になります。これは複素数システムに関しては現時点ではあまり大きな問題ではありませんが、仮に複素数の表現が二つではなく数百個あったとしたらどうでしょうか。また、抽象データインターフェイスにはメンテナンスするべきジェネリックセレクタが数多くあるとしたらどうでしょうか。また、すべてのインターフェイス手続きをすべての表現について知っているプログラマが誰もいないとしたらどうでしょうか。この問題は現実的なもので、大規模データベース管理システムのようなプログラムではこの問題に取り組む必要があります。

ここで、システム設計をさらにモジュール化するための手段が必要になります。*data-directed programming*(データ主導プログラミング) として知られるプログラミングテクニックはその手段を提供してくれます。データ主導プログラミングがどうやって動くのかを理解するために、まず次のこととに注目します。いろいろな型の集合に対して共通のものとなるジェネリック演算を扱う際にはいつも、実質的には、ひとつの軸が可能な演算で、もうひとつの軸が可能な型であるような二次元のテーブルを扱っていることになります。テーブルの項目は、それぞれの引数の型に対するそれぞれの演算を実装する手続きです。前の節で開発した複素数システムでは、演算の名前、データタイプ、実際の手続きの間の対応関係は、ジェネリックインターフェイス手続きのいろいろな条件節の中に広がっていました。しかし、それと同じ情報をFigure 2.22に示すようにテーブルの中に構造化することもできたところです。

データ主導プログラミングは、そのようなテーブルを直接扱うようにプログラムを設計するテクニックです。前のほうで、複素数算術演算のコードと、それぞれ型によって明示的なディスピッチを行う手続きの集合となっている二つの表現パッケージとの間の橋渡しをするメカニズムを実装しました。ここでは、そのインターフェイス部分を、演算の名前と引数の型の組み合わせをテーブルから検索して適用るべき正しい手続きを見つけ、それを引数の中身に適用するということを行なう、单一の手続きとして実装します。こうすると、新しい表現パッケージをシステムに追加する際に、既存の手続きには何の変更も加

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: 複素数システム演算のテーブル

えなくても大丈夫です。テーブルに新しい項目を追加するだけで十分です。

この計画を実装するために、演算-型テーブルを操作する `put` と `get` という二つの手続きがあるということにします。

- `(put <op> <type> <item>)` は、テーブルの `<op>` と `<type>` が指すところに `<item>` を入れる。
- `(get <op> <type>)` は、テーブルから `<op>`, `<type>` の項目を検索し、そこで見つかった項目を返す。見つからなければ、`get` は `false` を返す。

今のところは、`put` と `get` は言語に含まれているということにします。Chapter 3 (Section 3.3.3) では、これらを含め、テーブル操作演算の実装方法について学びます。

それでは、複素数システムでデータ主導プログラミングをどう使うかについて見てていきましょう。直行形式表現を開発した Ben は、元のものと同じようにコードを実装します。それから手続きの集合、つまり `package`(パッケージ) を定義し、システムに直交形式の数値の扱い方を知らせるテーブルに項目を追加することで、これらの手続きをシステムのほかの部分と接続します。これは、次の手続きを呼ぶことで行えます。

```
(define (install-rectangular-package)
  ;; 内部手続き
  (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)))))

;; システムのほかの部分とのインターフェイス
(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)

```

ここでの内部手続きが、Section 2.4.1で Ben が単独で書いたものと同じ手続きであることに注目してください。これらをシステムのほかの部分と接続するのに、変更を加える必要はありません。それに、これらの手続きの定義は組み込み手続きの内部にあるので、直交形式パッケージの外のほかの手続きと名前が衝突する心配もありません。これらの手続きをシステムのほかの部分と接続するために、Ben は自分の `real-part` 手続きを、演算名 `real-part` と型 (`rectangular`) のところに入れます。ほかのセレクタも同じようにします。⁴⁵ このインターフェイスは、システムのほかの部分から使うためのコンストラクタも定義しています。⁴⁶ コンストラクタは Ben が内部で定義していたものと同じですが、タグをつけるというところが違います。

Alyassa の極形式パッケージも似たようなものになります。

```
(define (install-polar-package)
  ;; 内部手続き
```

⁴⁵ ここでは、`rectangular` という記号ではなく `(rectangular)` というリストを使い、必ずしも同じ型であるとは限らない複数の引数を持つ手続きを持てるようにしています。

⁴⁶ コンストラクタを入れる場所の型はリストにしなくとも大丈夫です。コンストラクタは必ずある特定の型のオブジェクトを作るのに使われるからです。

```

(define (magnitude z) (car z))
(define (angle z) (cdr z))
(define (make-from-mag-ang r a) (cons r a))
(define (real-part z)
  (* (magnitude z) (cos (angle z))))
(define (imag-part z)
  (* (magnitude z) (sin (angle z))))
(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))
;; システムのほかの部分とのインターフェイス
(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 は、相手のものと同じ名前 (例えば、`real-part` など) によって定義された元々の手続きをまだ使っていますが、これらの定義は今では別々の手続きの内部定義 (Section 1.1.8 参照) になっているので、名前の衝突は起りません。

複素数算術演算セレクタは、`apply-generic` というジェネリック “演算” 手続きを使ってテーブルにアクセスします。これは、ジェネリック演算を引数に適用するものです。`apply-generic` は、テーブルから演算の名前と引数の型に対応する箇所を検索し、手続きがあればそれを適用します。⁴⁷

⁴⁷ `apply-generic` は、Exercise 2.20で説明したドット末尾記法を使います。それぞれのジェネリック演算が取る引数の数は違うかもしれないからです。`apply-generic` では、`op` は `apply-generic` の第一引数を値として持ち、`args` は残りの引数のリストを値として持ちます。

また、`apply-generic` は `apply` という基本手続きを使っています。この手続きは手続きとリストという二つの引数を取り、リストの要素を引数として手続きを適用します。例えば、

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "No method for these types: APPLY-GENERIC"
            (list op type-tags))))))
```

`apply-generic` を使うと、複素数演算のジェネリックセレクタは以下のように定義できます。

```
(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))
```

新しい表現がシステムに追加されてもこれらはまったく変わらないということに注意してください。

また、このパッケージの外部のプログラムが、実部と虚部や絶対値と偏角から複素数を作る際に使うコンストラクタを抽出することもできます。Section 2.4.2と同じように、実部と虚部がある場合は直交形式の複素数を構築し、絶対値と偏角がある場合は極形式の複素数を構築します。

```
(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では、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))

(apply + (list 1 2 3 4)))
```

は 10 を返します。

```

((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)))))


```

このプログラムは、微分する式の型によってディスパッチを実行していると捉えることができる。この場合、データの“タイプタグ”は代数演算記号 (+ など) で、行う演算は `deriv` ということになる。基本的な微分を行う手続きを次のように書き直すと、プログラムをデータ主導スタイルに変形できる。

```

(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp))
                (operands exp) var))))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))


```

- a 上で何をしているか説明せよ。手続き `number?` と `variable?` は、なぜデータ主導ディスパッチとして取り込むことができないのだろうか。
- b 和と積に対する微分手続きと、上記のプログラムで使っているテーブルにそれらを組み込む補助コードを書け。
- c 任意の微分規則 (例えば、Exercise 2.56の指數の微分など) を選び、それをこのデータ主導システムに組み込め。

- d この単純な代数操作では、式の型は式をまとめる代数演算子となっている。しかし、仮に手続きのインデックスを逆にして、`deriv` でディスパッチを行う箇所を次のようにする。

```
((get (operator exp) 'deriv) (operands exp) var)
```

これに対応して、微分システムにはどのような変更を加える必要があるだろうか。

Exercise 2.74: Insatiable Enterprises, Inc.(貪欲エンタープライズ社)は、世界中に散らばった多数の独立事業所からなる分散型複合企業である。この会社のコンピュータ設備は、どのユーザから見てもネットワーク全体がひとつのコンピュータのように見えるような巧妙なネットワークインターフェイス方式によって相互接続されたところだ。貪欲社の社長は、初めてネットワークの機能を使って事業所ファイルから管理情報を取り出そうとして愕然とした。事務所ファイルはすべて Scheme のデータ構造として実装されているのに、使われている個々のデータ構造は事業所ごとに違うのだ。事業所長会議が急遽開かれ、事業所の独立性をこれまで通りに保ったままで、本部の要求を満たせるようにファイルを統合する戦略を探ることになった。

そのような戦略をデータ主導プログラミングによって実装する方法を示せ。例として、各事業所の人事記録は単独のファイルからなり、従業員の名前をキーとしたレコードの集合を持っているとする。集合の構造は事業所ごとに異なる。さらに、各従業員のレコードはそれ自身が(事業所ごとに異なる構造を持つ)集合で、`address` と `salary` のような識別子をキーとした情報を含んでいるとする。具体的には、

- 指定した人事ファイルから指定した従業員のレコードを取得する `get-record` 手続きを本部向けに実装せよ。この手続きは、任意の事業所のファイルに適用できる必要がある。個々の事業所のファイルはどのように構造化しなければならないか説明せよ。具体的には、どのような型情報を提供する必要があるだろうか。
- 任意の事業所の人事ファイル内の与えられた職員のレコードから給与情報を返す `get-salary` 手続きを本部向けに実装せ

- よ。この演算が動くようにするには、レコードはどのように構造化しなければならないだろうか。
- c 本部向けに、`find-employee-record` 手続きを実装せよ。この手続きは、全事業所のファイルから与えられた従業員を検索し、該当レコードを返す。引数として、従業員名と全事業所のファイルのリストを取るとせよ。
 - d 貪欲社が新しい会社を吸収した場合、新しい人事情報を中央システムに組み入れるためにはどのような変更が必要になるだろうか。

メッセージパッキング

データ主導プログラミングでポイントとなる考え方は、プログラム中のジェネリック演算を Figure 2.22 のような演算-型テーブルを明示的に扱うということです。Section 2.4.2 で使ったプログラミングスタイルでは、それぞれの演算に自分のディスパッチの面倒を見させることによって、必要となる型ディスパッチを構築していました。実質的には、それぞれのジェネリック演算手続きがテーブルの行を表すというやり方で、演算-型テーブルを行に分解していましたことになります。

もうひとつの実装戦略は、テーブルを列に分解し、データ型によってディスパッチを行う“賢い演算”を使うのではなく、演算の名前によってディスパッチを行う“賢いデータオブジェクト”を使うというものです。これを実現するには、データオブジェクト（例えば直交形式の複素数）を手続きとして表現し、その手続きは実行するべき演算名を引数として取り、指定された演算を実行するというようにします。このような規律に従うと、`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)
```

`apply-generic` 手続きはジェネリック演算を引数に適用するものでしたが、このデータに対応する `apply-generic` 手続きは、単に演算の名前をデータオブジェクトに渡して、オブジェクトに仕事をさせるだけになります。⁴⁸

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

`make-from-real-imag` の返す値が手続き—内部手続き `dispatch`—であることに注意してください。これは、`apply-generic` が演算の実行を要求したときに呼び出される手続きです。

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) と呼ばれています。この名前は、データオブジェクトは要求された演算の名前を“メッセージ”として受け取る実体であるというイメージから来ています。メッセージパッシングの例については、Section 2.1.3 ですでに見ています。そこでは、`cons`, `car`, `cdr` がデータオブジェクトを使わずに手続きだけを使って定義できるということを見てきました。ここでは、メッセージパッシングがただの数学的トリックではなく、ジェネリック演算によってシステムを組み立てるうえでの便利な技術だということを示しました。この章の残りでは、ジェネリックな算術演算について検討するために、メッセージパッシングではなくデータ主導プログラミングを使い続けることにします。Chapter 3 ではメッセージパッシングに戻り、それがシミュレーションプログラムを構築するうえでの強力なツールになるということを見ていきます。

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で実装せよ。この手続きは、上で与えた `make-from-real-imag` と似たようなものになるだろう。

Exercise 2.76: ジェネリック演算を使った大きなシステムが発展するにつれ、新しいデータオブジェクトの型や新しい演算が必要になることがある。三つの戦略 (明示的ディスパッチによるジェネリック演算、データ主導スタイル、メッセージパッシングスタイル) それぞれについて、新しい型や新しい演算を追加するために必要な変更を記述せよ。新しい型がよく追加されるシステムでは、どの組み立て方が最も適しているだろうか。新しい演算を追加するシステムでは、どれが最も適しているだろうか。

⁴⁸ この仕組みの制約のひとつは、一引数のジェネリック手続きしか使えないということです。

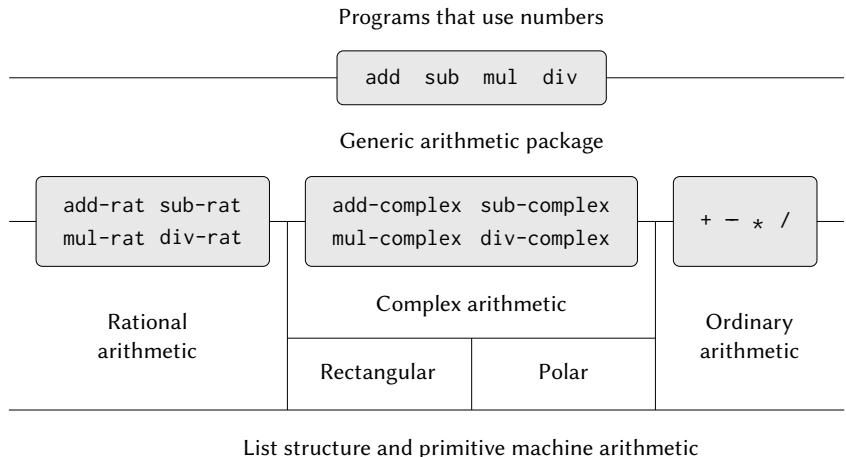


Figure 2.23: ジェネリック算術演算システム

2.5 ジェネリック演算によるシステム

前の節では、データオブジェクトの表現が二通り以上あるシステムの設計方法について見てきました。ポイントとなる考え方は、ジェネリックインターフェイスという手段によって、データ演算を指定するコードをいくつかの表現に結びつけるというものでした。ここでは、同じ考え方を使って、いろいろな表現に対してジェネリックな演算を定義するだけでなく、いろいろな種類の引数に対してジェネリックな演算を定義するやり方について見ていきます。ここまで、いくつもの算術演算パッケージに触れてきました。言語組み込みの基本算術演算 (+, -, *, /)、Section 2.1.1 の有理数演算 (`add-rat`, `sub-rat`, `mul-rat`, `div-rat`)、それに Section 2.4.3 で実装した複素数演算があります。今度はデータ主導テクニックを使い、これまでに構築してきたすべての算術演算パッケージを合併した算術演算パッケージを構築します。

Figure 2.23 は、これから構築するシステムの構造を示しています。抽象化の壁に注目してください。“数値”を使う人の視点からは、どんな種類の数が与えられても演算を行う `add` というたったひとつの手続きだけが見えています。`add` はジェネリックインターフェイスの一部で、別々になっている通常の

算術演算、有理数の算術演算、複素数の算術演算のパッケージに、数値を使うプログラムから統一的なアクセスができるようになります。個々の算術演算パッケージ(例えば複素数パッケージ)は、それ自身、異なる表現(例えば直交形式と極形式)のために設計された複数のパッケージを連結するジェネリック手続き(例えば `add-complex`)によってアクセスするようになっていることもあります。また、システムの構造は加法的なので、それぞれの算術演算パッケージは別々に設計して組み合わせてジェネリック算術演算システムを作れるようになっています。

2.5.1 ジェネリック算術演算

ジェネリック算術演算を設計するというタスクは、ジェネリック複素数演算を設計するのと似たようなものになります。例えば、通常の数値に対しては基本手続きの足し算 `+` のようにふるまい、有理数に対しては `add-rat` のようにふるまい、複素数に対しては `add-complex` のようにふるまうジェネリックな足し算手続き `add` がほしいところです。`add` その他のジェネリック算術演算は、[Section 2.4.3](#)で複素数のジェネリックなセレクタの実装に使ったのと同じ戦略によって実装できます。それぞれの種類の数値にタイプタグをくっつけ、ジェネリック手続きが引数のデータ型によって適切なパッケージにディスパッチを行なうようにします。

ジェネリック算術演算手続きは、次のように定義します。

```
(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))
```

まず、*ordinary*(通常の)数値、つまり言語の基本数値を扱うパッケージを組み込みます。通常の数値には `scheme-number` という記号のタグをつけることになります。このパッケージの算術演算は基本算術演算手続きです(つまり、タグなし数値を扱うのに新たに手続きを定義する必要はありません)。これらの演算はそれぞれ二つの引数を取るので、テーブルには (`scheme-number scheme-number`) というリストをキーとして組み込みます。

```
(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、次の手続きによって（タグつきの）通常の数値を作成します。

```

(define (make-scheme-number n)
  ((get 'make 'scheme-number) n))

```

ジェネリック算術演算システムのフレームワークがもうできているので、新しい種類の数値を加えることは簡単にできます。以下は、有理数算術演算を行うパッケージです。加法性のおかげで、Section 2.1.1 の有理数のコードを修正なしでパッケージの内部手続きとして使えるというところに注目してください。

```

(define (install-rational-package)
  ;; 内部手続き
  (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))))
;; システムのほかの部分とのインターフェイス
(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))

```

複素数を扱う同様のパッケージを、`complex` というタグを使って組み込みます。このパッケージを作る際に、直交形式と極形式のパッケージで定義した演算 `make-from-real-imag` と `make-from-mag-ang` をテーブルから取り出しています。加法性のおかげで、Section 2.4.1 の手続き `add-complex`, `sub-complex`, `mul-complex`, `div-complex` が内部手続きとして使えます。

```

(define (install-complex-package)
  ;; 直交形式パッケージと極形式パッケージからインポートした手続き
  (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))
  ;; 内部手続き
  (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))))
;; システムのほかの部分とのインターフェイス
(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部のプログラムは、複素数を構築するのに実部と虚部を使うこともできまし、絶対値と偏角を使うこともできます。元の手続きは、本来は直交形式パッケージと極形式パッケージの中で定義されたものですが、そこから複素数パッケージにエクスポートされ、さらにそこから外の世界へとエクスポートされているというところに注目してください。

```

(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))

```

ここでは、二つのレベルのタグシステムができています。 $3+4i$ のような典型的な複素数は、Figure 2.24に示すように表現されることになります。外側のタグ(complex)は、数値を複素数パッケージに送るために使われます。複素数パッケージに入ると、次のタグ数値を直行形式パッケージに送るのに(rectangular)

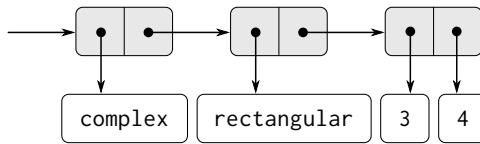


Figure 2.24: 直行形式による $3 + 4i$ の表現

タグが使われます。巨大で複雑なシステムでは、多くのレベルが、それぞれ次のレベルとジェネリック演算という手段によって接続されるという形で存在するということがあります。データオブジェクトが“下向きに”渡されるにつれ、適切なパッケージに送るために使われた外側のタグは (`contents` の適用によって) はがされ、(もしあれば) 次のレベルのタグが見えるようになります。さらなるディスパッチに使われます。

上記のパッケージでは、`add-rat`, `add-complex` その他の算術演算手続きは、最初に書いたときそのままの状態で使いました。しかし、これらの定義が異なるインストール手続きの内部に入ると、互いに識別できる名前である必要はなくなります。両方のパッケージで、単に `add`, `sub`, `mul`, `div` という名前をつけていたとしても大丈夫です。

Exercise 2.77: Louis Reasoner は、式 (`magnitude z`) を評価しようとした。ここで、`z` は Figure 2.24 に示したオブジェクトである。驚いたことに、5 という答えが返ってくるのではなく、`apply-generic` からのエラーメッセージが返ってきて、型 (`complex`) には `magnitude` という演算を行う手続きがないという。彼がこの対話を Alyssa P. Hacker に見せると、Alyssa は“複素数セレクタが `polar` と `rectangular` の数値にだけしか定義されていないくて、`complex` の数値に対して定義されていないのが問題なのよ。動くようにするには `complex` パッケージに以下の式を追加すれば大丈夫。”と言う。

```
(put 'real-part '(complex) real-part)
(put 'imag-part '(complex) imag-part)
(put 'magnitude '(complex) magnitude)
(put 'angle '(complex) angle)
```

なぜこれが動くのか、詳しく説明せよ。例として、`z` が Figure

2.24に示したオブジェクトであるとき、式 (`(magnitude z)`) を評価した際に呼ばれるすべての手続きをトレースせよ。具体的には、`apply-generic` は何回起動されるだろうか。それぞれの場合について、ディスパッチ先の手続きは何になるだろうか。

Exercise 2.78: `scheme-number` パッケージの内部手続きは、本質的に基本手続き `+`, `-` その他に対する呼び出しでしかない。ここでのタイプタグシステムでは、各データオブジェクトに型がくっついていないといけないようになっていたので、直接この言語の基本手続きを使うことはできなかった。しかし、実際のところ、Lisp の実装はすべて型システムを持っていて、内部で使用している。`symbol?` や `number?` のような基本述語は、データオブジェクトが特定の型を持つか識別している。Section 2.4.2 の `type-tag`, `contents`, `attach-tag` の定義を変更し、ここでのジェネリックシステムが Scheme の内部型システムを利用できるようにせよ。つまり、システムの挙動はそのままで、普通の数値を `car` が `scheme-number` という記号であるペアとして表すのではなく、単に Scheme の数値として表現されるようにせよ。

Exercise 2.79: 二つの数値の等価性をテストするジェネリックな等価性述語 `equ?` を定義し、ジェネリック算術演算パッケージに組み込め。この演算は、通常の数値、有理数、複素数に対して動作しなければならない。

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、ジェネリック算術演算パッケージに組み込め。この演算は、通常の数値、有理数、複素数に対して動作しなければならない。

2.5.2 異なる型のデータを組み合わせる

ここまで、統合算術演算システムの定義の仕方について見てきました。このシステムは、通常の数値、複素数、有理数を含み、またこのほかどんな数値のタイプでも作って組み込むことができるものでした。しかし、無視してきた重要な問題がひとつあります。ここまでで定義してきた演算は、異なるデータ型を完全に独立したものとして扱うようになっています。つまり、例えば二つの通常の数字を足すのと二つの複素数を足すのに別々のパッケージがあるということです。考えの対象外にしていたのは、型の境界を超えた演算、例えば複

素数と通常の数値との足し算のようなものについて考えることに意味があるという事実です。私たちは、プログラムの部品同士の間に壁を作り、独立して開発したり理解したりできるようにすることに多大な労力をつぎ込んできました。ここでは、注意深くコントロールされたやり方でクロスタイプ(異なる型同士)の演算を導入し、モジュール境界を大きく壊すことなくそれらの演算をサポートできるようにします。

クロスタイプ演算を扱う方法のひとつは、可能な型の組み合わせの中で、演算が有効であるものそれぞれに対して別々の手続きを設計するというものです。例えば、複素数パッケージを拡張し、複素数と実数の足し算を提供するようにして、(complex scheme-number) というタグを使ってテーブルに組み込むということができます。⁴⁹

```
;; 複素数パッケージに含める
(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いくのですが、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が、その型を扱う手続きのパッケージの構築だけでは終わらず、クロスタイプ演算を実装する手続きを構築して組み込む分まで必要になります。これはあつという間に、その型自身の演算を定義するのに必要な分よりずっと多い量になってしまいます。この手法はまた、別々のパッケージを加法的に組み合わせる能力を台無しにてしまいます。少なくとも、個々のパッケージを実装する人がほかのパッケージについてあまり考慮しないでいいようにするという能力は台無しになってしまいます。例えば、上の例では、複素数と通常の数値の混合演算を扱うのが複素数パッケージの責任だということは妥当に見えます。しかし、有理数と複素数を組み合わせるのは、複素数パッケージでやってもいいし、有理数パッケージでやってもいいし、これら二つのパッケージから演算を抽出する何らかの第三のパッケージでやってもいいかもしれません。複数のパッケージにわたる責任の分割について一貫したポリシーを策定するということは、多くのパッケージと多くのクロスタイプ演算を伴うシステムを設計する際には、どうしようもないほど大変なタスクになってしまうでしょう。

⁴⁹ このほかに、(scheme-number complex) という二つの型を扱うために、ほとんど同じ手続きをもうひとつ用意する必要があります。

強制型変換

まったく関係のない型同士のまったく関連のない演算という一般的な状況では、たとえ面倒でも、明示的なクロスタイプ演算を実装することぐらいしかできません。幸い普通は、型システムに隠れているかもしれない積み上げ式の構造を利用して、もっとうまくやることができます。別々のデータ型が完全に独立しているというわけではなく、ある型のオブジェクトをほかの型と見なす方法があるということはよくあります。このプロセスは、*coercion*(強制型変換)と呼ばれます。例えば、通常の数値と複素数を算術演算によって組み合わせることを求められたら、通常の数値を虚部がゼロの複素数と見なすことができます。こうすると、この問題を二つの複素数を組み合わせる問題に変換でき、複素数パッケージによって普通のやり方で扱うことができるようになります。

一般に、ある型のオブジェクトを等価なほかの型のオブジェクトに変換する強制型変換手続きを設計することによって、この考え方を実装できます。以下のものは典型的な強制型変換手続きで、与えられた通常の数値を、実部がその数値で虚部がゼロの複素数に変換します。

```
(define (scheme-number->complex n)
  (make-complex-from-real-imag (contents n) 0))
```

これらの強制型変換手続きは、特別な強制型変換テーブルに組み込むことになります。二つの型の名前をキーとして使います。

```
(put-coercion 'scheme-number
  'complex
  scheme-number->complex)
```

(このテーブルを操作するための手続き `put-coercion` と `get-coercion` が存在すると仮定しています) 一般に、このテーブルにはいくつか空きができます。すべての型の任意のデータオブジェクトをほかのすべての型に強制型変換するということは、一般的には可能でないからです。例えば、任意の複素数を実数に強制型変換する方法というものはありません。そのため、一般的な `complex->scheme-number` 手続きがテーブルに含まれるということはありません。

強制型変換テーブルが準備できたら、Section 2.4.3 の `apply-generic` 手続きを変更を加えることによって、強制型変換を統一的に扱うことができるようになります。演算を適用することを求められたら、最初はこれまでと同じように引数の型に対して演算が定義されているかどうかチェックします。定義されていれば、演算-型テーブルで見つかった手続きにディスパッチします。定義されていなければ、強制型変換を試みます。簡単にするために、ここでは引数

が二つの場合だけを考えます。⁵⁰強制型変換テーブルを見て、一つ目の型のオブジェクトが二つ目の型に強制型変換可能かどうか確認します。可能であれば、一つ目の引数を強制型変換し、もう一度演算を試みます。もし一つ目の型のオブジェクトが二つ目の型に一般的に強制型変換できないのであれば、逆に二つ目の引数を一つ目の引数の型に強制型変換できないか試してみます。最後に、どちらの型についてももう一方の型に強制型変換する既知の方法がなければ、諦めます。手続きは以下のようになります。

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((type1 (car type-tags))
                    (type2 (cadr type-tags))
                    (a1 (car args))
                    (a2 (cadr args)))
                (let ((t1->t2 (get-coercion type1 type2))
                      (t2->t1 (get-coercion type2 type1)))
                  (cond (t1->t2
                           (apply-generic op (t1->t2 a1) a2))
                        (t2->t1
                           (apply-generic op a1 (t2->t1 a2)))
                        (else (error "No method for these types"
                                     (list op type-tags))))))
              (error "No method for these types"
                     (list op type-tags)))))))
```

上でざっくり見たように、この強制型変換という構想は、明示的にクロスタイプ演算を定義するという手法に比べて多くのメリットがあります。型同士を関連づけるために強制型変換手続きを書く(n 個の型を持つシステムに対しては、最大で n^2 個の手続き)ということはやはり必要ですが、書く必要があるのはそれぞれの型のペアに対して手続きひとつだけで、それぞれの型の集合とそれとのジェネリック演算に対して別々の手続きを書く必要はありません。⁵¹ここ

⁵⁰一般化についてはExercise 2.82を参照。

⁵¹うまくやると、普通は n^2 よりも少ない数の強制型変換手続きで済ませることができます。例えば、もし型1から型2への変換方法と型2から型3への変換方法がわかつていれば、この知識を使って型1から型3に変換することができます。このやり方を使うと、システムに新しい型を追加する際に明示的に提供する必要のある強制型変換手続き

では、型同士の適切な変換は型そのものによって決まり、適用する演算には依存しないという事実を利用しています。

一方、この強制型変換構想の持つ汎用性では十分でない応用もあるかもしれません。組み合わせるオブジェクトのどちらももう一方の型に変換できないという場合でも、両方のオブジェクトを第三の型に変換することで演算の実行が可能かもしれません。そのような複雑な場合を扱いながらプログラムのモジュール性を維持するためには、普通は型同士の関係のさらに深い構造を利用するシステムを構築することが必要になります。これについては次節で検討します。

型の階層

上で紹介した強制型変換構想は、型のペアの間に自然な関係があるということを頼りにしていました。しかし、異なる型の間の関係には、より“グローバルな”構造があるということがよくあります。例えば、整数、有理数、実数、複素数を扱うジェネリック算術演算システムを構築しているとしましょう。そのようなシステムでは、整数を特別な種類の有理数と見なし、有理数を特別な種類の実数と見なし、実数を特別な種類の複素数と見なすというのはとても自然です。ここで実際に扱っているのはいわゆる *hierarchy of types*(型の階層) というもので、その中では、例えば整数は有理数の *subtype*(サブタイプ)(つまり、有理数に適用できる任意の演算は自動的に整数に適用できる) ということになります。逆に、有理数は整数の *supertype* (スーパータイプ) と呼びます。ここで扱っている特定の階層はとても単純なもので、それぞれの型は高々ひとつのスーパータイプと高々ひとつのサブタイプを持ちます。このような構造は *tower*(タワー) と呼ばれます。[Figure 2.25](#)にこの構造を示します。

もし扱っているのがタワー構造であれば、階層に新しい型を追加するという問題はとても簡単にできます。新しい型をすぐ上のスーパータイプに組み入れる方法と、新しい型がすぐ下の型のどのようなスーパータイプであるかを記述するだけで十分だからです。例えば、複素数と整数の足し算をしたい場合、明示的に `integer->complex` という特別な強制型変換手続きを定義する必要はありません。その代わりに、整数の有理数への変換方法、有理数の実数への変換方法、実数の複素数への変換方法を定義します。次に、システムがこれらの

の数を劇的に減らすことができます。もしシステムを必要なだけ高度なものにする準備があるのなら、システムが型同士の関係の“グラフ”を検索して、明示的に提供された強制型変換手続きから推論できる強制型変換手続きを自動的に生成させることもできます。

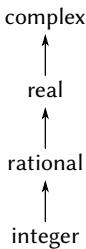


Figure 2.25: 型のタワー

ステップを通して整数を複素数に変換できるようにして、それから二つの複素数を足し合わせます。

`apply-generic` 手続きは、次のように設計し直すことができます。それぞれの型に対して、その型のオブジェクトをタワーの一階層に“上げる”`raise`という手続きを用意します。こうすると、システムが異なる型のオブジェクトの演算を行うよう求められたとき、すべてのオブジェクトが塔の同じ階に揃うようになるまで低い型を連続して上げていくということができるようになります ([Exercise 2.83](#)と[Exercise 2.84](#)は、そのような戦略の実装の詳細に関係するものです)。

タワーの別の利点として、すべての型がスーパータイプに定義されたすべての演算を“継承する”という概念を簡単に実装できるということがあります。例えば、整数に対して実部を求める特別な手続きを提供していないとしても、整数は複素数のサブタイプなので、整数に対しても `real-part` が定義されていることが期待されます。タワーであれば、`apply-generic` を修正して、統一的な方法でこれを実現できます。もし必要な演算が与えられたオブジェクトの型に対して直接定義されていなければ、オブジェクトをそのスーパータイプに上げて再試行します。こうやって、望む演算が実行可能になる階にたどり着くか、てっぺんに当たるか (その場合はあきらめることになります) するまで、引数を変換しながらタワーを登っていきます。

より一般的な階層に対してのタワーのもう一つの利点は、データオブジェクトを最も単純な表現に“下げる”ことが簡単にできるということがあります。例えば、 $2 + 3i$ と $4 - 3i$ を足す場合、複素数 $6 + 0i$ という形で答えを得るよりも、整数の 6 という答えを得るほうがいいでしょう。[Exercise 2.85](#)では、そのようなレベル下げ演算について検討します (注意すべきところは、レベル下げ

が可能な $6 + 0i$ のようなオブジェクトと、レベル下げが不可能な $6 + 2i$ のようなオブジェクトを見分ける一般的な方法が必要だというところです)。

階層の不適切さ

システムのデータ型が自然にタワーとして配置できる場合、ここまで見てきたように、異なる型同士のジェネリック演算の問題はとても単純になります。残念ながら、普通はそうはいきません。[Figure 2.26](#) はいろいろな型のより複雑な関係を図示したものです。この例では、さまざまな型の幾何学図形同士の関係を示しています。この図から、一般的にはひとつの型が二つ以上のサブタイプを持つということが見て取れます。例えば、三角形と四角形はどちらも多角形のサブタイプです。それに加えて、ひとつの型が二つ以上のスーパー・タイプを持つこともあります。例えば、直角二等辺三角形は二等辺三角形と見なすこともできますし、直角三角形と見なすこともできます。この複数スーパー・タイプ問題は特に厄介です。ある型を階層の中で“上げる”ための唯一の方法というものがいるということになるからです。あるオブジェクトにある演算を適用する際に、“正しい”スーパー・タイプを見つけるために、`apply-generic`のような手続きが型ネットワーク全体の中をかなり検索しないといけないことになるかもしれません。一般的に、ひとつの型には複数のサブタイプがあるので、値を型階層の中で“下げる”強制型変換にも同じような問題があります。巨大システムを設計する際に、モジュール性を保ちながら相互に関連する多くの型を取り扱うということは非常に難しく、現在多くの研究がなされている領域です。⁵²

⁵² この文は第一版にもあったのですが、12年前と同じように、現状に対しても当てはまります。いろいろな型を持つ実体同士の関係(哲学者が“オントロジー”と呼ぶもの)を表現する実用的で汎用的なフレームワークを開発することは、どうしようもないほど難しいように思えます。10年前に存在した混乱といま存在する混乱の主な違いは、今ではいろいろな不適切なオントロジーの理論が、同じように不適切なプログラミング言語の山に組み入れられているということです。例えば、オブジェクト指向言語の複雑性—それと、現代的なオブジェクト指向言語同士の微妙で混乱を招く相違点—の多くは、相互に関連する型に対するジェネリック演算の扱いが中心となっています。[Chapter 3](#)では計算オブジェクトについて検討しますが、これらの問題を完全に回避しています。オブジェクト指向言語に慣れた読者は、[Chapter 3](#)で局所状態についてかなり紙面を費やしているのに、“クラス”や“継承”には一言も触れていないことに気づくと思います。実際のところ、知識表現や自動推論の研究成果を利用することなしに、これらの問題をコンピュータ言語設計という観点のみから解決することはできないのではないかと私たちには疑っています。

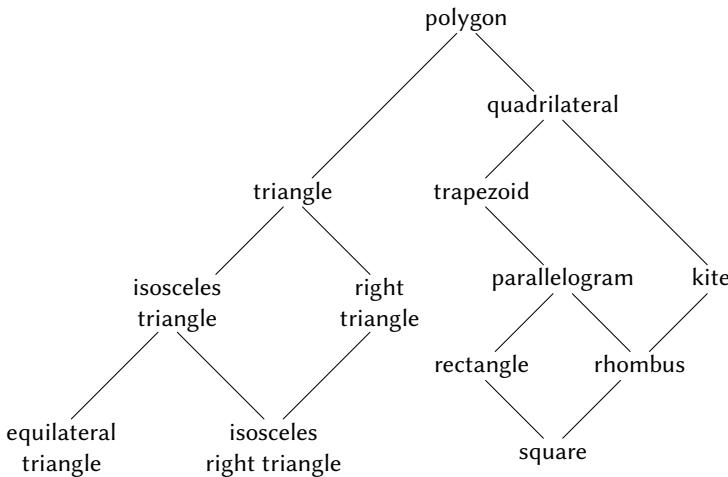


Figure 2.26: 幾何学図形の型の間の関係

Exercise 2.81: Louis Reasoner は、引数の型がすでに同じであっても、`apply-generic` は引数をお互いの型に強制型変換しようとしてもいいのではないかと気がついた。そのため、それぞれの型の引数をそれ自身の型に `coerce`(強制型変換) する手続きを強制型変換テーブルに入れる必要があると彼は考えた。例えば、上に示した `scheme-number->complex` という強制型変換に加え、彼は次のことを行う。

```

(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 の強制型変換手続きを組み込むと、もし `scheme-number` 型の二つの引数や `complex` 型の二つの引数とある演算に対

して `apply-generic` が呼ばれ、その演算がテーブル内でそれらの型に対して見つからない場合は、何が起こるだろうか。例えば、ジェネリックな指数関数演算を定義したとする。

```
(define (exp x y) (apply-generic 'exp x y))
```

そして、Scheme-number パッケージに対して指数関数手続きを追加し、ほかのパッケージには追加しないとする。

```
; 以下は Scheme-number パッケージに追加する
(put 'exp '(scheme-number scheme-number)
      (lambda (x y) (tag (expt x y))))
; 基本手続き expt を使う
```

`exp` を二つの複素数の引数で呼び出した場合、何が起こるだろうか。

- b 同じ型の引数に対する強制型変換について手を加えないといけないとする Louis の考え方は正しいだろうか。それとも、`apply-generic` はそのままの状態で正しく動作するだろうか。
- c `apply-generic` を修正し、二つの引数が同じ型であれば強制型変換を試行しないようにせよ。

Exercise 2.82: `apply-generic` を一般化し、複数の引数一般について強制型変換を扱うようにするやり方を示せ。戦略のひとつとしては、すべての引数を一つ目の引数の型に強制型変換することを試み、次に二つ目の引数の型に強制型変換することを試み、ということを続けるというものだ。この戦略について（また、上で述べた二引数バージョンについても）、それが十分に一般的でないような状況の例を挙げよ（ヒント：テーブルには適切な混合型演算があり、それが試行されないという場合について考える）。

Exercise 2.83: Figure 2.25 に示した型のタワー（整数、有理数、実数、複素数）を扱うジェネリック算術演算システムを設計しているとする。それぞれの型（複素数を除く）に対し、その型のオブジェクトをタワーの中で一階上げる手続きを設計せよ。それぞれの型（複素数を除く）に対して動作するジェネリックな `raise` 演算を組み込む方法を示せ。

Exercise 2.84: Exercise 2.83 の `raise` 演算を使って `apply-generic` 手続きを修正して、この節で検討した通り、連続して“上げる”という方法によって引数が同じ型を持つよう強制型変換を行うようにせよ。二つの型のどちらがタワーの中で高い位置にあるかをテストする方法を考える必要がある。システムのほかの部分と“互換性がある”ようなやり方でこれを行い、タワーに新しい階を追加する際に問題を引き起こさないようにせよ。

Exercise 2.85: この節では、あるデータオブジェクトを型のタワーの中で可能な限り下げていくことによって“単純化”するという方法について触れた。Exercise 2.83 で述べたタワーについて、これを実行する手続き `drop` を設計せよ。ポイントは、オブジェクトを下げることができるかどうかを何らかの汎用的なやり方で決めるということにある。例えば、複素数 $1.5 + 0i$ は `real` まで下げることができ、複素数 $1 + 0i$ は `integer` まで下げることができるが、複素数 $2 + 3i$ はまったく下げることができない。以下に、あるオブジェクトを下げることができるとどうかを決める計画のひとつを示す。まず、オブジェクトをタワーの中で一階下に“押す” ジェネリック演算 `project` (射影) を定義する。例えば、複素数の射影では虚部を捨てることになる。こうすると、ある数値を `project` して、その結果を元の型に `raise` したときに、最初のものと等しい何かになっていれば、その数値は下げができるということになる。可能な限りオブジェクトを落とす手続き `drop` を書くことで、この考え方を実装するやり方を詳しく示せ。いろいろな射影演算を設計し⁵³、`project` をジェネリック演算としてシステムに組み込むことが必要になる。また、Exercise 2.79 で述べたジェネリック等価性述語を利用することも必要となる。最後に、`drop` を使って Exercise 2.84 の `apply-generic` を書き直し、解答を“単純化”するようにせよ。

Exercise 2.86: 複素数の実部、虚部、絶対値、偏角に、通常の数値や有理数や、これからシステムに追加するその他の数値を使えるようにしたいとする。そのような複素数を使えるようにするために、システムにどのような変更を加える必要があるか説明し、それを実装せよ。通常の数値と有理数に対してジェネリックに使える `sine` や `cosine` のような演算を定義する必要があるだろう。

⁵³ 実数は、引数に最も近い整数を返す基本手続き `round` を使って整数に射影できる。

2.5.3 例: 記号代数

記号代数式の操作は複雑な処理です。それについて検討することで、大規模システムの設計の際に起こる多くの困難な問題について知ることができます。代数式は一般に、木構造の演算子をオペランドに適用する階層構造と見なすことができます。代数式を構築するには、定数や変数のような基本オブジェクトから始め、これらを加算や乗算のような代数演算子によって組み合わせます。ほかの言語の場合と同じように、複合オブジェクトを簡単に参照できるように抽象化を行います。記号代数での典型的な抽象化としては、線形結合、多項式、有理関数、三角関数などの考え方があります。これらは複合“型”と見なすことができ、そうすることで式の処理を指示することがやりやすくなります。例えば、次の式は、

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

整数を係数とする y の多項式の三角関数を係数とする x の多項式として記述することができます。

ここでは完全な代数処理システムを開発しようとしているわけではありません。そのようなシステムは、深い代数の知識と洗練されたアルゴリズムを具現化した、非常に複雑なプログラムになります。ここでは、代数操作の単純でありながら重要な部分である多項式の算術演算について見てていきます。また、そのようなシステムを設計する際にどのような決定をしなければいけないかということと、その取り組みに抽象データとジェネリック演算という考え方を適用する方法について説明します。

多項式の算術演算

多項式の数値演算を行うシステムを設計するにあたっての最初の仕事は、多項式とは何なのかを決ることです。多項式は通常、いくつかの変数 (多項式の *indeterminates* (不定元)) に関して定義されます。簡単にするために、ひとつしか不定元を持たない多項式 (*univariate polynomials* (一元多項式)) に限定することにします。⁵⁴ 多項式は項の和として定義し、その項は係数か、不定元の幕乗か、係数と不定元の幕乗の積であるとします。係数は、多項式の不定

⁵⁴一方で、多項式の係数自身が別の変数に関する多項式であることは許すことにします。こうすると、完全な多元システムと本質的に同じ表現力を持つことになりますが、この先で述べるように、強制型変換の問題が出てきます。

元と独立な代数式と定義します。例えば、

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

は単純な x の多項式で、

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

は係数が y の多項式である x の多項式です。

ここまでで、もういくつかの厄介な問題を避けて通ってきています。上の一一番目の多項式は、多項式 $5y^2 + 3y + 7$ と同じものでしょうか、それとも異なるものでしょうか。妥当な答えは、“多項式を純粋に数学の関数であると考えれば、同じものです。しかし、多項式を構文形式であると考えれば、異なるものです。”となるでしょう。また、二番目の多項式は、 x の多項式を係数とする y の多項式と代数学的に等価です。私たちのシステムはこのことを認識するべきでしょうか。しかも、多項式を表現する方法はほかにもあります—例えば、因数の積、(一元多項式の場合) 根の集合、特定の点の集合における多項式の値のリストなどによって表すこともできます。⁵⁵ この代数操作システムでは、“多項式”というのは特定の構文形式であって、その奥にある数学的意味ではないと決めることで、これらの問題を解決することができます。

次に、多項式の算術演算をどのようなものにするか考える必要があります。この単純なシステムでは、加算と乗算だけ考えることにします。さらに、組み合わせる二つの多項式は同じ不定元を持っていなければならないということになります。

このシステムの設計にあたっては、おなじみのデータ抽象化の規律に従います。多項式は *poly* というデータ構造を使って表現し、*poly* はひとつの変数と項の集まりからなります。*poly* からそれらの部品を抽出するセレクタ *variable*, *term-list* と、与えられた変数と項のリストから *poly* を組み立てるコンストラクタ *make-poly* があると想定します。変数はただの記号なので、変数の比較には Section 2.3.2 の *same-variable?* 手続きが使えます。以下の手続きは、*poly* の加算と乗算を定義するものです。

```
(define (add-poly p1 p2)
```

⁵⁵一元多項式について、与えられた点の集合における多項式の値を使うということが特によい表現となることもあります。こうすると、多項式の算術演算は非常に単純になります。例えば、この方式で表現された二つの多項式の和を求めるることは、対応する点における多項式の値を足し合わせるだけできます。より見慣れた表現に変形するには、 $n+1$ 個の点における多項式の値から n 次の多項式の係数を復元するラグランジュの補間公式が使えます。

```

(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))))

```

多項式を私たちのジェネリック算術演算システムに組み入れるためには、タイプタグをつける必要があります。タグとしては `polynomial` を使うことにして、タグつき多項式に対する適切な演算を演算テーブルに組み込みます。Section 2.5.1 同様、コードはすべて多項式パッケージの組み込み手続きの中に埋め込むことにします。

```

(define (install-polynomial-package)
  ;; 内部手続き
  ;; 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))
  ;(Section 2.3.2 の same-variable? と variable? 手続き)

  ;; 項と項リストの表現
  ;(下記の adjoin-term ... coeff 手続き)

  (define (add-poly p1 p2) ...)
  ;(add-poly が使う手続き)
  (define (mul-poly p1 p2) ...)
  ;(mul-poly が使う手続き)

```

```

;; システムのほかの部分と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)

```

多項式の加算は項ごとに実行します。同じ次数の項(つまり、同じ指数を持つ不定元)同士を組み合わせる必要があります。これは、足す項と足される項の係数の合計を係数とする、同じ次数を持つ新しい項を作ることで行えます。片方にだけ項があって、もう一方のほうに同じ次数の項がない場合は、構築中の和の多項式にその項をそのまま追加します。

項リストを操作するために、空の項リストを返す `the-empty-termlist` というコンストラクタと、項リストに新しい項を追加する `adjoin-term` というコンストラクタがあると想定します。さらに、与えられた項リストが空かどうか調べる `empty-termlist?` という述語と、項リストから最大次数の項を取り出すセレクタ `first-term`、最大次数の項以外の全ての項を返すセレクタ `rest-terms` もあると想定します。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` があると想定します。これらの演算によって、項と項リストがデータ抽象として捉えられるようになり、具体的な表現は別に考えることができます。

以下は、二つの多項式の和となる項リストを構築する手続きです。⁵⁶

```

(define (add-terms L1 L2)
  (cond ((empty-termlist? L1) L2)
        ((empty-termlist? L2) L1)
        (else
          (let ((t1 (first-term L1)))

```

⁵⁶この演算は、Exercise 2.62で開発した `union-set` 演算にとても似ています。実際、不定元の指數の順に並べた集合として多項式の項を考えると、和の項リストを生成するプログラムは `union-set` とほとんど同じものになります。

```

        (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)))))))))

```

ここでの最も重要なポイントは、項の係数同士を足し合わせるのにジェネリック加算手続き `add` を使っているというところです。以下で見ていくように、このことは強力な結果をもたらします。

二つの項リストを乗算するには、`mul-term-by-all-terms` を繰り返し使って、最初のリストの各項をもう一方のリストのすべての項と乗算します。`mul-term-by-all-terms` は、与えられた項と与えられた項リストのすべての項と乗算するものです。返される項リストの列(一つ目のリストの項それぞれに対してひとつずつ)を和として集積します。二つの項の乗算では、各乗数の次数の和を次数とし、各乗数の係数の積を係数とする項を作ります。

```

(define (mul-terms L1 L2)
  (if (empty-termlist? L1)
    (the-empty-termlist)
    (add-terms (mul-term-by-all-terms (first-term L1) L2)
               (mul-terms (rest-terms L1) L2))))
(define (mul-term-by-all-terms t1 L)
  (if (empty-termlist? L)
    (the-empty-termlist)
    (let ((t2 (first-term L)))
      (adjoin-term
        (make-term (+ (order t1) (order t2))
                  (mul (coeff t1) (coeff t2)))
        (mul-term-by-all-terms t1 (rest-terms L)))))))

```

多項式の加算と乗算は本当にこれがすべてです。項の演算にはジェネリック手続き `add` と `mul` を使っているため、ジェネリック数値演算パッケージにとって既知のどんな型の係数でも、自動的に多項式パッケージが扱えるようになっているというところに注意してください。Section 2.5.2で考察したような強制型変換メカニズムを組み込めば、以下のような係数の型が異なる多項式に対する演算も自動的に扱えるようになります。

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

`add-poly`, `mul-poly` という多項式の加算と乗算の手続きを `polynomial` 型の `add` と `mul` としてジェネリック算術演算システムに組み込んだので、私たちのシステムは自動的に次のような多項式演算を扱えるようになっています。

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

その理由は、システムが係数を組み合わせようとするとき、`add` と `mul` を通してディスパッチを行うためです。係数はそれ自身 (y の) 多項式なので、これらは `add-poly` と `mul-poly` を使って組み合わされることになります。結果として、これは“データ主導再帰”的な構造になります。例えば、`mul-poly` を呼び出すと、係数同士をかけ算するために `mul-poly` を再帰的に呼び出す結果になるということです。もし係数の係数がそれ自身多項式なら (この方法は三変数の多項式を表現するのに使えます)、データ主導の仕組みによってシステムがもう一レベルの再帰を行なうことが保証されます。データ構造によって規定されるレベルの数だけ、この再帰は続きます。⁵⁷

項リストの表現

最後に、項リストのよい表現を実装するという仕事に取りかからなければなりません。項リストというのは、実質的には、項の次数をキーとする係数の集合です。つまり、Section 2.3.3で検討したようなどんな集合の表現手法でも、

⁵⁷ この仕組みが完全にスムーズに動くようにするには、ジェネリック算術演算システムに“数値”を多項式に強制型変換する能力を追加しなければなりません。そのためには、数値を 0 次の多項式と見なし、係数がその数値であるとします。これは、次のような演算を行う場合に必要になります。

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

この例では、係数 $y + 1$ と係数 2 を足すことが求められます。

このタスクに適用することができます。一方、手続き `add-terms` と `mul-terms` は、常に高い次数から低い次数という方向で、項リストに順次アクセスします。このため、何らかの順序つきリスト表現を使うことにします。

項リストを表現するリストはどのように構造化するべきでしょうか。考慮するポイントのひとつは、操作対象となる多項式の“濃度”です。多項式は、ほとんどの次数の項が 0 でない係数を持つ場合、*dense*(密) と呼ばれます。もし 0 の項が多ければ、*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。

密な多項式の項リストは、係数のリストとして表現するのが最も効率的です。例えば上の A は、`(1 2 0 3 -2 -5)` という形でうまく表現できます。この表現では、その項の係数から始まるサブリストの長さから 1 を引いたものになります。⁵⁸ しかしこれは、 B のような疎な多項式の表現としてはひどいものになります。ほんの少しの孤立した非ゼロ項によって区切られた、巨大なゼロのリストになってしまふでしょう。疎な多項式のもうちょっとまともな表現としては、非ゼロ項のリストというものがあります。それぞれの項は、項の次数とその次数に対する係数を含むリストになります。その計画を使う場合、多項式 B は `((100 1) (2 2) (0 1))` という形で効率的に表現されることになります。ほとんどの多項式操作は疎な多項式に対して実行されるので、こちらの手法を使うことにします。項リストは高次から低次の順に並べられた項のリストとして表現されると想定します。このことを決めれば、セレクタとコンストラクタは素直に実装できます。⁵⁹

```
(define (adjoin-term term term-list)
```

⁵⁸これらの多項式の例では、Exercise 2.78で提案したような型メカニズムを使ってジェネリック算術演算システムを実装していると想定しています。つまり、通常の数値の係数は、`car` が記号 `scheme-number` であるペアとしてではなく、数値そのものによって表現されているということです。

⁵⁹項リストは順序つきだと想定していますが、`adjoin-term` の実装は、新しい項を既存の項リストに単純に `cons` するようになっています。`adjoin-term` を使う手続き (`add-terms` など) がこの手続きを呼ぶ際に、リスト内のものよりも高次の項を引数として呼び出しを行うということを保証するなら、このままでも問題はありません。そのような保証をしたくなければ、順序つきリストとしての集合の表現 (Exercise 2.61) に対する `adjoin-set` に近い形で `adjoin-term` を実装することもできます。

```

(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 で定義したものです (下の Exercise 2.87 も参照)。

多項式パッケージのユーザは、(タグつきの) 多項式を次の手続きによって作成します。

```

(define (make-polynomial var terms)
  ((get 'make 'polynomial) var terms))

```

Exercise 2.87: 多項式に対する `=zero?` をジェネリック数値演算パッケージに組み込め。こうすることで、係数がそれ自身多項式となっている多項式に対して `adjoin-term` が動作するようになる。

Exercise 2.88: 多項式システムを拡張し、多項式の減算ができるようせよ (ヒント : ジェネリックな符号反転演算を定義するといいかもしれない)。

Exercise 2.89: 密な多項式に適していると上で述べた項リスト表現を実装する手続きを定義せよ。

Exercise 2.90: 疎な多項式と密な多項式の両方に対して効率的な多項式システムを作りたいとする。そのためのひとつ的方法は、システム内でどちらの項リスト表現も使えるようにするということである。この状況は Section 2.4 の複素数の例と似ている。そのときは直行形式と極形式のどちらの表現も使えるようにした。これを行うためには、異なる型の項リストを識別し、項リストに対する演算をジェネリックにしなければならない。この一般化を実装するために、多項式システムを再設計せよ。これは局所的な変更ではなく、かなりの作業が必要になる。

Exercise 2.91: 1 変数多項式は、別の 1 変数多項式によって割ることができる。商と剰余は多項式となる。例えば、次のようになる

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

除算は筆算方式で実行できる。まず、被除数の最高次の項を除数の最高次の項で割る。その答えが商の第一項となる。次に、答えに除数をかけ、被除数からそれを引く。このようにして、再帰的に引き算の結果を除数で割り算するということを繰り返して残りの答えを求める。除数の次数が被除数の次数を上回ったら終わりとして、被除数を剰余とする。また、もし被除数がゼロになった場合には、商と剰余の両方をゼロとして返す。

`div-poly` 手手続きは、`add-poly` や `mul-poly` をベースに設計できる。この手続きは、二つの多項式が同じ変数を持つかどうかチェックする。同じ変数を持つなら、`div-poly` は変数を取り去って問題を `div-terms` に渡す。`div-terms` は項リストに対する除算を実行する。最後に、`div-poly` は `div-terms` の結果に変数をくっつけ直す。`div-terms` は、除算の商と剰余の両方を求めるようになるのが便利だ。`div-terms` は項リスト二つを引数として取り、商となる項リストと剰余となる項リストからなるリストを返す。

次の空欄を埋め、`div-terms` の定義を完成させよ。これを使って、二つの多項式を引数として取り、商と剰余の `poly` のリストを返す `div-poly` を実装せよ。

```
(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
                     (再帰的に残りを計算する)
                     )))
                (完全な結果を作る)
                ))))))
```

記号代数の型の階層

私たちの多項式システムは、ある型(ここでは多項式)のオブジェクトが、実際には多くの異なる型のオブジェクトを部品として持つ複雑なオブジェクトだということがありうることを示しています。このことは、ジェネリック演算を定義するうえで特に難しい問題にはなりません。必要な複合型の部品の操作を行うための適切なジェネリック演算を組み込むだけで大丈夫です。実際、多項式は“再帰的データ抽象化”的な一つとなっているということを見てきました。これは、多項式の部品がそれ自身多項式でありうるということです。ジェネリック演算とデータ主導プログラミングスタイルのおかげで、それほど苦労せずにこの複雑な問題を扱うことができています。

一方、多項式代数は、データ型を自然にタワー型に配置することができないシステムです。例えば、このシステムでは係数が y の多項式である x の多項式というものが作れます。また、係数が x の多項式である y の多項式を作ることもできます。これらの型は、自然に考えると、片方がもう一方の“上”であるとはどうやっても言えません。しかし、それぞれの集合の要素を足し合わせることが必要になるということはよくあります。ひとつの可能性としては、片方の多項式を展開したり再配置したりして、もう一方の多項式の型に変換し、両方の多項式が同じ主変数を持つようにするというものです。変数に順番をつけて、どんな多項式でも、高優先度の変数が主変数となり低優先度の変数が係数に埋め込まれるような“標準形”に常に変換することにすれば、タワー的な構造を押しつけることはできます。この戦略はそれなりにうまくいくのですが、変換の過程で多項式を不必要に展開して、読みにくくなったり、場合によっては非効率になってしまったりします。このタワー戦略というのは、確実にこの領域に対して自然なものとは言えません。ユーザが古い型をいろいろなやり方(三角関数、幕級数、積分など)によって組み合わせて動的に新しい型を発明できるようなどのような領域に対しても、同じことが言えます。

強制型変換の制御は大規模な代数処理システムを設計するうえで深刻な問題になります。これは驚くには当たりません。そのようなシステムでは、複雑性の多くはいろいろな型同士の関係に関わるところにあります。実際のところ、強制型変換についてはまだ完全に理解できているとは言えないところがあります。それどころか、データ型という概念についても、完全にわかっているとは言えません。それでも、私たちの知識は、大規模なシステムの設計を支える強力な構造化とモジュール化の原則を与えてくれます。

Exercise 2.92: 変数に順番をつけることによって、異なる変数の多項式同士の加算と乗算が動作するよう多項式パッケージを拡張せ

よ。(これは簡単ではない!)

拡張問題: 有理関数

ジェネリック算術演算システムを拡張し、*rational functions*(有理関数)を含むようにすることもできます。これは、次のように分子と分母が多項式の“分数”です。

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

システムは、有理関数の加算、減算、乗算、除算ができなければなりません。また、以下のような計算も実行できる必要があります。

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

(ここでは、加算結果は共通因数を取り除いて簡約されています。通常の“たすき掛け”であれば、分子が4次多項式で分母が5次多項式の分数になっていたところです。)

有理数算術演算パッケージを修正し、ジェネリック演算を使えるようにすれば、分数を既約にするという問題を除いて、望むことができるようになります。

Exercise 2.93: 有理数算術演算パッケージを修正し、ジェネリック演算を使うようにせよ。しかし、`make-rat` は分数を既約にしようと試みないように変更せよ。`make-rational` を二つの多項式に対して呼び出し有理関数を作ることで、システムをテストせよ。

```
(define p1 (make-polynomial 'x '((2 1)(0 1))))
(define p2 (make-polynomial 'x '((3 1)(0 1))))
(define rf (make-rational p2 p1))
```

次に、`add` を使って `rf` にそれ自身を足せ。この加算手続きが分数を既約にしないことが確認できるだろう。

多項式の分数を既約にするのは、整数のところで使ったのと同じ考え方ですることができます。`make-rat` を修正して、分子と分母の両方を、それらの最大公約数で割るようにするということです。“最大公約数”という概念は、多項式についても考えることができます。実際に、二つの多項式の GCD(最大公約数)

は、整数に対するユークリッドのアルゴリズムと本質的に同じものを使って求めることができます。⁶⁰整数版は次のようなものでした。

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

これを使って、自明な修正を加えて項リストに対して動く GCD 演算を定義できます。

```
(define (gcd-terms a b)
  (if (empty-termlist? b)
      a
      (gcd-terms b (remainder-terms a b))))
```

ここでの `remainder-terms` は、Exercise 2.91で実装した項リスト除算演算 `div-terms` が返すリストから、剩余の部分を取り出すものです。

Exercise 2.94: `div-terms` を使って手続き `remainder-terms` を実装し、それを使って `gcd-terms` を上記のように定義せよ。次に、二つの多項式に対して多項式 GCD を求める手続き `gcd-poly` を書け(この手続きは、二つの多項式が同じ変数のものでなければエラーを出すようにする)。システムに `greatest-common-divisor` というジェネリック演算を組み込み、多項式は `gcd-poly` によって簡約し、通常の数値は通常の `gcd` によって簡約するようにせよ。テストとして以下を実行せよ。

```
(define p1 (make-polynomial
  'x '((4 1) (3 -1) (2 -2) (1 2))))
```

⁶⁰ユークリッドのアルゴリズムが多項式に対して動作する働くということは、代数では、多項式が *Euclidean ring*(ユークリッド環)と呼ばれる代数領域の一種を形成するというように形式化されます。ユークリッド環とは、加算、減算、交換可能な乗算を認め、環の各元 x に対して、ある性質を持った“測度” $m(x)$ という正の整数を割り当てる方法を持つ領域です。その性質とは、任意の非ゼロな x と y に対し $m(xy) \geq m(x)$ となり、任意の x と y に対し、 $y = qx + r$ で $r = 0$ または $m(r) < m(x)$ となる q が存在するというものです。

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 p2 (make-polynomial 'x '((3 1) (1 -1))))
(greatest-common-divisor p1 p2)
```

その結果を手計算によって確認せよ。

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) を使って Q_1 と Q_2 の GCD を求めよ。答えが P_1 と同じにならないことに注意せよ。この例では計算に非整数演算の演算が絡んでくるので、GCD アルゴリズムが困難になっている。⁶¹ 何が起こっているか理解するために、GCD の計算時に `gcd-terms` をトレースせよ。または、この除算を手計算で実行せよ。

Exercise 2.95で出てきた問題は、以下のように GCD アルゴリズムを修正すると解決できます(これは、実際には整数係数の多項式の場合にしか動作しません)。GCD 計算で多項式の除算を行う前にはいつも、除算過程で分数が現れないことを保証するように選んだ整数定数を被除数にかけるようにします。こうすると、答えは実際の GCD と整数定数倍だけ違うものになりますが、このことは有理関数を既約に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のに使うので、整数定数倍は相殺されます。

より正確には、 P と Q が多項式であるとき、 O_1 を P の次数(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 の先頭の係数とします。すると、 P に *integerizing factor* (整数化因子) $c^{1+O_1-O_2}$ をかけると、答えの多項式を `div-terms` アルゴリズムを使って分数を出さずに Q で割ることができるということが示せます。被除数にこの定数をかけてから割るという操作は、 P の Q による *pseudodivision*(擬除算) と呼ばれることもあります。除算の剰余は *pseudoremainder*(擬剰余) と呼ばれます。

Exercise 2.96:

⁶¹ MIT Scheme のような実装では、この計算は確かに Q_1 と Q_2 の約数となる多項式を生成しますが、係数は有理数になります。ほかの多くの Scheme システムでは、整数の除算結果は有限精度の小数となることがあるので、正しい約数が得られないことがあります。

- a 手続き `pseudoremainder-terms` を実装せよ。これは `remainder-terms` と同じようなものになるが、`div-terms` を呼ぶ前に、上で述べた整数化因数を被除数にかけるというところが違う。`gcd-terms` を修正して `pseudoremainder-terms` を使うようにし、`greatest-common-divisor` が Exercise 2.95 の例に対して整数係数の答えを返すようになっていることを確認せよ。
- b これで GCD は整数係数になる。しかし、係数は P_1 の係数よりも大きくなる。`gcd-terms` を修正し、答えの係数すべての(整数の)最大公約数で係数を割ることによって答えの係数から共通因数を取り除くようにせよ。

まとめると、有理関数を既約にする方法は以下のようになります。

- Exercise 2.96版の `gcd-terms` を使って、分子と分母の GCD を求める。
- GCD を求めたら、分子と分母を GCD で割る前に、それらに同じ整数化因子をかけて、GCD による割り算で非整数の係数が出ないようにする。整数化因子としては、GCD の最初の係数を $1 + O_1 - O_2$ 乗したものが使える。ここで、 O_2 は GCD の次数、 O_1 を分子と分母の大きいほうの次数とする。こうすると、分子と分母を GCD で割っても分数が出てこないということが保証できる。
- この演算の結果は、整数係数を持つ分子と分母になる。いろいろ整数化因子をかけているため、係数は普通とても大きなものになっている。そのため、最終ステップとして、分子と分母の係数すべてに対する(整数の)最大公約数を求め、それで割ることによって、冗長な因数を取り除く。

Exercise 2.97:

- a このアルゴリズムを `reduce-terms` という手続きとして実装せよ。手続きは `n` と `d` という二つの項リストを引数として取り、上で説明したアルゴリズムによって `n` と `d` を既約にした `nn` と `dd` のリストを返す。また、`add-poly` と同様の手続き `reduce-poly` を実装せよ。これは二つの `poly` が同じ変数を持つかどうかチェックし、もし同じであれば変数を取り去って問題を `reduce-terms` に渡し、`reduce-terms` に返された二つの項リストにその変数をつけ直す。

b `reduce-terms` と似たような手続きとして、元々の `make-rat` が整数に対して行っていたことと同じことを実行するものを次のように定義せよ。

```
(define (reduce-integers n d)
  (let ((g (gcd n d))) (list (/ n g) (/ d g))))
```

そして、`reduce` というジェネリック演算を定義せよ。これは、`apply-generic` を呼んで、`reduce-poly`(polynomial 型引数に対して) または `reduce-integers`(scheme-number 型引数に対して) にディスパッチする。こうすると、`make-rat` が分子と分母を組み合わせて有理数を作る前に `reduce` を呼ぶようにすることで、有理数算術演算パッケージが有理数を既約にするようになります。これで、このシステムは整数と多項式のどちらの有理式も扱えるようになった。プログラムをテストするために、この拡張練習問題の最初に出てきた例を試してみよ。

```
(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)
```

正しく既約になっている正しい答えが返ってくるかどうかを確認せよ。

GCD の計算は、有理関数の演算を行うどんなシステムでも、中心的な存在になります。上で使ったアルゴリズムは、数学的には素直なのですが、非常に遅いものです。遅さの原因は、部分的には除算回数の多さにあり、部分的には擬除算によって生成される中間係数が巨大になるというところにあります。多項式の GCD を求めるよりよいアルゴリズムをどう設計するかということは、代数処理システム開発において活発な領域のひとつです。⁶²

⁶² 多項式 GCD 計算の非常に効率的でエレガントな手法が Richard Zippel (1979) によって発見されました。この手法は、Chapter 1 で検討した高速な素数性テストと同じく、確率的アルゴリズムです。Zippel の本 (Zippel 1993) では、この手法のほかにも、多項式の GCD を求めるほかのいくつかの方法について解説しています。

3

モジュール性、オブジェクト、状態

Μεταβάλλον ἀναπαύεται
(変化していくながら、静止している)
—Heraclitus

Plus ça change, plus c'est la même chose.
(変化すればするほど、同じものとなる)
—Alphonse Karr

ここまでこの章では、プログラムを作る基本的な要素を紹介してきました。基本手続きや基本データを組み合わせて合成物を作る方法について概観し、大きなシステムの複雑性に対処するには抽象化が重要だということを学びました。しかし、これらの道具はプログラムを設計するのに十分ではありません。プログラムを効率的に統合していくためには、プログラムの全体的な設計計画を立てるうえで助けになってくれる組織化原則が必要です。とりわけ、大きなシステムを *modular*(モジュール式) に構成する、つまり、システムをまとまりのある部品に“自然に”分割し、別々に開発、保守をすることを可能にするための戦略が必要となります。

物理システムをモデル化するプログラムの構成に特に適した強力な設計戦略のひとつとして、モデル化対象のシステムの構造をもとにプログラムの構造を決めるというものがあります。システムのそれぞれのオブジェクトに対して、

対応する計算オブジェクトを構築するというものです。この戦略をとることの狙いは、新しいオブジェクトやアクションを導入するためにモデルを拡張する際に、プログラムの戦略はそのままで、それらのオブジェクトやアクションの記号的な対応物を追加するだけですむようにすることです。システムの組織化がうまくいっていれば、新しい機能を追加したり古い機能をデバッグしたりするのに、システムの局所的部品に手を加えるだけでいいようになっているはずです。

すると、大きなプログラムを組織化する方法は、モデル化の対象となるシステムを私たちがどう知覚するかというところにかなり左右されることになります。この章では、システムの構造に対する二つのかなり異なった“世界観”から浮かび上がってくる、二つの顕著な組織化戦略について調べていきます。一つ目の組織化戦略は *objects* (オブジェクト) に集中するもので、はっきり分かれたオブジェクトの集合として大きなシステムを捉え、それらのオブジェクトのふるまいは時間とともに変化するとなります。もうひとつの組織化戦略は、システムを流れる情報の *streams* (ストリーム) に集中するというものです。これは、電子技術者が信号処理システムを見るのと同じやり方です。

オブジェクトによるアプローチもストリーム処理によるアプローチも、どちらもプログラムにおける大きな言語的な問題を引き起します。オブジェクトの場合、どうやって計算オブジェクトを変化させながら同一性を維持させるかという問題を取り組む必要があります。この問題のために、古い置換モデル (Section 1.1.5) はあきらめて、より機械的なものの理論的には扱いにくい、計算の *environment model* (環境モデル) を採用することになります。オブジェクト、変化、同一性を扱うことの難しさは、計算モデルの中で時間を扱うことの必要性から生まれる根本的な問題です。プログラムを並行実行できるようにしようとすると、これらの問題はますます大きくなります。ストリームというアプローチは、モデルの中でシミュレートしている時間と、評価の際にコンピュータの中で起こるイベントの順番とを切り離す場合に最も役に立ちます。このことは、*delayed evaluation* (遅延評価) として知られているテクニックによって達成できます。

3.1 代入と局所状態

私たちは普通、独立した物 (オブジェクト) があちこちにあるような場所としてこの世界を捉えています。それらのオブジェクトは、時間とともに変わる状態を持っています。オブジェクトのふるまいがその過去に影響されるとき、

そのオブジェクトは“状態を持つ”といいます。例えば銀行口座は、“100 ドル引き出せますか?”という質問の答えが預金と引き出しの取引履歴によって変わるという意味で、状態を持つといえます。オブジェクトの状態は、ひとつ以上の*state variables*(状態変数)によって特徴づけることができます。状態変数は、オブジェクトの現在のふるまいを決めるのに十分な履歴情報を保持します。単純な銀行システムでは、口座の取引履歴全体を記憶するのではなく、現在の残高によって特徴づけることができるでしょう。

多くのオブジェクトからなるシステムでは、オブジェクト同士が完全に独立しているということはほとんどありません。それぞれのオブジェクトは相互作用によってほかのオブジェクトの状態に影響を与え、その相互作用がオブジェクトの状態変数同士を結びつけます。実際に、システムがばらばらのオブジェクトによって構成されているという見方は、システムの状態変数がいくつかの密結合のサブシステムにグループ化することができて、サブシステム同士の結合は疎であるという場合に最も役に立つものです。

このシステムの見方は、システムの計算モデルを組織化する強力なフレームワークとなりえます。そのようなモデルをモジュール式にするには、システム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解する必要があります。それぞれの計算オブジェクトは、実際のオブジェクトの状態を表す*local state variables*(局所状態変数)を持たなければなりません。モデル化対象のシステムに存在するオブジェクトの状態は時間とともに変化するので、それらに対応する計算オブジェクトの状態変数も同じく変化しなければなりません。もしシステム内での時間の流れをコンピュータ内での経過時間としてモデル化することにするなら、プログラムの実行に伴ってふるまいの変わる計算オブジェクトを構築する方法が必要です。具体的には、もし状態変数をプログラミング言語の通常の記号名によってモデル化したいならば、プログラミング言語は*assignment operator* (代入演算子) を提供し、名前に関連づけられた値を変化させることができなければなりません。

3.1.1 局所状態変数

時間とともに変化する状態を持った計算オブジェクトを持つということがどういうことなのかを示すために、銀行口座からお金を引き出すという状況をモデル化することにします。そのために、引き出す *amount*(金額) を引数に取る *withdraw*(引き出し) という手続きを使います。引き出しを行うのに十分なお金が口座にあれば、*withdraw* は引き出し後に残る残高を返します。そうでなければ、*withdraw* は *Insufficient funds*(残高不足) というメッセージを返し

ます。例えば、口座に 100 ドルある状態から始めた場合、`withdraw`を使うことで、次のような一連の応答を受け取ります。

```
(withdraw 25)
75
(withdraw 25)
50
(withdraw 60)
"Insufficient funds"
(withdraw 15)
35
```

式 `(withdraw 25)` は二回評価されていますが、それぞれ違う値を返しているというところに注意してください。これは、手続きのふるまいとして、これまでになかった種類のものです。これまでには、手続きはすべて数学関数を計算する仕様として捉えることが可能でした。手続きの呼び出しを行うと、与えられた引数を関数に適用したときの値の計算が行われ、同じ手続きを同じ引数で二回呼び出すと、いつでも同じ値が返ってくることになっていました。¹

`withdraw` を実装するために、口座の残高を示す変数 `balance` を使い、`balance` にアクセスする手続きとして `withdraw` を定義します。`withdraw` 手続きは、`balance` が少なくとも要求された `amount` 以上の大ささがあるかをチェックします。もあるなら、`withdraw` は `balance` を `amount` 分だけ減らし、新しい `balance` の値を返します。なければ、`withdraw` は *Insufficient funds*(残高不足) というメッセージを返します。以下は、`balance` と `withdraw` の定義です。

```
(define balance 100)
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))
```

`balance` の減算は次の式によって行います。

¹ 実際には、これはあまり正しくありません。例外のひとつとして、Section 1.2.6 の乱数生成器があります。また別の例外としては、Section 2.4.3 で導入した演算-型テーブルに関連するものがあります。この場合、同じ引数で `get` を二回呼び出した場合の値は、その間の `put` の呼び出しによって変わります。一方、代入を導入するまで、そのような手続きを自分で作る方法はないということです。

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

ここでは **set!** という特殊形式を使っています。構文は次のようにになります。

```
(set! <name> <new-value>)
```

ここで、<name> は記号で、<new-value> は任意の式です。**set!** は<name> の値を <new-value> を評価して得られる結果に変更します。ここでの例では、balance を変更して、balance の以前の値から amount を引いた結果を新しい値にします。²

withdraw では、**if** のテストが真の場合に二つの式が評価されるようにするために、**begin** という特殊形式も使っています。二つの式では、まず balance を減らし、それから balance の値を返しています。一般的に、以下の式を評価すると、

```
(begin <exp1> <exp2> ... <expk>)
```

<exp₁> から <exp_k> までの式が順番に評価され、最後の式 <exp_k> の値が **begin** 形式全体の値として返されます。³

withdraw は意図した通りに動くのですが、変数 balance が問題になります。上で記述したように、balance はグローバル環境で定義された名前なので、どんな手続きからでもアクセスして値を調べたり変更したりすることができてしまいます。何とかして balance を **withdraw** の内部に置いて、**withdraw** が balance に直接アクセスできる唯一の手続きになるようにして、ほかの手続きは全部、balance には (**withdraw** の呼び出しを通して) 間接的にしかアクセスできないようにできればずっとよくなるでしょう。そうできれば、**withdraw** が口座の状態を覚えておくために使っている局所状態変数が balance であるという概念をより正確にモデル化できることになります。

balance を **withdraw** の内部に置くようにするには、定義を以下のように書き直します。

²**set!** 式の値は実装依存です。**set!** は作用のためだけに使うもので、値のために使うものではありません。

set! という名前は、Scheme で使われる命名規則に従っています。変数の値を変更する演算（または、Section 3.3 で学ぶデータ構造を変える演算）は、感嘆符で終わる名前をつけることになっています。これは、述語を指すのに疑問符で終わる名前を使うと似ています。

³Scheme では手続きの本体として式の連続が使えるので、すでに **begin** を暗黙的にプログラムの中で使正在することになります。また、**cond** 式の各節の <consequent> 部分も、单一の式ではなく、式の連続にすることができます。

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))))
```

ここでは、初期値 100 に束縛されたローカル変数 `balance` を持つ環境を `let` によって作っています。この局所環境の中で、`amount` を引数に取り以前の `withdraw` 手続きと同じようなふるまいをする手続きを `lambda` を使って作っています。`let` 式を評価した結果として返されるこの手続きは `new-withdraw` で、`withdraw` と正確に同じふるまいをしますが、変数 `balance` はほかのどの手続きからもアクセスできません。⁴

`set!` をローカル変数と組み合わせるというのは、今後局所状態を持つ計算オブジェクトを構築するのに使うことになる一般的なプログラミングテクニックです。残念ながら、このテクニックを使うと深刻な問題が出てきます。最初に手続きを導入したとき、手続きの適用とはどういうことかを解釈するために評価の置換モデル (Section 1.1.5) というのも導入しました。そのとき、手続きの適用とは、仮引数を値で置き換えたうえで手続きの本体を評価することとして解釈するべきだと述べました。問題は、言語に代入を導入したとたん、置換というのは手続き適用の適切なモデルではなくなるということです（どうしてそうなるのかはSection 3.1.3で見ていきます）。結果として、厳密に言うと、`new-withdraw` がなぜ上で述べたようにふるまうのかということについて現時点で理解する方法がないということになります。`new-withdraw` のような手続きをちゃんと理解するには、手続き適用についての新しいモデルを開発する必要があります。Section 3.2ではそのようなモデルを紹介し、さらに `set!` とローカル変数についての説明も加えます。しかし、とりあえず `new-withdraw` によって打ち立てられたテーマについて、いくつかの別のパターンを詳しく見ていきましょう。

⁴ プログラミング言語の業界用語では、変数 `balance` は手続き `new-withdraw` の中に *encapsulated*(カプセル化) されていると言います。カプセル化は、*hiding principle*(隠蔽原理) として知られる、一般的なシステム設計の原則を反映したものです。隠蔽原理とは、システムの各部品をほかの部品から守ることによって、システムをよりモジュール的で頑健なものにできるという考え方です。これはつまり、“知る必要”のあるシステムの部品にだけ、情報へのアクセス権を与えるということです。

次の手続き `make-withdraw` は“引き出し処理器”を作成します。`make-withdraw` の仮引数 `balance` は、口座の初期残高を指定します。⁵

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

`make-withdraw` を使うと、以下のように `w1` と `w2` という二つのオブジェクトを作ることができます。

```
(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` は完全に独立したオブジェクトで、それぞれ自分の局所状態変数 `balance` を持っているということがわかるでしょうか。片方から引き出しても、もう一方には影響しません。

引き出しだけでなく預け入れも扱うオブジェクトを作ることもできます。これによって、単純な銀行口座を表現することができるようになります。以下は、指定した初期残高を持つ“銀行口座オブジェクト”を返す手続きです。

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

⁵ 上の `new-withdraw` とは違い、`balance` をローカル変数にするために `let` を使う必要はありません。仮引数は元々ローカルだからです。このことは、Section 3.2で評価の環境モデルについて検討すると、よりはつきりします ([Exercise 3.10](#)も参照)。

```

        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)

```

`make-account` を呼び出すたびに、局所状態変数 `balance` を持つ環境が構築されます。この環境の中で、`make-account` は `balance` にアクセスする手続き `deposit`, `withdraw` と、また “メッセージ” を入力として取り二つの局所手続きのうちひとつを返す `dispatch` という追加の手続きを定義します。銀行口座オブジェクトを表現する値としては、`dispatch` 手続きそのものが返されます。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) のプログラミングスタイルですが、ここではローカル変数を変更する能力と合わせて使っています。

`make-account` は次のように使えます。

```

(define acc (make-account 100))
((acc 'withdraw) 50)
50
((acc 'withdraw) 60)
"Insufficient funds"
((acc 'deposit) 40)
90
((acc 'withdraw) 60)
30

```

`acc` を呼び出すたびに、局所的に定義された `deposit` または `withdraw` 手続きのどちらかが返され、それが指定された `amount` に適用されます。`make-withdraw` の場合と同じように、`make-account` をもう一度呼び出すと、

```
(define acc2 (make-account 100))
```

完全に別の口座オブジェクトを作り、自分のローカルな `balance` を保持します。

Exercise 3.1: `accumulator` は、ひとつの数値引数で繰り返し呼びられる手続きで、引数を合計に集積していく。呼び出されるたびに現在までに集積された合計を返す。それぞれ独立した合計を持つアキュムレータを返す手続き `make-accumulator` を書け。`make-accumulator` への入力は、合計の初期値を指定する。例えば、以下のようになる。

```
(define A (make-accumulator 5))
(A 10)
15
(A 10)
25
```

Exercise 3.2: ソフトウェアをテストするアプリケーションでは、計算の過程で、ある手続きが何回呼ばれたかを数えられると便利だ。一引数の手続き `f` を入力として取る手続き `make-monitored` を書け。`make-monitored` が返すのは第三の手続き（仮に `mf` とする）で、この手続きは内部カウンタを保持することによって呼び出された回数を覚えておく。`mf` に対する入力が `how-many-calls?` という特別な記号である場合、`mf` はカウンタの値を返す。入力が `reset-count` という特別な記号である場合、`mf` はカウンタをゼロにリセットする。それ以外の入力に対しては、`mf` はその入力によって `f` を呼び出した結果を返し、カウンタを 1 増やす。例えは、監視をつけたバージョンの `sqrt` 手続きを作ることもできる。

```
(define s (make-monitored sqrt))
(s 100)
10
(s 'how-many-calls?)
1
```

Exercise 3.3: `make-account` 手続きを修正し、パスワードで守られた口座を作成するようにせよ。具体的には、次のように `make-account` が追加の引数としてひとつの記号を取るようにする。

```
(define acc (make-account 100 'secret-password))
```

結果としてできる口座オブジェクトは、アカウント作成時のパスワードを伴っている場合にだけ要求を処理し、その他の場合にはエラーメッセージを出す。

```
((acc 'secret-password 'withdraw) 40)
60
((acc 'some-other-password 'deposit) 50)
"incorrect password"
```

Exercise 3.4: Exercise 3.3の make-account 手続きを変更して別のローカル状態変数を追加し、間違ったパスワードで口座に 7 回連続アクセスされると call-the-cops(警察を呼ぶ) という手続きを呼ぶようにせよ。

3.1.2 代入を導入する利点

今後見ていくことになりますが、プログラミング言語に代入を導入したことによって、我々は難しい概念的問題の藪の中に迷い込むことになります。それでもなお、ローカルな状態を持つオブジェクトの集合としてのシステムの見方は、モジュール化された設計を維持していくうえで強力なテクニックになります。単純な例として、rand 手続きの設計について考えてみましょう。rand 手続きを、呼ばれるたびにランダムに選ばれた整数を返します。

“ランダムに選ぶ”というのがどういうことなのかはまったくはつきりしていません。私たちが望んでいるのは、おそらく、rand を連続して呼び出すと一様分布という統計的性質を持つ数列を作るということです。ここでは、適切な数列を生成する方法については議論しません。その代わりに、ある数値 x_1 から始めると、望ましい統計的性質を持った数列 x_1, x_2, x_3 を以下のように生成するという性質を持つ手続き rand-update がもうあるとします。⁶

⁶rand-update を実装する一般的な方法としては、 a, b, m を適切に選ばれた整数として、 x を $ax + b$ modulo m に更新するというものがあります。Knuth 1981 の 3 章は、乱数列を生成し、それらの統計的性質を確立するテクニックについて広汎に考察しています。rand-update 手続きを数学関数を計算しているということに注意してください。同じ入力を 2 回与えられると、この手続きは同じ出力を返します。そのため、それぞれの数値が前の数値と関係していないというのが“ランダム”だとすると、rand-update によって生成される数列は、まったく“ランダム”とはいえません。“眞のランダム性”と、はつきりと規定された計算によって生成されるものの適切な統計的性質を持つ、いわゆる pseudo-random (擬似乱数) との関係は、数学と哲学の難しい課題が絡む複雑な問題で

```
x2 = (rand-update x1)
x3 = (rand-update x2)
```

rand は、局所状態変数 x を持ち、それをある固定値 random-init で初期化する手続きとして実装できます。rand を呼び出すたびに、現在の x の値に対する rand-update を計算し、それを乱数として返します。また、この値を x の新しい値として保存します。

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

もちろん、単純に rand-update を直接呼べば、代入を使わないので同じ乱数列を生成することもできます。しかし、そうするとプログラムの中で乱数を使うどの箇所でも、現在の x の値を明示的に覚えておいて、rand-update の引数として渡さないといけないということになります。これがどれだけ面倒なことになるかを実感するために、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使う場合について考えてみましょう。

モンテカルロ法は、大きい集合からランダムにサンプルの試行を選び、それらの試行結果を集計することによって推測した確率をもとに推定を行うというものです。例えば、ランダムに選んだ二つの整数が共通因子を持たない、つまり最大公約数が 1 であるという確率が $6/\pi^2$ であるという事実を使って、 π を近似することができます。⁷ π の近似値を求めるためには数多くの試行を行います。それぞれの試行では二つの整数をランダムに選び、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によって $6/\pi^2$ の近似値が得られるので、この値から π の近似値が得られます。

プログラムの中心となるのは手続き monte-carlo です。この手続きは試行回数と試行を引数に取ります。試行は、実行されるたびに真または偽を返す、引数なしの手続きとして表現されます。monte-carlo は、指定された回数だけ試行を実行し、試行が真となった割合を表す数値を返します。

```
(define (estimate-pi trials)
```

す。Kolmogorov, Solomonoff, Chaitin はこれらの問題の解明に大きく寄与しました。これに関する議論は Chaitin 1975 にあります。

⁷ この定理は E. Cesàro によるものです。議論と証明については、Knuth 1981 の 4.5.2 節参照。

```

(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))))
  (iter trials 0))

```

今度は、`rand` の代わりに `rand-update` を使って同じ計算をしてみましょう。局所状態をモデル化するために代入を使わないとすると、このやり方をするしかありません。

```

(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))

```

プログラムはまだ単純ではありますが、残念ながらモジュール性が破られてしまっているところがいくつかあります。プログラムの最初のバージョンでは、`rand`を使って、任意の `experiment` 手続きを引数として取る汎用の `monte-carlo` 手続きとしてモンテカルロ法を直接表現することができました。プログラムの二つ目のバージョンでは、乱数生成器の局所状態がないので、`random-gcd-test` は乱数 `x1` と `x2` を明示的に操作し、`rand-update` に対する新たな入力として `x2` を繰り返しのループの間ずっと再利用する必要があります。このように乱数を明示的に扱うと、テスト結果を集積していくという構造と、この特定の試行では乱数を二つ使うという事実とが結びついてしまいます。しかし、ほかのモンテカルロ法の試行では、使う乱数の数が一つだったり三つだったりするかもしれません。トップレベルの手続き `estimate-pi` まで、乱数の初期値を提供することについて面倒を見なければなりません。乱数生成器の内部がプログラムのほかの部分に漏れてしまっているため、モンテカルロ法という考え方を分離して、ほかのタスクにも適用できるようにするということが難しくなっています。プログラムの最初のバージョンでは、代入が乱数生成器の状態を `rand` 手続きの中にカプセル化しているので、乱数生成の細かいところはプログラムのほかの部分から独立することができます。

モンテカルロ法の例から、次の一般的な現象がわかります。複雑なプロセスのある部品という立場から見ると、ほかの部品は時間とともに変化しているように見えます。それらの部品は、時とともに変化する隠れた局所状態を持っています。このような分解方法を反映した構造を持つコンピュータプログラムを書こうと思うと、ふるまいが時間とともに変わる計算オブジェクト（銀行口座や乱数生成器など）を作ることになります。状態は局所状態変数によってモデル化し、状態の変化はそれらの変数への代入としてモデル化します。

こうなると、次のように結論づけてしまいたくなるところです。代入と、状態をローカル変数に隠すというテクニックを導入することで、追加の引数を渡すことによって全ての状態を明示的に操作するよりも、よりモジュール性の高いスタイルでシステムを構造化できるようになった、と。残念ながら、これから見ていくように、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によって定積分を推定する手法である。領域内の点 (x, y) に対しては真となり、領域外の点に対しては偽となるような述語 $P(x, y)$ によって記述される領域の面積を計算する場合について考える。例えば、中心 $(5, 7)$ で半径が 3 である円に含まれる領域は、 $(x - 5)^2 + (y - 7)^2 \leq 3^2$ となるかどうかをテストする述語によって記述される。そのような述語によって記述

された領域の面積を推定するには、まずその領域を包含する長方形を選ぶ。例えば、向かい合う頂点が(2, 4)と(8, 10)にある長方形は、上で述べた円を含む。求める積分は、長方形の中でその領域の中に入る部分の面積である。その積分は、ランダムに長方形の中の点 (x, y) を選んでいき、それぞれの点について $P(x, y)$ をテストし、その点が元の領域の中にあるかどうかを決定することによって推定できる。もしこの試行を多くの点について行えば、領域の中に入る点の割合から、その領域の中にある長方形の割合を推定することができる。よって、その割合に長方形全体の面積をかけると、積分の推定値が得られることになる。

モンテカルロ積分を、手続き `estimate-integral` として実装せよ。引数として、述語 `P`、長方形の上下界 `x1, x2, y1, y2`、推定値を生成するための試行回数を取る。この手続きは、上で π を推定するために使ったものと同じ `monte-carlo` 手続きを使わなければならぬ。この `estimate-integral` を使って、単位円の面積を測ることによって π の推定値を求めよ。

この手続きを書くのには、与えられた範囲からランダムに選んだ数値を返す手続きがあれば便利だ。以下の `random-in-range` 手続きは、Section 1.2.6で使った、入力よりも小さい非負数を返す `random` 手続きによってそれを実装している。⁸

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

Exercise 3.6: 亂数生成器をリセットして、与えられた値から始まる数列を作れるようにできると便利だ。記号 `generate` または `reset` のどちらかを引数として取り、次のようなふるまいをする新しい `rand` 手続きを設計せよ。`(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>` にリセットする。つまり、状態をリセットすることによって、再現可能な数列が生成できる。これは乱数を使うプログラムをテストしたりデバッグしたりするのにとても役に立つ。

⁸MIT Scheme はそのような手続きを提供しています。`random` に (Section 1.2.6のように) 整数が渡されれば整数を返しますが、(この課題のように) 小数を渡されると小数を返します。

3.1.3 代入を導入することのコスト

ここまで見てきたように、`set!` 演算があると局所状態を持つオブジェクトをモデル化することができるようになります。しかし、この利点は代償を伴うものです。このことによって、私たちのプログラミング言語はSection 1.1.5で紹介した手続き適用の置換モデルによって解釈することがもうできなくなっています。それだけでなく、プログラミング言語のオブジェクトと代入を扱う適切な枠組みとして使えるような、“いい感じの”数学的性質を持った単純なモデルというものは存在しません。

代入を使わない限り、同じ引数による同じ手続きの呼び出しは二回評価しても同じ結果になるので、手続きは数学関数の計算と見なすことができます。この本の一章と二章を通してしてきたような、まったく代入を使わないプログラミングというのは、そのために*functional programming* (関数型プログラミング) と呼ばれています。

代入によってどれだけ話が難しくなるかということを、Section 3.1.1の`make-withdraw` 手続きを単純化したバージョンを使って考えてみましょう。このバージョンでは、残額が十分かどうかのチェックをしないようにしています。

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

この手続きと、`set!` を使わない次の`make-decrementer` 手続きとを比べてみてください。

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

`make-decrementer` は、指定された残高 `balance` から入力値を引く手続きを返します。しかし、続けて呼び出しても、`make-simplified-withdraw` のような集積効果はありません。

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

`make-decrementer` の動作の説明には置換モデルが使えます。例えば、以下の式の評価を解析してみましょう。

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

まず、`make-decrementer` 本体の `balance` を 25 によって置き換えて、組み合わせの演算子を単純にします。式は以下のようになります。

```
((lambda (amount) (- 25 amount)) 20)
```

`lambda` 式本体にある `amount` を 20 によって置き換えて、演算子を適用します。

```
(- 25 20)
```

最終的な答は 5 になります。

しかし、`make-simplified-withdraw` に対して同じような置換による解析をしようとするはどうなるでしょうか。

```
((make-simplified-withdraw 25) 20)
```

まず、`make-simplified-withdraw` の本体にある `balance` を 25 に置き換えて演算子を単純にします。すると、式は次のようになります。⁹

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

今度は、`lambda` 式本体の `amount` を 20 に置き換えて演算子を適用します。

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

置換モデルにあくまでこだわるのであれば、この手続き適用の意味は、まず `balance` を 5 に設定して、それから式の値として 25 を返すというものだ、と言わなければならないことになります。これは間違った答えです。正しい答えを得るには、何とかして一つ目の `balance(set!)` の効果の前) と二つ目の

⁹`set!` 式に出てくる `balance` は置き換えません。`set!` 内の `(name)` は評価されていないからです。これを置き換えるとすると、`(set! 25 (- 25 amount))` というおかしな結果になってしまいます。

`balance(set! の効果の後)` とを区別する必要があります。しかし、これは置換モデルではできません。

ここで問題になるのは、置換というものが究極的に、私たちの言語の記号というものが本質的に値に対する名前であるという概念に基づいているということです。しかし、`set!` を導入し、変数の値は変えられるものだという考え方を導入したとたん、変数はもはや単純に名前であるとはいえないなります。変数は、値が格納される場所を何らかの方法で参照するもので、その場所に格納された値は変えることができるということになります。[Section 3.2](#)では、私たちの計算モデルの中で、環境がこの“場所”という役割を勤めるということを見ていきます。

同一性と変化

ここで浮上してきた問題は、ある特定の計算モデルが破綻するというだけではなく、より奥の深いものです。私たちの計算モデルに変更を導入したとたん、これまで単純だった多くの概念に問題が出てきます。例えば、二つのものが“同じ”であるという概念について考えてみます。

同じ引数で `make-decrementer` を二回呼び、二つの手続きを作成するとします。

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

`D1` と `D2` は同じでしょうか。無難な答えは「同じ」です。`D1` と `D2` は同じ計算上のふるまい—どちらも入力から 25 を引くという手続きだということ—を持っているからです。実際、どんな計算であっても、結果を変えることなく `D2` の代わりに `D1` を使うことができます。

これと、`make-simplified-withdraw` を二回呼ぶ場合とを対比してみましょう。

```
(define W1 (make-simplified-withdraw 25))  
(define W2 (make-simplified-withdraw 25))
```

`W1` と `W2` は同じでしょうか。これは確実に違います。`W1` と `W2` の呼び出しは、次の対話でわかるように、異なる効果を持っています。

```
(W1 20)  
5  
(W1 20)
```

-15

(W2 20)

5

W1 と W2 は、(make-simplified-withdraw 25) という同じ式を評価して作られたという意味では“同じ”ですが、どんな式であっても、式の評価結果を変えずに W2 の代わりに W1 が使えるということはありません。

式の値を変化させることなく、式の中で“等しいものは等しいものによって置き換えることができる”という概念をサポートしている言語は、*referentially transparent*(参照透明) であると言われます。私たちのコンピュータ言語に set! を含めたときに、参照透明性は破られました。そのため、どういう場合に等価な式を置き換えることによって式を単純にできるかということを決めるのは厄介な問題になります。結果として、代入を使うプログラムについて推論を行うことは劇的に難しくなります。

参照透明性をあきらめると、計算オブジェクトが“同じ”であるというのはどういう意味かという概念は、形式的に捉えることが難しくなります。実際のところ、私たちのプログラムのモデル化対象である現実世界において“同じ”というのがどういう意味かということ自身、はっきりしているとはとても言えません。一般的に、見たところ同一に見えるオブジェクトが実際に“同じもの”であるかどうかは、片方のオブジェクトを変更してみて、もう一方のオブジェクトも同じように変更されるかどうかを観測することによって決めるしかありません。しかし、あるオブジェクトが“変化”したかどうかを調べるには、“同じ”オブジェクトを二回観測して、そのオブジェクトの属性が最初の観測と二回目の観測で違うかどうかを調べる以外に方法がありません。このため、“変化”を測定することは、“同じ”ということについての *a priori*(先驗的) な概念なしにはできません。そして、同じかどうかを測定することは、変化の効果を観測することなしにはできません。

プログラミングの中でこの問題が起こってくる例として、Peter と Paul が 100 ドル入っている口座を持っているという状況について考えます。これを次のようにモデル化するのと、

```
(define peter-acc (make-account 100))
(define paul-acc (make-account 100))
```

次のようにモデル化するのとでは、大きな違いがあります。

```
(define peter-acc (make-account 100))
(define paul-acc peter-acc)
```

一つ目の状況では、二つの銀行口座は違うものです。Peter による取引は Paul の口座には影響せず、その逆も同じです。しかし、二つ目の状況では、`paul-acc` は `peter-acc` と同じものになるよう定義しています。実質的に、この場合は Peter と Paul が共同口座を持っていることになり、Peter が `peter-acc` から引き出しがすると、Paul は `paul-acc` の残額が減っていることを観測することになります。これら二つの、似ているものの異なる状況は、計算モデルを構築するうえで混乱を招きます。共同口座があると、具体的には、ひとつのオブジェクト（銀行口座）で二つの異なる名前（`peter-acc` と `paul-acc`）を持つものがあるということが特に混乱を招くものになります。仮に、プログラムの中で `paul-acc` が変更される可能性がある場所をすべて検索しようとしているとすると、`peter-acc` を変更するのも忘れずに探さなければなりません。¹⁰

“同じ”、“変化”について上で述べたことに関して、もし Peter と Paul が銀行の残高を調べることしかできず、残高を変えられるような操作ができないとすると、二つの口座が異なるかどうかという問題は意味がなくなるということに注意してください。一般的に、もしデータオブジェクトを決して変更しないのであれば、複合データオブジェクトはその部品を合わせたものだと見なすことができます。例えば、有理数は分母と分子によって決定されます。しかしこの見方は、変化がある場合には有効でなくなります。その場合、複合データは、複合データを構成する部品を合わせたものとは少し違う“アイデンティティ”を持つことになります。銀行口座は、引き出しを行って残高を変更しても、依然として“同じ”銀行口座です。その反対に、同じ状態情報を持つ二つの異なる銀行口座を作ることもできます。この複雑性の原因となっているのは、私たちのプログラミング言語ではなく、私たちが銀行口座をオブジェクトとして捉えているということです。例えば、有理数については、分子を変更しても“同じ”有理数であり続けることができるようなアイデンティティを持つ変更可能なオブジェクトを見なしたりすることは普通ありません。

¹⁰ ひとつの計算オブジェクトが二つ以上の名前によってアクセスされるという現象は、aliasing（エイリアス）と呼ばれます。共同口座の状況は、エイリアスのとて単純な例を示しています。Section 3.3では、“別の”複合データ構造が部品を共有するといった、より複雑な例について学びます。もし、二つの“異なる”オブジェクトが実際はひとつのオブジェクトに異なるエイリアスがついたものであることによって、あるオブジェクトを変更すると“副作用”として“異なる”オブジェクトを変更してしまうことがあるということを忘れていると、プログラムにバグが起こったりします。このような、いわゆる side-effect bugs(副作用バグ) というものは、特定が難しく解析も難しいため、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するべきだと提案する人もいます。(Lampson et al. 1981; Morris et al. 1980)

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対照的に、代入を多用するプログラミングは*imperative programming*（命令型プログラミング）といいます。計算モデルのいろいろな問題を引き起こすだけでなく、命令型スタイルで書いたプログラムは、関数型プログラムでは起こり得ないバグが起こりやすくなります。例えば、Section 1.2.1の反復階乗プログラムのことを思い出してみましょう。

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

内部の反復ループで、引数を渡す代わりに、変数 `product` と `counter` の値を更新するのに明示的な代入を使うことで、より命令型のスタイルにすることができます。

```
(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、プログラムの生成する結果を変えるわけではありません。しかし、ここで微妙な罠が入り込んでいます。代入の順序はどうやって決めたらいいのでしょうか。今回の場合、プログラムはたまたまこのままで正しいものになっています。しかし、代入を逆の順番で書いてしまうと、

```
(set! counter (+ counter 1))
(set! product (* counter product))
```

別の、間違った答えになるところです。一般的に、代入のあるプログラミングをする際には、代入の相対的な順番を注意深く考えて、それぞれの文が、中身

の変わった各変数の正しいバージョンを使うように気をつけないといけません。この問題は、関数型プログラミングでは最初から起こりません。¹¹

命令型プログラミングの複雑性は、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と、ますますひどいことになります。この点についてはSection 3.4で見ていくことにします。ここでは、まずは代入を含む式について計算モデルを与えるという問題に取り組み、またシミュレーションの設計をするのに局所状態を持つオブジェクトを使うことについて検討します。

Exercise 3.7: Exercise 3.3で述べた、パスワード機能を追加した make-account で作った銀行口座オブジェクトについて考える。私たちの銀行システムでは、共同口座を作る能力が必要だとする。これを実現する手続き make-joint を定義せよ。make-joint は三つの引数を取る。一つ目はパスワード保護された口座である。二つ目の引数は口座を定義したときのパスワードと一致している必要があり、そうでなければ make-joint 演算を進めることはできない。三番目の引数は新しいパスワードである。make-joint は、元の口座に新しいパスワードでもアクセスできるようにする。例えば、peter-acc が open-sesame というパスワードを持つ銀行口座だとすると、

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

このようにすることで、paul-acc という名前と rosebud というパスワードによって peter-acc に対する取引ができるようになる。この新しい機能に対応するには、Exercise 3.3の解答を修正するのがよいかもしない。

Exercise 3.8: Section 1.1.3で評価モデルを定義したときに、式を評価する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

¹¹このことを考えると、初步的なプログラミングを教える際に、非常に命令型に偏ったスタイルがよく使われるというのは皮肉なことです。これは、1960 年代と 1970 年代を通して一般的だった、手続きを呼ぶプログラムは代入を行うプログラムに比べて本質的に非効率的だ (Steele 1977 はそれが誤りだと証明しています) という信念の名残かもしれません。あるいは、初心者にとってはステップごとの代入のほうが手続きの呼び出しよりもイメージしやすいという考え方によるものかもしれません。理由はどうあれ、このことによって初級プログラマに “この変数の設定は、あの変数より前にしたほうがいいのか、後にしたほうがいいのか” という心配をさせることになり、プログラミングを複雑にして、重要な考え方を見えにくくさせてしまうということがよくあります。

た。しかし、部分式を評価する順番(例えば、左から右、または右から左)については規定していませんでした。代入を導入すると、手続きの引数を評価する順番によって結果が変わることが起こります。単純な手続き f を定義して、次の式

```
(+ (f 0) (f 1))
```

を評価する際に、 $+$ の引数の評価順が左から右であれば 0 を返し、右から左であれば 1 を返すようにせよ。

3.2 評価の環境モデル

Chapter 1で複合手続きを導入した際には、手続きを引数に適用するということの意味を定義するため、評価の置換モデル (Section 1.1.5) を使いました。

- 複合手続きを引数に適用するには、手続きの本体に出てくる仮引数を対応する引数で置き換えて、それを評価する。

プログラミング言語で代入を認めることにすると、このような定義はもはや適切ではなくなります。特に、Section 3.1.3では、代入があると変数はもはや単なる値に対する名前であるとは言えなくなるということを示しました。変数は、値が格納できる“場所”を何らかの方法で指していなければならないことになります。新しい評価モデルでは、これらの場所は *environments*(環境) と呼ばれる構造で保存されます。

環境というのは *frames*(フレーム) が並んだものです。それぞれのフレームは、変数の名前と、それに対応する値を結び付ける *bindings*(束縛) のテーブル(空のこともある)です(ひとつのフレームは、任意の変数に対して高々ひとつの束縛を持っています)。それぞれのフレームは、そのほかに *enclosing environment*(外側の環境)へのポインタも持っています。例外として、ここで説明のために *global*(グローバル) と呼ぶものは外側の環境へのポインタを持ちません。ある環境についての *value of a variable* (変数の値) とは、環境の中でその変数に対する束縛を含む最初のフレームの中で、その変数の束縛によって与えられる値ということになります。もしフレームの列の中にその変数に対する束縛を規定しているものがなければ、その変数はその環境の中で *unbound*(未束縛) であるといいます。

Figure 3.1は、三つのフレームからなる簡単な環境の構造を示したものです。フレームにはそれぞれ、I, II, III というラベルがついています。この図の

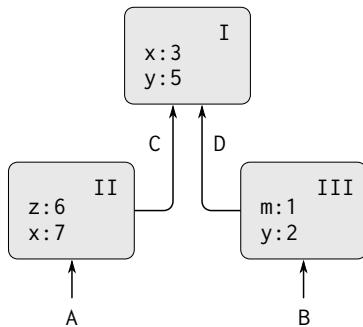


Figure 3.1: 単純な環境構造

中で、A, B, C, D は環境へのポインタです。C と D は同じ環境を指しています。変数 z と x がフレーム II に束縛されていて、 y と x がフレーム I に束縛されています。環境 D の中では、 x の値は 3 です。環境 B についても、 x の値は 3 です。これは次のように決定されています。まず、列の最初のフレーム (フレーム III) を調べます。 x に対する束縛は見つからないので、外側の環境 D に進み、フレーム I で束縛が見つかります。一方、環境 A では x の値は 7 です。列の最初のフレーム (フレーム II) が x と 7 の束縛を含んでいるからです。環境 A について、フレーム II の x と 7 の束縛が、フレーム I の x と 3 の束縛を *shadow(隠蔽)* しているといいます。

環境は評価プロセスにとって不可欠なものです。式を評価するコンテキストを決めるのは環境だからです。プログラミング言語の式は、それ自身意味を持たないとも言えます。式は、式が評価される環境があって初めて意味を持つことになります。 $(+ 1 1)$ のような単純な式を解釈するのにも、 $+$ が加算の記号だというコンテキストの中で演算をしていると理解していることが必要になります。そのため、私たちの評価モデルの中では、式の評価について語るときには、必ず環境についても語ることにします。また、インターフェリタとの対話について説明するために、グローバル環境というものがあると想定します。グローバル環境は (外側の環境のない) 単一のフレームからなり、そのフレームは記号の値として基本手続きとの関連づけを含んでいるとします。例えば、 $+$ が加算の記号であるということは、記号 $+$ はグローバル環境の中で基本要素の加算手続きに束縛されているというように捉えます。

3.2.1 評価規則

インタプリタが組み合わせを評価するやり方に関する全体的な仕様は、最初にSection 1.1.3で導入したものと同じものになります。

- 組み合わせを評価するには、
 1. 組み合わせの部分式を評価する。¹²
 2. 演算子となっている部分式の値を、オペランドとなっている部分式の値に適用する。

評価の環境モデルは、複合手続きを引数に適用することの意味を規定することによって、置換モデルに取って代わります。

評価の環境モデルでは、手続きは常に、何らかのコードと、ある環境へのポインタというペアになります。手続きを作る方法はひとつだけです。その方法とは、 λ 式を評価することです。 λ 式を評価することによって手続きが作られます。が、 λ 式のテキストが手続きのコードとなり、 λ 式が評価されたときの環境が手続きの環境になります。例えば、次の手続きの定義が、

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

グローバル環境の中で評価される場合について考えてみましょう。この手続き定義の文法は、その下にある暗黙の λ 式に対するシンタックスシュガーにすぎません。次のものを使っても同じことです。

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

これは、グローバル環境の中で `(lambda (x) (* x x))` を評価し、グローバル環境の中で `square` とその結果を束縛するものです。

Figure 3.2は、この`define`式を評価した結果を示したものです。手続きオブジェクトはペアであり、そのコード部分は、手続きが`x`という仮引数をひと

¹²代入があると、ステップ1に難しいところが出てきます。Exercise 3.8で示したように、代入があると、組み合わせの部分式がどういう順番で評価されるかによって違う値を返すような式が書けるようになるからです。そのため、正確には、ステップ1の評価順序(例えば、左から右や右から左)を規定しておく必要があります。しかし、この順序は常に実装の細部として捉えられるべきものであって、特定の順序に依存したプログラムは決して書いてはいけません。例えば、よくできたコンパイラであれば、部分式の評価順序を変えることによってプログラムを最適化したりすることもありえるからです。

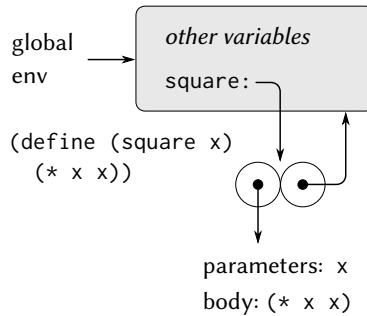


Figure 3.2: グローバル環境の中で `(define (square x) (* x x))` を評価することによって作られる環境

つ持つということと、手続きの本体が `(* x x)` であるということを規定しています。手続きの環境部分はグローバル環境へのポインタになります。 λ 式を評価して手続きを作ったのがグローバル環境の中だったからです。手続きオブジェクトと記号 `square` を結びつける束縛は、グローバルなフレームに追加されています。一般的に、`define` はフレームに束縛を追加することによって定義を作成します。

手続きの作られ方がわかったので、手続きがどのように適用されるかということも説明できるようになります。環境モデルは、以下のことを規定します。手続きを引数に適用するには、仮引数を引数の値に束縛するフレームを含む新しい環境を作ります。このフレームの外側の環境は、手続きによって規定される環境です。そして、この新しい環境の中で、手続きの本体を評価します。

この規則が実際にどのように適用されるか見てみましょう。Figure 3.3は、グローバル環境で `(square 5)` という式を評価することによって作られる環境構造を示したものです。ここで、`square` はFigure 3.2で生成された手続きです。手続きを適用すると新しい環境が作れます。図の中では `E1` というラベルをつけています。この環境の最初のフレームでは、手続きの仮引数である `x` が、引数の `5` に束縛されています。このフレームから上に伸びている矢印のポインタは、このフレームの外側の環境がグローバル環境であるということを示しています。ここでグローバル環境が選ばれているのは、`square` 手続きオブジェクトの環境部分が指しているのがグローバル環境だからです。E1 の中に、手

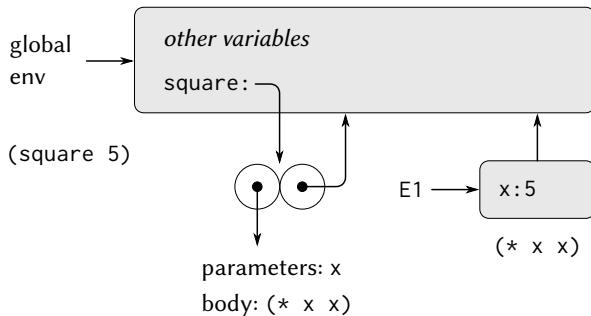


Figure 3.3: グローバル環境で `(square 5)` を評価することによって作られる環境

続きの本体である `(* x x)` を評価します。E1 の中の `x` の値は 5 であるため、結果は `(* 5 5)`、つまり 25 になります。

手続き適用の環境モデルを要約すると、次の二つの規則になります。

- 引数セットへの手続きオブジェクトの適用は、フレームを構築し、手続きの仮引数を呼び出しの引数に束縛し、それから新しく構築された環境という文脈の中で手続きの本体を評価することによって行う。新しいフレームは、適用する手続きオブジェクトの環境部分を外側の環境として持つ。
- 手続きは、与えられた環境について入式を評価することによって作る。結果としてできる手続きオブジェクトは、入式のテキストと、手続きが作成された環境へのポインタからなるペアである。

このほかに、`define` を使って記号を定義すると、現在の環境フレームの中に束縛を作り、その記号に指定の値を割り当てるということも規定します。¹³ 最後に、`set!` のふるまいを規定します。そもそも、環境モデルを導入しなければなら

¹³もしもその変数に対する束縛が現在のフレームにすでにあれば、その束縛は変更されます。こうすると記号が再定義できるので便利です。しかし、それができるということは、`define` を使って値の変更ができるということになります。つまり、明示的に `set!` を使っていないのに、代入に関する問題が出てくることになります。このため、すでに存在する記号を再定義しようとするとエラーや警告を出すようにしてほしいという人もいます。

なくなつたのはこの演算のせいでした。ある環境で、`(set! <variable> <value>)` という式を評価すると、その環境の中でのその変数に対する束縛を探し、その束縛を変更して新しい値を指すようにします。つまり、その変数に対する束縛を含む、その環境の中での最初のフレームを探し、そのフレームを修正するということです。もしその変数がその環境の中で束縛されていなければ、`set!` はエラーを発生させます。

これらの評価規則は、置換モデルよりはだいぶ複雑ですが、まだそれなりに単純です。それに、この評価モデルは抽象的ではあるものの、インタプリタが式を評価するやり方を正しく描写しています。[Chapter 4](#)では、ちゃんと動くインタプリタを実装するのに、このモデルが青写真となってくれるということを見ていきます。次の節では、いくつかの具体例としてのプログラムを解析していくことで、このモデルの詳細について解説していきます。

3.2.2 単純な手続きの適用

[Section 1.1.5](#)で置換モデルを導入した際に、以下のような手続き定義があるとき、`(f 5)` という組み合わせを評価した結果がどのようにして 136 になるかということを示しました。

```
(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)))
```

同じ例を環境モデルを使って分析することもできます。[Figure 3.4](#)は、グローバル環境で `f`, `square`, `sum-of-squares` の定義を評価することによって作られる三つの手続きオブジェクトを示したものです。それぞれの手続きオブジェクトは、何かしらのコードとグローバル環境へのポインタから構成されています。

[Figure 3.5](#)には、式 `(f 5)` を評価することによって作られる環境構造を示してあります。`f` を呼び出すことによって `E1` という新しい環境が作られ、その中のフレームで `f` の仮引数である `a` が引数の 5 に束縛されます。`E1` の中で、`f` の本体を評価します。

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

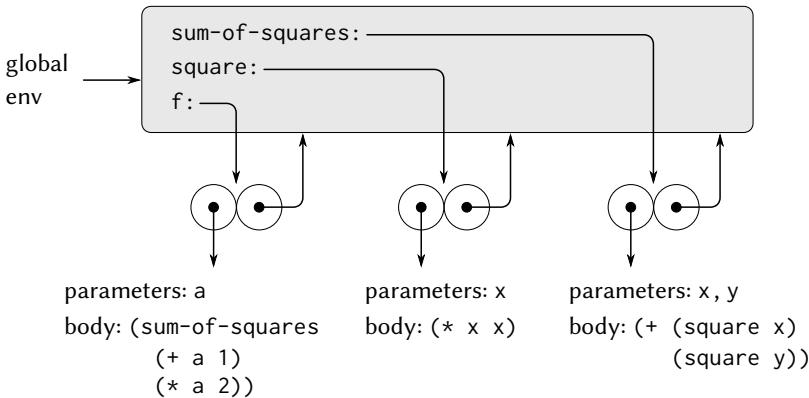


Figure 3.4: グローバルフレーム内の手続きオブジェクト

この組み合わせを評価するには、まず部分式から評価します。最初の部分式 `sum-of-squares` は、値として手続きオブジェクトを持っています（この値の見つけ方に注意してください。まず E1 の最初のフレームを見ますが、そこには `sum-of-squares` に対する束縛がありません。次に、外側の環境、つまりグローバル環境に進み、Figure 3.4 に示した束縛が見つかります）。残り二つの部分式の評価は、基本演算 `+` と `*` を適用して `(+ a 1)` と `(* a 2)` という二つの組み合わせを評価することによって行えます。結果はそれぞれ 6, 10 になります。

今度は、手続きオブジェクト `sum-of-squares` を引数 6, 10 に適用します。その結果、新しい環境 E2 が作られ、その中で仮引数 `x, y` がそれらの引数に束縛されます。E2 の中で、`(+ (square x) (square y))` という組み合わせを評価します。すると、次は `(square x)` を評価することになります。`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 を用意して、`x` を 6 に束縛し、E3 の中で `square` の本体 `(* x x)` を評価します。また、`sum-of-squares` の適用の中では、`(square y)` という部分式も評価する必要があります。ここで、`y` は 10 です。この二回目の `square` の呼び出しでは、また別の環境 E4 が作られ、その中で `square` の形式パラメタ `x` が 10 に束縛されます。そして、E4 の中で `(* x x)` を評価しなければなりません。

注意すべき重要な点は、`square` を呼び出すたびに、`x` の束縛を含む新しい環境が作られるということです。ここから、それぞれのフレームが、どれも `x`

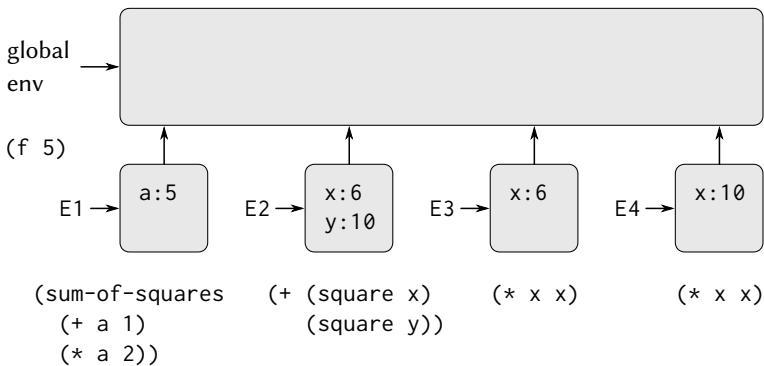


Figure 3.5: Figure 3.4内の手続きを用いて `(f 5)` を評価することで作られた環境

という名前を持つ異なるローカル変数をそれぞれ別々にしておくのに役に立っているということがわかります。`square` によって作られたそれぞれのフレームがグローバル環境を指していることに注意してください。これは、`square` の手続きオブジェクトが指している環境がグローバル環境だからです。

部分式が評価された後には、その結果が返されます。`square` の二回の呼び出しによって生成された値は、`sum-of-squares` によって加算され、その結果が `f` によって返されます。ここで私たちが着目しているのは環境構造なので、これらの返される値がどのように複数の呼び出しの間で渡されていくかということには深入りしません。しかし、これも評価プロセスの重要な側面のひとつです。この点については、Chapter 5で詳しく見ていきます。

Exercise 3.9: Section 1.2.1では、階乗を計算する二つの手続きを解析するのに置換モデルを使った。再帰版は以下の通りで、

```

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

反復版は以下の通りである。

```

(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)))
```

それぞれの `factorial` 手続きを使って (`factorial 6`) を評価することによって作られる環境構造を示せ。¹⁴

3.2.3 局所状態の保管場所としてのフレーム

局所状態を持つオブジェクトを表現するのに手続きと代入を使うやり方について考えるのには環境モデルが役に立ちます。ひとつの例として、次の手続きを呼び出すことによって作られる Section 3.1.1 の “引き出し処理器” について考えてみましょう。

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

次の式を評価し、

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

続けて次の式を評価する場合について説明します。

```
(W1 50)
```

```
50
```

Figure 3.6は、グローバル環境で `make-withdraw` 手続きを定義した結果を示しています。この定義は、グローバル環境へのポインタを持つ手続きオブジェクトを作成します。ここまでこれまで見た例と同じですが、手続きの本体がそれ自身入式であるというところが違います。

¹⁴Section 1.2.1で述べた、インタプリタは `fact-iter` のような手続きを末尾再帰を使うことによって一定の空間量で実行できるという主張は、環境モデルでは曖昧にしておきます。末尾再帰については、Section 5.4でインタプリタの制御構造を扱う際に考察します。

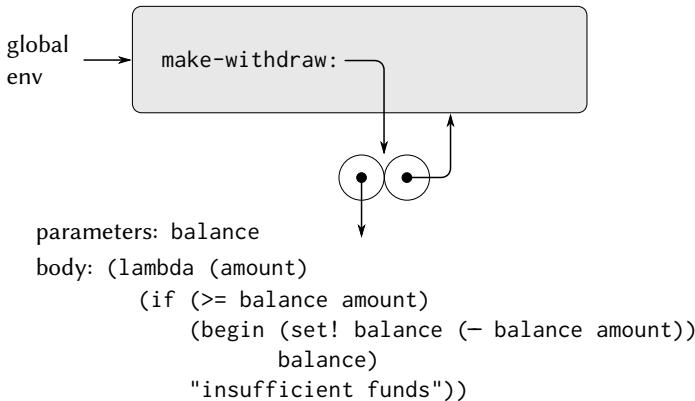


Figure 3.6: グローバル環境で `make-withdraw` を定義した結果

計算の面白いところは、手続き `make-withdraw` を引数に適用するときに起こります。

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

まず、いつも通り環境 E1 を立ち上げ、その中で仮引数 `balance` を引数の 100 に束縛します。この環境の中で、`make-withdraw` の本体、つまり入式を評価します。すると、新しい手続きオブジェクトが構築され、そのコード部分は `lambda` で規定されるものとなり、環境部分は、`lambda` が評価された環境である E1 になります。結果としてできる手続きオブジェクトは、`make-withdraw` の呼び出しに対して返される値になります。これがグローバル環境で `W1` に束縛されます。これは、`define` 自身がグローバル環境で評価されているからです。[Figure 3.7](#) に最終的な環境構造を示します。

これで、`W1` が引数に適用されるとどうなるか解析できるようになります。

```
(W1 50)
50
```

まずフレームを構築し、その中で `W1` の仮引数 `amount` を引数 50 に束縛します。注意すべき重要な点は、このフレームが持っている外側の環境が、グロ-

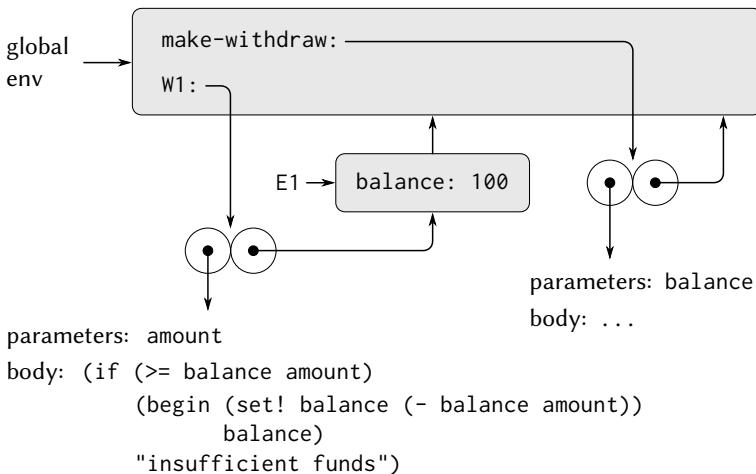


Figure 3.7: (`define W1 (make-withdraw 100)`) を評価した結果

バル環境ではなく環境 E1 だというところです。これは、W1 手続きオブジェクトによって規定されている環境が E1 だからです。この新しい環境の中で、手続きの本体を評価します。

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

結果の環境構造をFigure 3.8に示します。評価されている式は、`amount` と `balance` の両方を参照しています。`amount` は環境の最初のフレームで見つかり、`balance` は外側の環境のポインタをたどることによって E1 で見つかります。

`set!` が実行されると、E1 中の `balance` の束縛が変更されます。W1 の呼び出しが終わるときには `balance` は 50 になっていて、`balance` を含むフレームはそれまで通り手続きオブジェクト W1 によって指されています。`balance` を変更するコードを実行した、`amount` を束縛しているフレームは、もういらなくなりました。このフレームを構築することになった手続きの呼び出しは終了

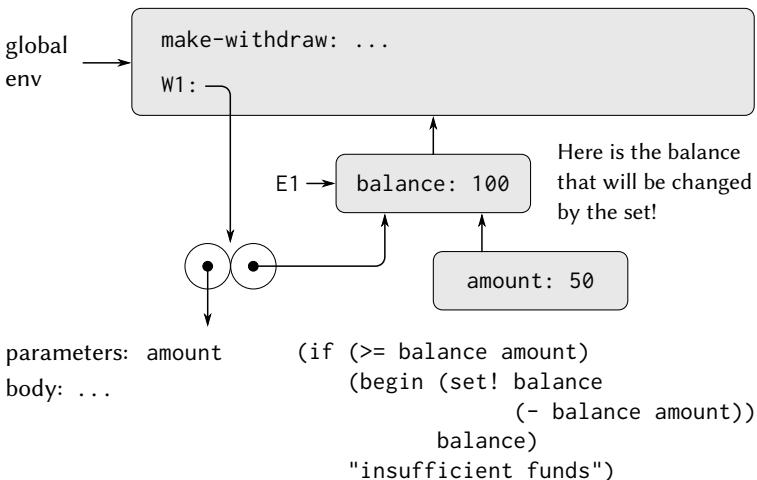


Figure 3.8: 手続きオブジェクト W_1 を適用することによって作成される環境

していますし、環境のほかの部分からこのフレームを指しているところもないからです。次に W_1 が呼ばれるときには、新しいフレームが構築されて $amount$ を束縛し、その外側の環境は E_1 になります。手続きオブジェクト W_1 の局所状態を持っておく“場所”としての役割を E_1 が果たしているということがわかります。Figure 3.9に、 W_1 を呼び出した後の様子を示します。

もう一度 $make-withdraw$ を呼び出して、二つ目の“withdraw”オブジェクトを作るとどうなるか見てみましょう。

```
(define W2 (make-withdraw 100))
```

こうすると、Figure 3.10のような環境構造が生成されます。この図は、 W_2 が手続きオブジェクトであるということ、つまり何らかのコードとひとつの環境のペアであるということを示しています。 W_2 の環境 E_2 は、 $make-withdraw$ を呼び出すことによって作られます。この環境は、 $balance$ に対する局所的な束縛を持つフレームを含んでいます。一方、 W_1 と W_2 のコードは同じです。ど

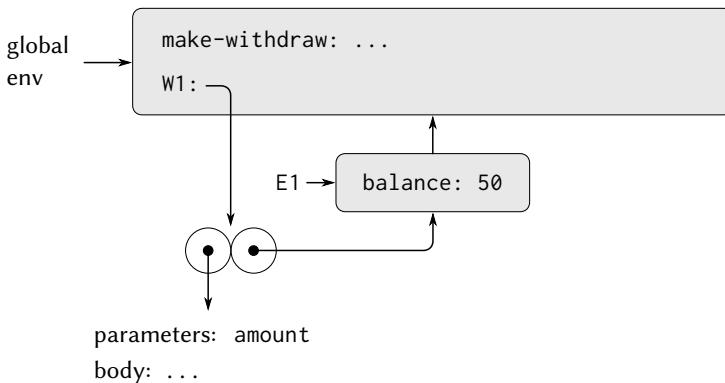


Figure 3.9: W1呼び出し後の環境

ちらも、`make-withdraw`本体の`λ`式によって規定されるコードです。¹⁵ これで、なぜ `W1` と `W2` が独立したオブジェクトとしてふるまうのかがわかりました。`W1` の呼び出しは `E1` に格納された状態変数 `balance` を参照し、`W2` の呼び出しは `E2` に格納された `balance` を参照しています。このため、一方のオブジェクトの局所状態に変更を加えても、もう一方のオブジェクトには影響がありません。

Exercise 3.10: `make-withdraw` 手続きでは、ローカル変数 `balance` は `make-withdraw` の仮引数として作られる。局所状態変数は、`let` を使って、以下のように明示的に作ることもできる。

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

¹⁵`W1` と `W2` はコンピュータ内の同じ物理コードを共有しているのか、それともそれぞれコードのコピーを持っているのかということは、実装上の詳細に属することです。Chapter 4で実装するインタプリタでは、コードは実際には共有されることになります。

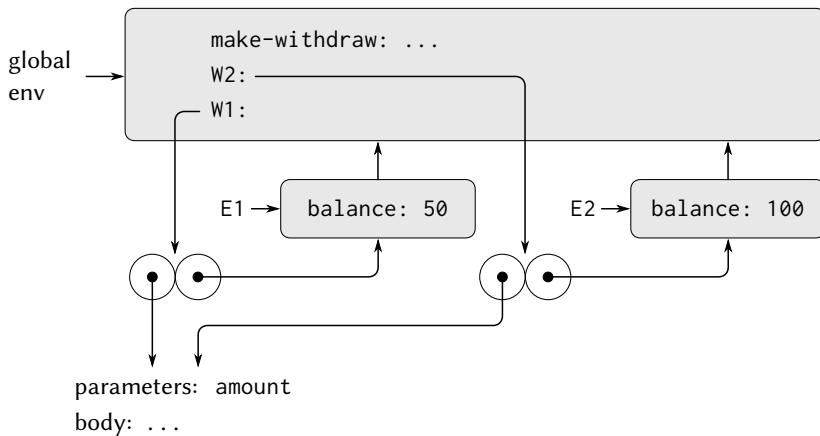


Figure 3.10: `(define W2 (make-withdraw 100))` を使って二つ目のオブジェクトを作成

`let` は手続き呼び出しに対する単純なシンタックスシュガーだという [Section 1.3.2](#) の内容を思い出そう。

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

上記の式は、次の式の別の書き方として解釈される。

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

環境モデルを使って、この `make-withdraw` の別のバージョンを解析し、上のような図を描いて次の対話を説明せよ。

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

`make-withdraw` のどちらのバージョンも同じふるまいを持つオブジェクトを作るということを示せ。環境構造は二つのバージョンの間でどのように異なるだろうか。

3.2.4 内部定義

Section 1.1.8では、手続きが内部定義を持つことができるという考えを導入し、それによって、次の平方根を計算する手続きに出てくるようなブロック構造が作れるようになりました。

```
(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))
```

今回は、これらの内部定義がどうして望み通りにふるまうのかを見るのに環境モデルを使うことができます。Figure 3.11は、式(`sqrt 2`)を評価する途中の、`guess`を1として内部手続き`good-enough?`に対する初回の呼び出しを行った時点の様子を示しています。

環境構造をよく見てください。`sqrt`はグローバル環境の中にある記号で、手続きオブジェクトに束縛されています。その手続きオブジェクトに関連づけられている環境はグローバル環境です。`sqrt`が呼ばれると、グローバル環境の下に新しい環境E1が作られます。その中では、仮引数`x`が2に束縛されています。それから、`sqrt`の本体がE1の中で評価されます。`sqrt`の本体の最初の式は以下の通りなので、

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

この式を評価するときに、手続き`good-enough?`が環境E1の中に定義されます。より正確に言うと、記号`good-enough?`がE1の最初のフレームに追加され、環境E1に関連づけられた手続きオブジェクトに束縛されるということになります。同じように、`improve`と`sqrt-iter`がE1の中で手続きとして定義されます。簡単にするために、Figure 3.11では`good-enough?`に対する手続きオブジェクトだけを示しています。

局所手続きが定義された後に、式(`sqrt-iter 1.0`)が評価されます。この式の評価が行われる環境は、やはりE1です。そのため、E1の中で`sqrt-`

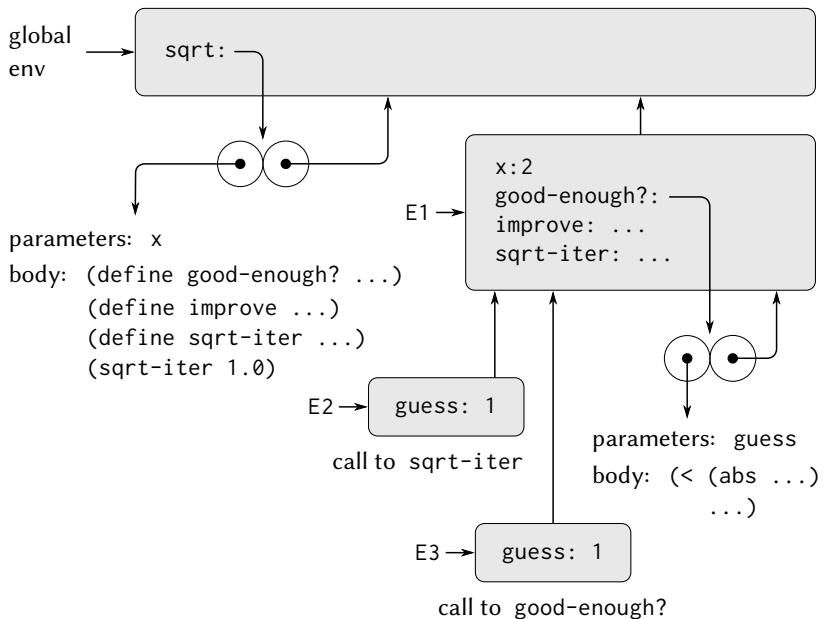


Figure 3.11: 内部定義を持つ `sqrt` 手続き

`iter` に束縛されている手続きオブジェクトが、1 を引数として呼び出されます。この呼び出しで環境 `E2` が作られ、その中では `sqrt-iter` の仮引数 `guess` が 1 に束縛されています。`sqrt-iter` は、(E2 の) `guess` の値を引数として `good-enough?` を呼び出します。こうすると別の環境 `E3` が構築され、`guess(good-enough? の引数)` は 1 に束縛されます。`sqrt-iter` と `good-enough?` はどちらも `guess` という名前の仮引数を持っているですが、これらは別々のフレームにある別々のローカル変数です。また、`E2` と `E3` は両方とも外側の環境として `E1` を持っています。手続き `sqrt-iter` と `good-enough?` はどちらも環境部分として `E1` を持っているためです。その結果、`good-enough?` の本体の中の記号 `x` は、`E1` に出てくる `x` の束縛を参照することになります。これは、元の `sqrt` 手続きの呼び出しに使われた `x` の値ということになります。

このように、環境モデルを使うと、プログラムをモジュール化する便利なテクニックとして局所手続きの定義を使ううえでの二つのキーとなる性質について説明することができます。

- 局所手続きの名前は、それらを囲む手続きの外側の名前と衝突しない。局所手続きの名前は、グローバル環境に束縛されるのではなく、手続きが実行されるときに作られるフレームに束縛されるからである。
- 局所手続きは、単純に仮引数の名前を自由変数として使うだけで、それらを囲む手続きの引数にアクセスできる。これは、局所手続きの本体が、外側の手続きの評価環境に属する環境で評価されるからである。

Exercise 3.11: Section 3.2.3では、局所状態を持つ手続きのふるまいがどのように環境モデルによって説明されるかを見た。ここでは、内部定義がどのように動くのかを見た。典型的なメッセージパッシング手続きは、これら両方の側面を持っている。Section 3.1.1の銀行口座手続きについて考えてみよう。

```
(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)
```

次の対話によって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保存されているだろうか。次のように、別の口座を定義したとする。

```
(define acc2 (make-account 100))
```

二つの口座の局所状態は、どのようにして別々に保存されることになるだろうか。acc と acc2 では、環境構造のどの部分が共有されているだろうか。

3.3 可変データによるモデル化

Chapter 2では、複合データを複数の部品からなる計算オブジェクトを作る手段として使う方法について取り上げました。これは、複数の側面を持つ現実世界のオブジェクトをモデル化するためのものでした。また、Chapter 2ではデータ抽象化の規律を導入しました。その規律に従うと、データ構造は、データオブジェクトを作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によって規定することになっていました。しかし今では、データというものにはChapter 2では触れなかつたような別の側面があるということがわかっています。状態が変化するオブジェクトからなるシステムをモデル化しようとすると、複合データを構築したり選択を行ったりするだけでなく、それらを修正する必要も出てきます。状態の変化する複合オブジェクトをモデル化するために、ここでセレクタとコンストラクタに加えて、データオブジェクトを変更する *mutators*(ミューテータ) と呼ばれる演算を含めたデータ抽象化を設計することにします。例えば、銀行システムをモデル化しようとすると、口座残高を変更する必要があります。このため、銀行口座を表すデータ構造は、例えば次のように指定された口座の残高を指定された新しい値に変更する演算ができるようにすることが考えられます。

```
(setbalance! <account> <newvalue>)
```

ミューテータが定義されているデータオブジェクトは、*mutable data objects* (可変データオブジェクト) と呼ばれます。

Chapter 2では、複合データを合成するための汎用的な“糊”としてペアを導入しました。この節では、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、可変データオブジェクトを構築するためのブロックとしてペアを使うことができるようになります。これらのミュータータはペアの表現力を大幅に拡張し、それによってSection 2.2で取り組んだ列や木構造以外のデータ構造も作れるようになります。また、シミュレーションの例もいくつか取り上げます。それらの例では、複雑なシステムを局所状態を持つオブジェクトの集合としてモデルすることになります。

3.3.1 可変リスト構造

ペアに対する基本演算—`cons`, `car`, `cdr`—は、リスト構造を構築し、またリスト構造から部品を選択するために使うことができます。しかし、これらはリスト構造を変更することはできません。これまで学んできた、`append` や `list` のようなリスト演算についても同じことが言えます。これらは、`cons`, `car`, `cdr` によって定義できるものだからです。リスト構造を変更するには、新しい命令が必要となります。

ペアに対する基本ミュータータは `set-car!` と `set-cdr!` です。`set-car!` は二つの引数を取り、一つ目の引数はペアでなければなりません。`set-car!` はこのペアを変更し、`car` ポインタを `set-car!` の第二引数へのポインタに置き換えます。¹⁶

例として、Figure 3.12に示すように、`x` がリスト `((a b) c d)` に束縛され、`y` がリスト `(e f)` に束縛されているという場合について考えます。式 `(set-car! x y)` を評価すると、`x` が束縛されているペアが変更され、その `car` を `y` の値で置き換えます。演算の結果をFigure 3.13に示します。構造 `x` は変更され、今では `((e f) c d)` と表示されるような構造になっています。置き換えられたポインタが指していたリスト `(a b)` を表す二つのペアは、今では元の構造から切り離されています。¹⁷

Figure 3.13とFigure 3.14を見比べてみてください。後者は、`x` と `y` が Figure

¹⁶`set-car!` と `set-cdr!` の返す値は実装依存です。`set!` 同様、これらの命令は効果のために使うものです。

¹⁷ここから、リストに対して変更演算を行うと、アクセス可能な構造の部品にならない“garbage”（ゴミ）ができるかもしれないということがわかります。Section 5.3.2では、Lispのメモリ管理システムは *garbage collector*（ガベージコレクタ）というものを含んでいて、それによって必要のないペアが使っていたメモリ空間をリサイクルということを学びます。

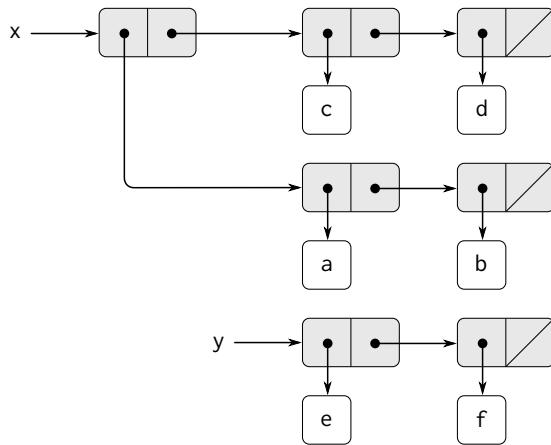


Figure 3.12: リスト $x: ((a\ b)\ c\ d)$ と $y: (e\ f)$

3.12の元のリストに束縛されている状態で `(define z (cons y (cdr x)))` を実行した結果を示したもので、変数 z は `cons` 演算によって作られた新しいペアに束縛されるようになっていますが、 x が束縛されているリストは変化していません。

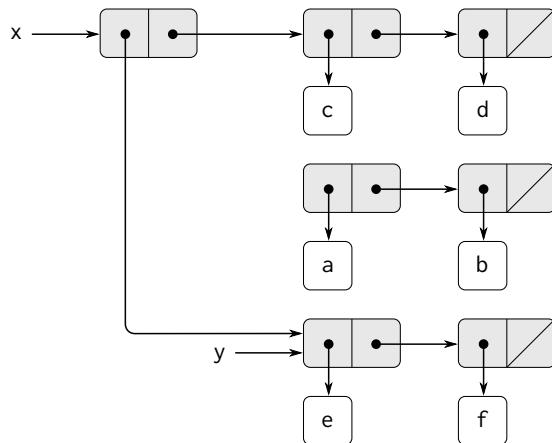


Figure 3.13: Figure 3.12のリストに対する (`set-car! x y`) の効果

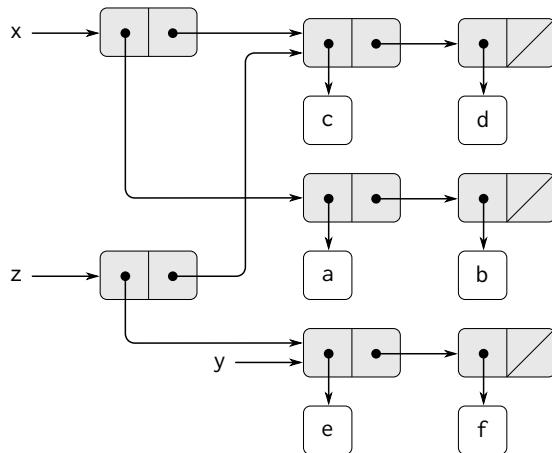


Figure 3.14: Figure 3.12のリストに対する (`define z (cons y (cdr x))`) の効果

`set-cdr!` 演算は `set-car!` と似ています。唯一の違いは、置き換えるのがペアの `car` ポインタではなく `cdr` ポインタだということです。[Figure 3.12](#) のリストに対して `(set-cdr! x y)` を実行したときの効果を [Figure 3.15](#) に示します。ここでは、`x` の `cdr` ポインタが `(e f)` に対するポインタに置き換えられています。また、`x` の `cdr` だったリスト `(c d)` は、今では構造から取り外されています。

`cons` は新しいペアを作ることによって新しいリスト構造を組み立てるものですが、`set-car!` と `set-cdr!` は既存のペアを修正するものです。実際のところ、これら二つのミューテータと、既存のリスト構造の一部でない新しいペアを返す `get-new-pair` という手続きがあれば、それらによって `cons` を実装することもできます。まず新しいペアを取得し、指定されたオブジェクトを `car` と `cdr` ポインタに設定し、その新しいペアを `cons` の結果として返します。¹⁸

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

¹⁸ `get-new-pair` は、Lisp の実装に必要となるメモリ管理の一部として実装する必要がある演算のひとつです。これについては [Section 5.3.1](#) で検討します。

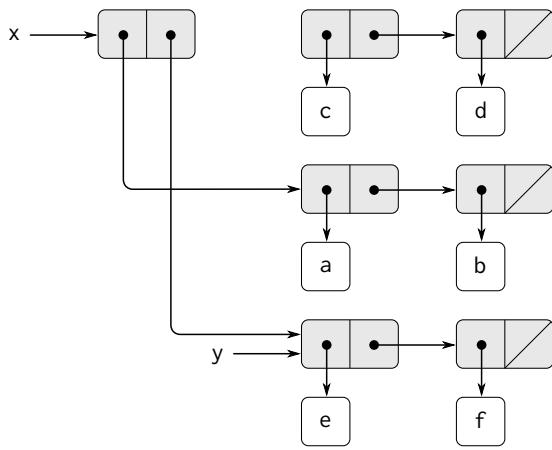


Figure 3.15: Figure 3.12のリストに対する (`set-cdr! x y`) の効果

Exercise 3.12:

Section 2.2.1では、次のリスト追加手続きを紹介した。

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

appendは、xの要素を連続でyにconsしていくことによって新しいリストを作る。手続きappend!はappendと似ているが、コンストラクタではなくミュータータである。これはxの最後のペアを修正し、cdrがyになるようにして二つのリストをつなぎ合わせることによって追加を行う(空のxを引数としてappend!を呼ぶとエラーになる)。

```
(define (append! x y)
  (set-cdr! (last-pair x) y)
  x)
```

ここで、last-pairは引数の最後のペアを返す手続きである。

```
(define (last-pair x)
  (if (null? (cdr x)) x (last-pair (cdr x))))
```

次の対話について考えよ。

```
(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>
```

空欄の<response>はどうなるだろうか。箱とポインタの図を描き、答えについて説明せよ。

Exercise 3.13: 次の make-cycle 手手続きについて考えよ。これは、Exercise 3.12で定義した last-pair 手手続きを使っている。

```
(define (make-cycle x)
  (set-cdr! (last-pair x) x)
  x)
```

次の式によって作られる構造 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) を計算しようとすると、どうなるだろうか。

Exercise 3.14: 次の手手続きはわかりにくいが、とても役に立つ。

```
(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 “一時的” な変数 temp を使って、x の cdr の古い値を保持する。これは、次の行の set-cdr! が cdr を破壊するためである。一般的に mystery が何をするのか説明せよ。v は (define v (list 'a 'b 'c 'd)) によって定義されたものとする。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だろうか。

共有とアイデンティティ

Section 3.1.3では、代入を導入することによって出てくる“同一性”と“変更”という理論的な問題について触れました。これらの問題は、個々のペアが別々のデータオブジェクトの間で shared(共有) されると実際に起こるようになります。例えば、以下のように作られる構造について考えてみましょう。

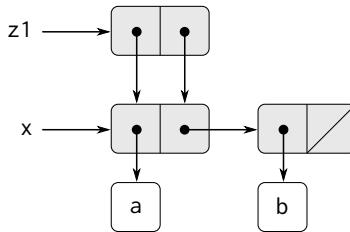


Figure 3.16: (`cons x x`) によって作られるリスト `z1`

```
(define x (list 'a 'b))
(define z1 (cons x x))
```

Figure 3.16に示したように、`z1` は、`car` と `cdr` が両方とも `x` という同じペアを指しています。このように、`z1` の `car` と `cdr` によって `x` が共有できるようになっているのは、`cons` が単純な方法で実装されているからです。一般的に、`cons` を使ってリストを構築すると、多くの個々のペアが多く別の構造によって共有される、ペアの絡まり合った構造ができます。

Figure 3.16に対して、Figure 3.17は次の式によって作られる構造を示しています。

```
(define z2 (cons (list 'a 'b) (list 'a 'b)))
```

この構造の中では、二つの (`a b`) リストに含まれるペアは、実際の記号は共有されていますが、区別可能です。¹⁹

リストとして考えると、`z1` と `z2` は両方とも“同じ”リスト、つまり ((`a b`) `a b`) を表現しています。一般的に、リストに対して `cons`, `car`, `cdr` だけを使って操作するのであれば、共有は完全に検出不可能です。しかし、リスト構造に対して変更を許可するとなると、共有は意味を持つようになります。共有によって違いが出てくる例として、次の手続きについて考えてみましょう。これは、適用対象の構造の `car` を修正するものです。

¹⁹ `cons` を呼び出すたびに新しいペアが返されるため、二つのペアは区別可能です。記号は共有されています。Scheme では、どんな名前に対しても唯一の記号が存在します。Scheme は記号を変更する手段を提供していないため、この共有を検出することはできません。また、そのように記号が共有されていることによって、単純にポインタが同じかどうかをチェックするだけの `eq?` によって記号の比較ができるようになっているということにも注意してください。

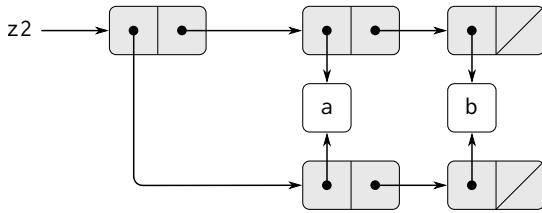


Figure 3.17: (cons (list 'a 'b) (list 'a 'b)) によって作られるリスト z2

```
(define (set-to-wow! x) (set-car! (car x) 'wow) x)
```

z1 と z2 は “同じ” 構造のはずなのに、これらに set-to-wow! を適用すると、異なる結果が返ってきます。z1 の場合は、car と cdr は同じペアなので、car を変更すると cdr も変更することになります。z2 の場合は、car と cdr は別のものなので、set-to-wow! は car しか変更しません。

```
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)
```

リスト構造の中にある共有を検出する方法として、述語 eq? を使うというものがあります。この述語は、二つの記号が同じかどうかテストする方法として Section 2.3.1 で紹介したものです。より一般的に、(eq? x y) は x と y が同じオブジェクトかどうか (つまり、x と y がポインタとして等しいかどうか) をテストします。そのため、Figure 3.16 と Figure 3.17 のように定義された z1 と z2 では、(eq? (car z1) (cdr z1)) は真になり、(eq? (car z2) (cdr z2)) は偽になります。

以降の節で見ていくように、共有を活用すると、ペアで表現可能なデータ構造の範囲を大きく広げることができます。一方、共有は危険なものになる可能性もあります。ある構造に加えた変更が、その変更部分をたまたま共有して

いる別の構造にも影響することになるからです。変更演算の `set-car!` と `set-cdr!` を使う際には注意する必要があります。扱っているデータオブジェクトについてよく理解していないと、変更が予期しない結果を招くことがあるかもしれません。²⁰

Exercise 3.15: 箱とポインタの図を描いて、上記の構造 `z1` と `z2` に対する `set-to-wow!` の効果を説明せよ。

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きを書くことにした。“簡単だよな”と彼は考えた。“どんな構造でも、`car` のぶんの数と `cdr` のぶんの数を足して、それにそのペア自身を数えるために 1 を足せば、ペアの数が求められる。”そして、Ben は次の手続きを書いた。

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
          (count-pairs (cdr x)))
      1)))
```

この手続きが正しくないということを示せ。具体的には、ちょうど三つのペアからなるリスト構造で、Ben の手続きが 3, 4, 7 を返すもの、また手続きが終了しないものを考え、それらを表す箱とポインタの図を描け。

Exercise 3.17: Exercise 3.16 の `count-pairs` 手続きの正しいバージョンを考え、任意の構造について異なるペアの数を返すようにせよ。(ヒント：カウント済みのペアを記録しておく補助データ構造を維持しながら構造をたどれ)。

²⁰ 可変データオブジェクトの共有を扱うことの難しさには、Section 3.1.3で提起した“同一性”と“変更”的問題が根底にあります。そこでは、私たちの言語で変更を許すようにするためには複合データは構成部品の集まりとは別の“アイデンティティ”を持たなければならないということに触れました。Lisp では、この“アイデンティティ”は、`eq?` によってテストされるような性質だと考えます。つまり、ポインタが同じかどうかということになります。多くの Lisp 実装では、ポインタの実装は本質的にはメモリアドレスなので、データオブジェクト“そのもの”は特定のメモリ上の場所のセットに格納された情報であると取り決めることによって、オブジェクトのアイデンティティを定義するという“問題を解決”しています。これは単純な Lisp のプログラムには十分ですが、計算モデルの“同一性”という問題を解く一般的な方法とはとても言えないものです。

Exercise 3.18: リストを検査し、循環を持つかどうか、つまり連続して `cdr` を取ることによってリストの終端を探そうとするプログラムが無限ループになるかどうかを判定する手続きを書け。Exercise 3.13でそのようなリストを構築した。

Exercise 3.19: 定数量の空間しか使わないアルゴリズムによって Exercise 3.18 をやり直せ(これにはとても巧妙なアイデアが必要となる)。

変更は代入にすぎない

複合データを導入したとき、Section 2.1.3ではペアが純粋に手続きだけによって表現できるということを見ました。

```
(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))
```

可変データについても同じことが言えます。可変データオブジェクトは、代入と局所状態を使って手続きとして実装できます。例えば、Section 3.1.1で `make-account` を使って銀行口座を実装したのと似たようなやり方で、上記のペアの実装を拡張して `set-car!` と `set-cdr!` を扱えるようにすることができます。

```
(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)

```

可変データのふるまいを説明するためには、理論的には、代入だけあれば十分です。私たちの言語で `set!` を認めたとたんに、代入の問題だけでなく、可変データ一般に関する問題がすべて引き起こされることになります。²¹

Exercise 3.20: 上記の手続きによるペアの実装を使って、次の式の列を評価する。

```

(define x (cons 1 2))
(define z (cons x x))
(set-car! (cdr z) 17)
(car x)
17

```

評価の様子を、環境の図を描いて説明せよ (Exercise 3.11と比較せよ)。

3.3.2 キューの表現

ミューーテータ `set-car!`, `set-cdr!` を使うと、`cons`, `car`, `cdr` だけでは構築できないようなデータがペアを使って構築できるようになります。この節では、ペアを使ってキューと呼ばれるデータ構造を表現する方法を学びます。Section 3.3.3では、テーブルと呼ばれるデータ構造を表現するやり方について学ぶことになります。

`queue(キュー)` とは、項目を一方の端 (キューの *rear*(終端)) に挿入し、もう一方の端 (*front*(先端)) から削除する列です。Figure 3.18は、最初は空だったキューにアイテム `a` と `b` が挿入される様子を示しています。次に `a` が削除され、`c` と `d` が挿入され、`b` が削除されています。項目は常に挿入した順番に削除さ

²¹一方、実装面から見ると、代入を実装するためには環境を変更することが必要となります。この環境というのはそれ自身が可変データ構造です。このように、代入と変更は等価だということです。どちらも、もう一方によって実装することができます。

<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: キュー演算

れるため、キューは*FIFO*(first in, first out) (先入れ先出し) バッファと呼ばれることもあります。

データ抽象化を使うと、キューは以下の演算の集合によって定義されるものと考えることができます。

- コンストラクタ : (`make-queue`) は、空のキュー (項目を持たないキュー) を返す。
- 二つのセレクタ :

`(empty-queue? <queue>)`

キューが空かどうかテストする。

`(front-queue <queue>)`

キューの先頭にあるオブジェクトを返す。キューが空ならエラーを出す。
キューは変更しない。

- 二つのミューテータ :

`(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値として返す。

`(delete-queue! <queue>)`

キューの先頭のアイテムを削除し、変更されたキューを値として返す。
削除前にキューが空であればエラーを出す。

キューはアイテムの列なので、当然通常のリストとして表現できるところです。キューの先頭はリストの `car` となり、キューへの項目の挿入はリストの終端に新しい要素を追加することとなり、キューからのアイテムの削除は単にリストの `cdr` を取るということになるでしょう。しかし、この表現は非効率的です。というのは、項目を挿入するためにはリストを終端まで走査しなければいけないからです。リストを走査するための手段は `cdr` 演算を連続して使うことだけなので、この走査は n 項目のリストに対して $\Theta(n)$ ステップが必要になります。リスト表現に単純な変更を加えるだけで、必要なステップが $\Theta(1)$ 、つまり必要なステップ数がキューの長さと独立なキュー演算の実装ができるようになり、この欠点を克服することができます。

リスト表現の難しいところは、リストの終端を見つけるためにリストを走査しないといけないということです。なぜ走査しないといけないかというと、リストをペアの鎖として表現する標準的なやり方では、先頭へのポインタはすぐに得られるのに対し、リストの終端へのポインタは簡単にアクセスできないようになっているからです。その欠点を避けるためには、キューをリストとして表現したうえで、リストの最終ペアを指す追加のポインタを加えるように修正します。こうすると、項目を挿入する際に終端ポインタを見ることによってリストの走査を避けられます。

このやり方では、キューは `front-ptr`, `rear-ptr` というポインタのペアとして表現できます。ポインタはそれぞれ通常のリストの先頭と最後のペアを指しています。キューを識別可能なオブジェクトにするために、`cons` を使って二つのポインタを結合します。つまり、キュー自身は二つのポインタの `cons` になります。Figure 3.19はこの表現を図示したものです。

キュー演算を定義するには以下の手続きを使います。これらは、キューの先端・終端ポインタを選択・変更できるようにするものです。

```
(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、実際のキュー演算を実装できるようになります。先端のポインタが空リストであればキューは空であるとします。

```
(define (empty-queue? queue) (null? (front-ptr queue)))
```

`make-queue` コンストラクタは、初期状態の空キューとして、`car` と `cdr` の両方が空リストであるペアを返します。

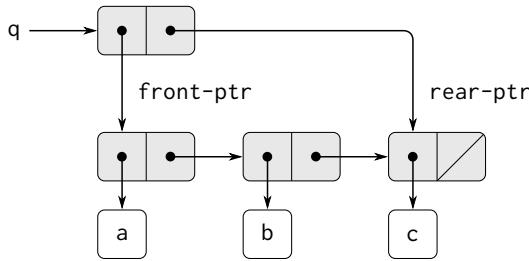


Figure 3.19: 先端と終端のポインタを持つリストとしてのキューの実装

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

キューの先頭の項目を選択するには、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には、結果としてFigure 3.20のようになるような手順を実行します。まず、`car` が挿入する項目で `cdr` が空リストという新しいペアを作ります。もしキューが元々空だったら、先端と終端のポインタが両方ともこの新しいペアを指すようにします。そうでなければ、キューの最終ペアが新しいペアを指すように変更し、また終端ポインタも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)))
```

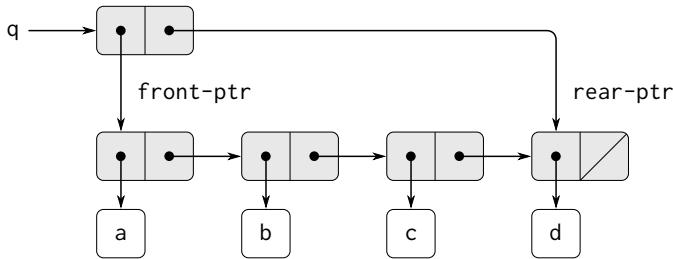


Figure 3.20: Figure 3.19のキューに (insert-queue! q 'd) を使った結果

```
(set-rear-ptr! queue new-pair)
(queue)))
```

キューの先頭の項目を削除するのは、先頭ポインタがキューの二番目の項目を指すようにするだけで大丈夫です (Figure 3.21参照)。²²

```
(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、上記のキューの実装をテストすることにした。彼は Lisp インタプリタにキューの手続きを入力し、それから試しに使ってみた。

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

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

²²最初の項目がキューの最終項目でもある場合、削除後に先端ポインタは空リストになります。これがキューが空だという印になります。終端ポインタは依然として削除済みの項目を指し続けることになりますが、わざわざ更新する必要はありません。empty-queue? は先端ポインタしか見ないからです。

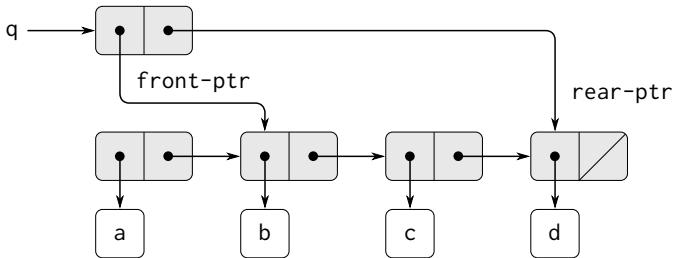


Figure 3.21: Figure 3.20のキューに (`delete-queue! q`) を使った結果

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

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

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

“全然ダメじゃないか”と彼は文句を言う。“インタプリタの応答を見てみろよ。最後の項目がキューに二回挿入されているじゃないか。それに、項目を両方とも削除しても、二つ目の `b` は残ったままだ。つまり、キューは空になっていないってことだ。空になっていないといけないので” Eva Lu Ator は、Ben が挙動を誤解していると言う。“項目がキューに二回入っているわけじゃなくて” 彼女は説明する。“Lisp 標準の表示プログラムが、キューの表現をどう解釈していいかわからないってだけ。キューをちゃんと表示させたいなら、キューを表示する手続きを自分で書かないと” Eva Lu の話について解説せよ。具体的には、なぜ Ben の例がそのような表示結果を出力するのか説明せよ。キューを入力として取り、キュー内の項目列を表示する手続き `print-queue` を定義せよ。

Exercise 3.22: キューは、ポインタのペアとして表現する以外に、局所状態を持つ手続きとして構築することもできる。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なる。したがって、手続き `make-queue` は以下のような形になる。

```
(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、この表現によるキューの演算を実装せよ。

Exercise 3.23: `deque(両端キュー)` (“double-ended queue”) は、先端と終端のどちらに対しても項目の挿入と削除が行える列である。`deque` に対する演算は、コンストラクタ `make-deque`、述語 `empty-deque?`、セレクタ `front-deque`、`rear-deque`、ミュータータ `front-insert-deque!`, `rear-insert-deque!`, `front-delete-deque!`, `rear-delete-deque!` である。ペアによって `deque` を表現するやり方を示せ。また、演算を実装せよ。²³ 全ての演算は $\Theta(1)$ ステップで完了しなければならない。

3.3.3 テーブルの表現

Chapter 2で集合を表現するいろいろなやり方について学んだ際に、Section 2.3.3では、識別用キーによって索引づけされたレコードのテーブルを保持するというタスクについて触れました。Section 2.4.3のデータ主導プログラミングを実装するにあたっては、二つのキーによって情報を格納したり取り出したりする二次元のテーブルが大活躍しました。ここでは、可変リスト構造としてテーブルを構築するやり方について見ていきます。

まず、一次元の表について考えます。一次元の表では、それぞれの値は単独のキーの下に格納されます。このテーブルをレコードの列として実装し、それぞれのレコードは、キーとそれに関連づけられた値とのペアとして実装します。レコードはペアによって連結して列状にします。連結用ペアの `car` は連結したレコードを指しています。これらの連結用ペアは、テーブルの *backbone*

²³ 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)。

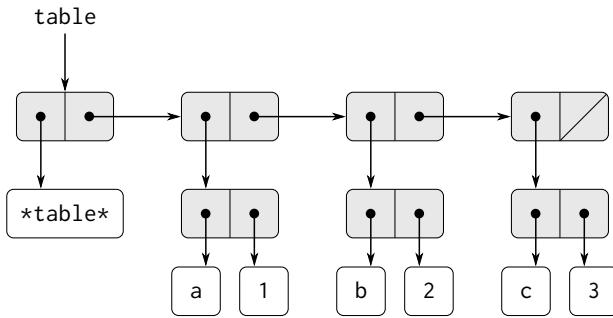


Figure 3.22: 頭つきリストとして表現されたテーブル

(**背骨**)といいます。テーブルに新しいレコードを追加するときに変更できる場所が必要なので、テーブルは*headed list*(頭つきリスト)として構築します。頭つきリストは、先頭に特別な背骨のペアを持っています。これはダミーの“レコード”—今回の場合、適当に選んだ記号 `*table*`—を持っています。Figure 3.22は、次のテーブルに対する箱とポインタの図を示したものです。

a: 1
b: 2
c: 3

テーブルから情報を取り出すには `lookup` という手続きを使います。この手続きは引数としてキーを取り、関連づけられた値(キーの下に格納された値がなければ偽)を返します。`lookup` は、引数としてキーとレコード列を取る `assoc` 演算によって定義します。`assoc` はダミーレコードを見ないことに注意してください。`assoc` は、与えられたキーを `car` として持つレコードを返します。²⁴ それから `lookup` は、`assoc` に返されたレコードが偽でないことをチェックし、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)))
```

²⁴`assoc` は `equal?` を使うため、記号・数値・リスト構造のキーを認識できます。

```

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

```

特定のキーの下に値を挿入するには、まず `assoc` を使って、すでにそのキーを持つレコードがテーブルにないか確認します。なければ、キーと値を `cons` して新しいレコードを作り、それをテーブルのレコード列の先頭、つまりダミーレコードの後に挿入します。すでにそのキーのレコードがあるなら、そのレコードの `cdr` を指定された新しい値にセットします。テーブルのヘッダは、新しいレコードを挿入するために修正する固定の場所という役割を果たしています。²⁵

```

(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のは、記号 `*table*` を持つリストを作るだけで終わりです。

```

(define (make-table)
  (list '*table*))

```

二次元テーブル

二次元テーブルでは、それぞれの値は二つのキーによって索引づけがされています。そのようなテーブルは、一次元テーブルのそれぞれのキーがサブテーブル

²⁵ このため、最初の背骨ペアは、テーブル“そのもの”を表現するオブジェクトということになります。つまり、テーブルを指すポインタというのは、このペアを指すポインタだということです。この同じ背骨ペアは、常にテーブルのスタート地点になっています。もしこういうやり方をしていなければ、`insert!` が新しいレコードを追加する際に、テーブルの新しいスタート地点の値を返さなければならなくなるところです。

ルを特定ようにすることで作れます。Figure 3.23は、次のテーブルに対する箱とポインタの図を示したものです。

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

このテーブルは、二つのサブテーブルを持っています。(サブテーブルには特別なヘッダ記号は必要ありません。サブテーブルを特定するキーがその目的を果たすためです)。

項目を探すときには、一つ目のキーを使って正しいサブテーブルを特定します。次に、二つ目のキーを使ってサブ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項目を挿入するには、`assoc` を使って一つ目のキーの下にサブテーブルが格納されているかどうか確認します。もしなければ、ひとつのレコード (`key-2, value`) を持つ新しいサブテーブルを構築し、それをテーブルの一つ目のキーの下に挿入します。もし一つ目のキーに対するサブテーブルがすでに存在するなら、上で述べた一次元テーブルの挿入法を使って、そのサブテーブルに新しいレコードを挿入します。

```
(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
```

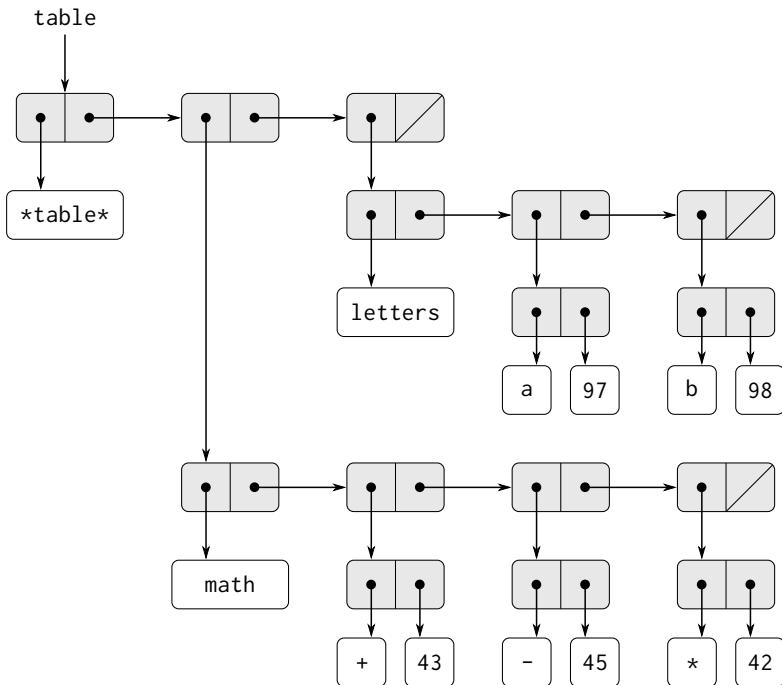


Figure 3.23: 二次元テーブル

```

        (cons (cons key-2 value)
              (cdr subtable)))))

(set-cdr! table
           (cons (list key-1
                        (cons key-2 value))
                 (cdr table)))))

'ok)

```

局所テーブルの作成

上で定義した `lookup` と `insert!` 演算は、引数としてテーブルを取ります。このことによって、二つ以上のテーブルにアクセスするプログラムが使えるようになります。複数のテーブルを扱う別の方法としては、それぞれのテーブルに対して別々の `lookup` と `insert!` 手続きを持つというものがあります。これは、内部テーブルを局所状態として保持するオブジェクトとしてテーブルを手続き的に表現することによって達成できます。適切なメッセージが送られると、この“テーブルオブジェクト”は内部テーブルに対する演算を行う手続きを提供します。以下に、この方法で表現した二次元テーブル生成プログラムを示します。

```
(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` を使うと、Section 2.4.3でデータ主導プログラミングに使った `get` と `put` を次のように実装できます。

```
(define operation-table (make-table))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))
```

`get` は二つのキーを取り、`put` は二つのキーと値を引数に取ります。どちらの操作も、`make-table` 呼び出し時に作られるオブジェクトにカプセル化された同じ局所テーブルにアクセスします。

Exercise 3.24: 上のテーブル実装では、キーの等価性は `equal?(assoc` に呼ばれる) によってテストされている。これは常に適切なテストというわけではない。例えば、数字をキーとするテーブルを扱っている場合、検索する数字に対して完全に一致する必要がなく、いくらかの誤差の範囲内のものであればいいということも考えられる。キーの“等価性”をテストするのに使う手続き `same-key?` 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、局所テーブルに対する適切な `lookup`, `insert!` 手続きにアクセスするために使用できる `dispatch` 手続きを返すようにすること。

Exercise 3.25: 一次元と二次元のテーブルを一般化し、任意の数のキーの下に値を格納するテーブルを実装するやり方を示せ。値によってキーの数は異なることもある。`lookup`, `insert!` 手続きは、テーブルにアクセスするためのキーのリストを入力として取る。

Exercise 3.26: 上のように実装したテーブルを検索するには、レコードのリストを走査する必要がある。これは、基本的にSection 2.3.3の順序なしリスト表現である。大きなテーブルに対しては、別のやり方でテーブルを構造化するほうが効率がよくなるかもしれない。(キー、値) のレコードを二分木を使ってまとめるようなテーブルの実装について説明せよ。キーは、何らかの方法(数の大小

順やアルファベット順など)によって順序づけができるものとする (Chapter 2のExercise 2.66と比較せよ)。

Exercise 3.27: *memoization(メモ化)(tabulation(テーブル化))* とは、以前計算したことのある値を手続きが局所テーブルに記録できるようにするテクニックである。このテクニックを使うことでプログラムのパフォーマンスが大きく変わることもある。メモ化された手続きは、以前の呼び出しの値をテーブルに保持する。キーにはその値を生成した引数を使う。メモ化された手続きが値を計算するよう求められると、まずテーブルをチェックしてその値が入っていないか見て、もし入っていればその値をそのまま返す。入っていないければ、普通のやり方で新しい値を計算し、それをテーブルに格納する。メモ化の例として、Section 1.2.2のフィボナッチ数を計算する指数的プロセスを思い出そう。

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

同じ手続きをメモ化したバージョンは次のようにになる。

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

ここで、*memoize* は以下のように定義される手続きである。

```
(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previously-computed-result
              (lookup x table)))
        (or previously-computed-result
            (let ((result (f x)))
              (put x result table)
              result))))
```

```
(insert! x result table)
result))))))
```

環境の図を描き、(memo-fib 3) の計算を分析せよ。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計算できるのか説明せよ。単に memo-fib を (memoize fib) と定義していたとしても、この構想はうまくいくだろうか。

3.3.4 デジタル回路シミュレータ

複雑なデジタルシステム（例えはコンピュータなど）の設計は重要な工学活動です。デジタルシステムは、単純な要素を相互接続することで構築されます。個々の要素のふるまいは単純ですが、それらが構成するネットワークのふるまいはとても複雑になることもあります。回路設計案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使わ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）と呼ばれる種類のプログラムの例となります。この種類のプログラムでは、アクション（“イベント”）は後になってさらなるイベントを引き起こし、それらのイベントがまたさらなるイベントを引き起こし…という繰り返しになっています。

本書での回路の計算モデルは、回路を構成する基本部品に対応するオブジェクトによって構成されることになります。まず、*digital signals*（デジタル信号）を伝える *wires*（線）があります。デジタル信号は、ある特定の瞬間には、二つの可能な値である 0 と 1 のどちらかしか取りません。ほかには、いろいろなタイプのデジタル *function boxes*（関数箱）もあり、入力信号を伝える回路と、ほかの出力回路とを接続しています。これらの箱は、入力信号から計算される信号を出力します。出力信号は、関数箱のタイプによって時間的な遅れが生じます。例えば、*inverter*（インバータ）は入力を反転する基本関数箱です。インバータに対する入力信号が 0 に変化すると、インバータ遅延時間 1 単位後にインバータは出力信号を 1 に変えます。インバータに対する入力信号が 1 に変化すると、インバータ遅延時間 1 単位後にインバータは出力信号を 0 に変えます。インバータは Figure 3.24 のような記号として表します。同じ Figure 3.24 に示している *and-gate*（AND ゲート）は、二つの入力とひとつの出力を持つ基本関数箱です。この関数箱は、入力の *logical and*（論理積）となる値に出力信号を駆動します。つまり、入力信号が両方とも 1 になると、AND ゲート遅延時間 1 単位後に、AND ゲートは出力信号を 1 にするということです。そうでなけ

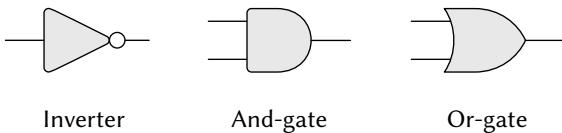


Figure 3.24: デジタル論理回路シミュレータにおける基本関数

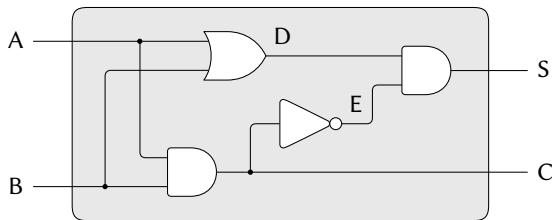


Figure 3.25: 半加算器回路

れば、出力は 0 です。*or-gate(OR ゲート)* は、似たような 2 入力の基本関数箱で、入力の *logical or*(論理和) となる値に出力を駆動します。つまり、少なくともひとつの入力信号が 1 であれば出力は 1 になり、そうでなければ出力は 0 になるということです。

基本関数を接続すると、より複雑な関数を構築することができます。そのためには、関数箱の出力を別の関数箱の入力に結線します。例えば、Figure 3.25 に示す *half-adder* (半加算器) は、ひとつの OR ゲート、二つの AND ゲート、ひとつのインバータからなります。入力信号としては A と B の二つを取り、出力信号は S と C の二つです。S は、A と B のどちらかひとつだけが 1 のときに 1 になり、C は A と B の両方が 1 のときに 1 になります。図を見ると、遅延があるために、出力が異なる時刻に生成されるかもしれないということがわかります。デジタル回路設計の設計の難しさは、多くはこの事実によって起こるものです。

ここからは、私たちの学習対象となる、デジタル論理回路をモデル化するプログラムを作っていきます。このプログラムでは、信号を“保持”する線をモデル化する計算オブジェクトを構築します。関数箱は、信号同士の正しい関

係を成り立たせる手続きによってモデル化します。

シミュレーションの基本要素のひとつは、回路を構築する手続き `make-wire` です。例えば、次のようにして 6 つの線を構築することができます。

```
(define a (make-wire))
(define b (make-wire))
(define c (make-wire))
(define d (make-wire))
(define e (make-wire))
(define s (make-wire))
```

ある関数箱を線につなげるには、そのタイプの関数箱を構築する手続きを呼びます。コンストラクタ手続きの引数は、その箱につなげる線です。例えば、AND ゲート、OR ゲート、インバータを構築できるようになっているとすると、Figure 3.25 に示す半加算器を結線できます。

```
(or-gate a b d)
ok
(and-gate a b c)
ok
(inverter c e)
ok
(and-gate d e s)
ok
```

さらに都合のいいことに、この演算には明示的に名前をつけることができます。そのためには、半加算器につなげる四本の外部の線が与えられたときにこの回路を構築する `half-adder` という手続きを定義します。

```
(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))
```

この定義の利点は、より複雑な回路を作成するための構築用ブロックとして `half-adder` 自身を使うことができるということです。例えば、Figure 3.26 は、

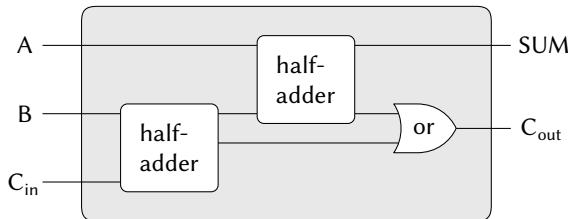


Figure 3.26: 全加算器回路

二つの半加算器とひとつのORゲートによって構成される*full-adder*(全加算器)を示したものです。²⁶ 全加算器は、次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))
```

`full-adder`を手続きとして定義したので、今度はそれを構築用ブロックとして使って、さらに複雑な回路を作ることができるようになります(例はExercise 3.30を参照)。

簡単にいうと、このシミュレータは回路を構築するための道具を提供してくれるものだといえます。Section 1.1でLispの学習に対するアプローチとして使ったような一般的な視点から見ると、基本関数箱は言語の基本要素を構成し、箱を結線することは組み合わせの手段であり、手続きとして線のパターンを規定することは抽象化の手段としての役割を果たしているといえるでしょう。

²⁶全加算器は、二つの二進数の加算に使われる基本的な回路要素です。ここで、AとBは加算する二つの数の対応する位置のビットで、C_{in}はひとつ右の位置での加算から来るキャリー(繰り上がり)ビットです。この回路は、対応する位置の合計ビットであるSUMと、左に伝播させるキャリービットであるC_{out}を生成します。

基本関数箱

基本関数箱は、ひとつの線上の信号の変化がほかの線上の信号に影響を与える“力”を実装したものです。関数箱を構築するには、線に対する次のような演算を使います。

- `(get-signal <wire>)`
これは、線上の信号の現時点での値を返します。
- `(set-signal! <wire> <new value>)`
これは、線上の信号の値を新しい値に変更します。
- `(add-action! <wire> <procedure of no arguments>)`
これは、線上の信号の値が変わったときに、指定された手続きが必ず実行されるようにするものです。そのような手続きは、線上の信号の値の変化をほかの線に伝える手段となります。

これらに加え、`after-delay` という手続きも使います。これは、遅延時間と実行する手続きを引数に取り、与えられた遅延時間後に与えられた手続きを実行するというものです。

これらの手続きを使って、基本のデジタル論理関数を定義できます。インバータを通して入力と出力を接続するには、`add-action!` を使って、入力線上の信号の値が変わるたびに呼ぶ手続きを入力線に関連づけます。この手続きは入力信号の `logical-not`(論理否定) を計算し、そして `inverter-delay1` 単位後に、出力信号にその新しい値をセットします。

```
(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 ゲートはもうちょっと複雑です。アクション手続きは、入力のどちらかひとつが変化したときに動くようにする必要があります。アクション手続きは、

入力線上の信号の値の logical-and(論理積) を (logical-not と似たような手続きを使って) 求め、and-gate-delay1 単位後に出力線上で新しい値への変化が起こるよう設定します。

```
(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 ゲートを基本関数箱として定義せよ。or-gate のコンストラクタは and-gate と似たようなものにすること。

Exercise 3.29: OR ゲートを構築する別の方針として、AND ゲートとインバータからなる複合デジタル論理デバイスとして組み立てるというものがある。これを実現する手続き or-gate を定義せよ。この OR ゲートの遅延時間は、and-gate-delay と inverter-delay によって表すと、どのようになるだろうか。

Exercise 3.30: Figure 3.27 は、 n 個の全加算器をつなげた ripple-carry adder (繰り上がり伝播加算器) を示している。これは、二つの n ビット二進数を足すための最も単純な形の並列加算器である。 $A_1, A_2, A_3, \dots, A_n$ と $B_1, B_2, B_3, \dots, B_n$ という入力は、足し合わせる二つの二進数 (各 A_k, B_k は 0 または 1) である。この回路は、和のうちの n ビットの $S_1, S_2, S_3, \dots, S_n$ と、足し算による繰り上がりの C を生成する。この回路を生成する手続き ripple-carry-adder を書け。この手続きは、 n 本の線のリスト三つ — A_k, B_k, S_k — と、もうひとつの線 C を引数として取るようにする。繰り上がり伝播加算器の大きな欠点は、キャリー信号が伝播するのを待つ必要があるということだ。 n ビットの繰り上がり伝播加算器から完全な出力を得るのにかかる遅延時間は、AND ゲート・OR ゲート・インバータの遅延時間によって表現すると、どのようになるだろうか。

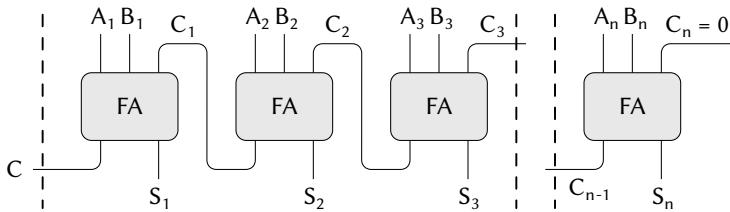


Figure 3.27: n bit の数を扱う繰り上がり伝播加算器

回路の表現

このシミュレーションでは、線は二つの局所状態変数を持つ計算オブジェクトになります。信号の値 `signal-value`(初期値は 0) と、信号の値が変わるとときに実行するアクション手続き `action-procedures` の集合です。メッセージパッキングスタイルを使い、局所手続きの集合に適切な局所演算を選ぶ `dispatch` 手続きを加えたものとして線を実装します。これは、Section 3.1.1で単純な銀行口座オブジェクトを実装したときと同じです。

```
(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))
```

局所手続き `set-my-signal!` は、新しい信号値によって線上の信号が変わるかどうかをテストします。変わるのであれば、以下の `call-each` 手続きをを使って各アクション手続きを実行します。`call-each` は、引数なし手続きのリストの各項目を呼び出すというものです。

```
(define (call-each procedures)
  (if (null? procedures)
      'done
      (begin ((car procedures)
              (call-each (cdr procedures))))))
```

局所手続き `accept-action-procedure!` は、与えられた手続きを実行する手続きのリストに追加し、それから新しい手続きを一回実行します (Exercise 3.31 参照)。

局所手続き `dispatch` が上で規定した通りに設定されていれば、以下のいくつかの手続きを提供することで線に対する局所演算にアクセスできるようになります。²⁷

```
(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))
```

線は、時とともに変化する信号を持ち、接続される装置が増えていくこともあるので、可変オブジェクトの典型といえます。ここでは、代入によって変化する局所状態変数を持つ手続きとして線をモデル化しています。新しい線が作られると、状態変数の新しいセットが (`make-wire` 中の `let` 式によって) 割り当てられ、それらの新しい状態変数を含む環境を持つ新しい `dispatch` 手続きが構築され、それが返されることになります。

²⁷これらの手続きは、通常の手続き的構文によってオブジェクトの局所手続きにアクセスできるようにするシンタックスシュガーにすぎません。“手続き”と“データ”的役割がこんなに簡単なやり方で交換できるということは特筆すべきことです。例えば、もし `(wire 'get-signal)` と書くようになると、`wire` は `get-signal` というメッセージを入力として呼び出す手続きとして考えるということになります。`(get-signal wire)` と書くようになると、`wire` は手続き `get-signal` の入力となるデータオブジェクトだと考えやすくなります。この問題の実際のところは、手続きをオブジェクトとして扱う言語においては“手続き”と“データ”的間に根本的な違いはなく、好きなスタイルでプログラミングできるようにシンタックスシュガーを選ぶことができるということです。

線は、それに接続されたいろいろなデバイスの間で共有されます。そのため、ひとつのデバイスとのやりとりによって起こった変化が、その線につながったほかのすべてのデバイスに影響することになります。線は、接続時に提供されるアクション手続きを呼ぶことによって、変化を近隣に伝えます。

予定表

後は、`after-delay` さえあればシミュレータが完成します。ここでの考え方は、すべきことのスケジュールを持っておく`agenda`(予定表)と呼ばれるデータ構造を保持するということです。予定表に対しては、以下の演算が定義されます。

- (`make-agenda`) は、新しい空の予定表を返します。
- (`empty-agenda?` (`agenda`)) は、指定した予定表が空のときに真になります。
- (`first-agenda-item` (`agenda`)) は、予定表の最初の項目を返します。
- (`remove-first-agenda-item!` (`agenda`)) は、最初の項目を削除することによって、予定表を変更します。
- (`(add-to-agenda! <time> <action> <agenda>)`) は、指定された時間に実行するアクション手続きを追加することによって、予定表を変更します。
- (`(current-time <agenda>)`) は、現在のシミュレーション時間を返します。

ここで使う特定の予定表は、`the-agenda`として示します。手続き`after-delay`は、`the-agenda`に新しい要素を追加します。

```
(define (after-delay delay action)
  (add-to-agenda! (+ delay (current-time the-agenda))
                  action
                  the-agenda))
```

シミュレーションを動かすのは、手続き`propagate`(伝播)です。この手続きは`the-agenda`を操作し、この予定表に載っているそれぞれの手続きを順番に実行します。一般的に、シミュレーションが進むにつれ、新しい項目が予定表に追加されていき、`propagate`は予定表に項目が存在する限りずっとシミュレーションを続けます。

```
(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))))
```

シミュレーション例

次の手続きは、線に“probe”(プローブ)を置くというもので、シミュレータの動作を表示します。プローブは、信号値が変わるたびに新しい信号値・現在時刻・線の識別名を表示するよう線に伝えます。

```
(define (probe name wire)
  (add-action! wire
    (lambda ()
      (newline)
      (display name)
      (display " ")
      (display (current-time the-agenda))
      (display " New-value = ")
      (display (get-signal wire)))))
```

まず、予定表を初期化し、基本関数箱の遅延時間を規定します。

```
(define the-agenda (make-agenda))
(define inverter-delay 2)
(define and-gate-delay 3)
(define or-gate-delay 5)
```

次に、四つの配線を定義し、そのうち二つにはプローブを設置します。

```
(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
```

それから、線を (Figure 3.25 のように) 半加算器回路に接続し、`input-1` 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` の信号は、時刻 8 で 1 に変わります。シミュレーション開始から 8 単位時間がたつことになります。ここで、`input-2` の信号を 1 に設定し、値が伝播するようにします。

```
(set-signal! input-2 1)
done
```

```
(propagate)
carry 11 New-value = 1
sum 16 New-value = 0
done
```

`carry` は時刻 11 で 1 に変わり、`sum` は時刻 16 で 0 に変わります。

Exercise 3.31: `make-wire` の中に定義される内部手続き `accept-action-procedure!` は、新しいアクション手続きが線に追加されるときに、その手続きがすぐに実行されるということを規定している。この初期化がなぜ必要なのか説明せよ。具体的には、上の段落の半加算器の例をトレースし、`accept-action-procedure!` を以下のように定義した場合にシステムの応答がどのように変わらか述べよ。

```
(define (accept-action-procedure! proc)
  (set! action-procedures
    (cons proc action-procedures)))
```

予定表の実装

最後に、予定表のデータ構造の詳細を記述します。このデータ構造は、将来実行するようスケジュールされた手続きを持っておくものです。

予定表は、複数の*time segment*(時間区分)によって構成されます。それぞれの時間区分は、数値(時刻)と、その時間区分の間に実行するようスケジュールされた手続きを持っておくキュー(Exercise 3.32参照)からなるペアです。

```
(define (make-time-segment time queue)
  (cons time queue))
(define (segment-time s) (car s))
(define (segment-queue s) (cdr s))
```

時間区分キューは、Section 3.3.2で説明したキュー演算によって操作します。

予定表自身は、時間区分の一次元テーブルです。Section 3.3.3で説明したテーブルと違うのは、時間区分は時間の前後順に並んでいるという点です。それに加えて、予定表の先頭には *current time*(現在時刻)(最後に処理したアクションの時刻)を保存します。新しく構築したばかりの予定表には時間区分はなく、現在時刻としては 0 を持っています。²⁸

```
(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)
```

²⁸ 予定表はSection 3.3.3のような頭つきリストですが、このリストには時刻というヘッダがあるので、追加のダミー・ヘッダ(テーブルで使った *table* 記号のようなもの)の必要はありません。

```
(cdr (segments agenda)))
```

予定表に時間区分がなければ、その予定表は空だということです。

```
(define (empty-agenda? agenda)
  (null? (segments agenda)))
```

予定表にアクションを追加するには、まず予定表が空かどうかチェックします。空であれば、アクションの時間区分を作り、それを予定表に組み込みます。空でなければ、それぞれの時間区分の時刻を調べながら予定表を走査します。指定時刻に時間区分が見つかれば、それに関連づけられたキューにアクションを追加します。指定時刻以降の時刻にたどり着いたら、その時刻のすぐ前に、新しい時間区分を挿入します。予定表の最後にたどり着いたら、末尾に新しい時間区分を作る必要があります。

```
(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)))

  (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))))
```

予定表の先頭の項目を削除する手続きは、先頭の時間区分にあるキューの先頭の項目を削除します。削除して時間区分が空になるなら、それを時間区分のリストから削除します。²⁹

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

予定表の先頭の項目は、先頭の時間区分のキューの先頭にあります。項目を取り出すたびに、現在時刻の更新も行います。³⁰

```
(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: 予定表の各時間区分で実行すべき手続きはキューに保存される。そのため、各時間区分の手続きは、予定表に追加した順(最初に入れたものが最初に取り出される)で呼び出される。どうしてこの順番でなければならないのか説明せよ。具体的には、ひとつの時間区分の中で入力が 0, 1 から 1, 0 に変わるとのふるまいをトレースし、時間区分の手続きを通常のリストと

²⁹ この手続きの if 式には `<alternative>` 式がありません。このような“片腕 if 文”は、二つの式から選ぶというのではなく、何かをするかどうかを決めるという場合に使います。if 式は、述語が偽であり、かつ `<alternative>` が存在しない場合、不定の値を返します。

³⁰ こうすると、現在時刻は常に直近に処理したアクションの時刻になります。この時刻を予定表の先頭に格納することで、関連づけられた時間区分が削除されていてもその時刻が確実にわかるようになっています。

して先頭だけで追加と削除を行う(最後に入れたものが最初に取り出される)ようにするとふるまいがどのように変わるか述べよ。

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計算という構造になっていて、あらかじめ決められた引数に演算を行い、望む出力を生成します。一方で、いくつかの量の間の関係を使ってシステムをモデル化するということもよくあります。例えば、機械構造の数学的モデルには、金属棒のたわみ d と棒に加わる力 F 、棒の長さ L 、断面積 A 、弾性係数 E との間に成り立つ、次の方程式によって表される関係についての情報が含まれるかもしれません。

$$dAE = FL.$$

このような方程式は一方向ではありません。量のうち任意の四つがあれば、それを使って五つ目の量を計算することができます。しかし、この方程式を伝統的なコンピュータ言語に翻訳しようとすると、どの量を残り四つの量を使って計算するのか選ばなければなりません。そのため、断面積 A とたわみ d の計算は同じ方程式から導かれるものなのに、 A を計算する手続きは d を求めるのに使えないということになります。³¹

この節では、関係そのものによってプログラミングできるような言語の設計についての概略を述べます。この言語の基本要素は *primitive constraints*(基本制約)です。これは、量と量の間に成り立つ関係を記述したものです。例えば、(adder a b c) は a, b, c という量の間に $a + b = c$ という方程式の関係が成り立っていないなければならないということを規定し、(multiplier x y z) は、 $xy = z$ という制約を表し、(constant 3.14 x) は x の値が 3.14 でなければならぬということを表しています。

この言語では、基本制約を組み合わせてより複雑な関係を表す方法を提供します。制約を組み合わせるには、*connectors*(コネクタ)を使って制約を接続し、*constraint networks*(制約ネットワーク)を構築します。コネクタは、最低ひとつずつの制約に含まれる値を“保持する”オブジェクトです。例を挙げると、

³¹ 制約伝播は、Ivan Sutherland (1963)による、信じられないほどの先進性を持った SKETCHPAD システムで初登場したものです。Xerox パロアルト研究センターでは、Smalltalk ベースの美しい制約伝播システムが Alan Borning (1977)によって開発されました。Sussman, Stallman, Steele は、制約伝播を電子回路解析に応用しました (Sussman and Stallman 1975; Sussman and Steele 1980)。TK!Solver (Konopasek and Jayaraman 1984) は、制約に基づく大規模モデリング環境です。

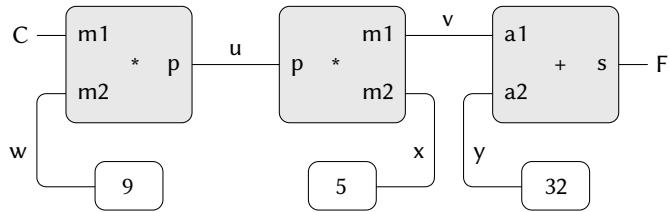


Figure 3.28: 制約ネットワークとして表現された関係

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

華氏温度と摂氏温度には次のような関係があるということを私たちは知っています。

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

このような制約は、加算器・乗算器・不变制約という基本制約からなるネットワーク (Figure 3.28) として考えることができます。図の中で、左には m_1, m_2, p というラベルのついた三つの端子を持つ乗算箱があります。これらの端子は、次のようにして乗算器をネットワークのほかの部分と接続します。まず、 m_1 端子は摂氏の気温を保持するコネクタ C につなげられます。 m_2 端子は w というコネクタにつなげられます。このコネクタは 9 を保持する定数箱にもつながっています。 p 端子は乗算箱によって m_1 と m_2 の積になるよう制約されていて、別の乗算箱の p 端子につながっています。この乗算箱では、 m_2 は整数 5 に、 m_1 は加算の項のうちのひとつに接続されています。

このようなネットワークによる計算の進行は以下のようになります。コネクタに (ユーザまたは接続された制約箱によって) 値が与えられると、コネクタは関連づけられた制約すべて (そのコネクタを起動した制約を除き) を起動し、値が与えられたことを知らせます。起動されたそれぞれの制約箱は、それに接続されているコネクタを調査して、コネクタの値を決めるのに十分な情報があるかどうかを見ます。もし値が決められるなら、制約箱はそのコネクタに値を設定し、するとそのコネクタは関連づけられた制約すべてを起動し ... という連鎖が続けます。例えば、摂氏と華氏の変換の場合、制約箱によって w, x, y はそれぞれ 9, 5, 32 にすぐに設定されます。それらのコネクタは乗算器と加算器を起動し、それらは計算を進行させるために十分な情報がないと判断します。もしユーザ (またはネットワークのどこか別の場所) が C にある値 (例えば 25) を設定すると、左端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 は二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。

制約システムを使う

上で概要を述べた温度計算を制約システムを使って実行するには、まずコンストラクタ `make-connector` を読んで C, F という二つのコネクタを作り、それらを適切なネットワークに接続します。

```
(define C (make-connector))
(define F (make-connector))
(celsius-fahrenheit-converter C F)
ok
```

ネットワークを作る手続きは、次のように定義します。

```
(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))
```

この手続きは内部コネクタ u, v, w, x, y を作成し、基本制約のコンストラクタ `adder`, `multiplier`, `constant` を使って Figure 3.28 に示したように接続します。Section 3.3.4 のデジタル回路シミュレータの場合と同じように、基本要素の組み合わせに手続きを使うことによって、この言語は自動的に複合オブジェクトを抽象化する手段を持つことになります。

ネットワークの動作を見るために、Section 3.3.4 で配線の監視を使ったのと似たような `probe` 手続きを使って、コネクタ C, F にプローブを置きます。コネクタにプローブを置くと、コネクタに値が与えられるたびにメッセージを表示するようになります。

```
(probe "Celsius temp" C)
(probe "Fahrenheit temp" F)
```

それから、C の値を 25 に設定します。`(set-value! C 25)` への三つ目の引数は、この指示が `user` によるものであることを C に伝えていきます。

```
(set-value! C 25 'user)
Probe: Celsius temp = 25
Probe: Fahrenheit temp = 77
done
```

C のプローブが起動され、値を報告します。C はまた、値を上記のネットワークに伝播させます。それによって F に 77 が設定され、それが F のプローブによって報告されます。

今度は、F に新しい値、例えば 212 を設定してみます。

```
(set-value! F 212 'user)
Error! Contradiction (77 212)
```

コネクタは矛盾を検知したというメッセージを出します。値は 77 なのに、誰かがそれを 212 に設定しようとしているということです。新しい値によってネットワークを再利用するには、C に古い値を忘れるよう指示します。

```
(forget-value! C 'user)
Probe: Celsius temp = ?
Probe: Fahrenheit temp = ?
done
```

C は、最初に値を設定した `user` がその値を取り下げたことを検知し、C はその値を捨てるに同意します。プローブの結果はそのことを示しています。そして、C はネットワークのほかの部分にその事実を伝えます。この情報は最終的に F にまで伝播します。F は、自分の値が 77 だと信じ続ける根拠がなくなったことに気づきます。このようにして、F もまた自分の値を捨てることになります。これもプローブによって表示されています。

これで F は値を持たなくなつたので、F に 212 を設定できるようになります。

```
(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を通して伝播されると、`C` の値が 100 に設定され、それが `C` のプローブに記録されます。`F` から `C` を計算するのにも、`C` から `F` を計算するのにも、完全に同じネットワークが使われていることに注意してください。この計算の無方向性は、制約ベースシステムの際立った特徴です。

制約システムの実装

制約システムは、Section 3.3.4 のデジタル回路シミュレータとよく似たやり方で、局所状態を持つ手続きオブジェクトによって実装します。制約システムの基本オブジェクトはもう少し複雑なものになりますが、予定表や論理遅延について考える必要がない分、システム全体としてはこちらのほうが単純です。コネクタに対する基本演算は次の通りです。

- `(has-value? <connector>)` は、コネクタが値を持っているかどうかを返します。
- `(get-value <connector>)` は、コネクタの現在の値を返します。
- `(set-value! <connector> <new-value> <informant>)` は、`informant` がコネクタに対して新しい値を設定するよう要求しているということを示しています。
- `(forget-value! <connector> <retractor>)` は、`retractor` が現在の値を忘れるよう要求しているということをコネクタに対して伝えます。
- `(connect <connector> <new-constraint>)` は、新しい制約に参加するようコネクタに伝えます。

コネクタは、手続き `inform-about-value` と `inform-about-no-value` によって制約と通信します。`inform-about-value` は与えられた制約にコネクタが値を持ったことを伝え、`inform-about-no-value` は制約にコネクタが値を失ったことを伝えます。

`adder` は、`a1, a2` という加数コネクタと `sum` コネクタと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（下記の `me` 手続き）を持つ手続きとして実装します。

```
(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、その加算器を値として返します。加算器を表現する手続き me は、局所手続きを起動するディスパッチャの役割を果たしています。

```

(define (inform-about-value constraint)
  (constraint 'I-have-a-value))
(define (inform-about-no-value constraint)
  (constraint 'I-lost-my-value))

```

加算器の局所手続き process-new-value は、コネクタのうちひとつが値を持ったことが加算器に伝えられたときに呼び出されます。加算器は、まず a1 と a2 の両方が値を持っているかチェックします。持っているなら、二つの加数の和を値として設定するよう sum に伝えます。set-value! の informant 引数

は、加算器オブジェクト自身である `me` になります。`a1` と `a2` の両方が値を持っているわけではない場合、`a1` と `sum` が値を持っているかもしれないのに、加算器はそれをチェックします。もしそうなら、`a2` にはその二つの差を設定します。最後に、`a2` と `sum` が値を持っているなら、加算器には `a1` の値を設定するのに十分な情報があることになります。コネクタのひとつが値を失ったことが加算器に伝えられると、加算器はすべてのコネクタに値を捨てるよう要求します（実際に捨てられるのはこの加算器によって設定された値だけです）。それから、加算器は `process-new-value` を実行します。この最後のステップの理由は、ひとつ以上のコネクタがまだ値を持っているかもしれません（つまり、元々その加算器によって設定されたのではない値を持っていたということ）、それらの値は加算器を通して伝播させ直す必要があるかもしれませんからです。

乗算器は加算器に非常によく似ています。乗算器は、因数の片方が 0 なら、もう片方の値がわからなくて `product` を 0 に設定します。

```
(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 や I-lost-my-value のメッセージが定数箱に送られるとエラーを発生させます。

```

(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に値が設定されたり解除されたりしたことについて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

コネクタは手続きオブジェクトとして表現します。局所状態変数として、コネクタの現在の値 `value`、コネクタの値を設定したオブジェクト `informant`、コネクタが参加する制約のリスト `constraints`を持ちます。

```

(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き `set-my-value` は、コネクタの値を設定するよう要求されたときには呼びられます。コネクタが現時点では値を持っていなければ、その値を設定し、値の設定を要求してきた制約を `informant` として覚えておきます。³² それからコネクタは、値の設定を要求してきた制約を除き、参加している制約すべてに通知を行います。これは、次のイテレータによって実現します。このイテレータは、与えられたひとつを除いて、リストのすべての項目に対して指定された手続きを適用します。

```

(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))

```

コネクタが値を忘れるよう依頼されると、局所手続き `forget-my-value` を実行します。この手続きは、その要求が元々その値を設定したオブジェクトと同じオブジェクトから来たものかどうかを確認します。そうであれば、コネクタは関連づけられた制約に値を失ったことを伝えます。

局所手続き `connect` は、もしリストになければ、制約リストに新しい制約を加えます。それから、コネクタが値を持っているなら、新しい制約にその事実を伝えます。

³²`setter` は制約とは限りません。温度の例では、`user` を `setter` として使いました。

コネクタの手続き `me` は、ほかの内部手続きへのディスパッチを行う役割を果たし、またコネクタのオブジェクトとしての表現にもなっています。以下の手続きは、ディスパッチ用の構文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: 三つのコネクタ `a`, `b`, `c` を入力として取り、`c` の値が `a` と `b` の値の平均となるような制約を作る手続き `averager` を、基本の乗算器・加算器・定数制約を使って定義せよ。

Exercise 3.34: Louis Reasoner は、`square` という制約を作りたいと思っている。`square` は二つの端子を持ち、二つ目の端子のコネクタ `b` は常に一つ目の端子の値 `a` の二乗になる。彼は、ひとつの乗算器からなる次の単純な装置を提案した。

```
(define (square a b) (multiplier a a b))
```

このアイデアには重大な欠陥がある。説明せよ。

Exercise 3.35: Ben Bitdiddle は Louis に、Exercise 3.34 の問題を避ける方法のひとつは、`square` を新しい基本制約として定義することだと伝えた。Ben が書いた手続き概略の欠けている部分を埋め、そのような制約を実装せよ。

```
(define (square a b)
  (define (process-new-value)
    (if (has-value? b)
        (if (< (get-value b) 0)
            (error "square less than 0: SQUARE")
            (get-value b))
        (alternative1)))
  (alternative2)))
```

```
(define (process-forget-value) (body1))
(define (me request) (body2))
(rest of definition)
me)
```

Exercise 3.36: グローバル環境で、次のような式の列を評価するとする。

```
(define a (make-connector))
(define b (make-connector))
(set-value! a 10 'user)
```

set-value! の評価中のある時点で、コネクタの局所手続きの中の以下の式が評価されることになる。

```
(for-each-except
  setter inform-about-value constraints)
```

上記の式が評価される環境を表す環境図を描け。

Exercise 3.37: celsius-fahrenheit-converter 手続きは、以下のような式指向スタイルによる定義に比べると面倒だ。

```
(define (celsius-fahrenheit-converter x)
  (+ (* (/ x 9) 5)
     32))
(define C (make-connector))
(define F (celsius-fahrenheit-converter C))
```

ここで、`c+`, `c*` 等は“制約”バージョンの数値演算である。例えば、`c+` は二つのコネクタを引数として取り、加算器制約によってそれらと関連づけられたコネクタを返す。

```
(define (c+ x y)
  (let ((z (make-connector)))
    (adder x y z)
    z))
```

似たような手続き `c-`, `c*`, `c/`, `cv(定数)` を定義し、上記の変換器の例のように複合制約を定義できるようにせよ。³³

3.4 並行性: 期限厳守

ここまで、モデル化のための道具として、局所状態を持つ計算オブジェクトというものが持つ力について見てきました。しかし、Section 3.1.3で警告したように、この力は代償を伴うものです。その代償とは、参照透明性を失ない、結果として同一性と変化ということについて厄介な問題を引き起こしたこと、評価の置換モデルを放棄し、より複雑な環境モデルを採用せざるを得なくなつたことです。

³³式指向形式を使うと計算の中間の式に名前をつける必要がなくなるので便利です。元々の制約言語による定式化は面倒でした。これは、多くの言語で複合データを扱うのが面倒なのと同じです。例えば、変数がベクトルを表現しているとして、積 $(a+b) \cdot (c+d)$ を求めたいとすると、指定されたベクトル引数に値をセットするが手続き自身はベクトルを値として返さないような手続きを使って、“命令型スタイル”で作業をすることもできます。

```
(v-sum a b temp1)
(v-sum c d temp2)
(v-prod temp1 temp2 answer)
```

別の方針として、ベクトルを値として返す手続きを使うことによって、式によって扱うこと也可能です。こうすると、明示的に `temp1` と `temp2` を使う必要がなくなります。

```
(define answer (v-prod (v-sum a b) (v-sum c d)))
```

Lisp では、手続きの値として複合オブジェクトを返すことができるので、ここまで命令型スタイルで作ってきた制約言語を、この練習問題のように式指向のスタイルに変形するということができます。複合データを扱う手段に乏しい Algol, Basic, Pascal(明示的に Pascal のポインタ変数を使わない限り)では、複合オブジェクトを操作しようとすると、命令型スタイルに縛られることになります。式指向形式の利点を考えると、この節でやったように命令型スタイルでシステムを実装する意味はあるのかと思う人もいるかもしれません。理由のひとつは、式指向でない制約言語はコネクタオブジェクトだけでなく制約オブジェクトに対するハンドル(例えば、`adder` 手続きの値)を提供するということです。コネクタ演算を通して間接的に制約を扱うのではなく、制約と直接通信する新しい演算によってシステムを拡張したいという場合には、このことが役に立ちます。命令型の実装によって式指向スタイルを実装するのは簡単なのですが、その逆はとても難しいのです。

状態、等価性、変化の複雑性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って計算モデルの中に *time(時間)* という概念を含めなければならなくなつたということです。代入を導入するまでは、値を持つ式はいつでも同じ値を持つという意味で、プログラムには時間という概念がありませんでした。それに対して、Section 3.1.1 の最初で導入した、銀行口座から引き出しを行い差引残高を返すということのモデル化の例を思い出してみましょう。

```
(withdraw 25)  
75  
(withdraw 25)  
50
```

ここでは、同じ式を連続して評価して、違う値が返されています。このふるまいは、代入文 (この場合、変数 `balance` への代入) を実行することが、値が変化する *moments in time(瞬間)* を示していることによります。ある式を評価した結果は、式そのもの以外に、その評価が起こったのがこれらの瞬間の前なのか、それとも後なのかということによっても変わることになります。局所状態を持つ計算オブジェクトによってモデルを構築すると、プログラミングの本質的な概念として、時間という問題に取り組む必要が出てきます。

物理世界の認知に合うように、計算モデルの構造化をさらに進めることができます。現実世界のオブジェクトは、ひとつずつ順番に変化するということはありません。オブジェクトは、*concurrently(並行)* に、つまり同時に動いているように認識されています。そのため、システムのモデル化する際に、並行に動く計算プロセスの集合という形で行なうことが自然になることはよくあります。別々の局所状態を持ったオブジェクトを使ってモデルを作り上げることによってプログラムをモジュール式にできるのと同じように、独立して並行に実行が進むいくつかの部品に計算モデルを分割するのが向いているということがあります。プログラムは逐次型コンピュータで実行することになるとしても、あたかも並行に実行されるかのようにプログラムを書くと、プログラムは本質的でない時間の制約を避けるようになり、そのためプログラムのモジュール性が高まります。

プログラムのモジュール性を高めるほかに、並行計算は逐次計算に対してスピード的な利点もあります。逐次計算は一度にひとつの演算しか実行できないので、タスクの実行時間は実行する演算の数に比例します。³⁴ しかし、もし

³⁴ 実際のプロセッサの多くは、*pipelining(パイプライン)* という戦略によって、同時にいくつかの演算を実行します。このテクニックはハードウェアの利用効率を大きく改善するのですが、これは直列の命令列の実行スピードを上げるために使われ、逐次的

問題を比較的独立した、時々しか通信を行わないいくつかの部品に分割できるなら、それぞれの部品を別々のプロセッサに割り当てる可能性があります。そうすると、使えるプロセッサの数に比例して速度を上げることができるようになります。

残念ながら、並行性という概念を持ち込むと、代入によって生じた複雑性がさらに問題のあるものになります。並行実行という現象は、並列に動いているのが現実世界であってもコンピュータであっても、時間というものの理解をさらに複雑にします。

3.4.1 並行システムにおける時間の性質

表面的には、時間というものは単純なもののように見えます。時間は、事象につけられる順番です。³⁵ 任意の事象 A と B があれば、ありうるパターンは A が B の前に起こる、 A と B は同時に起こる、 A は B の後に起こるというどれかになります。例えば、銀行口座の例に戻って考えてみましょう。最初の時点で 100 ドル入っている共同口座から Peter は 10 ドルを引き出し、Paul は 25 ドルを引き出して、口座には 65 ドルが残るとします。二回の引き出しの順番によって、残高の推移は $\$100 \rightarrow \$90 \rightarrow \$65$ か $\$100 \rightarrow \$75 \rightarrow \$65$ のどちらかになります。銀行システムのコンピュータ実装では、この変わっていく残高の推移は、変数 `balance` への連続した代入としてモデル化できます。

しかし、状況が複雑になると、そのような見方には問題が出てきます。Peter や Paul やほかの人たちが、世界中に分散した ATM のネットワークを通して、同じ銀行口座にアクセスしていると考えてみましょう。実際の口座残高の推移は、アクセスのタイミングや機械同士の通信の詳細に決定的に依存することになります。

このように、イベントの順番が決まっていないということは、並行システムを設計するうえで深刻な問題になります。例えば、Peter と Paul による引き出しが、共通の変数 `balance` を共有する二つの別々のプロセスとして実装されいるとします。各プロセスは、Section 3.1.1 の手続きによって規定されるものです。

```
(define (withdraw amount)
  (if (>= balance amount)
```

プログラムのふるまいはそのままです。

³⁵ケンブリッジの建物の壁に描かれた落書きを引用すると、“時間とは、何もかもが同時に起こらないようにするために発明された仕掛けだ”

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

もしこの二つのプロセスが独立に動作するとすると、次のようなことが起こる可能性があります。まず、Peter は残高を確認して、正当な金額を引き出そうとします。しかし、Peter が残高を確認してから引き出しを終えるまでの間に、Paul がお金を引き出してしまって、Peter の確認結果が無効になってしまいます。

もっとひどいことになる可能性もあります。それぞれの引き出しプロセスの一部として、次の式が実行されるとなります。

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

この式は、三つのステップからなります。(1) 変数 `balance` の値にアクセスし、(2) 新しい残高を計算し、(3) `balance` にこの新しい値を設定する、というものです。もし Peter と Paul の引き出し手続きがこの文を並行に実行したとすると、二つの手続きが `balance` にアクセスして新しい値を設定する順番が入れ違いになってしまふ可能性もあります。

Figure 3.29のタイミング図は、`balance` が 100 の状態から始まって、Peter が 10 引き出し、Paul が 25 を引き出しているのに、`balance` の最終的な値が 75 となるという状況で、イベントの発生順を表したものです。図に示したように、こういうおかしなことになった原因は、Paul の `balance` への 75 の代入が、引き算する `balance` の値が 100 であるという前提のもとで行われているということです。しかし、この前提は Peter が `balance` を 90 に変更したときに無効になっています。システム内のお金の総額が保たれていないので、これは銀行システムとしてはどうしようもない失敗です。二つの取引の前は、お金の総額は 100 ドルでした。取引の後は、Peter が 10 ドル、Paul が 25 ドル、銀行が 75 ドルを持っていることになっています。³⁶

³⁶二つの `set!` 演算が同時に残高を変更しようとすると、もっとひどい失敗が起こる可能性があります。この場合、メモリ上に残る実際のデータは、二つのプロセスが書き込んだ情報がでたらめに混ぜ合わされたものになってしまふかもしれません。ほとんどのコンピュータでは、基本メモリ書き込み演算にはインターロックがあり、そのような同時アクセスを防いでいます。しかし、このような一見単純な保護の仕組みさえ、マルチプロセスコンピュータを設計する際には実装上の問題が出てきます。設計にあたっては、メモリアクセスの速度を上げるために別々のプロセッサの間でデータが複製（“キャッシュ”）されていても、プロセッサ間で一貫したメモリ内容の見え方を保つことを保証するために、精巧な *cache-coherence*（キャッシュコヒーレンス）プロトコルが必要となります。

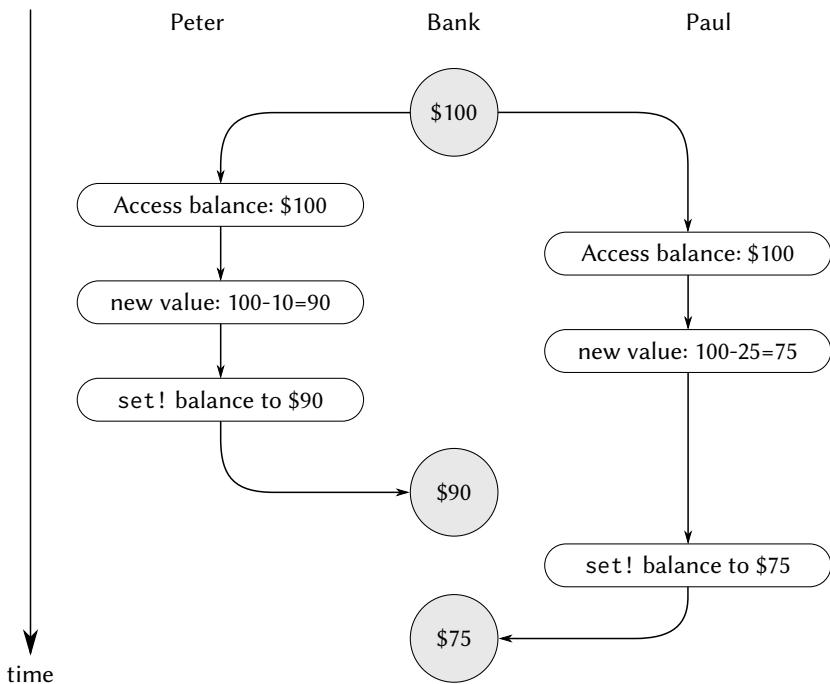


Figure 3.29: 二回の引き出しで、イベントの順番が入れ違いになって最終的に間違った残高になってしまう様子を示したタイミング図

ここでの例は、複数のプロセスが共通の状態変数を共有するという一般的な現象を示しています。これが面倒になるのは、二つ以上のプロセスが同時に共有された状態を操作しようとするかもしれないということによります。銀行口座の例では、それぞれの取引の間は、それぞれの顧客がほかに顧客がいないかのようにふるまうことができる必要があります。ある顧客が口座残高に依存する形で残高を更新しようとすると、変更のちょうどその前の瞬間まで口座が自分の考えている通りの残高を持っているという前提で顧客が行動できるようにしなければなりません。

並行プログラムの正しいふるまい

上の例は、並行プログラムに入り込みがちなバグの典型例を示しています。この複雑性の根源には、異なるプロセスの間で共有される変数に対する代入があります。ここまですでに、`set!`を使うプログラムを書く際には気をつけなければならないということがわかりました。これは、計算の結果が代入の起こる順番に依存するからです。³⁷ 並行プロセスでは、代入には特に気をつける必要があります。これは、別々のプロセスが行う代入の順番はコントロールできるとは限らないからです。そのような変更が（二人の預金者が共同口座にアクセスする場合のように）いくつか同時に起こる可能性があるなら、何らかの方法でシステムの正常な動作を保証しなければなりません。例えば、共同口座からの引き出しの場合であれば、金額が変わらないことを保証する必要があります。並行プログラムを正常に動作させるためには、並行実行に何らかの制限を加えることが必要になるかもしれません。

考えられる制限のひとつとして、共有された状態を変更するようなどんな二つの演算も同時に起こってはならないと決めてしまうことです。これは非常に厳しい要求です。分散銀行システムの場合であれば、同時にはひとつの取引しか行えないことを保証するようシステム設計者に要求することになります。これは非効率で、また保守的すぎます。Figure 3.30は、Peter と Paul が口座を共有していて Paul はほかに個人口座も持っているという状況を示したものです。図は、共有口座からの二回の引き出し（Peter が一回、Paul が一回）と、Paul の個人口座への預金を示しています。³⁸ 共有口座からの二回の引き出し

³⁷ Section 3.1.3の階乗プログラムは、单一の逐次処理の場合にもこの問題が起こることを示したものです。

³⁸ それぞれの列は、Peter の財布、(Bank1 内の) 共同口座、Paul の財布、(Bank2 内の) Paul の個人口座の中身が、それぞれの引き出し (W) と預け入れ (D) の前後でどうなっているかを示しています。Peter は Bank1 から 10 ドルを引き出しています。Paul は

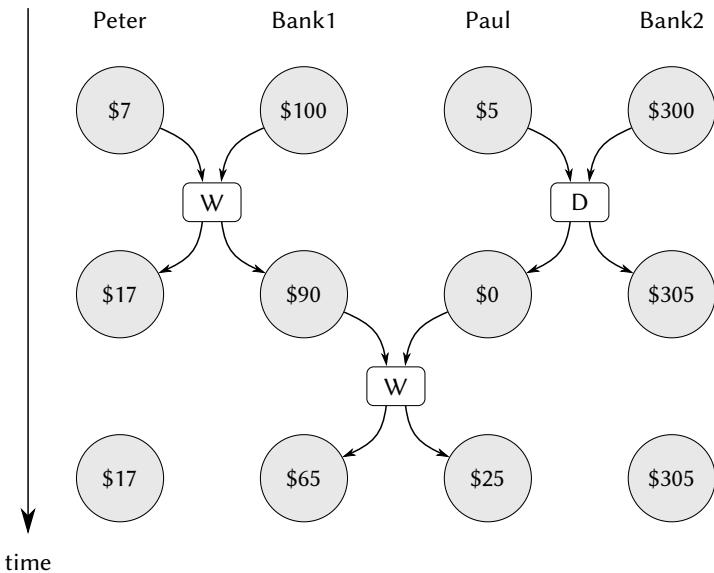


Figure 3.30: 銀行 1 の共同口座と銀行 2 の個人口座に対する並行な預け入れと引き出し

は、並行であってはいけません（どちらも同じ口座にアクセス・更新しているからです）。また、Paul の預金と引き出しは並行であってはいけません（どちらも Paul の財布にアクセス・更新を行っているからです）。しかし、Paul が自分の個人口座に預け入れを行うのと、Peter が共有口座から引き出しを行うのは、許可しても問題ないはずです。

並行性に対するもう少し緩い制限として、各プロセスが何らかの順番で逐次的に実行された場合と同じ結果になることを保証するというものがあります。この要求には、二つの重要な側面があります。一つ目は、各プロセスが別々に実行される必要はなく、あたかもそれらのプロセスが逐次的に実行されたかのような結果さえ出せばいいということです。Figure 3.30 の例でいうと、銀行口座システムを設計する人は、Paul の預け入れと Peter の引き出しが同時

5 ドルを Bank2 に預金し、次に Bank1 から 25 ドルを引き出しています。

に起こるようにも問題ないことになります。全体としての結果は、二つの操作が逐次的に行われたのと同じことになるからです。二つ目は、並行プログラムの実行結果には二つ以上の可能な“正解”がありうるということです。プログラムの結果は、何らかの順序で逐次的に実行したものと同じになつていればよいかどうです。例えば、Peter と Paul の共同口座の残高が最初は 100 ドルで、Peter が 40 ドルを預け入れると並行に Paul が残高の半額を引き出すという場合について考えてみましょう。この場合、逐次実行では、残高は最終的に 70 ドルか 90 ドルのどちらかになります (Exercise 3.38 参照)。³⁹

並行プログラムを正しく動作させる要件として、これより弱いものもあります。例えば、(物体内での熱の流れのような) 拡散シミュレーションプログラムを数多くのプロセスから構成されるものとして書く場合について考えてみます。それぞれのプロセスは小さな空間を表現し、値を並行に更新するものです。それぞれのプロセスは、自分自身の値と近傍の値の平均を取って、連続して自分の値を変えていきます。このアルゴリズムは、演算の行われる順番とは独立に、正しい解に収束します。共有される値を並行に使用する際に制限を設ける必要はありません。

Exercise 3.38: Peter, Paul, Mary が共同口座を持っていて、最初は 100 ドル入っているとする。以下に示すコマンドによって、Peter は 10 ドルを預け入れ、Paul は 20 ドルを引き出し、Mary は口座残高の半額を引き出すということを並行的に行う。

```
Peter: (set! balance (+ balance 10))
Paul: (set! balance (- balance 20))
Mary: (set! balance (/ balance 2)))
```

- これらの 3 つの取引が完了した後で `balance` が取り得る異なる値をすべて列挙せよ。銀行システムは、三つのプロセスが何らかの順序で逐次的に実行されるようにするとする。
- もしシステムがプロセスの実行が入れ違いになることを許すとすると、ほかにどのような値になる可能性があるだろうか。Figure 3.29 のようなタイミング図を描き、それらの値がどのようにして起こるか説明せよ。

³⁹ この考え方をより形式的に表現すると、並行プログラムは本質的に *nondeterministic*(非決定的) であるということになります。つまり、並行プログラムは单一の値を持つ関数として記述するのではなく、取り得る値の集合という形で結果を持つ関数として記述することになります。Section 4.3 では、非決定的計算を表現する言語について学びます。

3.4.2 並行性制御メカニズム

ここまで見てきたように、並行プロセスを扱ううえで難しいところは、別々のプロセスのイベントの順番が入れ違いになることを考えないといけないことによるものです。例えば、二つのプロセスがあって、片方は (a, b, c) 、もう片方は (x, y, z) という、それぞれ順序つきの三つのイベントを持っているとします。もしこの二つのプロセスが、実行が入れ違いになることに対する制限がない状態で並行して動作するとすると、個別プロセス内での順序と矛盾しないイベントの順序として、20通りが考えられることになります。

(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あたっては、これら20通りの順序の効果を考え、それぞれのふるまいが受け入れられるものかどうかチェックしなければならなくなります。そのようなやり方は、プロセスとイベントの数が増えるにつれて、あっという間に手に負えなくなります。

より現実的な並行システムの設計アプローチとしては、プログラムのふるまいの正しさを保証できる形で並行プロセスの入れ違い方を制約できるような一般的なメカニズムを考えるというものがあります。これまで、このことを目的とした多くのメカニズムが開発されてきました。この節では、その中のひとつであるserializer(直列変換器)について説明します。

共有状態へのアクセスの直列化

直列化というのは、次のような考え方を実装したものです。「プロセスは並行して実行されるが、並行して実行できない手続きの集合がある」。より正確には、直列化では互いに区別される複数の手続きの集合を作り、それぞれの直列化された集合の中では同時にひとつのプロセスしか実行されないようにするというものです。集合内の手続きが実行されているときには、その集合内の何かの手続きを実行しようとするプロセスは、一つ目の実行が終わるまで待たされることになります。

直列化は共有変数へのアクセスの制御に使えます。例えば、共有変数の値をその変数のそれまでの値に基づいて更新したいというとき、その変数のそれ

までの値に対するアクセスとその変数への新しい値の割り当てと同じ手続き内に置くようにします。それから、その変数に値を割り当てるほかの手続きもすべて同じ直列変換器にまとめることで、この手続きとそれらの手続きが同時に実行されないようにします。こうすることで、変数へのアクセスとそれに対応する値の割り当てとの間で変数の値が変更されないことが保証できます。

Scheme のシリアルライザ

上のメカニズムをもう少し具体的に考えるために、`parallel-execute` という手続きを持つ拡張 Scheme があると仮定します。

```
(parallel-execute <p1> <p2> ... <pk>)
```

各 $\langle p \rangle$ は、引数なしの手続きである必要があります。`parallel-execute` は、各 $\langle p \rangle$ に対して独立したプロセスを作り、それらのプロセスは $\langle p \rangle$ を（引数なしで）適用します。これらのプロセスは、すべて並行に実行されます。⁴⁰

これの使い方の例として、以下のものについて考えてみましょう。

```
(define x 10)
(parallel-execute
  (lambda () (set! x (* x x)))
  (lambda () (set! x (+ x 1))))
```

これは、二つの並行プロセスを作ります。 x に x かける x を設定する P_1 と、 x に 1 を足す P_2 です。実行が終わった後には、 P_1 と P_2 のイベントの入れ違い具合によって、 x は可能な五通りの値のうちのひとつになります。

- 101: P_1 が x に 100 を設定し、次に P_2 が x を 101 に増やす
- 121: P_2 が x を 11 に増やし、次に P_1 が x を $x * x$ に設定する
- 110: P_1 が $(* x x)$ を評価する際の x の値に対する二回のアクセスの間に P_2 が x を 10 から 11 に変える
- 11: P_2 が x にアクセスし、それから P_1 が x を 100 に設定し、それから P_2 が x を設定する
- 100: P_1 が x に（二回）アクセスし、それから P_2 が x を 11 に設定し、

⁴⁰`parallel-execute` は標準 Scheme の一部ではありませんが、MIT Scheme で実装することができます。ここでの実装では、新しい並行プロセスはオリジナルの Scheme プロセスと並行に実行できます。また、ここでの実装では、`parallel-execute` によって返される値は、新しく作ったプロセスを停止させるのに使える特別なコントロールオブジェクトです。

それから P_1 が x を設定する

直列化手続きを使うことによって、並行性に制約を加えることができます。直列化手続きは、*serializers(シリアルライザ)* によって作ります。シリアルライザは、*make-serializer* により構築されます。この実装は後で示します。シリアルライザは手続きを引数に取り、元の手続きと同じようにふるまう直列化手続きを返します。あるひとつのシリアルライザに対する呼び出しは、すべて同じ集合に属する直列化手続きを返します。

このため、次のものを実行すると、上の例とは異なり、 x の二つの可能な値である 101 または 121 しか返しません。

```
(define x 10)
(define s (make-serializer))
(parallel-execute
 (s (lambda () (set! x (* x x))))
 (s (lambda () (set! x (+ x 1)))))
```

ほかの可能性は排除されています。これは、 P_1 と P_2 の実行が入れ違いにならないようになっているからです。

Section 3.1.1 の *make-account* 手続きの、預け入れと引き出しが直列化されたバージョンを以下に示します。

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (protected withdraw))
            ((eq? m 'deposit) (protected deposit))
            ((eq? m 'balance) balance)
            (else (error "Unknown request: MAKE-ACCOUNT"
                         m)))))

  dispatch))
```

この実装では、二つのプロセスがひとつの口座に並行に預け入れをしたり引き出しをしたりすることはできなくなっています。こうすることで、Figure 3.29で図示したような、Paul が新しい値を計算するために残高にアクセスしてから Paul が実際に代入を行うまでの間に Peter が口座残高を変えてしまうといったエラーの原因は排除されます。一方、二つの口座はそれぞれ自分のシリアルайザを持っているので、別々の口座に対する預け入れと引き出しは並行に行うことができます。

Exercise 3.39: 並行実行の結果として上で示した五つの可能な値のうち、実行を次のように直列化したら、どれが残るだろうか。

```
(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: 以下のものを実行したときに x が取り得る値をすべて列挙せよ。

```
(define x 10)
(parallel-execute (lambda () (set! x (* x x)))
                  (lambda () (set! x (* x x x))))
```

以下の直列化手続きを使うと、これらの可能性のうちどれが残るだろうか。

```
(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、次のように銀行口座を実装したほうがいいのではないかと心配している（変更箇所はコメントつきの行）。

```
(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)))) ;直列化
          (else
            (error "Unknown request: MAKE-ACCOUNT"
                  m)))))

dispatch))

```

その理由は、銀行口座に対する非直列化アクセスを許していると、異常なふるまいが起こるかもしれないということだ。その意見に賛成するか。Ben の心配を実証するようなシナリオは存在するだろうか。

Exercise 3.42: Ben Bitdiddle は、`withdraw` と `deposit` メッセージが来るたびに新しい直列化手続きを作るのは時間の無駄だと提案した。彼の意見は、`make-account` を変更して、`protected` に対する呼び出しを `dispatch` 手続きの外で行うようにしたらしいというものだ。つまり、口座が引き出し手続きを要求されたときに、(口座作成と同時に作った) 同じ直列化手続きを返すことになる。

```

(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)))
  (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)))

```

これは安全な変更だろうか。具体的には、これら二つのバージョンの `make-account` によって許される並行性に違いはあるだろうか。

複数の共有リソース使う際の複雑性

シリアルライザは、並行プログラムの複雑性を隔離し、複雑性を注意深く、また(おそらくは)正しく扱うことができるよう、強力な抽象化を提供しています。しかし、共有リソースが(ひとつの銀行口座のように)ひとつしかない場合にはシリアルライザを使うのは比較的素直にうまくいくのですが、共有リソースが複数ある場合には、並行プログラミングは落とし穴の多い難しいものになります。

起こりうる困難のひとつを示すため、二つの銀行口座の残高を交換したいという例について考えます。この場合、それぞれの口座にアクセスして残高を調べ、残高同士の差を計算し、その差の分だけ片方の口座から引き出し、もうひとつの口座に預け入れることになります。これは、次のように実装できます。

⁴¹

```

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

```

⁴¹ `deposit` メッセージが負の金額を受け入れるということを利用して `exchange` を簡単にしています(これはこの銀行システムの深刻なバグです!)。

```
((account2 'deposit) difference)))
```

この手続きは、交換をしようとしているのがひとつのプロセスだけの場合にしかうまくいきません。ここで、Peter と Paul の二人が両方とも口座 a_1 , a_2 , a_3 にアクセスし、Peter が a_1 と a_2 を交換する間に Paul が並行に a_1 と a_3 を交換するという場合について考えてみましょう。個別口座に対する預け入れと引き出しが(この節の上で示した `make-account` 手続きのように)直列化されていましたとしても、`exchange` はやはり不正確な結果になる可能性があります。例えば、Peter が a_1 と a_2 の残高の差を求めるとき、Peter が交換を終える前に Paul が a_1 の残高を変更してしまうかもしれません。⁴² 正しいふるまいをさせるためには、交換が終わるまでの間ずっと、どちらの口座からもほかの並行アクセスを排除するように配慮する必要があります。

これを実現する方法のひとつとして、両方の口座のシリアライザを使って `exchange` 手続き全体を直列化するというものがあります。そのためには、口座のシリアライザにアクセスする準備をします。シリアライザを露出させることで、銀行口座オブジェクトのモジュール性を意図的に破っていることに注意してください。`make-account` の以下のバージョンは、Section 3.1.1 の元のバージョンと同じですが、balance 変数を守るためにシリアライザが提供されていることと、そのシリアライザがメッセージパッシングによってエクスポートされていることが違います。

```
(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)
```

⁴² 口座残高が最初 10 ドル, 20 ドル, 30 ドルだったとすると、並行の交換を何回行っても、順番はともかく、残高は 10 ドル, 20 ドル, 30 ドルのままでなければなりません。個別口座に対する預け入れを直列化するだけでは、このことを保証するには十分ではありません。Exercise 3.43 参照。

```

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

```

これを使って、預け入れと引き出しを直列化して行うことができます。しかし、前の直列化口座とは違って、直列化を明示的に管理するのは銀行口座オブジェクトのそれぞれのユーザの責任になっています。以下はその一例です。⁴³

```

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

```

このようにシリアルайザをエクスポートすることによって柔軟性が得られ、交換プログラムを直列化したものを実装できるようになります。単純に、両方の口座のシリアルайザによって元の `exchange` 手続きを直列化するだけです。

```

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

```

Exercise 3.43: 三つの口座の残高が最初は 10 ドル、20 ドル、30 ドルで、それから複数のプロセスが実行されて口座の残高を交換するとする。プロセスが逐次的に実行されたとしたら、並行の残高交換を何回行ったとしても、口座残高は何らかの順番で 10 ドル、20 ドル、30 ドルであるということを論証せよ。この章の最初のバージョンの `account-exchange` プログラムを使って残高交換を実装するとこの条件が破られることがあるということを、Figure 3.29のようなタイミング図を描いて示せ。一方、この `exchange` プログラ

⁴³Exercise 3.45では、なぜ預け入れと引き出しを自動的に口座によって直列化されないようになっているのかを調査します。

ムを使ったとしても、口座残高の合計は保存されることを論証せよ。個別口座の取引を直列化しなければこの前提まで破られるということをタイミング図を描いて示せ。

Exercise 3.44: ある口座から別の口座に振込をするという問題について考える。Ben Bitdiddle は、たとえ複数の人々が複数の口座間で並行にお金の振込をするという場合であっても、以下の手続きによって達成できると主張している。この手続きでは、例えば上で出てきたバージョンの `make-account` のように、預け入れと引き出しの取引を直列化する口座側の何らかのメカニズムを使う。

```
(define (transfer from-account to-account amount)
  ((from-account 'withdraw) amount)
  ((to-account 'deposit) amount))
```

Louis Reasoner は、これには問題があり、残高交換問題を扱う際に必要となったような、より洗練された手法を使う必要があると主張する。Louis は正しいだろうか。もし正しくないとしたら、振込問題と残高交換問題の本質的な違いは何だろうか (`from-account` の残高は `amount` 以上あると考える)。

Exercise 3.45: Louis Reasoner は、預け入れと引き出しが自動的に直列化されないようになったことで、この銀行口座システムが必要以上に複雑になり、またエラーが起こりやすくなつたと考えている。彼は、`make-account-and-serializer` が (`serialized-exchange` のような手続きで使えるように) シリアライザをエクスポートするだけでなく、その上で `make-account` のように口座と預け入れを直列化するためにもそれを使うようにしたほうがいいと提案した。彼は、口座を次のように再定義することを提案している。

```
(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、預け入れは元の `make-account` と同じように扱われるこ
とになる。

```

(define (deposit account amount)
  ((account 'deposit) amount))

```

Louis の考え方のどこがおかしいのか説明せよ。具体的には、`serialized-exchange` が呼ばれたときにどうなるか考えよ。

シリアルライザの実装

シリアルライザは、`mutex`(ミューテックス) と呼ばれる、より基本的な同期メカニズムによって実装します。ミューテックスは、`acquire(獲得)` と `release(解放)` という二つの演算をサポートするオブジェクトです。ミューテックスが一度獲得されると、そのミューテックスに対するほかの獲得演算は、そのミューテックスが解放されるまで続行できません。⁴⁴ ここで実装では、それぞれのシリアルライザは関連づけられたミューテックスをひとつ持っています。手続き `p` が与えられると、シリアルライザはミューテックスを獲得し、`p` を実行し、それから `mutex` を解放するという手続きを返します。これによって、このシリアルライザによって生成された手続きのうちひとつしか同時に実行されないことを保証します。これこそが、保証しなければならない直列化の特性です。

⁴⁴ “`mutex`” という用語は *mutual exclusion*(相互排除) の短縮形です。並行プロセスがリソースを安全に共有できる仕組みを準備するという一般的な問題は、相互排除問題と呼ばれます。ここでのミューテックスは、*semaphore*(セマフォ) メカニズムをちょっと変えて単純にしたもので ([Exercise 3.47](#) 参照)。セマフォというメカニズムは、アントホーフェン技術大学で開発された “THE” マルチプログラミングシステムで導入されたものです。“THE” という名前は、この大学のオランダ語のイニシャルに由来します ([Dijkstra 1968a](#))。獲得と解法の演算は、元は P と V と呼ばれていました。これらは鉄道で使われる腕木信号との関連で、オランダ語の単語 *passeren*(渡す) と *vrijgeven*(解放する) から来ています。ダイクストラの有名な解説 ([Dijkstra 1968b](#)) は、明確に並行性制御の問題を提示した最初期のもののひとつで、セマフォを使ってさまざまな並行性の問題を扱う方法を示しています。

```
(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)))
```

ミューテックスは、真または偽の値を持つことができる可変オブジェクト(ここでは1要素のリストを使い、それを`cell(セル)`と呼ぶことにします)です。値が偽のときはミューテックスは獲得可能です。値が真のときはミューテックスは使用中で、このミューテックスを獲得しようとするプロセスは待たざることになります。

ここでミューテックスのコンストラクタ`make-mutex`は、まずセルの中身を偽に初期化します。このミューテックスを獲得するには、セルを確認します。もしミューテックスが使用可能であれば、セルの中身を真にして続行します。そうでなければ、ミューテックスが使用可能になるまで、ループしながら何度も獲得を試みます。⁴⁵ ミューテックスを解放するには、セルの中身に偽を設定します。

```
(define (make-mutex)
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell)
                 (the-mutex 'acquire))) ; retry
            ((eq? m 'release) (clear! cell))))
      the-mutex))
  (define (clear! cell) (set-car! cell false)))
```

⁴⁵ほとんどの時分割OSでは、ミューテックスでブロックされたプロセスは、上記のように“ビジーウェイト”で時間を無駄にすることはありません。そうする代わりに、システムは最初のプロセスの待ち時間の間にほかのプロセスが動くようにスケジュールし、ブロックされたプロセスはミューテックスが使用可能になったときに起こされることになります。

`test-and-set!` は、セルをテストし、テスト結果を返します。それに加えて、もしテストが偽であれば、`test-and-set!` は偽を返す前にセルの中身に真を設定します。このふるまいは、次のような手続きとして表現できます。

```
(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。ここに非常に重要な難しい問題があります。これは並行性制御がシステムに入り込むことになる根本的なところなのですが、それは `test-and-set!` 演算は *atomically*(アトミックに) 実行されなければいけないということです。これは、あるプロセスがセルをテストし偽だと知ったら、ほかのプロセスがそのセルをテストするより前に、そのセルの内容は実際に真に設定されるよう保証しなければならないということを意味します。この保証をしなければ、Figure 3.29で銀行口座が失敗したのと同じように、ミューテックスが失敗することになります (Exercise 3.46参照)。

`test-and-set!` の実際の実装は、システムが並行プロセスをどのように実行するかに依存します。例えば、並行プロセスを動かすのに、逐次的なプロセッサ上で時分割メカニズムを使い、各プロセスに短い時間を割り当てて動かしたのちに割り込みをして次のプロセスに処理を移すというやり方でプロセスを巡回するというやり方があります。この場合、`test-and-set!` はテストと値の設定の間は時分割を停止するということによってうまくいきます。⁴⁶ また、マルチプロセスコンピュータはアトミック演算を直接ハードウェアでサポートする命令を提供します。⁴⁷

⁴⁶ 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!` は以下のように実装できます。

```
(define (test-and-set! cell)
  (without-interrupts
    (lambda ()
      (if (car cell)
          true
          (begin (set-car! cell true)
                 false)))))
```

`without-interrupts` は、引数の手続きが実行されている間、時分割割り込みを無効にするものです。

⁴⁷ そのような命令には、`test-and-set`, `test-and-clear`, `swap`, `compare-and-exchange`, `load-reserve`, `store-conditional` など、さまざまなものがあります。これらの設計は、そ

Exercise 3.46: テキストで示したように、`test-and-set!` を通常の手続きを使って実装し、演算をアトミックにしようとはしないと仮定する。Figure 3.29のようなタイミング図を描き、このミューテックスの実装が、二つのプロセスが同時にミューテックスを獲得するのを許してしまうという失敗を犯すことになるのはどのような場合か示せ。

Exercise 3.47: (サイズ n の) セマフォとは、ミューテックスの一般化である。ミューテックスと同じように、セマフォは獲得と解放の演算をサポートするが、並行に n プロセスまで獲得できるという意味でより一般的になっている。それ以上のプロセスがセマフォを獲得しようとすると、解放演算を待つ必要がある。セマフォの実装を、以下それぞれの条件で行え。

- a ミューテックスを使う
- b アトミックな `test-and-set!` 演算を用いる

デッドロック

シリアルライザの実装方法について学んだため、今度は上記の `serialized-exchange` を使っても口座の残高交換にはまだ問題があるというところを見ていきます。ここで、Peter が a_1 と a_2 の残高を交換しようとしたとき、Paul が並行に a_2 と a_1 の残高を交換しようとしている想像してみましょう。Peter のプロセスは、 a_1 を保護する直列化手続きに入った時点に達し、そのすぐ後に、Paul のプロセスが a_2 を保護する直列化手続きに入ったとします。このとき、 a_2 を保護する直列化手続きから Peter が抜けるまで、Peter は進む (a_2 を

の機械のプロセッサ - メモリ間インターフェイスに注意深く合わせたものでなければいけません。ここで起こる問題のひとつとして、そのような命令を使って二つのプロセスがちょうど同じ時間に同じリソースを獲得しようとした場合に何が起こるかを決めることがあります。この場合には、どちらのプロセスが制御を得るかについての決断をするメカニズムが必要になります。そのようなメカニズムは、*arbiter*(アービタ) と呼ばれます。アービタは通常、最終的には何らかのハードウェアデバイスに行き着きます。残念なことに、アービタの決定に任意の長い時間を許さない限り、100% の確率でちゃんと動作する公平なアービタを構築することは物理的に不可能だと証明可能です。ここでの基本的な現象について最初に触れているのは、アリストテレスの天体論に対する 14 世紀のフランス人学者ジャン・ビュリダンの注釈です。ビュリダンは、等しく魅力的な二つの食べ物の間に置かれた完全に理性的な犬は、どちらに最初に行くか決めることができないため飢え死にすると主張しました。

保護する直列化手続きに入る) ことができません。同じように、*a1* を守る直列化手続きから Peter が抜けるまで、Paul は進むことができません。どちらのプロセスも、お互いを待って、永遠に立ち往生することになります。この状況は*deadlock(デッドロック)* と呼ばれます。複数の共有リソースに対して並行なアクセスを提供するシステムでは、常にデッドロックに気をつける必要があります。

この状況でデッドロックを避けるひとつの方法として、それぞれの口座に固有の ID 番号を振り、プロセスが手続きに入る際に最小の番号を持つ口座を最初に保護するように *serialized-exchange* を書き換えるというものがあります。この手法は交換問題にはうまくいくのですが、さらに洗練されたデッドロック回避テクニックが必要になる状況や、デッドロックがまったく防げないという状況もあります ([Exercise 3.48](#)、[Exercise 3.49](#) 参照)⁴⁸

Exercise 3.48: 上で説明したデッドロック回避手法（口座に番号を振り、各プロセスは小さな番号の口座から獲得を試みる）によって、残高交換問題でのデッドロックが防げるのはなぜか、詳細に解説せよ。この考え方を取り入れるように *serialized-exchange* を書き直せ (*make-account* も変更して、各口座が番号つきで作られ、適切なメッセージを送ることによってその番号にアクセスできるようにする必要がある)。

Exercise 3.49: 上で説明したデッドロック回避メカニズムがうまくいかない場合のシナリオを示せ（ヒント：残高交換問題では、各プロセスは自分がどの口座にアクセスする必要があるか事前に知っている。プロセスが、どの追加リソースが必要になるかわからない状態で共有リソースにアクセスする必要があるという状況を考えよ）。

並行性、時間、通信

ここまで、並行システムのプログラミングでは、異なるプロセスが共有状態にアクセスする際にイベントの順番を制御することが必要となるということ

⁴⁸ 共有リソースに番号を振り、順番に獲得するという一般的なデッドロック回避技術は、[Havender \(1968\)](#) によります。デッドロックが防げない状況では、*deadlock-recovery* (デッドロック回復) 手法が必要となります。そのような手法では、プロセスにデッドロック状態から“身を引かせて”、再試行させます。デッドロック回復メカニズムはデータベース管理システムで広く使用されています。これについては、[Gray and Reuter 1993](#) で詳しく取り上げられています。

を見てきました。さらに、シリアルайザをうまく使うことによってその制御を行う方法についても見ました。しかし、並行性の問題はこれよりも深いところにあります。根本的な観点から見ると、“共有状態”というものが何を意味するかが常にしつきりしているとは限らないからです。

`test-and-set!` のようなメカニズムは、任意の時間にグローバルな共有フラグを調べることをプロセスに要求します。これは、現代の高速なプロセッサで実装するには問題が多く、効率がよくありません。そのようなプロセッサでは、パイプラインやキャッシュメモリといった最適化技術のために、すべての瞬間にメモリの内容が一貫した状態にあるとは限らないからです。そのため、最近のマルチプロセスシステムでは、シリアルайザという枠組みは新しい並行性制御法に取って代わられつつあります。⁴⁹

共有状態の難しい側面は、大きな分散システムでも問題になります。例えば、分散銀行システムで、個別の支店が残高のローカルな値を保持していて、これらの値をほかの支店で保持されているものと定期的に比較するという場合について想像してみましょう。そのようなシステムでは、“口座残高”的な値は同期直後以外では不確定になります。もし Peter が、Paul との共同口座にお金を預けたとしたら、口座残高が変わったのはいつということになるでしょうか。地域支店が残高を変更したときでしょうか、それとも同期の後でしょうか。また、もし Paul が別の支店から口座にアクセスしたとすると、ふるまいが“正しい”といえるようなものにするためには、銀行システムにどのような制約を加えるのが妥当でしょうか。正しさについて考えるうえで意味があるのは、Peter と Paul によって個別に観測されるふるまいと、同期直後の口座の“状態”的なものです。“本当の”口座残高がいくらかとか、同期の間のイベントの順番というのは、重要でなかったり、意味がなかったりするかもしれません。⁵⁰

ここで基本的な現象は、異なるプロセスの同期や、共有状態の設定や、イベントの順番を割り当てるといったことには、プロセス間の通信が必要となる

⁴⁹ このような直列化の代替手段のひとつとして、*barrier synchronization*(バリア同期)と呼ばれるものがあります。プログラマは並行プロセスを好き勝手に実行させますが、ある一定の同期ポイント(“バリア”)を設定しておき、すべてのプロセスがそのバリアに着くまで先に進めないようにするというものです。現代のプロセッサは、一貫性が必要となる部分にプログラマが同期ポイントを設定できるよう、機械語命令を提供しています。例えば、POWERPC はこの目的のために SYNC(同期) と EIEIO(Enforced In-order Execution of Input/Output) と呼ばれる二つの命令を含んでいます。

⁵⁰ これは奇妙な見方のように思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に対する国際課金は、通常は国ごとに精算され、別々の国で行われた課金は定期的に調整されます。このため、口座残高が国によって異なるということがあります。

ということです。本質的に、並行性制御における時間という概念は、通信と密接に結びついている必要があります。⁵¹面白いことに、相対論でも時間と通信との間には似たようなつながりがあります。相対論では、光速(イベントの同期に使える最も高速な信号)は、時間と空間に関する基本的な定数です。計算モデルの中での時間と状態を扱う際に遭遇する複雑性は、実際のところ、物理的宇宙の基本的な複雑性を反映しているかもしれません。

3.5 ストリーム

ここまででは、モデル化の道具としての代入について理解し、また代入が引き起こす複雑な問題に対して評価してきました。ここからは、別の取り組み方をすることによってこれらの問題のいくつかを回避することはできなかつたのかということについて考えていきます。この節では、状態をモデル化する別のアプローチについて検討します。このアプローチは、*streams*(ストリーム)と呼ばれるデータ構造に基づくものです。ここから先で見ていくように、状態をモデル化する際に生じる複雑性の一部をストリームによって軽減することができます。

一步下がって、この複雑性というものがどこから来ているのか再検討してみましょう。現実世界の現象をモデル化しようとする中で、見たところもつともな決定をいくつかしてきました。局所状態を持つ現実世界のオブジェクトを局所変数を持つ計算オブジェクトによってモデル化し、現実世界の時間の変化をコンピュータ上の時間変化と同一視し、コンピュータ上のモデルオブジェクトの状態の経時変化はモデルオブジェクトの局所変数に対する代入として実装しました。

ほかのアプローチはあるでしょうか。モデル対象の世界での時間とコンピュータ上の時間を同一視することは避けられないのでしょうか。変化する世界での現象をモデル化するには、時間とともにモデルを変化させるしかないのでしょうか。この問題について、数学関数を使って考えてみましょう。量 x の時間によって変化するふるまいは、時間の関数 $x(t)$ として記述できます。もし時刻ごとの x に注目するなら、それを変化する量としてとらえることになります。しかし、全時間を通しての値の履歴に注目するなら、変化については重要視し

⁵¹分散システムでは、この観点はLamport (1978)によって深く研究されています。彼は、通信によって“グローバル時計”を作り、それを使って分散システムでイベントの順番を確立する方法を示しました。

ないことになります。関数そのものは変化しないことになるからです。⁵²

もし時間を離散ステップで測るのであれば、時間関数は(無限かもしれない)列としてモデル化できます。この節では、モデル化対象システムの時間履歴を表現する列によって変化をモデル化する方法について見ていきます。これを達成するために、*stream*(ストリーム)と呼ばれる新しいデータ構造を導入します。抽象的な視点から見ると、ストリームはただの列です。しかし、ストリームを(Section 2.2.1のように)素直にリストとして実装したものでは、ストリーム処理の本来の力が見えてきません。その代わりに、ここでは*delayed evaluation*(遅延評価)というテクニックを導入します。遅延評価を使うことによって、非常に大きな(無限でもある)列をストリームとして表現することが可能になります。

ストリーム処理によって、状態のあるシステムを代入や可変データを使わずにモデル化することができるようになります。このことは、論理的にも実用的にも大きな意味があります。代入の導入につきもののマイナス点を避けてモデルを構築できるからです。一方、ストリームという枠組みにはそれはそれで難しいところがあります。どちらのモデル化テクニックが、よりモジュール化され、よりメンテナンスしやすいシステムの実現につながるかということは、未解決の問題です。

3.5.1 ストリーム=遅延リスト

Section 2.2.3で見てきたように、列はプログラムモジュールを組み合わせるための標準インターフェイスとして使うことができます。また、`map`, `filter`, `accumulate`のような列操作の強力な抽象化について定式化してきました。これらは、簡潔でエレガントに広い範囲の演算を表すことができるものです。

残念ながら、列をリストとして表すと、エレガントさのために計算に必要な時間と空間の効率性を大幅に犠牲にすることになります。列の操作をリストの変形として表すと、プログラムは処理の各ステップで(非常に大きいかもしれない)データ構造を構築し、コピーしないといけないことになります。

このことを確認するために、ある区間のすべての素数の和を求めるプログラム二つを比較してみましょう。一つ目のプログラムは、標準的な繰り返し

⁵²物理学では、運動について考える手段として粒子の“世界線”を導入することによって、この見方を採用することができます。また、Section 2.2.3では、信号処理システムに関して考えるうえではこれが自然な方法であると述べました。ストリームを信号処理に適用することについては、Section 3.5.3で検討します。

スタイルで書かれています。⁵³

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

二つ目のプログラムは、同じ計算をSection 2.2.3の列演算を使って行います。

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime?
                      (enumerate-interval a b))))
```

計算を実行する際に、一つ目のプログラムは、集積している和だけ格納すれば十分です。それに対して、二つ目のプログラムのフィルタは、`enumerate-interval` が区間内の数の完全なリストを構築し終わるまで、まったく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、それが `accumulate` に渡され、その後に和に畳み込まれます。そのような大きな中間ストレージは一つ目のプログラムでは必要ありません。一つ目のプログラムは、区間を順次列挙し、素数を生成しながら和に足していくと考えることができます。

リストを使うことによる非効率性は、次のような例で痛切に明らかになります。10,000 から 1,000,000 の区間に含まれる二番目の素数を計算するのに、列のパラダイムを使って、次の式を評価することによって行うような場合です。

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

この式は、確かに二番目の素数を見つけます。しかし、計算のオーバーヘッドはものすごいものになります。約 100 万の整数のリストを構築し、各要素の素数性をテストしてこのリストをフィルタし、それからほとんどすべての結果を無視することになってしまいます。より伝統的なプログラミングスタイルでは、

⁵³(Section 1.2.6で出てきたような) 素数性をテストする述語 `prime?` はすでにあるとします。

数え上げとフィルタをひとつにまとめ、二つ目の素数に到達した時点で停止することになるでしょう。

ストリームは、列をリストとして操作するコストを発生させることなしに列操作を使えるようにする巧妙な考え方でえます。ストリームを使うと、二つの世界のいいとこ取りができます。列操作としてプログラムをエレガントに定式化しながら、順次計算の効率性も得ることができます。基本的な考え方方は、ストリームを部分的しか構築しないで、構築した一部をそのストリームを消費するプログラムに渡すというものです。消費プログラムがストリームのまだ構築していない部分にアクセスしようとしたら、ストリームは自動的に必要な部分だけを構築し、ストリーム全体が存在するという幻想を維持します。言い換えると、プログラムはあたかも完全な列を扱っているかのように書くのですが、ストリームの実装は、ストリームの構築とストリームの使用が自動的かつ透過的に組み合わせられるように設計します。

表面的には、ストリームはただのリストで、それを走査する手続きの名前が違うだけです。ストリームにはコンストラクタ `cons-stream` と二つのセレクタ `stream-car`, `stream-cdr` があり、セレクタは以下の制約を満たしています。

```
(stream-car (cons-stream x y)) = x  
(stream-cdr (cons-stream x y)) = y
```

さらに、識別可能なオブジェクト `the-empty-stream` があります。これは決して `cons-stream` 演算の結果になることがなく、述語 `stream-null?` によって識別可能です。⁵⁴ これによって、リストを作ったり使ったりするのとまったく同じようにストリームを作ったり使ったりすることができ、列という形になっている集約データを表現することができます。具体的には、[Chapter 2](#)で学んだ `list-ref`, `map`, `for-each` のようなリスト演算について、ストリーム版を構築することができます。⁵⁵

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

⁵⁴MIT の実装では、`the-empty-stream` は空のリスト'()と同じもので、`stream-null?` は `null?` と同じものとなっています。

⁵⁵ここは気になるところだと思います。ストリームとリストに対してこのような似通った手続きを定義しているということは、何か隠れた抽象化を見逃しているということです。残念ながら、この抽象化を利用するためには、評価プロセスに対する細かい制御が必要なのですが、それは現時点ではまだできません。この点については、[Section 3.5.4](#) の末尾で詳しく検討します。[Section 4.2](#)では、リストとストリームを統合する枠組みを開発します。

```

(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)
  (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))

```

ストリームが自動的かつ透過的にストリームの構築と使用を組み合わせるよう実装するために、ストリームの cdr は、ストリームが cons-stream によって構築されたときでなく、 stream-cdr 手続きによってアクセスされたときに評価されるようにします。ここでの実装の選択は、Section 2.1.2で有理数について検討したときと似ています。有理数を実装する際には、分母と分子を既約にするのを構築時にするか選択時にするかを選ぶことができるということを見てきました。どちらの有理数の実装も結果として同じデータ抽象化になりますが、どちらを選ぶかによって効率に影響があります。ストリームと通常のリストの間にも、似たような関係があります。データ抽象化としては、ストリームはリストと同じです。違うの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 も cdr も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。

ここでは、 delay という特殊形式に基づいてストリームを実装します。`(delay <exp>)` を評価しても、式`<exp>` は評価されず、代わりにいわゆる *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>` を評価するという“約束”と考えることができます。`delay` と対になるものとして、 `force` という手続きがあります。これは、遅延オブジェクトを引数として取り、評価を実行するというものです。つまり、 `delay` に約束の遂行を迫るということになります。`delay` と `force` の実装方法については後ほど見ていくとして、

まずはこれらを使ってストリームを構築してみましょう。

`cons-stream` は、次の形が

```
(cons-stream <a> <b>)
```

以下と等価になるよう定義された特殊形式です。

```
(cons <a> (delay <b>))
```

つまり、ストリームを構築するのにはペアを使うということです。しかし、ストリームの残りの値を `cdr` に入れる代わりに、要求されることがあれば残りを計算するという約束を入れることにします。これで、`stream-car` と `stream-cdr` が次のように手続きとして定義できるようになります。

```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```

noindent `stream-car` はペアの `car` を選択します。`stream-cdr` はペアの `cdr` を選択し、そこに置かれた遅延表現を評価し、ストリームの残りを取得します。⁵⁶

ストリーム実装の動作

この実装のふるまいを見るために、上で見た“とんでもない”素数計算をストリームによって定式化し直したものを見解析してみましょう。

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

これが実際に効率的に動くということを見ていきます。

まず、`stream-enumerate-interval` を 10,000 と 1,000,000 という引数で呼び出します。`stream-enumerate-interval` は、`enumerate-interval` (Section 2.2.3) のストリーム版です。

⁵⁶ `stream-car` と `stream-cdr` は手続きとして定義できるのですが、`cons-stream` は特殊形式である必要があります。`cons-stream` が手続きだったとすると、私たちの評価モデルによると、`(cons-stream <a>)` を評価すると自動的に `` が評価されることになります。これは避けたいところです。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` は通常の手続きでも大丈夫です。

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
        low
        (stream-enumerate-interval (+ low 1) high))))
```

つまり、`stream-enumerate-interval` に返される結果は、`cons-stream` によって作られた次のようなものになります。⁵⁷

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

つまり、`stream-enumerate-interval` はストリームを返すのですが、そのストリームの `car` には 10,000 が入っていて、`cdr` には、要求されたら区間の数え上げを続けるという約束が入っていることになります。次に、このストリームにフィルタをかけて素数を残します。それには、ストリーム版の `filter` 手続き (Section 2.2.3) を使います。

```
(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`、つまり 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` を調べます。`stream-cdr` を呼び出すと、遅延化された `stream-enumerate-interval` の評価を強制します。評価すると、次のものが返ってきます。

```
(cons 10001
  (delay (stream-enumerate-interval 10002 1000000)))
```

⁵⁷ここで示した数値は、実際には遅延オブジェクトの中にはありません。実際にあるのは元の式で、環境の中で変数が適切な数値に束縛されています。例えば、10001 と示した場所には実際には `(+ low 1)` があり、その `low` が 10,000 に束縛されています。

`stream-filter` は、今度はこのストリームの `stream-car`、つまり 10,001 を見て、これも素数ではないので、さらに `stream-cdr` を強制します。これを `stream-enumerate-interval` が素数 10,007 を作り出すまで繰り返します。10,007 が返ってくると、`stream-filter` は定義通りに次のものを返します。

```
(cons-stream (stream-car stream)
             (stream-filter pred (stream-cdr stream)))
```

この場合は、以下のようになります。

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

この結果が、元の式の `stream-cdr` に渡されることになります。これによって、遅延化された `stream-filter` が強制され、`stream-filter` は次の素数である 10,009 が見つかるまで遅延化された `stream-enumerate-interval` の強制を続けます。最終的に、元の式の `stream-car` に渡される結果は次のようにになります。

```
(cons 10009
      (delay (stream-filter
              prime?
              (cons 10010
                  (delay (stream-enumerate-interval
                          10011
                          1000000))))))
```

`stream-car` は 10,009 を返し、これで計算が完了します。二つ目の素数を見つけるために必要なだけの整数のみがテストされ、素数フィルタに与えるのに必要な分だけの区間だけが列挙されています。

一般的に、遅延評価は“要求駆動”プログラミングと考えることができます。要求駆動プログラミングでは、ストリーム処理の各段階は、次の段階を満足させるのに十分なだけしか動かされません。ここまででやったことは、計算の中で起こることの実際の順番と、手続きの見かけ上の構造とを切り離すとい

うことです。手続きを書くときには、ストリームが“すべて同時に”存在するかのように考えて書くのですが、実際には、計算は伝統的なプログラミングスタイルの場合と同じように漸進的に実行されることになります。

delay と force の実装

delay と force はミステリアスな演算に見えるかもしれません、実際のところ、実装はとても単純です。delay は、後で要求されたときになって評価するように、式をパッケージする必要があります。これは、単純に式を手続きの本体として扱うことによって実現できます。delay は、次のようなシンタックス糖りによって

```
(delay <exp>)
```

以下のものを表すような特殊形式として実装できます。

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

force は、delay によって生成された(引数なしの)手続きを呼び出すだけです。ですので、force は次のような手続きとして実装できます。

```
(define (force delayed-object) (delayed-object))
```

delay と force が仕様通りに動くようになるだけなら、この実装で十分です。しかし、重要な最適化をひとつ取り入れる余地があります。多くの応用では、同じ遅延オブジェクトを何回も強制するはめになります。これは、ストリームを使った再帰プログラムで、深刻な非効率性の原因となることがあります(Exercise 3.57参照)。これを解決するには、遅延オブジェクトが最初に強制される際に計算した値を格納するように遅延オブジェクトを構築します。二回目以降の強制では、計算を繰り返すことはしないで、格納した値をそのまま返します。つまり、Exercise 3.27で説明したものと同じような、特別目的のメモ化手続きとして delay を実装するということです。これを実現する方法のひとつとして、(引数なしの)手続きをひとつ引数として取り、その手続きのメモバージョンを返す手続きを使うというものがあります。以下にその手続きを示します。メモ化手続きが一回目に実行されると、計算結果を格納します。二回目以降の評価では、その結果をそのまま返します。

```
(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))))
```

そして、(delay <exp>) が以下と等価になるように delay を定義します。

```
(memo-proc (lambda () (exp)))
```

force は以前の定義そのままです。⁵⁸

Exercise 3.50: 以下の定義を完成させよ。これは、Section 2.2.1Footnote 12 の map と同じように、複数の引数を取る手続きを使えるよう stream-map を一般化するものである。

```
(define (stream-map proc . argstreams)
  (if (<??> (car argstreams))
      the-empty-stream
      (<??>
       (apply proc (map <??> argstreams))
       (apply stream-map
              (cons proc (map <??> argstreams))))))
```

Exercise 3.51: 遅延評価についてより詳しく観察するために、引数を表示してからそのまま返すだけの以下の手続きを使うことにする。

```
(define (show x)
  (display-line x)
  x)
```

⁵⁸ストリームの実装は、この節で説明したもの以外にもいろいろ考えられます。ストリームを実用的なものにする鍵となっている遅延評価は、Algol 60 では *call-by-name*(名前呼び)という引数渡し手法にもともと備わっていました。このメカニズムを使ってストリームを実装するという考え方には、Landin (1965) で初めて記述されたものです。ストリームの遅延評価は、Friedman and Wise (1976) によって Lisp に導入されました。この実装では、*cons* は常に引数の評価を遅延させるようになっていて、そのためリストは自動的にストリームとしてふるまうようになっていました。メモ化による最適化は、*call-by-need*(必要呼び)としても知られています。Algol コミュニティであれば、ここでの元々の遅延オブジェクトは *call-by-name thunks*(名前呼びサンク)と呼び、最適化されたバージョンは *call-by-need thunks*(必要呼びサンク)と呼ぶところです。

以下のような一連の式を評価させると、インタプリタはどのような表示をするだろうか。

59

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

上記の式をそれぞれ評価した後、`sum` の値はいくつになるだろうか。式 `stream-ref` と `display-stream` を評価したとき、どのような応答が表示されるだろうか。`(delay <exp>)` を単純に `(lambda () <exp>)` として実装し、`memo-proc` によって提供される最適化

⁵⁹ Exercise 3.51 や Exercise 3.52 のような課題は、`delay` の動作についてちゃんと理解しているかどうかを確認できるという意味で、価値のあるものです。一方、遅延評価と表示を混在せると、頭が混乱しやすくなります。代入まで入ってくると、ますますひどくなります。コンピュータ言語の授業では、この節に出てきたような試験問題で講師が学生を苦しめるということが、伝統的によく行われてきています。言うまでもないことですが、そのような難しいところに依存するようなプログラムを書くというのは、醜い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、プログラムの中で実際に出来事が起こる順番を考えなくていいというところから来ています。残念ながら、代入があると、このような考え方方はできません。代入を使うときには、時間と変化について考えることが避けられないからです。

を使用しない場合、これらの応答は異なるものになるだろうか。説明せよ。

3.5.2 無限ストリーム

ここまで、実際にはアクセスする必要のある分しかストリームの計算をしていないのに、ストリームを完全な実体として扱っているという幻想を維持する方法について見てきました。このテクニックを使うと、列が非常に長いものでも、効率的にストリームとして表現することができます。さらに特筆すべきこととして、ストリームを使うと無限に長い列を表すこともできます。例えば、以下に示す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))
```

これは筋が通っています。`integers` は、`car` が 1 で、`cdr` が “2 から始まる整数を生成する約束” というペアになります。これは無限に長いストリームですが、しかしある時間のうちに調べられるのはそのその有限な一部だけです。そのため、プログラムからは無限ストリーム全体が存在するわけではないということはわかりません。

`integers` を使って、別の無限のストリームを定義することができます。例えば、7で割り切れない整数のストリームは以下のようになります。

```
(define (divisible? x y) (= (remainder x y) 0))
(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7)))
    integers))
```

こうすると、このストリームの要素にアクセスするだけで、7で割り切れない整数を探すことができます。

```
(stream-ref no-sevens 100)
117
```

`integers` と同じようなやり方で、フィボナッチ数の無限ストリームを定義することもできます。

```
(define (fibgen a b) (cons-stream a (fibgen b (+ a b))))
(define fibs (fibgen 0 1))
```

`fibs` は、`car` が 0 で、`cdr` が “(`fibgen 1 1`) を評価する約束” というペアです。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`) を評価する約束” というペアを生成し… という繰り返しになります。

もっと刺激的な無限ストリームについても見てみましょう。`no-sevens` の例を一般化すると、*sieve of Eratosthenes*(エラトステネスのふるい) という手法によって素数の無限ストリームを構築することができます。⁶⁰ まず、最初の素数である 2 から始まる整数列を考えます。残りの素数を得るには、まず整数列の残りから 2 の倍数をフィルタリングします。こうすると、次の素数である 3 から始まるストリームができます。次に、このストリームの残りから 3 の倍数をフィルタリングします。こうすると、次の素数である 5 から始まるストリームができます。これをずっと続けていきます。これは、言い換えると次のようなふるいのプロセスで素数列を作るということになります。あるストリーム `S` について、`S` の最初の要素を最初の要素とし、`S` の残りから `S` の最初の要素の倍数をフィルタリングした結果をふるいにかけたものを残りの要素とするようなストリームを作ることによって、`S` にふるいをかけるということです。このプロセスは、ストリーム演算によってすぐに記述できます。

```
(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、質問をするだけで特定の素数を得ることができます。

```
(stream-ref primes 50)
```

233

⁶⁰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ことでも有名です。彼は、夏至の日の正午の影を観察することによってこれを計算しました。エラトステネスのふるいは非常に古いものですが、最近まで巨大素数を見つける最も強力なツールであった特別目的ハードウェアによる“ふるい”は、このエラトステネスのふるいを基礎としています。しかし、70 年代からは、これらの手法はSection 1.2.6で検討したような確率的な技術の発展によって取って代わられています。

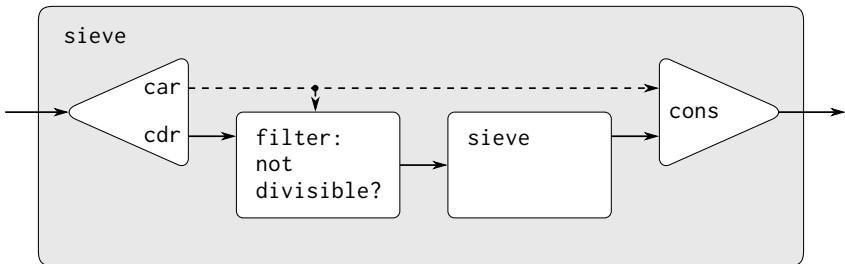


Figure 3.31: 信号処理システムとして見た素数のふるい

`sieve` によって作られる信号処理システムに注目すると面白いことがわかります。このシステムを“ヘンダーソン図”としてFigure 3.31に示します。⁶¹ 入力ストリームは“unconser”に流し込まれ、そこでストリームの最初の要素がストリームの残りから切り離されます。最初の要素を使って整除性フィルタが構築され、ストリームの残りがそのフィルタを通ります。このフィルタの出力は、また別のふるい箱に流されます。そして、元の最初の要素が内部のふるいの出力と `cons` され、それが出力ストリームとなります。つまり、無限なのはストリームだけではなく、信号処理器も無限だということです。ふるいは中にふるいを持っているからです。

ストリームの暗黙定義

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ひとつずつ計算する“生成”手続きを規定することによって定義されていました。ストリームを規定する別のやり方として、遅延評価を利用して暗黙的にストリームを定義するというものがあります。例えば、次の式は `ones` というストリームを 1 の無限ストリームとして定義します。

```
(define ones (cons-stream 1 ones))
```

⁶¹ 私たちは Peter Henderson にちなんでこれらの図をヘンダーソン図と呼んでいます。ストリーム処理について考える方法のひとつとして、この種の図を初めて見せてくれたのが Henderson だったからです。各実践は、転送される値のストリームを表しています。`car` から `cons` と `filter` に伸びる点線は、これがストリームではなく、単一の値であることを示しています。

これは再帰手続きの定義と同じような仕組みです。`ones` は、`car` が 1 で、`cdr` が `ones` を評価する約束というペアになります。`cdr` を評価すると、1 と、`ones` を評価する約束がまた得られます。この繰り返しです。

演算によってストリームを操作すると、もっと面白いことができます。例えば、`add-streams` という、二つの与えられたストリームの要素ごとの和を生成する演算について考えます。⁶²

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

こうすると、整数列を次のように定義することができます。

```
(define integers
  (cons-stream 1 (add-streams ones integers)))
```

これは、一つ目の要素が 1 で、残りが `ones` と `integers` の和であるようなものとして、`integers` を定義するものです。つまり、`integers` の二つ目の要素は、`1 + integers` の一つ目の要素で、2 になります。`integers` の三つ目の要素は、`1 + integers` の二つ目の要素で、3 になります。この繰り返しです。この定義がうまくいくのは、どの時点でも十分な `integers` ストリームが生成されているので、それを定義にフィードバックすることによって次の整数が生成できるからです。

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))))
```

この定義の意味は、`fibs` は 0, 1 から始まるストリームで、残りのストリームは自分自身に `fibs` をひとつずらしたもの足すことによって生成できるということです。

1	1	2	3	5	8	13	21	...	=	(stream-cdr fibs)		
0	1	1	2	3	5	8	13	...	=	fibs		
0	1	1	2	3	5	8	13	21	34	...	=	fibs

そのようなストリーム定義を式化するのに便利な手続きとしては、ほかに `scale-stream` というものもあります。これは、ストリームの各要素に与えられた定数をかけるものです。

⁶²これは、Exercise 3.50で出てきた一般化されたバージョンの `stream-map` を利用しています。

```
(define (scale-stream stream factor)
  (stream-map (lambda (x) (* x factor))
              stream))
```

例えば、

```
(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, ...

素数ストリームの別の定義の仕方は、整数列から始めて、素数性テストによってフィルタリングするというものです。まず、最初の素数である2が必要となります。

```
(define primes
  (cons-stream
    2
    (stream-filter prime? (integers-starting-from 3))))
```

この定義は、ある整数 n が素数かどうかをテストするのに、 \sqrt{n} 以下の（整数ではなく）素数で割り切れるかどうかをチェックすることになるので、思ったほど単純なものにはなりません。

```
(define (prime? n)
  (define (iter ps)
    (cond ((> (square (stream-car ps)) n) true)
          ((divisible? n (stream-car ps)) false)
          (else (iter (stream-cdr ps))))))
  (iter primes))
```

これは再帰定義です。`primes` は `prime?` 述語を使って定義されているのですが、`prime?` 自身は `primes` ストリームを使っているからです。この手続きが動くのは、どの時点でも、次にチェックする数の素数性をテストするのに十分なだけ `primes` ストリームが生成されているからです。つまり、素数性をテストするすべての n について、 n は素数であるか（その場合、 n を割り切る素数がすでに生成されています）、素数であるか（その場合、 \sqrt{n} より大きい素数 — n よりは小さい — がすでに生成されています）のどちらかです。⁶³

⁶³ この最後の点は非常に難しいところです。これは、 $p_{n+1} \leq p_n^2$ という事実によります（ここで、 p_k は k 番目の素数を表しています）。このような推定は立証がとても難しいものです。素数が無限に存在するということに対する古代のユークリッドによる証明は

Exercise 3.53: プログラムを実行することなしに、以下によって定義されるストリームの要素を述べよ。

```
(define s (cons-stream 1 (add-streams s s)))
```

Exercise 3.54: `add-streams` と同じような仕組みで二つの入力ストリームの要素ごとの積を生成する手続き `mul-streams` を定義せよ。これと `integers` ストリームを組み合わせて、(0 から始まる) n 番目の要素が $n+1$ の階乗となるような以下のストリームの定義を完成させよ。

```
(define factorials (cons-stream 1 (mul-streams (λ () (λ () ...))))
```

Exercise 3.55: 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手続き `partial-sums` を定義せよ。例えば、(`partial-sums integers`) は 1, 3, 6, 10, 15, ... というストリームになる。

Exercise 3.56: R. Hamming が提起した有名な問題に、素因数が 2, 3, 5 のみからなる正の整数を昇順に重複なく列挙するというものがある。すぐ思いつく方法は、整数をひとつずつ見ていって、2, 3, 5 以外の素因数がないかどうか調べるというものだ。しかし、これは非常に非効率である。これは、整数が大きくなるにつれて、要求を満たすものがどんどん少なくなっていくことによる。別の方針として、求める数値ストリームを s として、以下の事実に注目してみる。

- s は 1 から始まる。
- (`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`) についても言える。
- これらが s の要素のすべてである。

$p_{n+1} \leq p_1 p_2 \dots p_n + 1$ ということを示していますが、これよりずっとよい結果が証明されたのは 1851 年になってのことでした。この年、ロシア人の数学者 P. L. Chebyshev が、すべての n に対し、 $p_{n+1} \leq 2p_n$ であることを証明しました。この結果は元々 1845 年に推測されていたもので、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 22.3 節にあります。

つまり、これらの供給源から来る要素を組み合わせればよいということになる。そのために、二つの順序つきストリームを、重複を除きつつ、ひとつの順序つきストリームにまとめたものを返す手続き `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)))))))))
```

こうすると、求めるストリームは `merge` を使って次のように構築できる。

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

⟨??⟩ となっている箇所に欠けている式を埋めよ。

Exercise 3.57: `add-streams` 手続きによる `fibs` の定義を使って n 番目のフィボナッチ数を計算する際に、加算は何回実行されるだろうか。`(delay <exp>)` の実装を単純に `(lambda () <exp>)` として、[Section 3.5.1](#)で述べた `memo-proc` 手続きによって提供される最適化を使わない場合、加算回数が指数的に増加することを示せ。

Exercise 3.58: 以下の手続きによって計算されるストリームを解釈せよ。

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

(`quotient` は、二つの整数の、整数の商を返す基本要素である。
`(expand 1 7 10)` はどのような要素列を生成するだろうか。また、
`(expand 3 8 10)` はどうか。

Exercise 3.59: Section 2.5.3では、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の実装方法を学んだ。似たような方法で、以下のような *power series*(幂級数) を、

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

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

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

無限ストリームとして表現したものを扱うことができる。級数 $a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$ は、係数 $a_0, a_1, a_2, a_3, \dots$ を要素として持つ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。幂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、その級数の積分の係数(定

⁶⁴この練習問題は、必要呼びが Exercise 3.27 で述べた通常のメモ化と密接に関係していることを示しています。Exercise 3.27 では、局所テーブルを明示的に構築するために代入を使いましたが、ここでの必要呼びによるストリームの最適化では、ストリームの以前に強制された部分の値を格納することによって、事実上、そのようなテーブルを自動的に構築していることになります。

数項を除く) のストリーム $a_0, \frac{1}{2}a_1, \frac{1}{3}a_2, \dots$ を返す手続き `integrate-series` を定義せよ (結果は定数項を持たないため、冪級数を表していない。`integrate-series` を使う際には、適切な定数に `cons` することにする)。

- b 関数 $x \mapsto e^x$ は、自分自身の導関数となっている。つまり、 e^x と、 e^x の積分が、定数項 $e^0 = 1$ を除いて同じ級数になるとということになる。したがって、 e^x の級数は次のように生成できる。

```
(define exp-series
  (cons-stream 1 (integrate-series exp-series)))
```

\sin の導関数が \cos で、 \cos の導関数が負の \sin であるということに基づき、 \sin と \cos の級数を生成する方法を示せ。

```
(define cosine-series (cons-stream 1 (??)))
(define sine-series (cons-stream 0 (??)))
```

Exercise 3.60: Exercise 3.59 のように係数のストリームとして表現された冪級数では、級数の加算は `add-streams` により実装される。級数を乗算する以下の手続きの定義を完成させよ。

```
(define (mul-series s1 s2)
  (cons-stream (??) (add-streams (??) (??)))))
```

Exercise 3.59 の級数を使って $\sin^2 x + \cos^2 x = 1$ となることを確認することで、手続きをテストできる。

Exercise 3.61: S を、定数項が 1 の冪級数 (Exercise 3.59) とする。冪級数 $1/S$ 、つまり $SX = 1$ となるような級数 X を求めたいとする。 S_R を S の定数項の後の部分として、 $S = 1 + S_R$ と書く。すると、 X は以下のように解ける。

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

言い換えると、 X は定数項が 1 で、高次の項が $-S_R \cdot X$ として与えられる冪級数ということになる。この考え方を使って、定数項 1

を持つ冪級数 S に対する $1/S$ を求める手続き `invert-unit-series` を書け。Exercise 3.60 の `mul-series` を使う必要がある。

Exercise 3.62: Exercise 3.60 と Exercise 3.61 の結果を使って、2つの冪級数を割る手続き `div-series` を定義せよ。`div-series` は任意の 2 つの級数について動作するようにする（もし分母の定数項がゼロなら `div-series` はエラーを出すようする）。`div-series` と Exercise 3.59 の結果と組み合わせて `tan` の冪級数を生成する方法を示せ。

3.5.3 ストリームパラダイムの活用

遅延評価つきのストリームは強力なモデル化の道具として使うことができ、局所状態や代入が持つ利点の多くを提供します。そのうえ、プログラミング言語に代入を持ち込むときに生じる理論的に面倒なところをいくらか避けることができます。

ストリームというアプローチは、局所状態への代入を中心に組み立てるシステムとは違うモジュール境界を持つシステムを構築できるという意味で、学ぶところが多くあります。例えば、個別の瞬間における状態変数の値ではなく、時系列（または信号）全体を興味の中心に据えることができるようになります。このことで、異なる瞬間の状態の構成要素を組み合わせたり比較したりするのが便利になります。

ストリームプロセスとしての反復の定式化

Section 1.2.1 では、状態変数を更新することによる反復プロセスを導入しました。今では、状態を表現するのに、変数を更新していくのではなく、“時間のない” 値のストリームを使うことができます。この視点から Section 1.1.7 の平方根手続きを考え直してみましょう。そこで考え方は、推測値を改善する手続きを何度も適用することによって、 x の平方根の推測値の列を生成するというものでした。

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

元の `sqrt` 手続きでは、状態変数の一連の値がこれらの推測値となっていました。その代わりにここで考える方法は、推測値の無限ストリームを生成すると

いうものです。ストリームの最初の値は初期値の 1 です。⁶⁵

```
(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
...
```

ストリームの項を生成していくことによって、推測値を改善していくことができます。答えが十分よいものになるまで項を生成し続ける手続きを書くこともできます (Exercise 3.64 参照)。

このほかに、Section 1.3.1 で見た交代級数に基づいて π の近似値を生成する反復も同じように扱えます。

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

まず、級数の加数 (項ごとに符号が逆になる奇数の逆数) のストリームを生成します。次に、(Exercise 3.55 の partial-sums 手続きを使って、対象項数が増えていく部分和のストリームを作ります。それから結果を 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))
```

⁶⁵let を使って局所変数 `guesses` を束縛することはできません。`guesses` の値は `guesses` 自身に依存しているからです。Exercise 3.63 では、なぜここで局所変数が必要なのかを考えます。

```
(display-stream pi-stream)
4.
2.6666666666666667
3.4666666666666667
2.8952380952380956
3.3396825396825403
2.9760461760461765
3.2837384837384844
3.017071817071818
...

```

これは、収束は比較的遅いのですが、改善し続ける π の近似値のストリームになっています。列の8個の項で、 π の値は3.284から3.017の範囲になります。

ここまで範囲では、状態のストリームというアプローチは、状態変数を更新するというやり方とそれほど違いがありません。しかし、ストリームを使うと、面白いトリックを使う余地が出てきます。例えば、*sequence accelerator*(列加速)というものを使うと、ストリームを変形して、近似値の列を同じ値により速く収束する新しい列に変換することができます。

そのような加速のひとつとして、18世紀のスイスの数学学者 Leonhard Euler(レオンハルト・オイラー)によるものがありますが、これは交代級数(項ごとに符号が逆になる列)の部分和の列に対してうまく働きます。オイラーの手法では、 S_n を元の和の列の n 番目の項とすると、加速した列は次のような項を持つことになります。

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

したがって、元の列が値のストリームとして表されているなら、変換した列は以下によって与えられます。

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

これで、 π の近似値の列を使ってオイラーの加速を実際にやってみることができます。

```
(display-stream (euler-transform pi-stream))
3.16666666666667
3.133333333333337
3.1452380952380956
3.13968253968254
3.1427128427128435
3.1408813408813416
3.142071817071818
3.1412548236077655
...
```

さらに都合のよいことに、加速した列はさらに加速することができ、さらに再帰的にそれを加速し...と続けていくこともできます。つまり、ストリームのストリーム(この構造については、ここでは *tableau*(タブロー)と呼ぶことになります)を作ることになります。タブローの中では、それぞれのストリームはひとつ前のストリームを変換したものになっています。

```
(define (make-tableau transform s)
  (cons-stream s (make-tableau transform (transform s))))
```

タブローは以下のような形をしています。

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って列を作ります。

```
(define (accelerated-sequence transform s)
  (stream-map stream-car (make-tableau transform s)))
```

このタイプの“超加速”を π の列で実際にやってみます。

```
(display-stream
  (accelerated-sequence euler-transform pi-stream))
4.
3.16666666666667
```

```
3.142105263157895  
3.141599357319005  
3.1415927140337785  
3.1415926539752927  
3.1415926535911765  
3.141592653589778  
...
```

得られる結果は目覚ましいものです。8個の項を取ると、小数点以下14桁まで正しい π の値が求められます。元の π の列だけ使っていたら、同じぐらい正確な答えを求めようと思うと、 10^{13} 項ぐらいまで計算する(つまり、各項が 10^{-13} より小さくなるまで級数を展開する)必要があるところです。

これらの加速テクニックは、ストリームを使わないでも実装できたかもしれません。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。状態列全体をデータ構造として使うことができ、それを統一された演算の集合によって操作することができるからです。

Exercise 3.63: Louis Reasoner は、`sqrt-stream` 手続きを書くのに、局所変数 `guesses` を使わない以下のような素直な実装をなぜしないのかと尋ねた。

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

Alyssa P. Hacker は、このバージョンは冗長な計算を行うのでだいぶ非効率だと答えた。Alyssa の答えについて説明せよ。もし `delay` の実装が `memo-proc` (Section 3.5.1) による最適化を使わずに `(lambda () <exp>)` だけを使っていとすると、それでも二つのバージョンに効率の違いはあるだろうか。

Exercise 3.64: ストリームと数値(許容誤差)を引数として取る手続き `stream-limit` を書け。この手続きは、ストリームの二つの連続する要素の差の絶対値が許容誤差以下になるまでストリームを調べていき、その二つの値のうち後のものを返す。これを使うと、与えられた許容誤差まで平方根を求めることができる。

```
(define (sqrt x tolerance)
  (stream-limit (sqrt-stream x) tolerance))
```

Exercise 3.65: 級数

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

を使い、 π について上で行ったのと同様に、2の自然対数についても近似値の列を三つ計算せよ。これらの列はどのくらい速く収束するか。

ペアの無限ストリーム

Section 2.2.3では、列というパラダイムで、伝統的な入れ子ループをペアの列に対して定義されるプロセスとして扱う方法について見てきました。このテクニックを無限ストリームに一般化して、無限の集合について“ループ”しないといけないためにループとして簡単に表現できないようなプログラムを書くことができます。

例えばSection 2.2.3の prime-sum-pairs 手続きを一般化して、 $i \leq j$ の整数で $i + j$ が素数である整数ペア (i, j) すべてのストリームを作りたいとします。int-pairs を $i \leq j$ の整数ペア (i, j) すべての列とすると、必要なストリームは単純に次のようになります。⁶⁶

```
(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、二つのストリーム $S = (S_i)$ と $T = (T_j)$ があるとして、無限の長方形の配列を想像してみましょう。

$$\begin{array}{ccccccc}
(S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \dots \\
(S_1, T_0) & (S_1, T_1) & (S_1, T_2) & \dots \\
(S_2, T_0) & (S_2, T_1) & (S_2, T_2) & \dots \\
& \dots
\end{array}$$

⁶⁶Section 2.2.3と同じように、ここでは整数ペアを Lisp のペアとしてではなくリストとして表現します。

生成したいのは、対角線上またはその上にあるペア、つまり以下に示すようなペアすべてを含むストリームです。

(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

(S と T を両方とも整数のストリームとすると、これが求めるストリーム `int-pairs` になります。)

一般的なペアのストリームを `(pairs S T)` と呼び、それを三つの部分からなるものとして考えます。ペア (S_0, T_0) 、一行目の残りのペア、それ以外のペアです。⁶⁷

(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

この分解の三つ目の部分 (一行目以外にあるペア) は、`(stream-cdr S)` と `(stream-cdr T)` によって (再帰的に) 作られるペアです。また、二番目の部分 (一行目の残り) は以下のように表現されます。

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

よって、求めるペアのストリームは次のように作ることができます。

```
(define (pairs s t)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (<何らかの方法で組み合わせる>
      (stream-map (lambda (x) (list (stream-car s) x))
                  (stream-cdr t)))
    (pairs (stream-cdr s) (stream-cdr t)))))
```

手続きを完成させるには、二つの内部ストリームを組み合わせる何らかの方法を選ぶ必要があります。考え方のひとつとして、Section 2.2.1 の `append` 手続きのストリーム版を使うというものがあります。

⁶⁷このような分解方法を選ぶ理由となる洞察についてはExercise 3.68を参照してください。

```
(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向いていません。二つ目のストリームを組み込む前に、一つ目のストリームから要素をすべて取ることになるからです。具体的には、以下のようにしてすべての正の整数のペアを生成しようとすると、

```
(pairs integers integers)
```

結果のストリームは、一つ目の要素が 1 に等しいすべてのペアをまず列挙しようとします。そのため、ほかの値が一つ目の要素となるようなペアは永遠に作られないことになってしまいます。

無限ストリームを扱うには、プログラムを十分長い時間実行したら、すべての要素にいつかは到達することを保証するような組み合わせの順番を考え出す必要があります。これを達成するよくできた方法として、以下の `interleave` 手続きを使うというものがあります。⁶⁸

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

`interleave` は要素を二つのストリームから交互に取るので、最初のストリームが無限であっても、二つ目のストリームの要素はすべて、混ぜ合わせたストリームにいつかは入ることになります。

よって、求めるペアのストリームは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)))
```

⁶⁸組み合わせの順番について求められる性質は、正確には、一つ目のストリームの i 番目の要素と二つ目のストリームの j 番目の要素に対応するペアが出力ストリームの $f(i, j)$ 番目に出現するという二引数の関数 f が存在するということです。`interleave` を使ってこれを達成するトリックは David Turner が示したもので、彼は KRC 言語でこれを使っています。(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`) について詳しく観察せよ。ペアがストリームに配置される順番について、全体的なコメントをせよ。例えば、ペア (1, 100) の前にはおよそどれだけの数のペアが出てくるだろうか。(99, 100) と (100, 100) の場合はどうだろうか。(正確な数学的表現ができるならなおさらよい。しかし、行き詰まつたら定性的な答えでもよい。)

Exercise 3.67: `pairs` 手続きを修正し、(`pairs integers integers`) がすべての整数のペア $(i, j) (i \leq j)$ という条件なしで) 生成するようにせよ。ヒント：追加のストリームを混ぜる必要がある。

Exercise 3.68: Louis Reasoner は、ペアのストリームを構築するのに三つの部分に分けるのは必要以上に複雑なのではないかと考えた。彼は、ペア (S_0, T_0) を最初の行のそれ以外のペアと分ける代わりに、次のように最初の行全体を扱うことを提案した。

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

これはうまくいくだろうか。`(pairs integers integers)` を Louis の `pairs` の定義を使って評価した場合にどうなるか考えよ。

Exercise 3.69: 三つの無限ストリーム S, T, U を取り、 $i \leq j \leq k$ の三つ組 (S_i, T_j, U_k) のストリームを生成する手続き `triples` を書け。`triples` を使って、すべての正の整数のピタゴラス数の三つ組、つまり $i \leq j$ かつ $i^2 + j^2 = k^2$ であるような三つ組 (i, j, k) のストリームを生成せよ。

Exercise 3.70: ペアが出てくる順番が、アドホックな混ぜ合わせ処理の結果ではなく、何か役に立つ順番となっているようなストリームが作れれば便利だ。もし、ある整数のペアがほかのペ

アと比べて“小さい”かどうかを判断する方法を定義できるなら、Exercise 3.56の `merge` 手続きに似たテクニックを使うことができる。これを行う方法のひとつとして、“重み関数” $W(i, j)$ を定義し、 $W(i_1, j_1) < W(i_2, j_2)$ なら (i_1, j_1) は (i_2, j_2) より小さいと取り決めるというものがある。`merge` に似ているが、追加の引数 `weight` 手続きを取る `merge-weighted` を書け。`weight` はペアの重みを計算する手続きで、マージ結果のストリームの中で要素が現れる順番を決めるのに使われる。⁶⁹ これを使って、`pairs` を `weighted-pairs` という手続きに一般化せよ。`weighted-pairs` は、二つのストリームに加えて重み関数を計算する手続きを引数に取り、重みに従って並べられたペアのストリームを生成する。その手続きを使って、以下のそれぞれのストリームを生成せよ。

- a 和 $i + j$ の順に並んだ、すべての正の整数のペア $(i, j) i \leq j$ のストリーム
- b i と j のどちらも 2, 3, 5 で割り切れないすべての正の整数のペア $(i, j) i \leq j$ が $2i + 3j + 5ij$ の順に並んでいるストリーム

Exercise 3.71: 二つの立方数の和で表す方法が複数ある数は、数学者 Srinivasa Ramanujan(シュリニヴィーサ・ラマヌジャン) にちなんで *Ramanujan numbers*(ラマヌジャン数) と呼ばれることがある。⁷⁰ ペアの順序つきストリームを使うと、エレガントな方法でこれらの数を計算することができる。二つの立方数の和として表す方法が二通りある数を見つけるには、 $i^3 + j^3$ という和によって重みづけをした整数のペア (i, j) のストリームを生成し (Exercise 3.70 参照)、それから同じ重みを持つ連続したペアをストリームから探すだけよい。ラマヌジャン数を生成する手続きを書け。そのような最初の数は 1,729 である。次の五個を求めよ。

⁶⁹ペアの配列の中で、行沿いに右に移動したり、列沿いに下に移動したりすると、ペアの重みが増加するようになっていることが必要になる。

⁷⁰以下は 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追悼文 (Hardy 1921) からの引用です。“‘すべての自然数は彼の友達だった’と言ったのはリトルウッドだった（と思う）。彼が病気で Putney にいたとき、一度彼を見舞いに行つたときのことを覚えている。私が乗つて行ったタクシーの番号は 1729 だった。私は、この数字はあまり面白くない、よくない兆しでなければいいが、と言つた。すると彼は‘いいえ、とても面白い数字ですよ。それは、二つの立方数の和として表す方法が二つある最小の数です’と答えた。”重みづけをしたペアを使ってラマヌジャン数を生成するトリックは Charles Leiserson が示してくれたものです。

Exercise 3.72: Exercise 3.71と似たようなやり方で、二つの平方数の和として表す方法が三通りあるすべての数のストリームを生成せよ(そのように書ける理由も示せ)。

信号としてのストリーム

ストリームの話を始めたとき、ストリームは信号処理システムの“信号”に対するコンピュータ上での同等物であるという言い方をしました。実際、ストリームを使うと、信号処理システムを非常に直接的なやり方でモデル化することができます。このとき、間隔を置いて連続した信号の値をストリームの連続した要素として表現します。例えば、ある入力ストリーム $x = (x_i)$ と初期値 C 、小さい増分 dt に対し、和

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

を集積して、値 $S = (S_i)$ のストリームを返す*integrator(積分器)* (*summer(加算器)*とも)を実装することができます。以下の `integral` 手続きは、整数ストリームの“暗黙的なスタイル”的定義 (Section 3.5.2)に似ています。

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

Figure 3.32は、`integral`手手続きに相当する信号処理システムの図です。入力ストリームは dt 倍され、加算器を通り、その出力がまた同じ加算器に戻されます。`int`の定義にあった自己参照は、この図では加算器の出力を入力のひとつに接続するフィードバックループとなっています。

Exercise 3.73: ストリームを使って時系列上の電流や電圧の値を表すことによって電子回路をモデル化することができる。例えば、抵抗値 R の抵抗と静電容量 C のコンデンサが直列になっている RC circuit (**RC**回路)があるとする。注入電流 i に対する回路の電圧応答 v は Figure 3.33の式によって決まる。この構造は信号流れ図によって示してある。

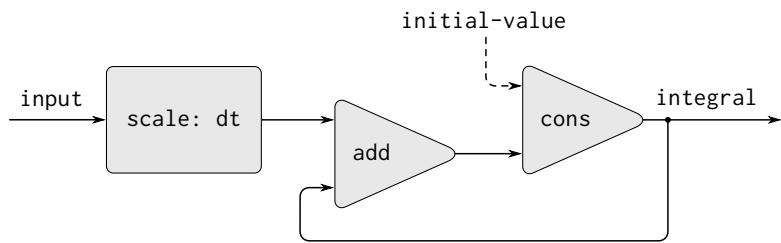


Figure 3.32: 信号処理システムとして見た integral 手続き

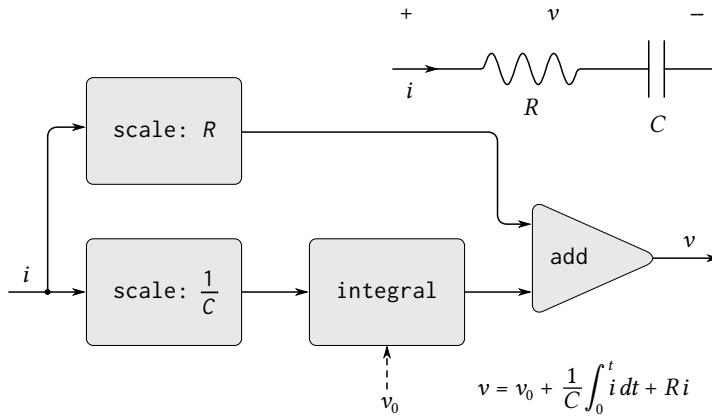


Figure 3.33: RC 回路と対応する信号流れ図

この回路をモデル化した手続き `RC` を書け。`RC` は値 R , C , dt を入力として取り、手続きを返す。返り値の手続きは、電流 i を表すストリームとコンデンサの初期電圧 v_0 を引数として取り、出力として電圧 v のストリームを生成する。例えば、`RC` を使って、(`define RC1 (RC 5 1 0.5)`) を評価することによって R が $5[\Omega]$ 、 C が $1[F]$ 、タイムステップが 0.5 秒の RC 回路をモデル化できる。これは、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手続き `RC1` を定義する。

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のひとつとして、入力信号の *zero crossings* (ゼロ交差) を記録する信号というものがある。これは、入力信号が負から正に変わったときに結果の信号が $+1$ となり、正から負に変わったときには -1 となり、その他の場合には 0 となるというものである (0 という入力の符号は正とする)。例えば、典型的な入力信号と、対応するゼロ交差信号は以下のようになる。

```
... 1 2 1.5 1 0.5 -0.1 -2 -3 -2 -0.5 0.2 3 4 ...
... 0 0 0 0 -1 0 0 0 0 1 0 0 ...
```

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、対応するゼロ交差のストリームはストリーム `zero-crossings` とする。Alyssa は、二つの値を引数として取り、値の符号を比較し、0, 1, -1 を適切に返す手続き `sign-change-detector` を書いた。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))
```

そこに通りがかった Alyssa の上司 Eva Lu Ator は、このプログラムは以下のプログラムとほぼ等価だと言った。このプログラムでは、Exercise 3.50 で出てきた、一般化したバージョンの `stream-map` を使っている。

```
(define zero-crossings
  (stream-map sign-change-detector
              sense-data
              (expression)))
```

`(expression)` によって示された部分を埋め、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 のゼロ交差識別器は十分でないということがわかった。センサからノイズの交じった信号が来ると、余計なゼロ交差を検出してしまうのだ。ハードウェアのスペシャリスト Lem E. Tweakit は、ゼロ交差を検出する前に信号を平滑化してノイズをフィルタリングすることを Alyssa に提案した。Alyssa は彼のアドバイスに従い、センサデータの値と直前の値との平均を取ることによって構築した信号から

ゼロ交差を検出することにした。彼女はこの問題について助手の Louis Reasoner に説明した。彼は、Alyssa のプログラムを次のように修正して、この考え方を実装しようとした。

```
(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探し、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 での Louis のアプローチに批判的だ。彼が書いたプログラムは、平滑化演算とゼロ交差検出が交じってしまっていて、モジュール化されていないというのが彼女の意見だ。例えば、入力信号を調整するよりよい方法を Alyssa が見つけていれば、検出器は変更する必要がなかったはずだ。Louis を手伝って、入力としてひとつのストリームを取り、入力ストリームの連続する二つの要素の平均を要素とするストリームを生成する手続き `smooth` を書け。次に、`smooth` をコンポーネントとして使い、ゼロ交差判定機をよりモジュラー化されたスタイルで実装せよ。

3.5.4 ストリームと遅延評価

前節の終わりのほうにあった `integral` 手続きは、ストリームを使うことによっ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する方法を示しています。Figure 3.32 に示した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` が自分自身によって定義されることによってモデル化されています。

```
(define int
```

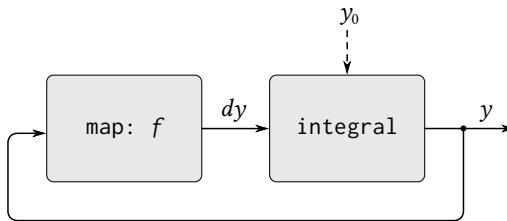


Figure 3.34: 方程式 $dy/dt = f(y)$ を解く“アナログ計算回路”

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

暗黙的定義のようなものを扱うインタプリタの能力は、`cons-stream`に組込まれている`delay`に依存しています。この`delay`がなかったとすると、インタプリタは`cons-stream`の二つの引数を評価してからでないと`int`を構築できないのですが、二つの引数を評価するには`int`がすでに定義されている必要がある、ということになってしまいます。一般的に、`delay`はループを含む信号処理システムをストリームを使ってモデル化するときに鍵となる部分です。`delay`がなければ、出力を返す前に、信号処理コンポーネントに対する入力が完全に評価されているようにモデルを定式化しなければいけません。この場合、ループは禁止されることになります。

残念ながら、ループを含むシステムに対するストリームモデルを構築するには、`cons-stream`の提供する“隠れた”`delay`だけでは足りず、それを超えて`delay`を使うことが求められることがあります。例えば、Figure 3.34は、 f を与えた関数として、微分方程式 $dy/dt = f(y)$ を解く信号処理システムを示しています。この図には、 f を入力信号に適用するマップコンポーネントがあります。マップコンポーネントは、このような方程式を解くために実際に使われるアナログ計算機回路と似たやり方で、積分器に対するフィードバックループに接続されています。

y の初期値 y_0 が与えられているとすると、モデル化に次のような手続きを使うことが考えられます。

```
(define (solve f y0 dt)
```

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

この手続きはうまくいきません。`solve` の一行目では `integral` を呼び出していますが、この呼び出しのためには入力 `dy` が定義されている必要があります。しかし、`dy` は `solve` の二行目まで定義されません。

一方で、この定義の意図するところは理解できます。原則として、`dy` を知らなくても、`y` のストリームの生成を始められるからです。実際、`integral` 等の多くのストリーム演算は、引数に関する部分的な情報しかない状況で答えの一部を生成できるという意味で、`cons-stream` に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` によって与えられます。そのため、出力ストリームの最初の要素は被積分値 `dy` を評価しなくとも生成できます。`y` の最初の要素さえわかれば、`solve` の二行目にある `stream-map` が動作し、`dy` の最初の要素を生成します。これによって `y` の次の要素が生成されます。以下、この繰り返しとなります。

この考え方を生かして、`integral` を定義し直し、被積分ストリームを *delayed argument*(遅延引数) として受け入れるようにします。`Integral` は、出力ストリームの二番目以降の要素の生成が要求されて初めて、被積分値を `force` し、評価されるようにします。

```
(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、`y` の定義の中で `dy` の評価を遅延させることによって、`solve` 手続きが実装できます。⁷¹

```
(define (solve f y0 dt)
```

⁷¹ この手続きは、すべての Scheme 実装で動くことが保証されているわけではありません。とはいえ、どの実装でも単純な変更で動作させることができます。問題は、Scheme 実装が内部定義を扱うやり方に微妙な違いがあるということにあります (Section 4.1.6 参照)。

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

一般的に、`integral` を呼び出す側は、被積分引数を `delay` しなければいけないようになっています。`solve` 手続きがちゃんと動作するということは、初期条件 $y(0) = 1$ として、微分方程式 $dy/dt = y$ の解について $y = 1$ での値を求めることで $e \approx 2.718$ の近似値を求めるこによって示すことができます。

```
(stream-ref (solve (lambda (y) y) 1 0.001) 1000)
2.716924
```

Exercise 3.77: 上で使った `integral` 手続きは、Section 3.5.2 に出てきた、整数の無限ストリームの“暗黙的”定義に似ている。別のやり方として、`integral` の定義を、(同じく Section 3.5.2 に出てきた)`integers-starting-from` に似たものにすることもできる。

```
(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))))
```

ループを含むシステムについて使うと、この手続きは `integral` の元のバージョンと同じ問題を持つことになる。手続きを修正し、遅延された引数を `integrand` として受け入れるようにして、上に示した `solve` 手続きを使えるようにせよ。

Exercise 3.78: 同次 2 階線形微分方程式について調べるために信号処理システムを設計するという問題について考えよ。

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

y をモデル化した出力ストリームは、ループを含む回路によって生成される。これは、 d^2y/dt^2 の値が y と dy/dt の値に依存し、またこれらの値は両方とも d^2y/dt^2 を積分することによって決まるから

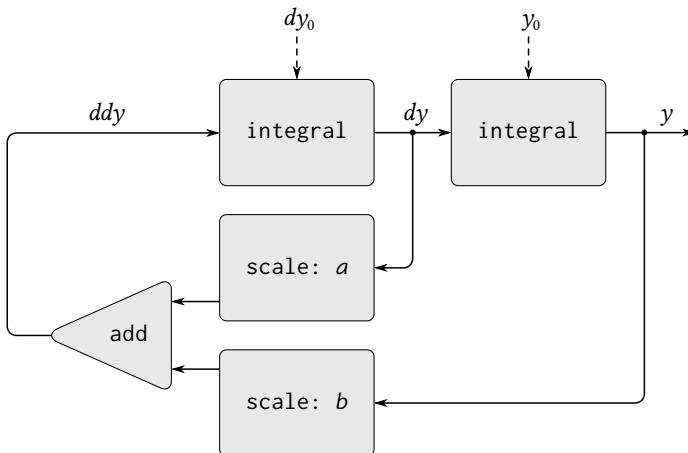


Figure 3.35: 二次線形微分方程式を解くための信号フロー図

である。コード化したいのは、図Figure 3.35に示すようなものだ。引数として、定数 a, b, dt と、 y の初期値 y_0 と dy/dt の初期値 dy_0 を取り、 y の一連の値のストリームを生成する手続き `solve-2nd` を書け。

Exercise 3.79: Exercise 3.78の `solve-2nd` 手続きを一般化し、一般的な二次微分方程式 $d^2y/dt^2 = f(dy/dt, y)$ を解くのに使えるようにせよ。

Exercise 3.80: *series RLC circuit*(直列 RLC 回路)とは、抵抗・コンデンサ・コイルが Figure 3.36 のように直列に接続されているものをいう。ここで、 R, L, C がそれぞれ抵抗・コイル・コンデンサであるとすると、これら三つのコンポーネントの電圧 (v) と電流 (i) の関係は、以下の方程式に示すようなものである。説明される。

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

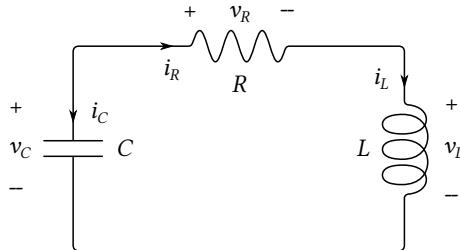


Figure 3.36: 直列 RLC 回路

また、回路の接続によって、次の関係が成立する。

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

これらの式を組み合わせると、回路の状態 (コンデンサの電圧 v_C とコイルの電流 i_L によって表される) は、以下の二つの微分方程式によって記述される。

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

この微分方程式のシステムを表す信号フロー図を Figure 3.37 に示す。

引数として、回路のパラメタ R, L, C と、時間の増分 dt を取る手続き RLC を書け。Exercise 3.73 の RC 手続きと同じように、RLC は手続きを返す。返り値の手続きは、状態変数の初期値 v_{C_0} と i_{L_0} を取り、 v_C と i_L の状態ストリームのペアを (cons を使って) 生成する。RLC を使って、 $R = 1[\Omega]$, $C = 0.2[F]$, $L = 1$ henry, $dt = 0.1[s]$, また初期値を $i_{L_0} = 0[A]$, $v_{C_0} = 10[V]$ としたときの連結 RLC 回路のふるまいをモデル化するストリームのペアを生成せよ。

正規順序評価

この節で示した例は、`delay` と `force` を明示的に使うことによってプログラミングを柔軟にできるということを示していますが、一部の例からは、この

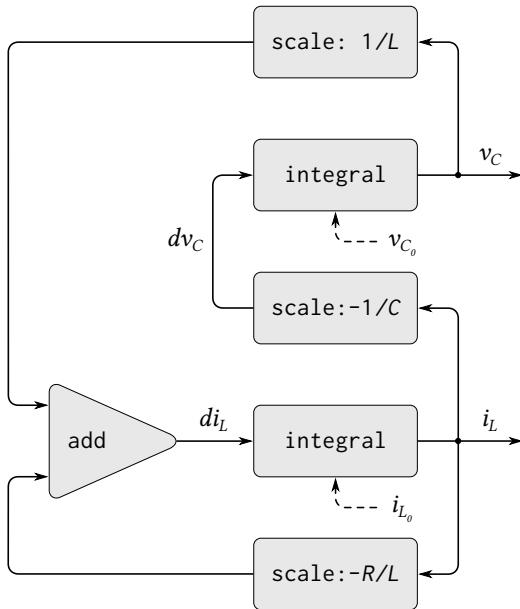


Figure 3.37: 直列 RLC 回路を解くための信号フロー図

ことによってプログラムがより複雑になってしまうということもわかります。例えば、新しい `integral` 手続きを使うことによってループを含むシステムをモデル化する力が得られるのですが、こうすると `integral` が遅延化された被積分関数を引数として呼び出す必要があるということを覚えておかないといけないことになり、`integral` を使う手続きはすべてこのことを認識していなければなりません。ここでは事実上、通常の手続きと、遅延化された引数を取る手続きという二つのクラスの手続きを作ったことになります。一般的に、別々のクラスの手続きを作ると、別々のクラスの高階手続きを同じように作らなければいけないことになってしまいます。⁷²

⁷²Pascal のような従来型の強い型付け言語で高階手続きを扱おうとすると面倒なことになるのですが、この件は、Lisp にそれが少し反映されたものです。強い型付け言語では、引数や返り値の型を手続きごとにプログラマが指定しなければいけません。数値、論

二つの別々のクラスの手続きを作るはめになることを避けようと思うと、ひとつの方法として、すべての手続きが遅延化された引数を取るようにするというものがあります。手続きの引数がすべて自動的に遅延化され、実際に必要になったとき(例えば、基本演算で必要になったとき)になって初めて強制されるような評価モデルを採用することも考えられます。こうすると、私たちの言語は正規順序評価を使うものになります。正規順序評価については、Section 1.1.5で評価の置換モデルを導入した際に説明しました。正規順序評価に変えることになると、遅延評価を統一的でエレガントなやり方で扱うことができるようになります。もしストリーム処理だけについて考えるのであれば、この戦略を採用するのが自然でしょう。Section 4.2では、評価器について学んだ後に、私たちの言語をこの方式に転換するやり方を見ていきます。残念ながら、手続き呼び出しに遅延を採用すると、代入・可変データ・入出力といった、イベントの順番に依存するプログラムを設計する能力がめちゃくちゃになってしまいます。`cons-stream` で `delay` をひとつ使うだけでも、Exercise 3.51やExercise 3.52で例示したように、ひどい混乱が起こることがあります。知られている限りでは、可変性と遅延評価はプログラミング言語の中でうまく同居させることができません。これら二つを同時に扱う方法を工夫することは活発な研究領域となっています。

3.5.5 関数型プログラムのモジュール性とオブジェクトのモジュール性

Section 3.1.2で見たように、代入を導入することの主な利点のひとつは、大きなシステムの状態の一部をカプセル化する、つまり“隠す”ことによって、システムのモジュール性を高められるというところにあります。ストリームモデルは、代入を使わずに同等なモジュール性を提供します。ひとつの実例として、

理値、配列などです。その結果として、“与えられた手続き `proc` を列のすべての要素にマップする”といった抽象化を、`stream-map` のようなひとつの高階手続きによって表現することができません。その代わりに、`proc` が取るかもしれない引数と返り値のデータ型の組み合わせすべてに対して、別々のマップ手続きが必要になってしまいます。高階手続きの存在を考えたとき、実用的な概念として“データ型”というものを維持しようとすると、いろいろと難しい問題が起ります。言語 ML(Gordon et al. 1979) は、この問題に対処する方法のひとつを示しています。この言語の“多相データ型”は、データ型同士の高階変換のためのテンプレートを含んでいます。そのうえ、ML のほとんどの手続きで、データ型がプログラマによって明示的に宣言されることはありません。その代わりに、ML には *type-inferencing*(型推論) メカニズムがあり、環境の情報を使って新しく定義された手続きのデータ型を推論します。

Section 3.1.2で詳しく調べたモンテカルロ法による π の推定を、ストリーム処理という視点から再実装してみます。

モジュール性という問題の鍵となっていたのは、乱数を使うプログラムから、乱数生成器の内部状態を隠したいということでした。そのときは、まず `rand-update` という手続きを作りました。この連続する値が乱数を供給することになり、これを使って乱数生成器を生成するということをしていました。

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

ストリームによって定式化する場合は、乱数生成器そのものは存在せず、`rand-update` を連続して呼ぶことによって生成される乱数ストリームがあるだけです。

```
(define random-numbers
  (cons-stream random-init
    (stream-map rand-update random-numbers)))
```

これによって、`random-numbers` ストリームの連続したペアに対して Cesàro(チエザロ) 実験を行った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` は、次に `monte-carlo` 手続きに供給されます。この手続きは、確率の推定値のストリームを生成します。この結果が、今度は π の推定値のストリームに変換されます。このプログラムのこのバージョンでは、試行を何回行うかという引数は必要ありません。(より多くの試行を行うことによる) よりよい π の推定値は、`pi` ストリームを後ろのほうまで取得していくことによって得られます。

```

(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)))

```

このアプローチにはかなりのモジュール性があります。このアプローチでも、任意の実験を扱える一般的な `monte-carlo` 手続きが定式化できているからです。それなのに、代入やローカル変数といったものは使われていません。

Exercise 3.81: Exercise 3.6では、乱数生成器を一般化して、乱数列をリセットできるようにすることで、再現可能な“ランダム”な数の列を生成することについて検討した。同じ生成器をストリームによって定式化せよ。この生成器は要求の入力ストリームに基づいて動作し、望み通りの乱数ストリームを生成する。要求は、新しい乱数を生成する `generate`、または列を指定された値にリセットする `reset` である。解答には代入を使わないこと。

Exercise 3.82: Exercise 3.5のモンテカルロ積分をストリームによって再度行え。ストリーム版 `estimate-integral` は試行回数の引数を持たない。その代わりに、連続した試行に基づく推定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

ここで、この章の初めて提起したオブジェクトと状態の問題に戻って、これらの問題を新しい観点から検討してみましょう。この章の初めでは代入と可変オブジェクトを導入し、状態を持つシステムをモデル化するプログラムをモジュール式に構築するメカニズムを提供しました。また、局所変数を持つ計算オブジェクトを構築し、これらの変数を代入によって変更しました。そして、

この世界に存在するオブジェクトの経時的なふるまいを、対応する計算オブジェクトの経時的なふるまいによってモデル化しました。

その後、ストリームを使うことによっても、局所状態を持つオブジェクトをモデル化することができるということを見てきました。オブジェクトの局所状態のような変化していく量は、逐次的な状態の時間履歴を表現するストリームによってモデルできます。本質的に、ストリームによって時間を明示的に表現することによって、シミュレートされた世界の中での時間と、評価の間に起こるイベントの順番とを切り離すということをしていることになります。実際に、`delay`というものが存在するので、モデル中のシミュレートされた時間と、評価の間に起こるイベントの順番との間に、ほとんど関係がないということもあります。

これら二つのモデル化アプローチの違いを見るために、銀行口座の残高を管理する“引き出し処理器”の実装について再度考えてみましょう。Section 3.1.3では、引き出し処理器の簡単なバージョンを実装しました。

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

`make-simplified-withdraw` を呼び出すと計算オブジェクトが生成されます。各オブジェクトはそれぞれ局所状態変数 `balance` を持っています。`balance` は、オブジェクトに対する呼び出しを行うことで減少していきます。オブジェクトは `amount`(金額) を引数として取り、新しい残高を返します。銀行口座のユーザが、そのようなオブジェクトに入力列をタイプし、ディスプレイの画面に返り値の列が表示されるのを見ているところを想像するとよいでしょう。

引き出し処理器をモデル化する別のやり方として、入力として残高と引き出す金額のストリームを取り、一連の口座残高のストリームを生成する手続きを使うというものがあります。

```
(define (stream-withdraw balance amount-stream)
  (cons-stream
    balance
    (stream-withdraw (- balance (stream-car amount-stream))
      (stream-cdr amount-stream))))
```

`stream-withdraw` は、出力が入力によって完全に決定される、明確に定義された数学関数を実装しています。しかし、ここで入力 `amount-stream` がユーザ

のタイプする一連の値のストリームで、結果の残高ストリームは画面に表示されていると考えてみてください。そうすると、値をタイプし結果を見ているユーザの視点からは、このストリーム処理は `make-simplified-withdraw` によって作られたオブジェクトと同じふるまいを持っていることになります。しかしながらストリーム版では、代入も局所状態もなく、このためSection 3.1.3で直面したような理論的な困難もありません。それなのに、システムには状態があるのです！

これは実に注目すべきところです。`stream-withdraw` は、ふるまいの変化しない、明確に定義された数学関数を実装しているのに、ここでユーザの知覚は、変化する状態を持つシステムと対話しているときと同じなのです。このパラドックスを解くひとつの方法は、状態を持つことをシステムに求めているのは、ユーザの時間的な存在だということを認識するということです。もしユーザがシステムとの対話から一步下がって、個々の取引ベースで考えるのをやめて残高のストリームという考え方をするなら、システムは状態を持たないよう見えるでしょう。⁷³

複雑な処理のある一部から見ると、ほかの部分は時間とともに変化するように見えます。それらは時間によって変わる局所状態を持っています。私たちの世界に存在するこの種の自然な分解方法（これは、私たちが世界の一部としての視点から世界を見るやり方と同じです）をモデル化したプログラムをコンピュータ中の構造を使って書こうと思うと、関数的でない計算オブジェクトを作ることになります。オブジェクトは変化しなければならないからです。状態は局所状態変数によってモデル化し、状態の変更はこれらの変数に対する代入によってモデル化します。こうすることによって、計算実行の時間が私たちのいるこの世界の時間をモデル化するようにします。その結果、コンピュータの中に“オブジェクト”ができることがあります。

オブジェクトを使ったモデル化は強力で直感的です。その大きな理由となっているのは、このやり方が私たちがこの世界と関わる際の認知にマッチしているということです。しかし、この章を通して何度も見てきたように、これらのモデルは、イベント順の制約や、複数プロセスの同期といった、厄介な問題を引き起こします。*functional programming languages*（関数型プログラミング言語）の発展は、これらの問題を避けられるという見込みによって促進されました。関数型プログラミング言語は代入や変更可能なデータを提供しません。このような言語では、すべての手続きは引数に対する明確な数学関数を実

⁷³物理でも同じように、動く粒子を観測しているとき、粒子の位置（状態）は変化しているというように言います。しかし、時空の中での粒子の世界線という視点からは、変化というものは存在しません。

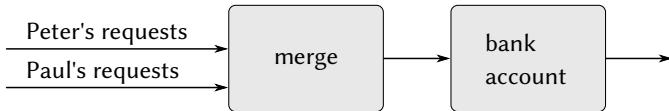


Figure 3.38: 取引要求リクエストのストリーム二つをマージすることでモデル化した共同銀行口座

装することになり、挙動は変化しません。並行システムを扱うには、関数型のアプローチは非常に魅力的です。⁷⁴

その一方、詳しく見ると、時間に関する問題が関数型のモデルにも紛れ込んでいることがあります。問題が多いのは、インタラクティブなシステムを設計しようとする場合です。中でも、独立した実体同士のやりとりをモデル化するようなシステムで特に問題となります。例えば、共同口座を持つてのような銀行システムの実装について、もう一度考えてみましょう。代入とオブジェクトを使う従来型のシステムでは、Peter と Paul が口座を共有しているという事実を、Section 3.1.3で見たように、Peter と Paul が取引要求を同じ銀行口座オブジェクトに送るようにすることによってモデル化します。ストリームという視点からは、“オブジェクト”そのものではなく、銀行口座のモデル化は、取引要求のストリームに対して演算を行い応答のストリームを生成する処理という形になるということは、すでに述べました。ということは、Peter と Paul が共同口座を持っているという事実は、Peter の取引要求ストリームと Paul の取引要求ストリームをマージし、その結果を銀行口座ストリーム処理に送るという形でモデル化できるのではないかでしょうか。この様子を Figure 3.38 に示します。

この定式化で問題になるのは、*merge*(マージ) という概念です。二つのストリームをマージするのに、Peter の要求をひとつ、Paul の要求をひとつ、といった具合に、単純に交互に取っていくというやり方ではうまくいきません。Paul が口座にめったにアクセスしないとしましょう。二回目の取引をするのは Paul が口座にアクセスするまで待てと Peter に言うというのは無理な話です。マージをどのように実装するにせよ、実装する際には、何らかの形で Peter と Paul が知覚する “実時間” によって制約されるように、二つの取引ストリーム

⁷⁴Fortran の開発者 John Backus が 1978 年に ACM チューリング賞を受賞したとき、彼は関数型プログラミングを一躍有名にしました。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するものでした。関数型プログラミングについてのよい概説としては、Henderson 1980 と Darlington et al. 1982 があります。

を織り交ぜなければなりません。これはつまり、Peter と Paul が会ったとすると、ある一部の取引は会う前に処理されていて、残りの取引は会った後に処理されるということについて、合意できるようにするということです。⁷⁵ これはまさに、Section 3.4.1で扱うことになった制約と同じものです。そのときは、状態を持つオブジェクトを並行処理するにあたって、イベントの“正しい”順番を保証するために、明示的な同期を導入する必要があることがわかりました。つまり、関数型スタイルをサポートしようとしていたところで、複数の主体からの入力をマージする必要が出てくると、関数型スタイルによって排除しようとしていた問題と同じものがまた現れるということになります。

この章は、モデル化対象である実世界の認知にマッチした構造を持つ計算モデルを構築することをゴールとするところから始めました。世界をモデル化するのには、分離した、時間制約のある、相互作用する、状態を持つオブジェクトの集まりとして扱うというやり方があり、单一の、時間のない、状態のないまとまりというやり方もあります。どちらの見方にも強力な利点がありますが、どちらの見方も単独では完全に十分ではありません。大統一は未だ現れていません。⁷⁶

⁷⁵ どんな二つのストリームについても、それらを織り交ぜる適切な順番は、一般的に複数あるということに注意してください。このため、厳密に言うと、“マージ”は関数ではなく関係ということになります。答えは入力に対する決定的な関数ではないということです。並行性を扱ううえで、非決定性が本質となっているということはすでに述べました(Footnote 39)。マージという関係は、この同じ本質的な非決定性について、関数型という視点から示したものです。Section 4.3では、非決定性をまた別の視点から見ていくことになります。

⁷⁶ オブジェクトモデルは、世界を独立した部分に分けることによって近似します。関数型モデルは、オブジェクト境界によってモジュール化することはしません。オブジェクトモデルは、“オブジェクト”的な共有されていない状態が、共有されている状態よりもずっと大きい場合に有用です。オブジェクトという視点がうまくいかない場合のひとつの一例として量子力学があります。量子力学では、物を個別の粒子として考えると、パラドックスや混乱が生まれます。オブジェクトという視点と関数型の視点を統一することは、プログラミングとはあまり関係がなく、根本的な認識論の問題と関係があるのかもしれません。

4

メタ言語抽象化

... 魔法は言葉に宿る —アブラカダブラ、開けゴマ、などなど —しかし、あるお話の魔法の言葉は、次のお話では魔法を持たない。本当の魔法は、どの言葉が、いつ、何に対して効くかということを理解するというところにある。そのコツは、コツを学ぶことにあるんだ。

... そして、それらの言葉は、普通のアルファベットの文字でできている。20かそこらの、ペンで描けるぐにやぐにやした線。これこそが鍵なんだ！それに手が届きさえすれば、宝物にもなる。まるで —まるで宝物の鍵そのものが宝物みたいなんだ！

—John Barth, *Chimera*

プログラム設計について学んだ際に、熟練プログラマが設計の複雑さをコントロールするやり方は、ほかの複雑なシステムを設計する人たちが使うのと同じ一般的なテクニックだということを見てきました。基本要素を組み合わせて複合オブジェクトを作り、複合オブジェクトを抽象化して高レベルの構築用プロックを作り、適切なマクロ的な見方でシステム構造を見渡すことによってモジュール性を維持するのです。これらのテクニックを説明するにあたっては、Lisp を使ってきました。Lisp は、プロセスを記述し、また計算データオブジェクトやプロセスを構築して実世界の複雑な現象をモデル化するための言語として役割を果たしてきました。しかし、取り組む問題が複雑になっていくにつれ、

Lisp が、というよりもどんな固定したプログラミング言語でも、必要なことをやり遂げるのに十分ではないと感じるようになります。考えをより効率的に表現しようとすると、絶えず新しい言語に目を向けることになります。新しい言語を作るというのは、工学的設計の複雑さをコントロールする強力な戦略です。複雑な問題に取り組む際に、問題を別のやり方で記述する（また、それによって思考する）ことのできる新しい言語を採用することで、私たち自身の能力を向上させるということはよくあります。手元の問題に特にうまく合った基本要素、組み合わせ手段、抽象化手段を持つ言語を使うということです。¹

プログラミングは多くの言語に恵まれています。言語には、特定のコンピュータの機械語といった物理的言語もあります。物理的言語が扱うのは、ストレージの個々のビットや基本機械語命令というレベルでのデータや制御の表現です。機械語のプログラマは、与えられたハードウェアを使って、資源に制約のある計算を効率的に実装するシステムやユーティリティを構築することに焦点を当てています。高級言語は、機械語という基層の上に構築され、ビットの集合としてのデータの表現や基本命令の列としてのプログラムの表現といったところを気にしないでいいように隠してくれます。これらの言語には、手続き定義のような組み合わせと抽象化の手段があります。これらの手段は大規模なシステムの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——これは、工学設計のすべての分野で重要な役割を果たしています。コンピュータプログラミングでは特に重要です。プログラミングでは、新しい言語を策定することができるうえに、それらの言語を評価器を構築することができます。

¹ この考え方は、工学全体に広く行き渡っているものです。例えば、電気工学では、回路の記述にさまざまな言語を使います。その中の二つとして、電気的 *network*(ネットワーク) の言語と、電気的 *system*(システム) の言語があります。ネットワーク言語は、離散的電子素子によって物理的に装置をモデル化することに焦点を当てています。ネットワーク言語の基本オブジェクトは、抵抗・コンデンサ・コイル・トランジスタなどの基本的な電気部品で、それらは電流と電圧という物理的変数を特性として持ります。ネットワーク言語で回路を記述する際には、エンジニアが注目しているのは設計の物理的特性です。それに対して、システム言語の基本オブジェクトは、フィルタやアンプといった信号処理モジュールです。ここで関わってくるのはこれらのモジュールの機能的ふるまいのみで、信号を操作するにあたってはモジュールの電圧や電流といった物理的実体は気にしません。信号処理システムの要素が電気的ネットワークによって構築されているという意味で、システム言語はネットワーク言語の上に成り立っているものです。しかし、ここでやりたいことは、与えられた応用問題を解くために電子装置を大規模に構成することです。部品の物理的実現可能性は前提となっています。このような階層化された言語の集合は、Section 2.2.4の图形言語で示したような階層化設計テクニックのまた別の例となっています。

実装することもできるからです。あるプログラミング言語の *evaluator*(評価器)(*interpreter*(インタプリタ)とも)とは、その言語の式に適用されたとき、その式を評価するのに必要なアクションを実行するような手続きを指します。

次のことは、プログラミングについての最も根本的な考え方といつても過言ではありません。

プログラミング言語の式の意味を決める評価器もまた、ただのプログラムにすぎない。

この点を理解すると、プログラマとしての私たち自身のイメージが変わります。他人の設計した言語のただのユーザではなく、言語の設計者として自分自身を見るようになります。

実際、ほとんどのプログラムは何らかの言語の評価器であると考えることができます。例えば、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規則を具体化し、リスト構造のデータに対する演算という形で実装したものです。このシステムを拡張して、多項式を読み込み、表示する手続きを追加したら、記号数学の問題を扱う特定目的言語のコアができたことになります。Section 3.3.4のデジタル論理シミュレータや Section 3.3.5の制約伝播は、それら自身がちゃんとした言語になっていて、それぞれ独自の基本要素、組み合わせ手段、抽象化的手段を持っています。この視点から見ると、大規模コンピュータシステムに取り組む技術と新しいコンピュータ言語を構築する技術が合流し、計算機科学そのものが、適切な記述言語を構築するという分野にほかならないものとなります。

これから私たちは、言語をほかの言語によって構築する技術への旅に出発します。この章では、Lisp を基盤として使い、いろいろな評価器を Lisp の手続きとして実装していきます。Lisp は、記号式を表現し、操作する能力のために、特にこのタスクに向いています。まずは、Lisp 自身の評価器を構築することによって、言語がどのように実装されるかを理解するための最初の一歩を踏み出します。ここでの評価器によって実装される言語は、この本で使っている Lisp の Scheme 方言のサブセットになります。この章で記述する評価器は、Lisp の特定の方言に対してのものですが、逐次式コンピュータのプログラムを書くために設計された式指向言語の評価器が持つ本質的な構造を持っています(実際のところ、ほとんどの言語処理系は、奥深いところで、小さな“Lisp”評価器を持っているものです)。この評価器は説明と議論のために簡単にしたもので、製品レベルの Lisp システムにとっては重要な一部の機能が省略されています。それでも、この単純な評価器は、この本のプログラムのほとんどを実行するの

に十分です。²

評価器を Lisp プログラムとしてアクセスできるようにすることの重要な利点のひとつは、別の評価規則を実装することができるようになるということがあります。これは、評価器プログラムに対する修正としてそれらの規則を記述することによって可能になります。この力を活用できる場所のひとつは、計算モデルが時間という概念を具体化する方法について、より制御しやすくするという場面です。時間という概念は、Chapter 3の議論で非常に重要なものとなっていました。そのときは、ストリームを使って実世界の時間の表現とコンピュータの時間とのつながりを断ち切ることによって、状態と代入の複雑さをいくらか緩和することができました。しかし、そこでのストリームプログラムは、Scheme の適用順序評価という制約があったため、面倒なことになることもありました。Section 4.2では、評価器に手を加えて *normal-order evaluation* (正規順序評価) に対応させることで、よりエレガントなアプローチができるように基層言語を変えてしまします。

Section 4.3では、もっと大胆に言語を変更します。ここでは、式がひとつの値だけではなく、複数の値を持つようにします。この *nondeterministic computing*(非決定性計算) 言語では、式のすべての可能な値を生成し、その後で何らかの制約を満たす値を検索するというプロセスを自然に表現できます。計算と時間のモデルということで言うと、これは時間を“可能な複数の未来”に分岐させ、それから適切な時系列を検索するようなものです。この非決定性評価器では、複数の値を管理して検索を行うということが、基層となる言語の仕組みによって、自動的に行われることになります。

Section 4.4では、*logic-programming*(論理プログラミング) 言語を実装します。論理プログラミング言語では、知識を入出力のある計算としてではなく、関係によって表します。こうすると、この言語は Lisp とは (Lisp だけでなく、既存のどんな言語とも) 大幅に違うものになります。そうでありながら、この論理プログラミング言語は本質的には Lisp 評価器と同じ構造を持っているということを、ここで見ていきます。

² この評価器に含めなかった機能のうちで最も重要なものは、エラー処理とデバッグ支援の仕組みです。評価器についてのより詳細な議論については、Friedman et al. 1992 を参照してください。これは、Scheme で書かれた一連の評価器によって進行する複数のプログラミング言語について解説しています。

4.1 メタ循環評価器

ここで作る Lisp 評価器は、Lisp プログラムとして実装していきます。Lisp プログラムを評価するのに、それ自身 Lisp で実装されている評価器を使うというのは、ちょっと考えると循環論のように思えるかもしれません。しかし、評価というのはプロセスです。そのため、評価プロセスを Lisp によって記述するというのは妥当なことです。Lisp というのは、結局のところ、プロセスを記述するための道具だからです。³評価する言語と同じ言語で書かれた評価器は、*metacircular* (メタ循環) であると言われます。

メタ循環評価器は、本質的にはSection 3.2で記述した評価の環境モデルを Scheme によって形式化したものです。モデルには、以下の二つの基本部品があつたことを思い出しましょう。

1. 組み合わせ (特殊形式以外の複合式) を評価するには、部分式をすべて評価し、それから演算子となる部分式の値を被演算子となる部分式の値に適用する。
2. 複合式を引数の集合に適用するには、手続きの本体を新しい環境で評価する。この環境を構築するには、手続きオブジェクトの環境部分にフレームをひとつ加えて拡張し、その中で手続きの仮引数を手続き適用対象の引数に束縛する。

これら二つの規則は、評価プロセスの本質を記述したものです。環境の中で評価する式は引数に適用する手続きに簡約され、それが今度は新しい環境の中で評価する新しい式に簡約され…を繰り返し、環境の中で値が見つかる記号や直接適用する基本手続きにたどり着くまで続けるというのが、評価プロセスの基本サイクルです (Figure 4.1 参照)。⁴

³これは確かにそうなのですが、この評価器では明らかにできない評価プロセスの重要な側面がいくつか残ります。の中でも最も重要なのが、手続きがほかの手続きを呼び出し、呼び出し元に返り値を返すメカニズムの詳細です。これらの問題については、Chapter 5で見ていくことにします。そこでは、評価器を単純なレジスタマシンとして実装することによって、評価プロセスをより詳しく調べることになります。

⁴基本手続きを適用してもいいということになると、評価器の中で実装すべきものは何が残るのでしょうか。この評価器の仕事は、言語の基本要素を実装することではなく、基本要素の集合を結合することによって言語を形成するための結合組織—組み合わせ手段と抽象化手段—を提供するというところにあります。具体的には、

- この評価器は入れ子の式を扱うことができます。例えば、単純に基本要素を適用するだけでは、式 $(+ 1 6)$ の評価には十分ですが、 $(+ 1 (* 2 3))$ を扱うには不十分です。

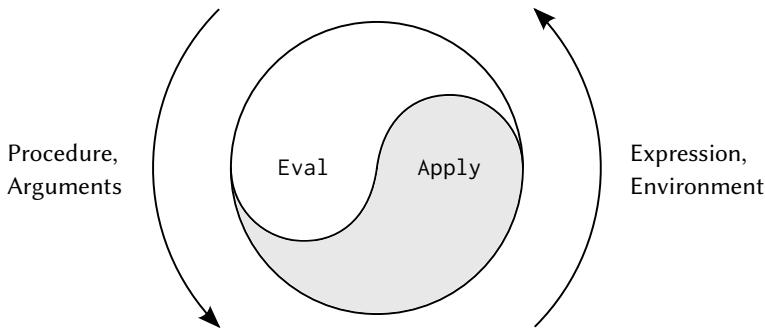


Figure 4.1: eval-apply サイクルが明らかにするコンピュータ言語の本質

評価器の実装は、評価する式の *syntax*(構文) を定義する手続きに依存することになります。ここでは、データ抽象化を使って、評価器を言語の表現に依存しないようにします。例えば、代入を表すのに `set!` 記号から始まるリストを使うというように決め打ちするのではなく、抽象述語 `assignment?` を使って代入をテストし、代入の部品にアクセスするには抽象セレクタ `assignment-variable` と `assignment-value` を使うことにします。式の実装は [Section 4.1.2](#)で詳しく説明します。また、手続きと環境の表現を規定する演算もあり、これらについては [Section 4.1.3](#)で記述します。例えば、`make-procedure` は複合手続きを構築し、`lookup-variable-value` は変数の値にアクセスし、`apply-primitive-procedure` は基本手続きを与えられた引数のリストに対して適用します。

基本手続き + に関するいうと、引数は数値である必要があり、引数として式 `(* 2 3)` を渡すと処理できないでしょう。評価器の重要な役割のひとつは、手続き合成をうまく指揮して、+ の引数として渡される前に `(* 2 3)` を 6 に簡約するということです。

- この評価器では変数が使えます。例えば、加算の基本手続きは `(+ x 1)` のような式を扱うすべがありません。評価器に変数を管理させて、基本手続きを呼び出す前に変数の値を得るようにする必要があります。
- この評価器では複合手続きが定義できます。これは、手続き定義を管理し、それらの定義を式評価の中で使うことができるようにして、手続きが引数を受け入れられるようにするメカニズムを提供するということです。
- この評価器は特殊形式を提供します。特殊形式は、手続き呼び出しとは別の評価の仕方をする必要があります。

4.1.1 評価器のコア

評価プロセスは、`eval` と `apply` という二つの手続きの相互作用として記述できます。

`eval`

`eval` は、引数として式と環境を取り、式を分類して評価を振り分けます。`eval` は、評価対象の式が持つ構文上の型による場合分けという構造を取ります。手続きに一般性を持たせるため、式の型決定を抽象的に表現し、式の持つさまざまな型の特定の表現に決め打ちしないようにします。式の型はそれぞれ、その型かどうかをテストする述語と、部品を選択する抽象手段を持っています。この*abstract syntax*(抽象構文) を使うことで、同じ評価器を使いつつ別の構文手続きの集合を使うことによって言語の構文を変更できるということがわかりやすくなります。

基本式

- 数値のような自己評価式については、`eval` は式そのものを返します。
- `eval` は、環境内で変数を検索して、変数の値を調べる必要があります。

特殊形式

- クオート式については、`eval` はクオートされた式を返します。
- 変数への代入 (あるいは変数の定義) は、再帰的に `eval` を呼び出し、変数に関連づける新しい値を計算する必要があります。さらに、環境を修正して、変数束縛を変更 (または作成) する必要があります。
- `if` 式の部品は特殊な処理をする必要があります。述語が真であれば結果式を評価し、偽であれば代替式を評価するようにするためにです。
- ラムダ式は、ラムダ式の規定する引数と本体、それと評価環境をまとめてパッケージ化し、適用可能な手続きに変形する必要があります。
- `begin` 式は、一連の式を出現順に評価する必要があります。
- 場合分け (`cond`) は、入れ子の `if` 式に変形し、それから評価します。

組み合わせ

- 手続きを適用するためには、`eval` は組み合わせ中の演算子部分とオペランド部分を再帰的に評価する必要があります。その結果得られる手続きと引数は `apply` に渡し、`apply` が実際の手続き適用を扱います。

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))))
```

簡単にするために、eval は cond を使った場合分けとして実装しています。これの欠点は、この手続きが扱えるのは限られた識別可能な式の型のみで、新しいものを定義しようと思うと eval の定義を編集しないといけないということです。ほとんどの Lisp 実装では、式の型によるディスパッチはデータ主導スタイルで行われています。そのことによって、eval が識別できる新たな式の型を、eval そのものの定義を修正することなしにユーザが追加できるようになっています (Exercise 4.3 参照)。

apply

apply は二つの引数を取ります。手続きと、その手続きを適用する引数のリストです。apply は手続きを二種類に分類します。基本演算の場合は、apply-primitive-procedure を呼んで適用します。複合手続きの場合は、手続きの本体を構成する式を順番に評価していくことによって適用を行います。複合手続きの本体を評価する環境は、手続きの持つ基本環境を拡張して、手続きの仮引

数を手続き適用対象の引数に束縛するフレームを追加することによって構築します。`apply` の定義は以下のようになります。

```
(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))))
```

手続きの引数

`eval` が手続き適用を処理する際には、`list-of-values` を使って、手続き適用対象の引数のリストを作ります。`list-of-values` は、組み合わせのオペランドを引数に取り、各オペランドを評価し、対応する値のリストを返します。

⁵

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

⁵`eval` の `application?` 節は、明示的に `list-of-values` 手続きを書くよりも、`map` を使えば（また、`operands` がリストを返すよう規定すれば）もっと簡単に書けたところです。ここで `map` を使わないことにしたのは、評価器がサポートする言語が高階手続きを持っているとしても、評価器は高階手続きを使わずに（つまり、高階手続きを持たない言語でも）実装できるということを強調したかったからです。

条件文

`eval-if` は、与えられた環境の中で `if` 式の述語部分を評価します。もし結果が真であれば、`eval-if` は結果式 (consequent) を評価し、そうでなければ代替式 (alternative) を評価します。

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

`eval-if` 内で `true?` を使うことは、被実装言語と実装言語の接続という問題を浮き彫りにします。`if-predicate` は被実装言語によって評価されるため、これはその言語での値を返すことになります。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。真であることのメタ循環表現は、基層となる Scheme の表現と異なるかもしれないからです。⁶

列

`eval-sequence` は、`apply` が手続きの本体中の式の列を評価するときや、`eval` が `begin` 式中の式の列を評価するときに使います。引数として式の列と環境を取り、出現順に式を評価していきます。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

以下の手続きは変数への代入を扱います。`eval` を呼ぶことによって代入する値を求め、変数とその値を `set-variable-value!` に渡すことによって指定された環境に設定します。

⁶ 今回は、実装言語と被実装言語は同じです。ここでの `true?` の意味をよく考えると、薬物に頼ることなく、意識を拡張することができます。

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

変数の定義も似たようなやり方で扱います。⁷

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

ここでは、代入や定義の値として記号 `ok` を返すことになっています。⁸

Exercise 4.1: メタ循環評価器がオペランドを評価する順番は、左から右なのか、右から左なのかわからないということに注意せよ。評価順は、基層となる Lisp のものを引き継ぐ。つまり、`list-of-values` にある `cons` の引数が左から右という順で評価されるのであれば、`list-of-values` はオペランドを左から右に評価することになる。`cons` の引数が右から左という順で評価されるのであれば、`list-of-values` はオペランドを右から左に評価することになる。基層となる Lisp の評価順によらず、オペランドを左から右に評価するようにしたバージョンの `list-of-values` を書け。また、オペランドを右から左に評価するバージョンの `list-of-values` も書け。

4.1.2 式の表現

この評価器は、Section 2.3.2で検討した記号微分プログラムと共に通すところがあります。どちらのプログラムも記号式の演算を行います。どちらのプログラムでも、複合式の演算結果は、式の各部品に対して演算を行い、その結

⁷ この `define` の実装は、多くの場合正しく動くのですが、内部定義の扱いに関する難しい問題を無視しています。問題の性質や解決方法についてはSection 4.1.6で見ていきます。

⁸ `define` と `set!` を紹介したときに述べたように、これらの値は Scheme の実装依存です—つまり、返す値は実装者が決めることができます。

果を式の型に従って組み合わせることによって得られます。どちらのプログラムでも、データ抽象化を使って、演算の一般的な規則と式の表現方法の詳細を切り離しています。微分プログラムでは、このことによって、同じ微分手続きを使って前置・中置・後置その他の形式の数式を扱えるようになっていました。評価器の場合は、評価対象の言語の文法が、純粹に式の部品を分類し抽出する手続きのみによって決まるという結果になります。

以下は、この言語の構文の仕様です。

- 自己評価式は数値と文字列しかありません。

```
(define (self-evaluating? exp)
  (cond ((number? exp) true)
        ((string? exp) true)
        (else false)))
```

- 変数は記号によって表現されます。

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

- クオート式は (quote <text-of-quotation>) という形式です。⁹

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

quoted? は、リストが指定された記号から始まるかどうかを確認する手続き tagged-list? を使って定義します。

```
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
```

- 代入は (set! <var> <value>) という形式です。

```
(define (assignment? exp) (tagged-list? exp 'set!))
(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))
```

⁹Section 2.3.1で触れたように、評価器はクオート式を quote から始まるリストとして扱います。これは、クオート式に引用符が使われている場合でも同じです。例えば、式'a'は、この評価器によって (quote a) として扱われます。Exercise 2.55参照。

- 定義は以下の形式か、

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

以下の形式です。

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

後者の形式(標準手続き定義)は、以下のシンタックスシュガーです。

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

これに対応する構文手続きは以下の通りです。

```
(define (definition? exp) (tagged-list? exp 'define))
(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)           ; formal parameters
                   (cddr exp))))    ; body
```

- lambda式は、記号lambdaから始まるリストです。

```
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))
```

また、lambda式のコンストラクタも提供します。これは、上記のdefinition-valueで使われています。

```
(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

- 条件式は `if` で始まり、述語・結果式・(オプションで) 代替式を持ちます。式が代替式部分を持たない場合は、代替式として `false` を使います。

¹⁰

```
(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` が `cond` 式を `if` 式に変形するのに使われます。

```
(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))
```

- `begin` は、式の列を単独の式にパッケージ化します。構文演算として、`begin` 式から実際の列を取り出す演算と、列の最初の式と残りの式を返すセレクタを用意します。¹¹

```
(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` で使うために、コンストラクタ `sequence->exp` も用意します。これは列を单一の式に変形するもので、必要であれば `begin` を使います。

¹⁰述語が `false` で代替式が存在しない場合の `if` 式の値は、Scheme では規定されていません。ここでは `false` にすることにしました。変数 `true` と `false` は、グローバル環境で束縛することによって、式の中で使えるようにします。Section 4.1.4 参照。

¹¹式のリストに対するこれらのセレクタ — また、これに対応するオペランドのリストに対するセレクタ — は、データ抽象化のためのものではありません。これらは、Section 5.4 の明示的制御評価器の理解を助けるために導入した、基本的なリスト演算を覚えやすくするための名前です。

```

(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))

```

- 手続き適用は、ここまで挙げた式の型に含まれない任意の複合式です。式の car は演算子で、cdr はオペランドのリストです。

```

(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))

```

派生式

この言語の特殊形式の中には、直接実装するのではなく、ほかの特殊形式を使った式によって定義できるものもあります。そのひとつの例が cond で、入れ子の if 式として実装できます。例えば、次の式を評価するという問題は、

```

(cond ((> x 0) x)
      ((= x 0) (display 'zero) 0)
      (else (- x)))

```

if と begin 式を持つ次の式を評価するという問題に簡約することができます。

```

(if (> x 0)
    x
    (if (= x 0)
        (begin (display 'zero) 0)
        (- x)))

```

cond をこのように実装することで、評価器を単純にできます。明示的に評価プロセスを規定しなけれ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` 式の部品を抽出する構文手続きと、`cond` 式を `if` 式に変形する手続き `cond->if` を用意します。場合分けは、`cond` から始まり、述語-アクション節のリストを持ちます。節は、述語が記号 `else` であれば `else` 節となります。¹²

```
(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))
(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))
(define (cond-predicate clause) (car clause))
(define (cond-actions clause) (cdr clause))
(define (cond->if exp) (expand-clauses (cond-clauses exp)))
(define (expand-clauses clauses)
  (if (null? clauses)
      'false ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last: COND->IF"
                      clauses))
            (make-if (cond-predicate first)
                  (sequence->exp (cond-actions first))
                  (expand-clauses rest)))))))
```

`cond` のような、構文変形として実装するようにした式のことは *derived expressions* (派生式) といいます。`let` 式も同じく派生式です (Exercise 4.6 参照)。¹³

¹²すべての述語が `false` で `else` 節が存在しない場合、Scheme では `cond` 式の値は規定されていません。ここでは `false` としています。

¹³実用的な Lisp システムでは、ユーザが新しい派生式を追加し、評価器を修正することなく構文変形としての実装を規定することができる仕組みを提供しています。そのようなユーザ定義の変形は *macro*(マクロ) と呼ばれます。単純なマクロ定義の仕組みを追加することは簡単なのですが、そうすると言語が名前の衝突という厄介な問題を持つことになります。そのような問題のないマクロ定義の仕組みについては多くの研究がなされています。例えば、Kohlbecker 1986, Clinger and Rees 1991, Hanson 1991を参照してください。

Exercise 4.2: Louis Reasoner は、`eval` の `cond` を並べ替えて、手続き適用の節が代入の節より前に来るようにしてやっている。彼は、こうするとインタプリタの効率がよくなると言う。プログラムは通常、代入や定義などより適用を多く含んでいるのだから、修正版の `eval` は、元の `eval` よりも式の型を識別するまでにチェックする節の数が少なくなると言うのだ。

- a Louis の計画はどこがまずいのだろうか (ヒント : Louis の評価器は、式 (`define x 3`) をどのように処理することになるだろうか)。
- b Louis は計画がうまくいかないのでイライラしている。彼は、何が何でも、ほかの大部分の型に対するチェックよりも先に、評価器に手続き適用を認識させたいと思っている。評価対象の言語の構文を変更し、手続き適用が `call` で始まるようすることで、彼の手伝いをせよ。例えば、(`factorial 3`) と書いていたところで、変更後は (`call factorial 3`) と書かなければならなくなる。`(+ 1 2)` は、`(call + 1 2)` と書かなければならない。

Exercise 4.3: ディスパッチをデータ主導スタイルで行うように `eval` を書き直せ。これを [Exercise 2.73](#) のデータ主導型微分手続きと比較せよ (式の型としては、複合式の `car` を使ってよい。この節で実装した構文ではそれが適切である)。

Exercise 4.4: [Chapter 1](#)で出てきた特殊形式である `and` と `or` の定義を思い出そう。

- `and`: 式は左から右へと評価される。式のいずれかが偽と評価されたら偽を返す。残りの式はすべて評価されない。すべての式が真の値として評価されるなら、最後の式の評価値を返す。式がなければ真を返す。
 - `or`: 式は左から右へと評価される。式のいずれかが真の値として評価されたらその値を返す。残りの式はすべて評価されない。すべての式が偽と評価されるか、式が存在しない場合は、偽を返す。
- `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、または式が全く存在しない場合は、`false` が返される。

適切な構文手続きと、評価手続き `eval-and`, `eval-or` を定義し、評価器の新たな特殊形式として `and` と `or` を組み込め。また、別のやり方として、`and` と `or` を派生式として実装する方法を示せ。

Exercise 4.5: Scheme には、`cond` 節の別の構文として、`(<test> => <recipient>)` というものがある。これは、`<test>` が真の値として評価される場合、`<recipient>` が評価される。その値は 1 引数の手続きである必要がある。そして、この手続きが`<test>` という値で呼び出され、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。この拡張構文をサポートするよう `cond` の処理を修正せよ。

Exercise 4.6: `let` 式は派生式である。以下は、

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

次のものと等価だからだ。

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

`let` 式の評価を、上記のような組み合わせの評価に簡約する構文変形 `let->combination` を実装せよ。また、`eval` に適切な節を追加し、`let` 式を扱えるようにせよ。

Exercise 4.7:

`let*` は `let` に似ているが、`let*` 変数の束縛は左から右へ順番に行われ、それぞれの束縛は、それまでの束縛がすべて見える環境の中で行われるというところが違う。例えば、

```
(let* ((x 3) (y (+ x 2)) (z (+ x y 5)))
      (* x z))
```

は 39 を返す。`let*` 式を入れ子の `let` 式の集合として書き直す方法を説明せよ。また、この変形を実行する手続き `let*->nested-lets` を書け。もしくすでに `let` を実装していて (Exercise 4.6)、`let*` を扱えるように評価器を拡張したいとしたら、以下のアクションを持つ節を `eval` に追加することで足りるだろうか。

```
(eval (let*->nested-lets exp) env)
```

それとも、非派生式によって明示的に `let*` を拡張する必要があるだろうか。

Exercise 4.8: “名前つき `let`” は `let` の変種で、以下の形式を持つ。

```
(let <var> <bindings> <body>)
```

`<bindings>` と `<body>` は通常の `let` と同じである。違うのは、`<body>` 内部で `<var>` が、本体が`<body>` で引数が`<bindings>` 内の変数である手続きに束縛されるというところだ。このことによって、`<var>` という名前の手続きを呼び出すことによって、繰り返し`<body>` を実行できる。例えば、反復フィボナッチ手続き (Section 1.2.2) は、名前つき `let` を使って次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` を修正し、名前つき `let` もサポートするようにせよ。

Exercise 4.9: 多くの言語では、`do`, `for`, `while`, `until` のようないろいろな反復構造をサポートしている。Scheme では、反復処理は通常の手続き呼び出しによって表現できるので、特別な反復構造

を入れても計算能力が本質的に上がるわけではない。一方で、そのような構造を持っていると便利なことも多い。いくつかの反復構造を設計し、使用例を示し、それらを派生式として実装するやり方を示せ。

Exercise 4.10: データ抽象化を使うことで、評価対象言語の個別の文法とは独立した eval 手続きを書くことができた。そのことを示すために、eval と apply を変更せずに、この節の手続きを変更することによって Scheme の新しい文法を設計・実装せよ。

4.1.3 評価器のデータ構造

評価器を実装するときには、式の外部文法の定義だけではなく、評価器がプログラム実行の一部として内部で操作するデータ構造も定義する必要があります。これには、手続きや環境の表現、true と false の表現のようなものがあります。

述語のテスト

条件式については、明示的な false オブジェクト以外はすべて true として受け入れます。

```
(define (true? x) (not (eq? x false)))
(define (false? x) (eq? x false))
```

手続きの表現

基本手続きを扱うときには、以下の手続きが利用できるものとします。

- (apply-primitive-procedure <proc> <args>)

これは、与えられた基本手続きをリスト <args> 中の引数の値に適用し、その結果を返します。

- (primitive-procedure? <proc>)

これは、<proc> が基本手続きかどうかテストします。

これらの基本手続きを扱う仕組みについては、Section 4.1.4 でさらに詳しく説明します。

複合手続きは、コンストラクタ `make-procedure` を使って、仮引数・手続き本体・環境によって構築します。

```
(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) (cadddr p))
```

環境に対する演算

評価器には環境を操作する演算が必要となります。Section 3.2で説明したように、環境はフレームの列で、各フレームは変数を対応する値に関連づける束縛テーブルです。環境を操作するには以下の演算を使います。

- `(lookup-variable-value <var> <env>)`
これは、環境(`env`) 内で記号(`var`) に束縛されている値を返します。変数が束縛されていない場合、エラーを発生させます。
- `(extend-environment <variables> <values> <base-env>)`
これは、リスト(`variables`) の記号がリスト(`values`) の対応する要素に束縛されている新しいフレームひとつからなる新しい環境を返します。環境(`base-env`) が外側の環境となります。
- `(define-variable! <var> <value> <env>)`
これは、環境(`env`) の一番目のフレームに、変数(`var`) と値(`value`) を関連づける新たな束縛を追加します。
- `(set-variable-value! <var> <value> <env>)`
これは、環境(`env`) 中の変数 (`var`) の束縛を変更して、変数を値(`value`) に束縛し直します。変数が未束縛であればエラーを発生させます。

これらの演算を実装するために、環境はフレームのリストとして表現することにします。リストの `cdr` が外側の環境となります。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 '())
```

環境のそれぞれのフレームはリストのペアとして表現します。ひとつはそのフレームで束縛される変数のリストで、もうひとつは関連づけられる値のリストです。¹⁴

```
(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))))
```

変数を値に関連づける新しいフレームによって環境を拡張するには、変数のリストと値のリストからなるフレームを作り、それを環境に追加します。変数の数と値の数が一致しなければエラーを発生させます。

```
(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))))
```

環境内の変数を探すには、一番目のフレームの変数リストを走査します。求める変数が見つかれば、値リスト内の対応する要素を返します。現在のフレームで変数が見つからなければ、外側の環境、さらにその外側の環境と検索していきます。空の環境に達したら、“未束縛変数”エラーを発生させます。

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
```

¹⁴以下のコードでは、フレームはデータ抽象化されていません。`set-variable-value!`と`define-variable!`では、`set-car!`を使ってフレームの値を直接変更しています。このフレーム手続きの目的は、環境操作の手続きを読みやすくすることです。

```

        (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))

```

指定した環境の中で変数を新しい値に設定するには、`lookup-variable-value`と同じように変数を走査し、見つかったら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、その変数の束縛を一番目のフレームで探し、束縛がすでにあればその束縛を変更します（これは`set-variable-value!`と同じです）。束縛がなければ、一番目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))))

```

ここで記述した手法は、環境を表現する数多くの方法のひとつにすぎません。データ抽象化を使って、どういう表現を選択するかという細かいところを評価器のほかの部分から分離させたので、やろうと思えば環境の表現方法を変えることもできます (Exercise 4.11参照)。製品品質の Lisp システムでは、評価器の環境演算速度 (特に、変数探索の速度) がシステムの性能に大きく影響します。ここで記述した表現は、概念的には単純なのですが、効率的ではないので普通は製品システムで使われることはないでしょう。¹⁵

Exercise 4.11: フレームをリストのペアとして表現するのではなく、束縛のリストとして表現することもできる。この場合、それぞれの束縛は名前と値のペアとなる。環境演算を書き直し、この表現を使うようにせよ。

Exercise 4.12: 手続き `set-variable-value!`, `define-variable!`, `lookup-variable-value` は、環境構造を順にたどる、より抽象的な手続きによって表現できる。共通パターンを取り出した抽象化を定義し、それらの抽象化によって上記三つの手続きを再定義せよ。

Exercise 4.13: Scheme では、`define` によって新しい束縛を作ることができる一方で、束縛を削除する方法は提供していない。与えられた記号の束縛を評価環境から削除する特殊形式 `make-unbound!` を実装せよ。この問題の仕様は完全ではない。例えば、削除を行うのは環境の一番目のフレームだけでよいのだろうか。仕様を完成させ、行った選択について根拠を述べよ。

4.1.4 評価器をプログラムとして実行する

評価器があれば、Lisp 式が評価されるプロセスを (Lisp によって) 記述したものを手にしたことになります。評価器をプログラムとして表現することの利点のひとつは、プログラムを実行できるということです。こうすると、Lisp の中で実行することによって、Lisp 自身が式を評価する方法の実行モデルが得ら

¹⁵ この表現 (Exercise 4.11 のバージョンも同様です) の欠点は、ある変数の束縛を見つけるのに、評価器がたくさんのフレームを見ていかないといけないことがあるということです (このようなアプローチは *deep binding* (深い束縛) と呼ばれます)。この非効率さを防ぐ方法のひとつとして *lexical addressing* (レキシカルアドレッシング) という戦略を使うというものがあります。これについては Section 5.5.6 で検討します。

れます。これは、評価規則を実験するフレームワークの役割を果たします。この章の後のほうでは、そのような実験を行います。

評価器プログラムが式を簡約していくと、最終的には基本手続きの適用になります。そのため、評価器を実行するのに必要なのは、基本手続きの適用をモデル化するために基層となる Lisp システムを呼び出す仕組みを作ることだけです。

基本手続きの適用を評価したときに、`eval` がオブジェクトを見つけて `apply` に渡せるように、それぞれの基本手続きの名前に対して束縛が存在する必要があります。そのため、グローバル環境を作成し、評価する式に出てくる可能性のある基本手続きの名前にそれぞれ固有のオブジェクトを関連づけます。グローバル環境は、評価する式の中で使えるように、記号 `true`, `false` の束縛も含んでいます。

```
(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))
```

基本手続きを表現する方法は、どのようなものでもかまいません。`apply` が `primitive-procedure?` を使って識別することができ、`apply-primitive-procedure` を使って適用することができればそれで十分です。ここでは、基本手続きはリストとして表現することにします。リストは記号 `primitive` から始まり、その基本手続きを実装している基層 Lisp の手続きを含むものとします。

```
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))
(define (primitive-implementation proc) (cadr proc))
```

`setup-environment` は、基本手続きの名前と実装手続きをリストから取ります。¹⁶

¹⁶ 基層の Lisp で定義されている手続きは、すべてメタ循環評価器の基本手続きとして使うことができます。この評価器に組み込む基本手続きの名前は、基層の Lisp での実装の名前と同じである必要はありません。ここで名前が同じになっているのは、メタ循環

```

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

```

基本手続きを適用するには、基層 Lisp システムを使って実装手続きを引数に適用するだけです。¹⁷

```

(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
    (primitive-implementation proc) args))

```

メタ循環評価器を手軽に動かせるようにするために、基層の Lisp システムの対話型評価環境 (REPL) をモデル化した *driver loop* (ドライバループ) を用意します。これは、*prompt*(プロンプト) を表示し、入力式を読み込み、その式をグローバル環境の中で評価し、その結果を表示するというものです。表示する結果の前には *output prompt* (出力プロンプト) を置いて、式の値をほかの出力から

評価器が実装しているのが Scheme そのものだという理由です。つまり、例えば (list 'first car) や (list 'square (lambda (x) (* x x))) を primitive-procedures のリストに含めるということも可能です。

¹⁷apply-in-underlying-scheme は、以前の章で使ってきました apply 手続きです。メタ循環評価器の apply 手続き (Section 4.1.1) は、この基本手続きの動作をモデル化したもので、apply という名前のものが二つあると、メタ循環評価器を実行する際に技術的な問題が起ります。これは、メタ循環評価器の apply を定義すると、基本手続きの定義が隠されてしまうことによります。回避する方法のひとつとしては、メタ循環の apply を別の名前にして、基本手続きの名前との衝突しないようにするというものがあります。ここではそうせずに、メタ循環の apply を定義する前に以下のようにして基層の apply への参照を保存してあると仮定しています。

```
(define apply-in-underlying-scheme apply)
```

こうすると、別の名前で元の版の apply にアクセスできることになります。

区別できるようにします。¹⁸

```
(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))
```

ここでは、`user-print` という特別な表示手続きを使い、複合手続きの環境部分を表示しないようにしています。環境部分はとても長いリストになったり、循環を含んでいたりするかもしれないからです。

```
(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))
```

これで、後はグローバル環境を初期化してドライバループを開始するだけで、評価器を実行することができます。以下は対話の例です。

```
(define the-global-environment (setup-environment))
(driver-loop)
;;; M-Eval input:
(define (append x y)
```

¹⁸ 基本手続き `read` はユーザからの入力を待ち、入力された次の完全な式を返します。例えば、ユーザが `(+ 23 x)` と入力した場合、`read` は記号 `+`、数値 `23`、記号 `x` を含む 3 要素のリストを返します。ユーザが '`x`' と入力した場合、`read` は記号 `quote` 記号 `x` を含む 2 要素のリストを返します。

```

(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 はそれぞれメタ循環評価器の実験をしていた。Eva は `map` の定義を入力し、それを使うテストプログラムを実行した。テストは問題なくうまくいった。一方、Louis はシステム版の `map` をメタ循環評価器の基本手続として組み込んだ。しかし、それを動かそうとすると、ひどいことになった。Eva の `map` はうまくいくのに Louis の `map` は失敗する理由を説明せよ。

4.1.5 プログラムとしてのデータ

比喩を使うと、Lisp 式を評価する Lisp プログラムについて考える助けになるかもしれません。プログラムの意味についての運用上の見方のひとつとして、プログラムは（無限に大きいかもしれない）抽象機械の記述であるというものがあります。例えば、階乗を計算する見慣れた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)))

```

このプログラムのことを機械と考えて、その機械には減算・乗算・等価テストという部品があり、また切り替えスイッチと別の階乗機械が入っているというように考えることもできます。（この階乗機械は、別の階乗機械を中心に含んでいるため無限です。）Figure 4.2はこの階乗機械の流れ図で、部品の配線を示しています。

同じように、評価器についても、機械の記述を入力として取る非常に特殊な機械として考えることができます。この入力をもとに、評価器は記述された機械の動作を模倣するよう自分自身を設定します。例えば、この評価器

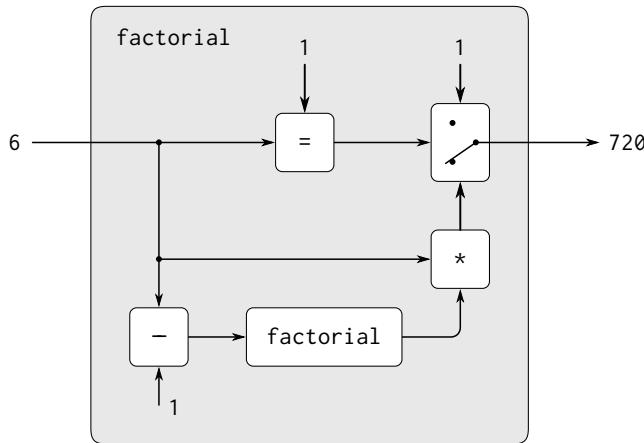


Figure 4.2: 抽象機械として見た階乗プログラム

にFigure 4.3のような `factorial` の記述を与えると、評価器は階乗の計算ができるようになるということです。

この視点から見ると、この評価器は *universal machine*(万能機械) であるといえます。この機械は、Lisp プログラムとして記述されているほかの機械について、それらの動作の真似をすることができます。¹⁹ これは特筆すべきことで

¹⁹ 模倣対象の機械が Lisp で記述されているということは本質ではありません。この評価器に、ほかの言語(例えば C 言語)の評価器としてふるまう Lisp プログラムを与えると、この Lisp 評価器は C 言語評価器の動作を模倣することになり、つまり C 言語のプログラムとして記述されたどんな機械の動作も模倣できるということになります。同じように、C 言語で Lisp 評価器を書くと、どんな Lisp プログラムでも実行できる C 言語プログラムができます。ここで本質的なことは、どんな評価器でも、ほかの評価器の動作を模倣することができるということです。このため、“何が原理上計算可能なのか”(時間とメモリの実用性は度外視して) という概念は、言語やコンピュータとは独立のもので、根底にある *computability*(計算可能性) という概念を反映しています。このことを最初にはっきりと示したのは Alan M. Turing (1912-1954) です。彼の 1936 年の論文は理論計算機科学の基礎を築きました。この論文で、チューリングは単純な計算モデル—現在 *Turing machine*(チューリングマシン) として知られている—を提示し、任意の“実効的処理”はそのような機械のプログラムとして定式化できることを主張しました(この主張は、*Church-Turing thesis*(チャーチ・チューリングのテーゼ) として知られています)。

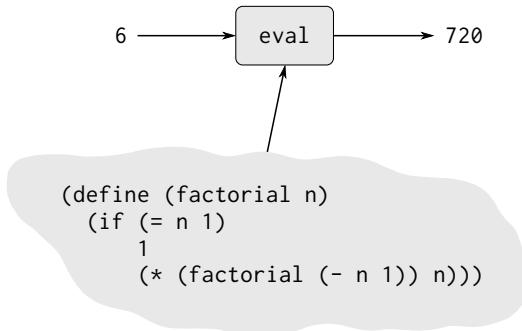


Figure 4.3: 階乗マシンの動作を模倣する評価器

す。例えば、電子回路について、同じような評価器を想像できるでしょうか。その評価器は、例えばフィルタのような、ほかの回路の設計図をコード化した信号を入力として取る回路ということになるでしょう。この入力を与えられると、この回路評価器はまるで同じ仕様を持つフィルタのようにふるまうことになります。そのような万能電子回路というものは、ほとんど想像もできないぐらい複雑なものです。プログラムの評価器が比較的単純なプログラムであるということは注目すべきことです。²⁰

評価器のもうひとつの特筆すべき側面は、プログラミング言語によって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のものとのつなぐ橋の役

次に、チューリングは万能機械、つまりチューリングマシン用プログラムの評価器としてふるまうチューリングマシンを実装しました。彼はこの枠組みを使って、チューリングマシンによって計算できない (Exercise 4.15 参照)、つまり“実効的な処理”として定式化できない良設定問題が存在することを実証しました。チューリングはさらに、実用的な計算機科学への基礎的な貢献も続けました。例えば、彼は汎用目的サブルーチンを使ってプログラミングを構造化するという考え方を発明しました。チューリングの伝記についてはHodges 1983を参照してください。

²⁰ 単純な手続きによって実装された評価器が、評価器そのものよりも複雑なプログラムの動作を模倣できるということが、直感に反すると思う人もいます。万能評価機械が存在するということは、計算というものの深遠ですばらしい性質を表しています。Recursion theory (再帰理論) という数理論理学の一部門では、計算の論理的限界を扱っています。Douglas Hofstadter(ダグラスホフスタッター) の美しい本 *Gödel, Escher, Bach* (ゲーデル、エッシャー、バッハ) では、これらの考え方のいくつかについて探求しています (Hofstadter 1979)。

割を果たすというところです。ここで、(Lisp で実装された) 評価器プログラムが動いていて、ユーザがその評価器に式を入力して、結果を見ているところを想像してみてください。ユーザの視点からは、 $(* x x)$ のような入力式はプログラミング言語の式で、評価器が実行するべきものです。しかし、評価器の視点からは、式は明確に定義されたルール集合に従って操作するべき単純なリスト（この場合、 $*$, x , x という三つの記号からなるリスト）ということになります。

ユーザにとってのプログラムは評価器にとってはデータということになりますが、これは必ずしも混乱するようなことではありません。実際のところ、この違いを無視して、データオブジェクトを Lisp の式として明示的に評価する能力をユーザに与えるために、eval 手続きをプログラム中で使えるようにすることが便利であることもあります。Lisp 方言の多くは、引数として式と環境を取り、その環境について式を評価する基本手続きの eval を提供しています。²¹ このため、

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

と

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

は、両方とも 25 を返します。²²

Exercise 4.15: 1 引数の手続き p とオブジェクト a について、式 (p a) が (エラーメッセージを出して終了したり、無限に動き続けるのではなく) 値を返すとき、p は a について “停止” するという。任意の手続き p と任意のオブジェクト a に対して、p が a について停止するかどうかを正確に決定するような手続き halts? というものを書くことは不可能であることを示せ。以下のように推論せよ。

²¹警告：この基本手続きの eval は、Section 4.1.1で実装した eval 手続きを同一のものではありません。というのは、この基本手続きの eval は実際の Scheme 環境を使うもので、Section 4.1.3で構築したようなサンプル環境構造を使うわけではないからです。これらの実際の環境はユーザが通常のリストとして操作することはできず、eval やほかの特別な演算によってアクセスする必要があります。同じように、以前の章で出てきた基本手続きの apply も、メタ循環 apply と同一のものではありません。基本手続きの apply は、実際の Scheme 手続きを使うもので、Section 4.1.3や Section 4.1.4で構築した手続きオブジェクトを使うわけではないからです。

²²Scheme の MIT 実装には eval が含まれていて、また user-initial-environment という、ユーザの入力式が評価される初期環境に束縛された記号も含まれています。

もしそのような手続き `halts?` が存在するとなれば、次のようなプログラムが実装できるはずだ。

```
(define (run-forever) (run-forever))
(define (try p)
  (if (halts? p p) (run-forever) 'halted))
```

ここで、式 `(try try)` を評価した場合について考え、起こりうる結果（停止するにしても、無限に動き続けるにしても）がすべて、意図する `halts?` の動作と矛盾することを示せ。²³

4.1.6 内部定義

評価の環境モデルとメタ循環評価器は、定義を順に実行し、環境のフレームを一定義分づつ拡張していきます。インタラクティブなプログラム開発では、プログラマは手続きの適用と新しい手続きの定義を自由に行ったり来たりする必要があるので、これは特に便利です。しかし、(Section 1.1.8で導入した) ブロック構造を実装するために使う内部定義についてよく考えると、名前ごとに環境を拡張するというのは、局所変数を定義する方法としてあまり優れているとはいえないのではないかと気づくと思います。

次のような、内部定義のある手続きについて考えてみましょう。

```
(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>)
```

ここでしたいことは、手続き `even?` の本体に含まれる名前 `odd?` が、`even?` の後に定義されている手続き `odd?` を参照するようにするということです。名前 `odd?` のスコープは `f` の本体全体であり、`odd?` の定義のある場所より後の `f` の本体の一部というわけではありません。また、`odd?` 自身も `even?` を使って定義されているということを考えると—つまり、`even?` と `odd?` は相互再帰手続きということになります—二つの `define` のまともな解釈方法としては、`even?`

²³ ここでは `halts?` が手続きオブジェクトを与えられるものと規定していますが、`halts?` が手続きのテキストとその環境にアクセスできるとしても、依然としてこの推論が成り立つということに注意してください。これはチューリングの有名な *Halting Theorem* (停止性問題) で、これが *non-computable* (計算不可能) な問題、つまり計算手続きとして実行不可能な良設定問題の最初の明確な例となりました。

と `odd?` という名前が同時に環境に追加されているかのように見なす以外にならないということがわかります。より一般的には、ブロック構造では、局所的な名前のスコープは、`define` が評価されている手続きの本体全体だということです。

私たちのインタプリタは、たまたま `f` の呼び出しを正しく評価できるようになっていますが、そうなっているのは“偶然の”理由のためです。内部手続きの定義は最初に来るので、これらの手続きへの呼び出しは、これらの手続きがすべて定義される前には評価されないからです。そのため、`even?` が実行されるときには `odd?` はすでに定義されていることになります。実際のところ、内部定義が本体の先頭に来て、定義された変数に対する値の式の評価が定義された変数をひとつも実際に使わない手続きの場合、ここでの逐次評価メカニズムは同時定義を直接実装したメカニズムと同じ結果を返します（これらの制約に従わず、逐次定義が同時定義と等価にならないような手続きの例については、Exercise 4.19を参照してください）。²⁴

しかし、内部定義が真に同時スコープを持つように定義を扱う単純な方法があります。現在の環境にあるすべての局所変数を、値の式を評価する前に作ってしまうのです。その方法のひとつとしては、`lambda` 式に対する構文変形を使うというものがあります。`lambda` 式の本体を評価する前に、本体内の内部定義を“走査”し、削除してしまうのです。内部で定義された変数は `let` によって作られ、それから代入によってそれらの持つ値に設定されることになります。例えば、以下の手続きは

```
(lambda (vars)
  (define u (e1))
  (define v (e2))
  (e3))
```

以下のように変形されます。

²⁴Chapter 1のFootnote 28で“責任を負いかねます”と書いたのは、プログラムにこの評価メカニズムに依存してほしくないというのがその理由です。内部手続きが最初に来て、定義が評価されている間にお互いを使わないと決めてあるため、Scheme の IEEE 標準を実装する際には、これらの定義を評価するメカニズムに選択の余地があります。どの評価ルールを選び、どの評価ルールを選ばないかということは、“不適格な”プログラムを解釈する場合以外には影響のない小さな問題のようにここでは思えるかもしれません。しかし、内部定義の同時スコープモデルに移行することによって、そうしない場合にコンパイラの実装で起こるいくつかの厄介な問題を避けられるようになるということを、Section 5.5.6では見ていきます。

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

ここで、`*unassigned*` は特別な記号で、未割り当ての変数の値を使おうとしたらエラーを発生させるものです。

内部定義を走査する以外の戦略のひとつは、Exercise 4.18に示されています。上記の変形とは違って、こちらは定義された変数の値はそれらの変数の値を使わないでも評価できるという制約を強制するものです。²⁵

Exercise 4.16: この練習問題では、上で説明した内部定義の手法を実装する。評価器は `let` をサポートしているとする (Exercise 4.6 参照)。

a `lookup-variable-value` (Section 4.1.3) を変更し、見つけた値が記号 `*unassigned*` であればエラーを発生させるようにせよ。

b 手続きの本体を引数に取り、上記の変形を施すことによって内部手続きを持たない等価な手続きを返す `scan-out-defines` を書け。

c `scan-out-defines` をインタプリタの `make-procedure` または `procedure-body` (Section 4.1.3 参照) に組み込め。どちらに組み込むのがよいだろうか。また、それはなぜか。

Exercise 4.17: 本文の手続き中の式 `<e3>` を評価しているときに有効である環境の図を描き、定義が逐次的に解釈される場合と定義を上記のように掲き出す場合とでそれぞれどのようになるか比較せよ。変形したプログラムでは、なぜフレームがひとつ余分にあるのだろうか。環境構造のこの違いが正しいプログラムのふるまいには決して影響を及ぼさない理由を説明せよ。インタプリタに

²⁵Scheme の IEEE 標準では、この制約に従うのはプログラマの責任で、実装側でそれを強制する責任はないと規定しているので、実装戦略に幅を持たせています。MIT Scheme を含め、いくつかの Scheme 実装は、上に示した変形を使っています。そのため、この制約に従わないプログラムでも、そのような実装では実際には動くこともあります。

余分なフレームを構築することなしに“同時”スコープルールを実装させる方法を設計せよ。

Exercise 4.18: 定義を掃き出す別の戦略として、本文中の例を以下のように変形するものを考える。

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

ここで、`a`, `b` はインタプリタの作成した新しい変数名を表現しているものとする。これらはユーザのプログラムには現れない。Section 3.5.4 の `solve` 手続きをについて考える。

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

この手続きは、この練習問題で示したように内部定義を掃き出した場合に、うまく動くだろうか。本文中で示したように掃き出した場合はどうだろうか。説明せよ。

Exercise 4.19: Ben Bitdiddle, Alyssa P. Hacker, Eva Lu Ator は、以下の式を評価したときの結果として望ましいものは何かということで議論している。

```
(let ((a 1))
  (define (f x)
    (define b (+ a x))
    (define a 5)
    (+ a b))
  (f 10))
```

Ben は、`define` に逐次ルールを使って結果を得るべきだと主張する。`b` は 11 と定義され、その次に `a` は 5 と定義されるので、結果

は 16 であるべきだということだ。Alyssa は、相互再帰には内部手続き定義の同時スコープルールが必要で、手続きの名前とほかの名前で扱いを変えるのは不合理だと反対する。このため、彼女はExercise 4.16で実装したメカニズムを支持している。これを使うと、`b` の値を計算する時点で、`a` は未割り当てということになる。よって、Alyssa の見方では、手続きはエラーを発生させるべきだということだ。Eva の意見はどちらとも違う。彼女は、`a` と `b` の定義が真に同時だとするなら、`b` を評価する際には `a` の値として 5 を使うべきだと言う。よって、Eva の見方では、`a` は 5 となり、`b` は 15 となり、結果は 20 となるべきだということになる。これらの見方の中で、(支持できるものがあるとすれば) あなたはどの見方を支持するか。また、Eva の言うようにふるまう内部定義を実装する方法を考えよ。²⁶

Exercise 4.20: 内部定義は逐次的に見えるのに実際は同時だということから、内部定義を完全に回避し、代わりに特殊形式 `letrec` を使うという人もいる。`letrec` は `let` に似ているので、そこで束縛され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自然ではない。上記の例の手続き `f` は、内部手続きを使わずに、まったく同じ意味を持つものを書くことができる。

```
(define (f x)
  (letrec
    ((even? (lambda (n)
              (if (= n 0) true (odd? (- n 1)))))
     (odd? (lambda (n)
              (if (= n 0) false (even? (- n 1))))))
    (f の本体の残り)))
```

`letrec` 式は以下の形式を持つ。

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

²⁶MIT Scheme の実装者たちは、次のような理由から Alyssa を支持しています。まず、Eva は原理上は正しいことになります。定義は同時だと見なされるべきだからです。しかし、Eva が求めることを行う一般的で効率的な仕組みを実装するのは難しいようです。そのような仕組みが作れない以上は、同時定義の難しいケースではエラーを発生させる (Alyssa の考え方) ほうが、正しくない答えを出す (Ben のやり方) よりもよいということです。

`letrec` 式は `let` の亜種で、変数 $\langle var_k \rangle$ に初期値を与える式 $\langle exp_k \rangle$ は、すべての `letrec` の束縛を含む環境の中で評価される。このことによって、上の例での `even?` と `odd?` の相互再帰や、以下のように 10 の階乗を評価する場合のように、束縛内での再帰が使えるようになる。

```
(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。上の本文やExercise 4.18で示したように、`letrec` 式を `let` 式に変形すること。つまり、`letrec` の変数を `let` によって作り、値を `set!` で代入することになる。

b Louis Reasoner は、内部定義について考えるうちに混乱してしまった。手続きの中で `define` を使いたくないなら `let` を使えばいいじゃないかというのが彼の考えだ。この練習問題で定義した `f` について、式 $(f\ 5)$ の評価中に $\langle f\ \text{の本体の残り} \rangle$ が評価される環境を示した環境図を描き、彼の考えの穴を示せ。同じ式の評価で、`f` の定義中の `letrec` の代わりに `let` が使われている場合の環境図も書け。

Exercise 4.21: 驚くべきことに、Exercise 4.20での Louis の直感は正しい。`letrec` を（それどころか、`define` さえ）使わずに再帰手続きを規定することは実際に可能だ。しかし、これを実現する方法は Louis の想像よりもずっと難しいものになる。次の式は、再帰的な階乗手続きを適用することによって 10 の階乗を求めている。²⁷

```
((lambda (n)
  ((lambda (fact) (fact fact n)))
```

²⁷ この例は、`define` を使わずに再帰手続きを定式化するプログラミングトリックを示したもので、この種のトリックとして最も一般的なものは *Y operator* (*Y* コンビネータ) です。これを使うと、“純粹ラムダ計算”による再帰の実装が可能ですが (λ 計算の詳細については Stoy 1977 を、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)。

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

- a これが実際に階乗を計算することを(式を評価して)確認せよ。フィボナッチ数を計算する同様の式を考えよ。
- b 相互再帰の内部定義を含む以下の手続きについて考える。

```
(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))
```

以下の空欄に式を埋め、内部定義も `letrec` も使わない `f` の別の定義を完成させよ。

```
(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 構文解析を実行から分離する

上で実装した評価器は単純なのですが、とても非効率的です。これは、式の構文解析と式の実行が交互に行われることによります。このため、プログラムが何回も実行される場合、構文解析が何回も行われてしまします。例えば、次の `factorial` の定義を使って (`factorial 4`) を評価することを考えてみましょう。

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

`factorial` が呼ばれるたびに、評価器は本体が `if` 式であることを判断し、述語を取り出す必要があります。述語を評価して値によってディスパッチを行う

のはその後になります。式 (* (factorial (- n 1)) n) を評価したり、部分式 (factorial (- n 1)) や (- n 1) を評価したりするたびに、評価器は eval の中で場合分けを行い、式が手続き適用だということを判断し、演算子とオペランドの抽出をする必要があります。この解析は高コストなので、何度も行うのは無駄です。

評価器を変形することで、構文解析が一度だけ実行されるようにしてずっと効率をよくすることができます。²⁸ まず、式と環境を引数に取る eval を二つに分割します。手続き analyze は式だけを引数に取り、構文解析を行って新しい手続きを返します。この新しい手続きは *execution procedure*(実行手続き) で、解析済みの式を実行するうえでするべき仕事をカプセル化したものです。実行手続きは環境を引数に取り、評価を完了します。こうすると、実行手続きが何度も呼ばれたとしても、analyze はひとつの式に対して一回しか呼ばれないで、作業量を減らすことができます。

解析と実行を分けることで、eval は以下のようになります。

```
(define (eval exp env) ((analyze exp) env))
```

analyze呼び出しの返り値は実行手続きで、それを環境に適用します。analyze 手続きは、Section 4.1.1 の元の eval で実行したのと同じ場合分けです。違うのは、ディスパッチ先の手続きで行うのが解析だけで、完全な評価を行わないというところです。

```
(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))))
```

²⁸ このテクニックは、Chapter 5 で検討するコンパイル処理には不可欠のものです。Jonathan Rees は、1982 年ごろ T プロジェクトのためにこのような Scheme インタプリタを書いています (Rees and Adams 1982)。Marc Feeley (1986) (Feeley and Lapalme 1987 も参照) は、このテクニックを修士論文で独立に開発しました。

```
((application? exp) (analyze-application exp))
(else
  (error "Unknown expression type: ANALYZE" exp))))
```

自己評価式を扱う最も単純な形の構文解析は以下のようになります。環境引数を無視して式を返すだけの実行手続きを返します。

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

クオート式については、実行フェーズではなく解析フェーズでテキストを一度だけ取り出すことになるので、少しだけ効率をよ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う必要があります。変数の値の検索は環境の知識に依存するからです。²⁹

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

analyze-assignment も、環境が与えられる実行時まで、変数を実際に設定するのを遅らせる必要があります。しかし、解析時に assignment-value 式を(再帰的に) 解析できるというのは効率の向上に大きく寄与します。assignment-value 式は一回しか解析されないことになるからです。定義についても同じことが言えます。

```
(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)
```

²⁹しかし、変数探索の中にも、構文解析の一部として実行可能な重要な部分があります。Section 5.5.6で示すように、環境構造の中で変数の値が見つかる場所は決定することができます。そのため、変数にマッチするエントリを探すために環境を走査しないで もよくなります。

```
(let ((var (definition-variable exp))
      (vproc (analyze (definition-value exp))))
  (lambda (env)
    (define-variable! var (vproc env) env)
    'ok)))
```

`if` 式については、解析時に述語、結果式、代替式を取り出し、解析します。

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env) (if (true? (pproc env))
                        (cproc env)
                        (aproc env)))))
```

`lambda` 式の解析も、大きな効率向上が可能です。`lambda` を評価した結果の手続きが何回も適用されることになるとしても、`lambda` の本体は一回しか解析されないからです。

```
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp)))
    (bproc (analyze-sequence (lambda-body exp))))
  (lambda (env) (make-procedure vars bproc env))))
```

式の列 (`begin` や、`lambda` 式の本体に出てくるようなもの) の解析はより複雑です。³⁰ 列のそれぞれの式は解析され、実行手続きが作られます。これらの実行手続きは、引数に環境を取り、その環境を引数としてそれぞれの実行手続きを順番に呼び出していくひとつの実行手続きにまとめられます。

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

³⁰列の処理について理解を深めるにはExercise 4.23を参照してください。

```
(let ((procs (map analyze exps)))
  (if (null? procs) (error "Empty sequence: ANALYZE"))
  (loop (car procs) (cdr procs))))
```

適用を解析するには、まず演算子とオペランドを解析します。それから、(実際に適用する手続きを得るために) 演算子の実行手続きを呼び出し、(実際の引数を得るために) オペランドの実行手続きを呼び出す実行手続きを構築します。次に、これらを `execute-application` に渡します。これはSection 4.1.1の `apply` と似たようなものです。`execute-application` が `apply` と違うのは、複合手続きの本体は解析済みで、それ以上の解析を行う必要がないということです。代わりに行うのは、拡張した環境を引数として本体の実行手続きを呼び出すことです。

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

新しい評価器は、節Section 4.1.2, Section 4.1.3, Section 4.1.4に出てきたものと同じデータ構造、構文手続き、実行時サポート手続きを使います。

Exercise 4.22: この節の評価器を拡張し、特殊形式 `let` をサポートするようにせよ (Exercise 4.6 参照)。

Exercise 4.23: Alyssa P. Hacker は、`analyze-sequence` がどうしてこんなに複雑になるのか理解できない。ほかの解析手続きは、どれも Section 4.1.1 に出てきた対応する評価手続き (または `eval` 節) を素直に変形したものだ。彼女は、`analyze-sequence` は以下のようになると思っていた。

```
(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、本文中のバージョンでは解析時に列を評価するという仕事以上のことを行っていると Alyssa に説明する。Alyssa の列実行手続きは、個別の実行手続きに対する呼び出しを組み込むのではなく、手続きを呼ぶためにループしている。実質的には、列内の個別の式は解析されているが、列そのものは解析されていないことになる。

`analyze-sequence` の両バージョンを比較せよ。例えば、列が式をひとつだけ含んでいるというよくあるケース (手続きの本体ではそれが普通だ) について考える。Alyssa のプログラムに生成された実行手続きはどのような仕事をするか。上の本文中のプログラムに生成された実行手続きについてはどうだろうか。二つの式を持つ列については、両バージョンはどのように異なるだろうか。

Exercise 4.24: 元のメタ循環評価器とこの節のバージョンの速度を比較する実験をいくつか設計し、実行せよ。その結果を使って、いろいろな手続きについて、解析と実行に使われる時間の割合を見積もれ。

4.2 Scheme の変異版 — 遅延評価

Lisp のプログラムとして表現された評価器を手に入れたので、評価器を修正するだけで言語設計上の別の選択肢の実験を行うようになります。実際、新言語の開発は、最初は既存の高級言語の中に新しい言語を埋め込む評価器を書くことによって行われることが多いのです。例えば、Lisp の修正提案のいくつかの側面について Lisp コミュニティのほかのメンバと議論をしたければ、その変更を実装した評価器を提供することができます。こうすると、受け取った人はその新しい評価器で実験を行い、さらなる修正という形でコメントを返すことができます。高レベルの実装ベースは、評価器のテストとデバッグを簡単にするだけではありません。既存の言語に埋め込むことで、ここでの組み込み Lisp 評価器が基層の Lisp の基本要素や制御構造を使っているのと同じように、言語を設計する人が機能をパクる (snarf)³¹ ことができるようになります。言語設計者が低レベル言語やハードウェアで完全な実装を (必要に応じて) 構築するのは、後になってからでも大丈夫です。この節と次の節では、表現力を大きく向上させる Scheme の変異版について探求していきます。

4.2.1 正規順序と適用順序

Section 1.1 で評価モデルの議論を始めたとき、Scheme は *applicative-order* (適用順序) 言語だと述べました。つまり、Scheme の手続きの引数はすべて手続きの適用時に評価されるということです。それに対して、*normal-order* (正規順序) 言語では、実際の引数の値が必要になるまで手続き引数の評価を遅らせます。手続き引数の評価を可能な限り (例えば、基本演算に必要になるまで) 遅らせることは、*lazy evaluation* (遅延評価) と呼ばれます。³² 以下の手続きについて考えてみましょう。

```
(define (try a b) (if (= a 0) 1 b))
```

³¹Snarf: “(特に、大きな文書やファイルを) 所有者の許可を得て、または許可を得ずに、使うために取ってくること。” Snarf down: “snarfすること。時に、吸収する・処理する・理解するという意味を帯びる。”(これらの定義は Steele et al. 1983 から snarf したものです。Raymond 1993 も参照。)

³²“遅延”という用語と“正規順序”という用語の違いは少々曖昧です。一般的に、“遅延”は特定の評価器の仕組みについて言いますが、“正規順序”は特定の評価戦略とは独立の言語の意味論について言います。しかし、これは厳格な区別というわけではなく、二つの用語を区別せずに使われることもよくあります。

`(try 0 (/ 1 0))` を評価すると、Scheme ではエラーになります。遅延評価の場合はエラーは起こりません。この式を評価すると 1 を返します。引数の `(/ 1 0)` は一度も評価されることがないからです。

次の手続き `unless` の定義は、遅延評価を利用した例となっています。

```
(define (unless condition usual-value exceptional-value)
  (if condition exceptional-value usual-value))
```

これは、次のような式で使うことができます。

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

これは、適用順序の言語では動作しません。`unless` が呼ばれる前に、通常の値と例外の値が両方とも評価されてしまうためです ([Exercise 1.6](#) と比較してください)。遅延評価の利点のひとつは、`unless` のようなある種の手続きが、引数のうち評価するとエラーになったり停止しなかったりするものがあったとしても、役に立つ計算ができるということです。

ある引数を評価し終わる前に手続きの本体に入るという場合、その手続きはその引数について *non-strict*(非厳密) であると言います。ある引数が手続きの本体に入る前に評価された場合、その手続きはその引数について *strict*(厳密) であると言います。³³ 純粹な適用順序言語では、すべての複合手続きはすべての引数について厳密で、基本手続きは厳密であることも非厳密であることもあります。このほかに、定義する手続きの厳密さをプログラマが細かくコントロールできるようにしている言語もあります ([Exercise 4.31](#) 参照)。

非厳密にすることで有用な結果が得られる手続きの顕著な例は `cons`(に限らず、一般的にほとんどすべてのデータ構造のコンストラクタ) です。要素の値がわかっていない場合でも、要素を組み合わせ、データ構造を作り、結果となるデータ構造に対して演算を行うことで、有用な計算ができます。例えれば、リストの個別の要素の値を知らない今までリストの長さを計算するという考え方には、どこにもおかしなところはありません。[Section 4.2.3](#) では、この考え方を利用し、非厳密な `cons` ペアからなるリストとして [Chapter 3](#) のストリームを実装します。

³³ “厳密”、“非厳密”という用語は、本質的には“適用順序”、“正規順序”と同じ意味です。違うのは、“厳密”、“非厳密”は個別の手続きと引数について言うもので、言語全体について言うものではないというところです。プログラミング言語のカンファレンスでは、次のような話を聞くかも知れません。“正規順序言語の Hassle は、厳密な基本手続きをいくつか持っています。そのほかの手続きは引数を遅延評価で取ります。”

Exercise 4.25: `unless` を (通常の適用順序 Scheme で) 上で示したように定義し、それから `factorial` を `unless` を使って以下のように定義するとする。

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

`(factorial 5)` を評価すると何が起こるだろうか。この定義は正規順序言語でも動くだろうか。

Exercise 4.26: Ben Bitdiddle と Alyssa P. Hacker は、`unless` のようなものを実装する際に遅延評価がどれだけ重要かというところで意見が分かれている。Ben は、適用順序でも特殊形式として `unless` を実装できると指摘した。Alyssa は、そういうふうにしてしまうと、`unless` はただの構文になってしまって、高階手続きと組み合わせて使える手続きにはならないと反論した。両者の議論の細部を補え。`unless` を (`cond` や `let` のように) 派生手続きとして実装する方法を示せ。また、`unless` を特殊形式でなく手続きとして持つほうが役に立つ状況の例を挙げよ。

4.2.2 遅延評価のインタプリタ

この節では、正規順序の言語を実装していきます。この言語は Scheme と同じですが、複合手続きがすべての引数について非厳密であるという点が違います。基本手続きは厳密なままで。Section 4.1.1 の評価器を修正して、解釈対象の言語がそのようにふるまうようにするのは難しくありません。必要な変更は、ほとんどすべて手続き適用が中心となっています。

ここでの基本的な考え方は、手続き適用の際に、どの引数を評価してどの引数を遅延させるかインタプリタが決めないといけないということです。遅延された引数は評価されません。その代わり、それらの引数は *thunk*⁵ (サンク) と呼ばれるオブジェクトに変形されます。³⁴サンクは、引数の値が必要に

訳注 5 “think” の過去分詞は本来は “thought” ですが、考案者たちが夜中まで議論していて頭が朦朧としているときにつけたのでこの形になったという説があります。

³⁴サンクという単語は、Algol 60 の call-by-name の実装について議論していた非公式な作業部会によって考え出されたものです。彼らは、式のほとんどの分析 (“thinking

なったときに、あたかも適用時に評価されていたかのようにその値を返します。サンクはそのために必要な情報を含んでいる必要があります。よって、サンクは引数の式と、手続き適用が評価された環境を持っていなければならないことになります。

サンク中の式を評価するプロセスは*forcing*(強制)と呼ばれます。³⁵一般的に、サンクが強制されるのはその値が必要になったときだけです。これには、サンクの値を使う基本手続きにサンクが渡される場合、条件式の述語の値である場合、サンクが手続きとして適用される演算子の値があります。設計上の選択肢として、Section 3.5.1で遅延オブジェクトに対して行ったように、サンクを*memoize*(メモ化)するかどうかということがあります。メモ化をする場合、サンクが初めて強制されたとき、サンクは計算された値を格納します。それ以降に行われる強制では、計算を再び行うことはせずに、格納された値を単純に返します。ここでは、インタプリタがメモ化を行ないようにします。多くの応用ではそのほうが効率的だからです。しかし、これには気をつけないといけないところもあります。³⁶

評価器の修正

遅延評価器とSection 4.1の評価器との主な違いは、`eval`と`apply`での手続き適用の扱いです。

`eval`の`application?`節は次のようにになります。

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

about")はコンパイル時にできるということに気づきました。つまり、実行時には式はすでに「考えられて」("thunk" about(Ingberman et al. 1960))いることになります。

³⁵これは、Chapter 3でストリームを表現するために導入した遅延オブジェクトに対する`force`の使用と似ています。ここで行っていることとChapter 3で行ったことの決定的な違いは、ここでは遅延と強制を評価器に組み込んでいて、言語全体を通して統一的かつ自動的に扱えるようにしているということです。

³⁶メモ化と遅延評価を組み合わせたものは、*call-by-name*(名前呼び)引数渡しに対して*call-by-need*(必要呼び)引数渡しと呼ばれることがあります。(call-by-nameはAlgol 60で導入されたものですが、これはメモ化しない遅延評価に似ています)。言語設計者である私たちは、評価器にメモ化させることも、メモ化させないことも、その選択をプログラマに任せることもできます(Exercise 4.31)。Chapter 3で見てきたことから予測できるように、代入がある場合、この選択は難しく混乱を招く問題を引き起こします(Exercise 4.27、Exercise 4.29参照)。Clinger (1982)による優れた論文では、ここで起こる混乱の複数の側面について明らかにすることを試みています。

```
(operands exp)
env))
```

これはSection 4.1.1の eval の application? 節とほとんど同じです。しかし、遅延評価では、apply の引数として、オペランド式を評価して得られたものを渡すのではなく、オペランド式をそのまま渡します。引数を遅延化する場合にはサンクの構築に環境が必要になるので、環境も一緒に渡す必要があります。遅延評価器でも演算子は評価します。手続きの型(基本手続きもしくは複合手続き)によってディスパッチを行い、適用するためには、apply には実際の手続きが必要だからです。

式の実際の値が必要になったときには、単純に eval をする代わりに以下のようにします。

```
(define (actual-value exp env)
  (force-it (eval exp env)))
```

こうすることで、式の値がサンクであれば強制されることになります。

apply の新しいバージョンも、Section 4.1.1 のバージョンとほとんど同じです。違うのは、eval が未評価のオペランド式を渡していくというところです。基本手続きについては、(厳密であるため) 適用する前にすべての引数を評価します。複合手続きについては、(非厳密であるため) 手続き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きはほとんどSection 4.1.1のlist-of-valuesそのままでですが、違うのはlist-of-delayed-argsが引数を評価するのではなく遅延化すること、list-of-arg-valuesがevalではなくactual-valueを使うということです。

```
(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))))
```

ほかに評価器を修正しなければならないところはifの処理の中にあります。ここでは、述語式の真偽をテストする前に、evalではなくactual-valueを使って述語式の値を取る必要があります。

```
(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手続きを変更して、evalの代わりにactual-valueを使うようにする必要があります。これは、遅延化された値がREPLまで返された場合、表示前に強制するようにするために、また、この評価器が遅延評価器であることを示すためにプロンプトも変更しておきます。

```
(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))
```

これらの変更を行つたら、評価器を動かしてテストできます。Section 4.2.1で検討した `try` 式の評価がうまくいったら、インタプリタが遅延評価を実行しているということになります。

```
(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

この評価器は、手続きが引数に適用されるときにサンクを作り、後でそれらのサンクを強制するようにしなければなりません。サンクは、後で引数を生成できるように、式と環境をまとめておく必要があります。サンクを強制するには、単純にサンクから式と環境を取り出し、その環境の中で式を評価します。ここでは `eval` ではなく `actual-value` を使います。これは、式の値がそれ自身サンクであった場合、サンクでないものにたどり着くまで強制を続けるようにするためです。

```
(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簡単な方法のひとつは、式と環境を含むリストを作るということです。そこで、サンクは次のように作ることにします。

```
(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))
```

実際のところは、評価器に入れたいものはこれとはちょっと違います。必要なのはメモ化されたサンクです。そのため、サンクが強制されたときには格納された式をその値で置き換え、`thunk` タグを別のものに変えて、サンクが評価済みだということがわかるようにします。³⁷

```
(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)
                     '()) ;必要のなくなった env を忘れる
           result)))
```

³⁷ここで気をつけるところは、式の値を評価したら、サンクから `env` も消すということです。こうしても、インタプリタから返される値は変わりません。しかし、空間の節約にはなります。Section 5.3で検討するように、必要がなくなり次第サンクから `env` への参照を削除するようになると、この構造は*garbage-collect*(ガベージコレクション)され、空間はリサイクルされるからです。

同様に、Section 3.5.1のメモ化した遅延オブジェクトの場合も、必要のなくなった環境は削除できるところです。そのためには、`memo-proc` で `(set! proc '())` のようなことを行って、(`delay`が評価された環境を含む) 手続き `proc` を値を格納した後に捨てるようになります。

```
((evaluated-thunk? obj) (thunk-value obj))  
(else obj)))
```

delay-it 手続きは、そのままでメモ化ありでもメモ化なしでも動くというところに注意してください。

Exercise 4.27: 以下の定義を遅延評価器に入力したとする。

```
(define count 0)  
(define (id x) (set! count (+ count 1)) x)
```

以下の一連の対話の空欄の値を埋め、回答について説明せよ。³⁸

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

Exercise 4.28: eval は、演算子を apply に渡す前に、eval ではなく actual-value を使って演算子を評価する。これは、演算子の値を強制するためである。ここで強制を行う必要があるということがわかるような例を示せ。

Exercise 4.29: メモ化しなければ、メモ化した場合よりも実行がずっと遅くなりそうなプログラムを示せ。また、以下の対話について考えよ。この対話の中では、id 手続きの定義は Exercise 4.27と同じで、count は 0 から始める。

³⁸ この練習問題は、遅延評価と副作用との干渉や副作用が大きな混乱のもとになるとすることを示しています。これは、Chapter 3 の議論から予想できる範囲内のことです。

```
(define (square x) (* x x))
;; L-Eval input:
(square (id 10))
;; L-Eval value:
⟨response⟩
;; L-Eval input:
count
;; L-Eval value:
⟨response⟩
```

評価器がメモ化されている場合とメモ化されていない場合の両方について、応答を示せ。

Exercise 4.30: 改心した C 言語プログラマの Cy D. Fect は、遅延評価器では列の中の式を強制しないので、副作用が起こらないことになるのではないかと心配している。列の中の式の値は最後のもの以外は使われない（式は変数代入や表示などの作用のために存在する）ので、（例えば基本手続きの引数として）値が使われて強制されるということが起こらないかもしれないと言うのだ。そのため Cy は、列を評価するときは、最後の式以外のすべての式を強制するべきだと考えている。彼は、Section 4.1.1 の eval-sequence を修正し、eval でなく actual-value を使用したほうがよいと提案した。

```
(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 は間違っていると考えている。彼は、Exercise 2.23 で説明した for-each 手続きを Cy に見せた。これは、副作用のある列の重要な例となっている。

```
(define (for-each proc items)
  (if (null? items)
      'done
      (begin (proc (car items))
             (for-each proc (cdr items)))))
```

彼は、本文中の評価器（元の eval-sequence を持つ）はこれを正しく扱えると主張する。

```
;; L-Eval input:  
(for-each (lambda (x) (newline) (display x))  
          (list 57 321 88))  
57  
321  
88  
;; L-Eval value:  
done
```

for-each のふるまいについて、なぜ Ben が正しいのか説明せよ。

- b Cy は、for-each については Ben が正しいと認めたが、eval-sequence に対する変更を提案したときに考えていたのはこの種のプログラムではないと言う。彼は、次の二つの手続きを遅延評価器で定義した。

```
(define (p1 x)  
  (set! x (cons x '(2)))  
  x)  
(define (p2 x)  
  (define (p e)  
    e  
    x)  
  (p (set! x (cons x '(2)))))
```

元の eval-sequence を使うと、(p1 1) と (p2 1) の値はどうなるだろうか。Cy が提案する eval-sequence に対する変更を適用した場合はどうだろうか。

- c Cy はまた、彼の提案する eval-sequence に対する変更は、a の例でのふるまい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 あなたの考えでは、遅延評価器で列を扱うのはどのようにするのがよいだろうか。Cy のアプローチだろうか、本文中のアプローチだろうか。それとも、別の何かのアプローチだろうか。

Exercise 4.31: この節でのアプローチはちょっと気持ちの悪いところがある。というのは、互換性のない変更を Scheme に加えてしまっているからだ。遅延評価を *upward-compatible extension* (上位互換性のある拡張) として、つまり通常の Scheme が前と同じように動くように実装できれば、そのほうがよいだろう。これは、手続き定義の構文を拡張し、引数を遅延させるかどうかをユーザがコントロールできるようにすれば実現できる。ついでに、遅延をメモ化するかどうかの選択もユーザができるようにしてもらいたい。例えは、

```
(define (f a (b lazy) c (d lazy-memo))
  ...)
```

この定義は、4引数の手続きで、1番目と3番目の引数は手続きが呼ばれたときに評価され、2番目の引数は遅延化され、4番目の引数は遅延化とメモ化が行われるものとして `f` を定義する。つまり、通常の手続き定義をすると通常の Scheme と同じふるまいになるが、すべての複合手続きの各仮引数に `lazy-memo` 宣言を加えると、この節で定義した遅延評価器のふるまいになる。Scheme にそのような拡張を加えるのに必要な変更を設計し、実装せよ。`define` の新しい構文を扱うには、新しい構文手続きを実装する必要がある。また、`eval` や `apply` が引数をどういうタイミングで遅延化するかを決定できるようにし、その決定に従って引数を強制したり遅延化したりできるようにしなければならない。強制についてメモ化するかどうかを適切に扱えるようにする必要がある。

4.2.3 遅延化リストとしてのストリーム

Section 3.5.1では、遅延化されたリストとしてストリームを実装するやり方を示しました。そのときは、特殊形式 `delay` と `cons-stream` を導入し、ストリームの `cdr` を計算するという “promise”(約束) を、実際の実行を先延ばしにしながら構築することができるようになります。評価プロセスについてコントロールしたいことが増えたときは、この特殊形式を導入するという一般的なやり方を使うことができます。しかし、このやり方は不自然です。問題のひとつとして、特殊形式が手続きのような第一級オブジェクトではないので、高階手続きと組み合わせて使えないということがあります。³⁹ そのうえ、リストに似

³⁹ これはまさしく、Exercise 4.26の `unless` で起こった問題です。

ているもののリストと同一ではない新しい種類のデータオブジェクトとしてストリームを作らざるを得ず、そのことによって通常のリスト演算の多く(`map`, `append`など)をストリーム用に再実装しなければならないことになりました。

遅延評価があれば、ストリームとリストは同じものにできます。そうすると、特殊形式を作ったり、リストとストリームに別々の演算を用意したりする必要はなくなります。そのためには、`cons` が非厳密になるようにできれば十分です。これを実現するひとつの方法は、遅延評価器を拡張して非厳密な基本手続きを許すようにして、そのひとつとして`cons` を実装するということです。もっと簡単な方法は、`cons` を基本手続きとして実装する根本的な必要性はない(Section 2.1.3) ということを思い出して、ペアを手続きとして表現するということです。⁴⁰

```
(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)))
```

これらの基本演算を使うと、リスト演算の標準定義は有限リストだけでなく無限リスト(ストリーム)に対しても動作するようになります。ストリーム演算はリスト演算として実装できることになります。以下にいくつかの例を示します。

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items)) (map proc (cdr items)))))
(define (scale-list items factor)
  (map (lambda (x) (* x factor)) items))
(define (add-lists list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
```

⁴⁰この手続きによる表現はExercise 2.4で説明したものです。本質的には、どんな手続きによる表現(例えば、メッセージパッキングによる実装)を使っても問題はありません。これらの定義は、ドライバーランプに入力すれば遅延評価器に組み込むことができます。元々`cons`, `car`, `cdr` をグローバル環境の基本手続きに入れていたとしたら、それらは再定義されることになります(Exercise 4.33とExercise 4.34も参照してください)。

```

(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るということに注意してください。この場合、リストの `cdr` だけでなく、`car` も同様に遅延化されています。⁴¹ 実際のところ、遅延ペアの `car` や `cdr` にアクセスする場合であっても、リスト要素の値を強制する必要はありません。値は、本当に必要になったときになって—例えば、基本手続きの引数になる場合や答えとして表示される場合など—初めて強制されることになります。

遅延ペアは、Section 3.5.4でストリームを扱った際に起こったような問題を解決するのにも役に立ちます。そのときの問題は、ループのあるシステムのストリームモデルを定式化するには、`cons-stream` によって提供されるもの以外にも、`delay` を明示的にプログラムの中にちりばめないといけないということでした。遅延評価では、手続きのすべての引数は統一的に遅延化されます。例えば、リストを積分して微分方程式を解く手続きを、Section 3.5.4で元々やりたかったように実装することができます。

```

(define (integral integrand initial-value dt)
  (define int
    (cons initial-value
          (add-lists (scale-list integrand dt) int)))
  int)
(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (map f y))
  y)
;;; L-Eval input:
(list-ref (solve (lambda (x) x) 1 0.001) 1000)

```

⁴¹このことによって、列に限らず、より一般的なリスト構造の遅延化版を作れるようになっています。Hughes 1990では、“遅延木”的いくつかの応用について検討しています。

```
;;; L-Eval value:  
2.716924
```

Exercise 4.32: Chapter 3のストリームと、この節で説明した“より遅延化された”遅延リストとの間の違いを示す例をいくつか挙げよ。この遅延度の高さを利用するにはどのようにすればいいだろうか。

Exercise 4.33: Ben Bitdiddle は、次の式を評価することで、上記の遅延リストの実装をテストした。

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

驚いたことに、これはエラーになった。少し考えて、彼はクオート式を読み込むことによって得られる“リスト”は `cons`, `car`, `cdr` の新しい定義によって操作されるリストとは違うということに気がついた。評価器のクオート式の扱いを修正し、ドライバループに入力されたクオート式のリストが真の遅延リストを生成するようにせよ。

Exercise 4.34: 評価器のドライバループを修正し、遅延ペアと遅延リストが何らかの方式で適切に表示されるようにせよ（無限リストについてはどうするのがよいだろうか？）。評価器が表示する際に遅延ペアを識別できるように、遅延ペアの表現も修正する必要があるかもしれない。

4.3 Scheme の変異版 — 非決定性計算

この節では、Scheme 評価器を拡張して、*nondeterministic computing*（非決定性計算）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にします。そのために、自動探索をサポートする機能を評価器に組み込みます。これは、Section 4.2での遅延評価の導入に比べ、ずっと深い言語に対する変更になります。

非決定性計算は、ストリーム処理と同じように、“生成してテストする”応用に役に立ちます。例えば、二つの正の整数のリストから始めて、和が素数となる整数ペア（ひとつは一番目のリストから取ったもの、ひとつは二番目のリストから取ったもの）を探すというタスクについて考えてみましょう。有限列

の演算によってこの問題を扱う方法についてはSection 2.2.3で、無限ストリームを使う方法についてはSection 3.5.3で見てきました。やり方は、すべての可能なペアの列を生成し、それらをフィルタして和が素数になるペアを選択するというものでした。Chapter 2のように実際にペアの列全体をまず生成するか、Chapter 3のように生成とフィルタリングを交互に配置するかというところは、計算がどのように構成されるかという本質的なイメージに比べると些細な問題です。

非決定性のアプローチは、違うイメージになります。単純に一番目のリストから数値を(何らかの方法で)ひとつ選び、二つ目のリストからも数値をひとつ選び、それらの和が素数であるよう(何らかの仕組みによって)要求するというものを想像してみてください。これは次の手続きによって表現できます。

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))
```

この手続きは、問題をただ書き直しただけで、解き方を規定しているようには見えないかもしれません。しかしながら、これはちゃんとした非決定性プログラムです。⁴²

ここで鍵となる考え方は、非決定性言語の式は二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を返す可能性もあります。ここでの非決定性プログラム評価器は、自動的に可能な値を選択し、その選択を覚えておくことによって動作します。後続の要求が満たされなければ、評価器は別の選択を試します。そして、評価が成功するか、または選ぶものがなくなるまで、新しい選択を試し続けることになります。遅延評価器によってプログラマが値を遅延化したり強制したりする方法の細かいところを考えないでもよくなったように、非決定性プログラム評価器の場合は、プログラマは選ぶ方法の細かいところを考えないでもいいようになります。

⁴² ここでは、数値が素数かどうかテストする手続き `prime?` をすでに定義してあるとしています。`prime?` が定義されているとしても、`prime-sum-pair` 手続きは、Section 1.1.7 の最初で記述した、平方根関数を定義しようとした役に立たない “Lisp もどき” に怪しいぐらいそっくりに見えるかもしれません。実は、あの考え方による平方根手続きは、実際に非決定性プログラムとして定式化できます。探索メカニズムを評価器に組み込むことで、純粹に宣言的な記述と、答えを計算する方法の命令的な仕様との間の区別は曖昧になってきます。Section 4.4 では、この路線をさらに進めていきます。

非決定性評価とストリーム処理がそれぞれ連想させる、時間についての異なるイメージを比べると考えを深める役に立ちます。ストリーム処理は、可能な答えのストリームが組み立てられる時間と実際のストリームの要素が生成される時間とを切り離すために、遅延評価を使います。評価器は、すべての可能な答えが時間のない列として目の前に並べられているという幻想を支えています。非決定性評価の場合、式は可能な世界の集合の探索を表現しています。それぞれの可能世界は、ある選択の集合によって決められたものです。非決定性プログラム評価器は、時間は分岐していて、プログラムは異なる可能な実行履歴を持っているという幻想を支えています。行き止まりにたどり着いた場合は、前の選択地点に戻って異なる分岐に沿って進むことができます。

以下で実装する非決定性プログラム評価器は、`amb` と呼ばれる新しい特殊形式に基づいているため、`amb` 評価器と呼ばれれます。上記の `prime-sum-pair` の定義を (`prime?`, `an-element-of`, `require` の定義とともに) `amb` 評価器のドライバループに入力すると、次のように手続きを実行できます。

```
;; 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、条件に合う選択が行われるまで評価器がそれぞれのリストから要素を選択し続けた結果として得られたものです。

Section 4.3.1では `amb` を導入し、どのようにしてそれが評価器の自動探索メカニズムを通じて非決定性を支えているかを説明します。Section 4.3.2では非決定性プログラムの例を示し、Section 4.3.3では通常の Scheme 評価器を修正して `amb` を実装する方法の詳細を示します。

4.3.1 `amb` と探索

Scheme を拡張して非決定性をサポートするようにするために、`amb` と呼ばれる新しい特殊形式を導入します。⁴³

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

⁴³ 非決定性プログラミングのための `amb` という考え方方は、1961 年に John McCarthy によって初めて書かれたものです (McCarthy 1963 参照)。

この式は、 n 個の式 $\langle e_i \rangle$ のうちひとつの値を“曖昧に”返します。例えば、次の式は

```
(list (amb 1 2 3) (amb 'a 'b))
```

次の 6 つの可能な値を持ちます。

```
(1 a) (1 b) (2 a) (2 b) (3 a) (3 b)
```

選択対象がひとつの amb は通常の（ひとつの）値を生成します。

選択対象のない amb—式 (amb)—は、受け入れ可能な値のない式です。操作的には、(amb) というものを、評価すると計算を“失敗”させる式と考えることができます。計算は異常終了し、値は生成されません。この考え方を使うと、ある特定の述語式 p が真でなければならないことを次のように表現できます。

```
(define (require p) (if (not p) (amb)))
```

amb と require があれば、上で使った an-element-of を実装できます。

```
(define (an-element-of items)
  (require (not (null? items)))
  (amb (car items) (an-element-of (cdr items))))
```

an-element-of はリストが空であれば失敗します。空でなければ、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かのどちらかを曖昧に返します。

無限の範囲を持つ選択を表現することもできます。次の手続きは、ある与えられた n 以上の任意の整数を潜在的に返します。

```
(define (an-integer-starting-from n)
  (amb n (an-integer-starting-from (+ n 1))))
```

これは、Section 3.5.2 で説明したストリーム手続き integers-starting-from に似ていますが、ひとつ重要な違いがあります。ストリーム処理は n で始まるすべての整数の列を表すオブジェクトを返すのに対し、amb 手続きはひとつの整数を返します。⁴⁴

⁴⁴ 実際は、非決定的にひとつの選択を返すこととすべての選択を返すこととの違いは、見方によるところがあります。値を使うコードの視点からは、非決定的な選択はひとつの値を返すものです。コードを設計するプログラマの視点からは、非決定的な選択とは、すべての可能な値を潜在的に返し、そこから計算が分岐して、すべての値が別々に調査されるというものです。

抽象的には、amb式を評価すると時間の分岐が起こり、それぞれの分岐ではその式の可能な値のひとつによって計算が進むと考えることができます。ambは *nondeterministic choice point*(非決定性選択点)を表現しているという言い方をします。動的割り当ての可能な十分な数のプロセッサのあるコンピュータを持っていたとしたら、探索を素直なやり方で実装できるところです。amb式にぶつかるまでは、実行は逐次的な機械と同じように進みます。amb式にぶつかった時点で、より多くのプロセッサが割り当て・初期化され、その選択の示すすべての並列実行が続けられます。それぞれのプロセッサは、失敗によって終了させられたり、さらに分岐したり、終了したりするまで、その選択が唯一の選択であったかのように計算を続けます。⁴⁵

一方、もしひとつつのプロセス(または、少しの並行プロセス)しか実行できない機械を持っている場合は、別のやり方を逐次的なやり方で考える必要があります。例えば、評価器を修正して、選択点にぶつかるたびにランダムな分岐を選ぶようにすることも考えられるでしょう。しかし、ランダムに選ぶと、失敗する値になりやすくなります。評価器を何度も何度も動かして、ランダムな選択をし続けて、失敗しない値を見つけるのを願うという方法もありますが、それよりはすべての可能な実行パスを *systematically search*(体系的に探索)したほうがよいでしょう。この節で開発し、実験を行う amb 評価器は、体系的探索を次のように実装します。評価器が amb の適用にぶつかると、最初は一番目の選択肢を選びます。この選択は、先でさらに選択肢にぶつかるかもしれません。評価器は、どの選択点でも常に一番目の選択肢を選びます。もある選択が失敗に終われば、評価器はオートマジックに⁴⁶直近の選択点に backtrack(バックトラック)し、次の選択肢を試します。ある選択点で残っている選択肢がなくなったら、評価器はその前の選択点に戻ってそこから再開します。この処理は、*depth-first search*(深さ優先探索)や *chronological backtracking*(時間的バ

⁴⁵これは絶望的に非効率な仕組みだと反対する人がいるかもしれません。このやり方では、この方式で記述された簡単な問題を解くだけでも何百万ものプロセッサが必要になります、ほとんどの時間ほとんどのプロセッサはアイドル状態になるんじゃないのか、という意見です。この反対意見は歴史的な文脈の中で扱われるべきです。以前はメモリは非常に効果な商品だと考えられていました。1964年には、1メガバイトのRAMが40万ドルもしたのです。今では、PCはすべて何メガバイトものRAMを搭載していて、ほとんどの時間ほとんどのメモリは使われていません。大量生産される電子製品のコストは過大評価してしまいがちなのです。

⁴⁶automagically: “自動的に、しかし何らかの理由で(一般的には複雑すぎたり、汚すぎたり、時には明白すぎたりして)話者が説明する気にならないようなやり方で” (Steele et al. 1983, Raymond 1993)

ックトラック)として知られる探索戦略に通じます。⁴⁷

ドライバループ

`amb` 評価器のドライバループには、変わった性質がいくつかあります。このドライバループは、式を読み込み、上の `prime-sum-pair` の例のように、最初の失敗しなかった実行の値を表示します。次の成功した実行の値を見たければ、バックトラックして二番目の失敗しなかった実行を生成するよう試みることをインタプリタに依頼します。これは記号 `try-again` を入力することによって伝えます。`try-again` 以外の任意の式が与えられると、インタプリタは前の問題の調べていない選択肢を捨て、新しい問題を開始します。以下に対話例を

⁴⁷自動探索戦略をプログラミング言語に統合するということには、長い波瀾万丈の歴史があります。探索と自動バックトラックによって非決定性アルゴリズムをエレガントなやり方でプログラミング言語に組み込むことができるのではないかという考えは、Robert Floyd (1967) が初めて提案したものです。Carl Hewitt (1969) は、自動的時間的バックトラックを明示的にサポートし、組み込みの深さ優先探索戦略を持つ Planner というプログラミング言語を発明しました。Sussman et al. (1971) はこの言語のサブセットとなる MicroPlanner と呼ばれる言語を実装し、これは問題解決とロボットの計画の仕事をサポートするのに使われました。また、論理学や定理証明といった分野からも似たような考えが出てきて、エディンバラとマルセイユで Prolog というエレガントな言語が生み出されることになります (Prolog については Section 4.4 で検討します)。その後、自動探索に対して多くの不満が出てきた後に、McDermott and Sussman (1972) が Conniver という言語を開発しました。この言語は、探索戦略をプログラマのコントロール下に置くための仕組みを備えていました。しかし、これは扱いにくいということがわかり、Sussman and Stallman 1975 が電子回路の記号分析の方法を調査している間により扱いやすいアプローチを発見しました。彼らは、事実と事実をつなげる論理的依存性を追跡することによる非時間的バックトラック計画を開発しました。この技術は、dependency-directed backtracking (依存主導バックトラック) として知られるようになりました。この手法は複雑でしたが、冗長な探索がほとんどないため、それなりに効率のよいプログラムを生成しました。Doyle (1979) と McAllester (1978; 1980) は Stallman と Sussman の手法を一般化・明確化し、探索を定式化する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。現代的な問題解決システムはすべて、何らかの形で真理維持システムを基層として利用しています。真理維持システムや真理維持を使ったアプリケーションを構築するエレガントな方法についての議論は、Forbus and deKleer 1993 を参照してください。Zabih et al. 1987 は、`amb` に基づく Scheme の非決定性拡張について記述しています。これはこの節で説明しているインタプリタに似ていますが、時間的バックトラックではなく依存主導バックトラックを使っているため、より高度なものとなっています。Winston 1992 には、これら二種類のバックトラック両方についての紹介があります。

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 つの与えられた限界の間の整数を返す手続き `an-integer-between` を書け。これはピタゴラス数、つまり与えられた範囲の間の整数の三つ組(i, j, k)で $i \leq j$ かつ $i^2 + j^2 = k^2$ となるものを求める手続きを実装するのに使える。

```
(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、探索対象の整数の大きさに上限を設けることなく“すべての”ピタゴラス数のストリームを生成する方法を検討した。an-integer-betweenを単純にExercise 4.35の手続き内のan-integer-starting-fromで置き換えるというやり方が、任意のピタゴラス数を生成する方法としてなぜ適切でないのか説明せよ。これを実際に達成する手続きを書け(つまり、try-againを繰り返し入力すると原理的にはすべてのピタゴラス数を生成することになる手続きを書け)。

Exercise 4.37: Ben Bitdiddleは、ピタゴラス数を生成する以下の手法がExercise 4.35のものよりも効率的だと主張している。彼は正しいだろうか。(ヒント：探索すべき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では、amb評価器の実装について記述します。しかし、まずはそれをどう使うかについての例をいくつか挙げます。非決定性プログラミングの利点は、探索の実行方法の詳細を隠すことによって、プログラムをより高い抽象レベルで表現できるということです。

論理パズル

次のパズル (Dinesman 1968から取ったもの) は、単純な論理パズルという大きなジャンルの中の典型的なものです。

Baker, Cooper, Fletcher, Miller, Smith は 5 階しかないアパートのそれぞれ別々の階に住んでいます。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 のすぐ上やすぐ下の階には住んでいません。Fletcher は Cooper のすぐ上やすぐ下の階には住んでいません。みんなはそれぞれどの階に住んでいるのでしょうか。

すべての可能性を列挙してから与えられた制約を課すことによって、誰がどの階に住んでいるかを素直なやり方で判断することができます。

48

```
(define (multiple-dwelling)
  (let ((baker      (amb 1 2 3 4 5)) (cooper (amb 1 2 3 4 5))
        (fletcher   (amb 1 2 3 4 5)) (miller  (amb 1 2 3 4 5))
        (smith      (amb 1 2 3 4 5)))
    (require
     (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)          (list 'cooper cooper)
          (list 'fletcher fletcher) (list 'miller miller)
          (list 'smith smith))))
```

48 このプログラムでは、リストの要素がそれぞれ別々かどうかを判断するのに次の手続きを使っています。

```
(define (distinct? items)
  (cond ((null? items) true)
        ((null? (cdr items)) true)
        ((member (car items) (cdr items)) false)
        (else (distinct? (cdr items)))))
```

`member` は `memq` と同じようなものですが、等価性のテストに `eq?` ではなく `equal?` を使うところが違います。

式 (multiple-dwelling) を評価すると、次の結果が生成されます。

```
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
```

この簡単な手続きはうまくいくのですが、非常に時間がかかります。Exercise 4.39と Exercise 4.40では、考えられるいくつかの改善方法について検討します。

Exercise 4.38: multiple-dwelling 手続きを修正し、Smith と Fletcher がすぐ上下の階に住んでいないという制約を取り除け。この変更したパズルにはいくつの解があるだろうか。

Exercise 4.39: multiple-dwelling 手続き内の制約の順番は解に影響するだろうか。答えを見つけるのにかかる時間には影響するだろうか。もし順番が関係あると考えるのであれば、与えられた問題の制約の順番を変えることによって得られるより速いプログラムを示せ。もし関係がないと考えるのであれば、そのことを論じよ。

Exercise 4.40: 複数居住問題で、各人を各階に割り当てるやり方は、階の割り当てがそれぞれ別々でなければならないという制約を設定する前と後で、それぞれ何組あるだろうか。各人を各階に割り当てるすべての可能なやり方を生成してからバックトラックにそれらを排除させるのは、非常に非効率的だ。例えば、ほとんどの制約は、一つか二つの人と階の変数にしか依存しないので、すべての人に対して階が選択される前にその制約を課すことができる。それまでの制約によってすでに排除されていない可能性だけを生成することによってこの問題を解く、ずっと効率のよい非決定性手続きを書き、実演せよ。

Exercise 4.41: 複数居住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より) を解け。

5人の女子生徒が試験のために座っている。彼女らは、両親が結果に過度の関心を持っていると考えている。そのため彼女らは、試験の結果について家に手紙を送る際に、ひとつ本当のことを書き、ひとつ嘘のことを書くよう申し合わせた。以下に、彼女らの手紙のそれに関連する部分を示す。

- Betty: “試験は Kitty が 2 位。私は残念ながら 3 位。”
- Ethel: “喜んで、私トップ取ったよ。Joan が 2 位。”

- Joan: “私は 3 位で、かわいそうな Ethel は最下位。”
- Kitty: “私は 2 位。Mary は 4 位しか取れなかつたんだ。”
- Mary: “私は 4 位。トップは Betty だつたよ。”

実際には、5 人の女の子の成績はどういう順番だったのだろうか。

Exercise 4.43: amb 評価器を使って次のパズルを解け。⁴⁹

Mary Ann Moore の父親と彼の四人の友達の Downing 大佐、Hall 氏、Barnacle Hood 倩、Parker 博士は、それぞれ一台ずつクルーザーを持っていて。五人にはそれぞれ娘が一人いて、自分のクルーザーにはほかの誰かの娘の名前をついている。Barnacle 倩のクルーザーは Gabrielle で、Moore 氏のは Lorna、Hall 氏のは Rosalind という名前だ。Downing 大佐のクルーザー Melissa は Barnacle 倩の娘の名前をつけたものだ。Gabrielle の父親は、Dr. Parker の娘の名前をつけたクルーザーを持っている。Lorna の父親は誰か。

効率よく動くプログラムを書くよう努めること (Exercise 4.40 参照)。また、もし Mary Ann の姓が Moore だと知らされていない場合はいくつの解が存在するだろうか。

Exercise 4.44: Exercise 2.42では、“8 クイーンパズル”という、どの 2 つも互いに利き筋に入らないように 8 つのクイーンをチェス盤に置くパズルについて記述した。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、普通はまずその入力を *parse(構文解析)* することを試みます。構文解析というのは、入力を何らかの文法構造に当てはめることです。例えば、“The cat eats.” のような冠詞・名詞・動詞からなる簡単な文を認識しようとしているといいます。そのような解析を達成するには、個々の単語の品詞を判定できる必要があります。まずは、単語を分類する数個のリストを作つてみましょう。⁵⁰

⁴⁹ これは 1960 年代に Litton Industries から出版された“問題のレクリエーション”という本から取つたものです。これは Kansas State Engineer によるものとされています。

⁵⁰ ここでは、各リストの最初の要素がリストの残りの単語の品詞を示すという仕様にしています。

```
(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。文法とは、より単純な要素から文法要素を組み立てる方法を記述した規則の集合です。非常に単純な文法として、文は常に二つの要素—名詞句とそれに続く動詞—にからなり、名詞句は定冠詞とそれに続く名詞からなると規定するようなものが考えられます。この文法では、“The cat eats.” という文は次のように構文解析されます。

```
(sentence (noun-phrase (article the) (noun cat))
          (verb eats))
```

それぞれの文法规則に対して別々の手続きを持つ単純なプログラムによって、このような解析結果を生成することができます。文を解析するには、二つの構成要素を識別し、それら二要素のリストに記号 `sentence` のタグをつけたものを返します。

```
(define (parse-sentence)
  (list 'sentence
        (parse-noun-phrase)
        (parse-word verbs)))
```

名詞句も同じように、定冠詞とそれに続く名詞を見つけることによって解析します。

```
(define (parse-noun-phrase)
  (list 'noun-phrase
        (parse-word articles)
        (parse-word nouns)))
```

一番下のレベルでは、構文解析は、要求される品詞の単語リストに次の未解析語が入っているかどうかを繰り返しチェックするということになります。これを実装するために、未解析の入力を持つグローバル変数 `*unparsed*` を保持します。単語をチェックするたびに、`*unparsed*` が空でないこと、また指定されたリストの単語から始まることを要求します。要求通りであれば、その単語を `*unparsed*` から削除し、単語とその品詞(リストの先頭にあります)と一緒に返します。⁵¹

⁵¹`parse-word` が未解析の入力リストを変更するのに `set!` を使っていることに注意してください。これうまく動作させるには、`amb` 評価器がバックトラックする際に、`set!`

```
(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)))
```

解析を始めるには、`*unparsed*` を入力全体に設定し、文の解析を試み、残っているものがないことをチェックするだけで大丈夫です。

```
(define *unparsed* '())
(define (parse input)
  (set! *unparsed* input)
  (let ((sent (parse-sentence)))
    (require (null? *unparsed*)) sent))
```

これで、構文解析器を試し、先ほどの簡単なテスト文に対してうまく動くか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))
```

`amb` 評価器は、構文解析上の制約を `require` の助けを借りて便利に表現する能够という意味で役に立っています。しかし、自動探索とバックトラックが本当に効果を上げるのは、要素を分解する方法に複数の選択肢があるようなより複雑な文法について考える場合です。

こ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”）を前置詞に名詞句が続くものとして定義します。

```
(define (parse-prepositional-phrase)
  (list 'prep-phrase
```

演算の効果を元に戻す必要があります。

```
(parse-word prepositions)
(parse-noun-phrase)))
```

文は名詞句に動詞句が続くものと定義できます。動詞句は、動詞か、または前置詞句が後ろにくついた動詞句のどちらかであるとします。⁵²

```
(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)))
```

ついでに、名詞句の定義にも手を加えて、“a cat in the class”のようなものを認めるようにします。ここまで名詞句と呼んできたものは、ここからは単純な名詞句と呼ぶことにします。そして、今後は名詞句は、単純な名詞句か、前置詞句が後ろにくついた名詞句のどちらかということになります。

```
(define (parse-simple-noun-phrase)
  (list 'simple-noun-phrase
    (parse-word articles)
    (parse-word nouns)))
(define (parse-noun-phrase)
  (define (maybe-extend noun-phrase)
    (amb noun-phrase
        (maybe-extend
          (list 'noun-phrase
            noun-phrase
            (parse-prepositional-phrase))))))
  (maybe-extend (parse-simple-noun-phrase)))
```

この新しい文法を使うと、より複雑な文を構文解析できます。例えば、次の文を解析すると、

⁵²この定義が再帰的であることに注意してください。動詞には、任意の数の前置詞句が続けられることになります。

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

以下の結果となります。

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

ある入力に対して二つ以上の合法な解析結果がある可能性があるというところに注意してください。文 “The professor lectures to the student with the cat” では、教授が猫と一緒に講義をしているのかもしれませんし、学生が猫を持っているのかもしれません。私たち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))
```

これを実行すると、以下の結果となります。

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

評価器に再試行を依頼すると、以下の結果となります。

```
(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: 上の文法では、次の文の構文解析結果には 5 通りある。“The professor lectures to the student in the class with the cat”。5 通りの解析結果を示し、それぞれのニュアンスの違いを説明せよ。

Exercise 4.46: Section 4.1 と Section 4.2 の評価器は、オペランドの評価順を決めていなかった。amb 評価器はオペランドを左から右に評価するということを今後見ていく。オペランドの評価順がそれと違うものであつたら我々の構文解析器がうまく動作しないのはなぜか説明せよ。

Exercise 4.47: Louis Reasoner は、動詞句は、動詞か、前置詞句が後ろに続く動詞句なのだから、手続き parse-verb-phrase は次のように定義するほうがずっと素直なのではないかと言う（名詞句も同じようにする）。

```
(define (parse-verb-phrase)
  (amb (parse-word verbs)
        (list 'verb-phrase
              (parse-verb-phrase)
              (parse-prepositional-phrase))))
```

これはうまく動くだろうか。amb 内の式の順番を入れ替えたら、プログラムのふるまいは変わるだろうか。

Exercise 4.48: 上の文法を拡張子、より複雑な文が扱えるようにせよ。例えば、形容詞や副詞を含むように名詞句や動詞句を拡張し

てもよい。または、重文を取り扱えるようにしてもよい。⁵³

Exercise 4.49: Alyssa P. Hacker は、文を構文解析するより、面白い文を生成することに興味がある。彼女は、手続き `parse-word` を変更して“入力文”を無視するようにして、常に成功して適切な単語を生成するようにするだけで、構文解析のために作ったこのプログラムを使って文の生成ができるのではないかと考えた。Alyssa の考えを実装し、生成された最初の数個の文を示せ。⁵⁴

4.3.3 amb 評価器の実装

通常の Scheme の式を評価すると、値を返すか、ずっと停止しないか、エラーを発生させるかのどちらかになります。非決定性 Scheme では、式を評価すると、それに加えて行き止まりの発見という結果になることもあります。その場合、評価は前の選択点にバックトラックしなければならないことになります。非決定性 Scheme の解釈は、このケースが加わるため、複雑になります。

非決定性 Scheme のための `amb` 評価器は、[Section 4.1.7](#) の解析評価器を修正して構築することにします。⁵⁵ 解析評価器と同じように、式の評価は、式の解析によって生成される実行手続きを呼ぶことによって達成されます。通常の Scheme の解釈と非決定性 Scheme の解釈の違うところは、完全に実行手続きの中だけに存在します。

⁵³ この種の文法は、いくらでも複雑なものにすることができます。しかし、実際の言語理解を考えると、このような文法はおもちゃにすぎません。実際の自然言語をコンピュータで理解するためには、構文解析と意味解釈を巧妙に組み合わせる必要があります。一方、おもちゃの構文解析器であっても、情報検索システムのようなプログラムの柔軟なコマンド言語をサポートするのには役に立ちます。[Winston 1992](#)では、コンピュータによる自然言語理解のアプローチと、簡単な文法のコマンド言語への応用について論じています。

⁵⁴ Alyssa のアイデアはちゃんと動きます（また、驚くほど単純です）が、生成される文はちょっと退屈です—生成される文は、この言語で可能な文の中から興味深いものをサンプリングしたものにはなっていないからです。実際のところ、この文法はいろいろなところでかなり再帰的で、Alyssa のやり方ではそれらの再帰のひとつに“はまって”抜け出せなくなってしまうのです。これに対処するひとつの方法については、[Exercise 4.50](#) を参照してください。

⁵⁵ [Section 4.2](#) の遅延評価器を実装したときは、[Section 4.1.1](#) の通常のメタ循環評価器を修正するよう選択しました。それに対して、`amb` 評価器は[Section 4.1.7](#) の解析評価器をベースにすることにします。この評価器内の実行手続きは、バックトラックを実装するのに便利なフレームワークとなっているからです。

実行手続きと継続

通常の評価器の実行手続きは、実行環境というひとつの引数を取るようになっていました。それに対して、amb評価器の実行手続きは三つの引数を取るようになっています。環境と、*continuation procedures*(継続手続き)と呼ばれる二つの手続きです。式の評価は、これらの二つの継続のうちひとつを呼んで終わります。評価の結果が値になったら、*success continuation*(成功継続)がその値で呼ばれます。評価の結果が行き止まりの発見になったら、*failure continuation*(失敗継続)が呼ばれます。継続を適切に構築して呼び出すことが、非決定性評価器がバックトラックを実装する仕組みになっています。

値を受け取り計算を続行するのが成功継続の仕事です。成功継続には、値のほかにまた別の失敗継続が渡されます。これは、値を使った結果が行き止まりになった場合に呼ばれることになります。

非決定性処理のほか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、式が分岐の選択を表現しうるというところにあります。そのような式を評価すると、どの選択肢が受け入れられる結果になるか前もってわからなくて、示された選択肢のひとつに進まなければなりません。これに対処するために、評価器はひとつの選択肢を選び、その値を成功継続に渡します。この値とともに、評価器は失敗継続を構築して渡します。失敗継続は、後で呼ぶことによって別の選択肢を選ぶことができます。

評価の間にユーザプログラムが現在の試行を明示的に拒絶した場合(例えば、`require`を呼び出すと(amb)という常に失敗する式が実行されることになります—[Section 4.3.1](#)参照)、失敗がトリガーされることになります。その時点で手元にある失敗継続が、直近の選択点にほかの選択肢を選ばせます。その選択点に考慮すべき選択肢が残っていなければ、それより前の選択点の失敗がトリガーされ…という繰り返しになります。失敗継続は、`try-again`要求に応じて、式のほかの値を見つけるためにドライバーループから呼び出されることがあります。

それに加えて、選択の結果としての処理の分岐で(変数への代入のような)副作用のある演算が起こった場合、処理が行き止まりを見つけたときに、新しい選択をする前に副作用を取り消す必要があるかもしれません。これは、副作用を持つ演算に、副作用を取り消し失敗を伝播させる失敗継続を生成することによって達成できます。

まとめると、失敗継続は以下のように構築されることになります。

- amb式—amb式によって行われた現在の選択が行き止まりに当たったとき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—選択肢を使い果たしたとき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。

失敗が起動されるのは、行き止まりにぶつかった場合だけです。これが起こるのは、以下のような場合です。

- ユーザプログラムが (`amb`) を実行する。
- ユーザがトップレベルドライバで `try-again` を入力する。

失敗継続は失敗を処理する間にも呼ばれます。

- 代入によって作られた失敗継続が副作用の取り消しを終えると、割り込んだ失敗継続を呼び出し、この代入に到った選択点からトップレベルまで失敗を伝播させます。
- ある `amb` の失敗継続が選択肢を使い切ると、その `amb` に元々渡されていた失敗継続を呼び出し、前のチェックポイントからトップレベルまで失敗を伝播させます。

評価器の構造

`amb` 評価器の文法とデータを表現する手続きと、基本的な `analyze` 手続きは、Section 4.1.7 の評価器のものとまったく同じものです。違うのは、特殊形式 `amb` を認識できるように追加の構文手続きが必要になるということだけです。⁵⁶

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

`analyze` のディスパッチにも、この特殊形式を認識して適切な実行手続きを生成する節を追加する必要があります。

```
((amb? exp) (analyze-amb exp))
```

トップレベル手続き `amb eval` (Section 4.1.7 版の `eval` に似たもの) は、与えられた式を解析し、与えられた環境と二つの与えられた継続に結果となる実行手続きを適用します。

⁵⁶評価器は、非決定性プログラムで使った `let` (Exercise 4.22 参照) をサポートしていると想定しています。

```
(define (ambeval exp env succeed fail)
  ((analyze exp) env succeed fail))
```

成功継続は 2 引数の手続きです。引数のひとつはちょうど得られたばかりの値で、もうひとつは値が失敗に終わった場合に使うことになるまた別の失敗継続です。失敗継続は無引数の手続きです。つまり、実行手続きの一般的な形は次のようにになります。

```
(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))
```

与えられた式の評価を試み、式の値（評価が成功した場合）か記号 failed（評価が失敗した場合）のどちらかを返します。下に示す、ドライバループでの ambeval の呼び出しは、ループを継続し try-again 要求をサポートするずっと複雑な継続手続きを使います。

amb 評価器の複雑なところは大部分、複数の実行手続きがお互いを呼び出す間、継続を持ち回らないといけないところから来るものです。以下のコードを読むときには、それぞれの実行手続きを [Section 4.1.7](#) の通常の評価器でそれに対応する手続きと比較してください。

単純な式

一番単純なタイプの式の実行手続きは、本質的には通常の評価器のものと同じになりますが、違うのは継続を管理する必要があるというところです。実行手続きは単純に成功し、その式の値と、それとともに自分に渡された失敗継続を返します。

```
(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))))

```

変数の検索はいつも‘成功’するというところに注意してください。`lookup-variable-value`が変数を見つけるのに失敗すると、通常通りエラーを発生させます。そのような“失敗”は、プログラムのバグ—未束縛変数への参照—を示しています。これは、いま試している非決定性選択肢の代わりに別の選択肢を試すべきだと示しているわけではありません。

条件文と列

条件文も、通常の評価器の場合と同じように扱われます。`analyze-if`によって生成される実行手続きは、述語実行手続きの`pproc`に成功継続を渡して起動します。この成功継続は、述語の値が真かどうかをチェックし、それに応じて結果部または代替部の実行に移ります。`pproc`の実行が失敗した場合は、`if`式に対する元の失敗継続が呼ばれます。

```

(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp))))
    (aprocs (analyze (if-alternative exp)))))
  (lambda (env succeed fail)
    (pproc env
           ;; pred-valueを得るための
           ;; 述語の評価に対する成功継続

```

```

(lambda (pred-value fail2)
  (if (true? pred-value)
      (cproc env succeed fail2)
      (aproc env succeed fail2)))
;; 述語の評価に対する失敗継続
fail)))

```

列も、これまでの評価器と同じように扱われます。違うのは、継続を渡すために必要となる内部手続き `sequentially` の中の仕掛けだけです。その仕掛けというのは、`a, b` と順番に実行していくときに、`b` を呼び出す成功継続を渡して `a` を呼び出すというものです。

```

(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail)
      (a env
        ;; a の呼び出しの成功継続
        (lambda (a-value fail2)
          (b env succeed fail2))
        ;; a の呼び出しの失敗継続
        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

定義の場合も、継続の管理にひと手間かけなければいけません。これは、実際に新しい変数を定義する前に、定義の値の式を評価する必要があるからです。これを達成するために、定義の値の実行手続き `vproc` が環境、成功継続、

失敗継続とともに呼び出されます。vproc の実行が成功して、定義される変数の値 val が得られたら、その変数が定義され、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))))
```

代入はもっと面白くなります。ここで初めて、継続を持ち回るだけではなく、ちゃんと利用することになります。代入の実行手続きは、最初のほうは定義の実行手続きと似ています。まず、変数に代入する新しい値を得ることを試みます。この vproc の評価が失敗した場合は、代入は失敗します。

しかし、vproc が成功し、代入の実行に進む場合は、この計算の分岐が後になってから失敗する可能性について考えなければなりません。その場合、その代入から抜け出してバックトラックすることが求められます。そのため、バックトラック処理の一部として、その代入を取り消す準備をしておく必要があります。⁵⁷

これは、新しい値を変数に代入して先に進む前に、古い値を保存する成功継続（下のコメント “*1*” の箇所）を vproc に渡すことによって実現します。代入する値とともに渡された失敗継続（下のコメント “*2*” の箇所）は、失敗を継続する前に、変数の古い値を復元します。つまり、成功する代入は、それに続く失敗に割り込む失敗継続を提供するということです。fail2 を呼んでいたはずの失敗はこの手続きを代わりに呼び、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)
      ...)))
```

⁵⁷ 定義の場合、取り消すことについては考えませんでした。内部定義は、走査によって消去されると想定できるためです。（Section 4.1.6）

```
(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))))
```

手続きの適用

適用の実行手続きには新しい考え方はありませんが、違うのは継続を管理することの技術的な複雑さという点です。この複雑さは、`analyze-application`の中で、オペランドを評価する際に成功継続と失敗継続を管理しないといけないというところから来ています。オペランドのリストの評価には、通常の評価器のように単純な `map` を使うのではなく、手続き `get-args` を使います。

```
(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))
          fail2)))))
```

`get-args` の中では、リスト中のそれぞれの `aproc` を、再帰的に `get-args` を呼ぶ成功継続とともに呼ぶことによって、`aproc` 実行手続きリストの `cdr` ダウンと、結果の `args` のリストを `cons` アップを達成していることに注意してください

い。これらの再帰的な `get-args` の呼び出しはそれぞれ、集積された引数リストに新しく得られた引数を `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 の成功継続
       (lambda (arg fail2)
         (get-args
          (cdr aprocs)
          env
          ;; get-args の再帰的呼び出しのための
          ;; 成功継続
          (lambda (args fail3)
            (succeed (cons arg args) fail3))
          fail2))
       fail)))
```

`execute-application` によって実行される実際の手続き適用は、継続を管理する必要があるということ以外は、通常の評価器と同じように達成されます。

```
(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

特殊形式 `amb` は、非決定性言語の鍵となる要素です。ここでは、解釈処理の本質と、継続を管理する理由について学ぶことになります。`amb` の実行手続きは、`amb` 式のすべての可能な値に対する実行手続き巡回する `try-next` というループを定義します。それぞれの実行手続きは、次の実行手続きを試す失敗継続とともに呼ばれます。試行する選択肢がなくなったら、`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))))
```

ドライバーループ

`amb` 評価器のドライバーループは、式の評価を再試行をユーザに許す仕組みのために複雑になっています。ドライバは、引数として `try-again` という手続きを取る `internal-loop` と呼ばれる手続きを使います。これの意図するところは、`try-again` を呼び出すと、非決定性評価での次のまだ試していない選択肢に進まなければいけないということです。`internal-loop` は、ユーザがドライバーループで `try-again` と入力した場合には応答として `try-again` を呼び、それ以外の場合には `ambeval` を呼んで新しい評価を開始します。

この `ambeval` 呼び出しに対する失敗継続は、それ以上値がないということをユーザーに伝え、ドライバーループを再起動するというものです。

`ambeval` 呼び出しに対する成功継続はより難解です。ここでは、得られた値を表示してから、次の選択肢を試行することができる `try-again` 手続きを引数として内部ループを再起動します。この `next-alternative` 手続きは、成功継続に二番目の引数として渡されたものです。通常であれば、この二番目の引数は、現在の評価の分岐が後に失敗した場合に使われる失敗継続として考え

ます。しかしこの場合は、評価は成功して終わっているので、“失敗”の別の分岐を起動することによって成功する評価を追加で検索できるということになります。

```
(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 を最初に呼び出すときには、現在の問題がないというメッセージを出してドライバループを再起動する try-again 手続きを使います。こ

これは、進行中の評価がないときにユーザが `try-again` を入力したときに起こるふるまいです。

Exercise 4.50: 新しい特殊形式 `ramb` を定義せよ。これは `amb` に似ているが、次の選択肢を左から右に探索するのではなく、ランダムな順番で探索するというところが違う。これがあると Exercise 4.49 の Alyssa の問題を解くのにどう役に立つか示せ。

Exercise 4.51: `permanent-set!` という、失敗時に取り消されることのない新しい種類の代入を実装せよ。例えば、以下のように、二つの異なる要素をリストから選択し、成功する選択をするのに要した試行回数を数えることができる。

```
(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!` ではなく `set!` を使ったら、どのような値が表示されるだろうか。

Exercise 4.52: `if-fail` という、ユーザが式の失敗を捕捉することを可能にする新しい構造を実装せよ。`if-fail` は二つの式を取る。`if-fail` は最初の式を通常通りに評価し、その評価が成功したら通常通りに戻る。しかし、その評価が失敗したら、以下の例のように二つ目の式の値が返される。

```
;; 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で述べた permanent-set! と Exercise 4.52 の if-fail があるとき、以下を評価した結果はどのようなものになるだろうか。

```

(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: require が amb を使う通常の手続きとして実装できて、ユーザが非決定性プログラムの一部として定義できるということに気づいていなければ、require を特殊形式として実装する必要があつただろう。そのためには、以下の構文手続きが必要になる。

```

(define (require? exp)
  (tagged-list? exp 'require))
(define (require-predicate exp)
  (cadr exp))

```

また、analyze 内のディスパッチにも新しい節をひとつ加えなければならない。

```
((require? exp) (analyze-require exp))
```

さらに、require 式を扱う手続き analyze-require も必要となる。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、数学では宣言型(何であるか)の知識を扱うのに対し、計算機科学では命令型(どうやるか)の知識を扱うということを強調しました。実際、プログラミング言語は、特定の問題を解くためのステップバイステップの方法を示す形で知識を表現することをプログラマに求めていました。一方で高級言語は、言語の実装の一部として、規定された計算がどのように進行するかという膨大な細かい部分について心配しないでもいいようにしてくれる、かなりの量の方法論的知識を提供してくれます。

Lisp 含め、ほとんどのプログラミング言語は、数学の関数の値を計算することを中心に構成されています。式指向の言語(Lisp、Fortran、Algolなど)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できるという“言葉遊び”を利用しています。このため、多くのプログラミング言語は、単方向計算(明確に定義された入力と出力を持つ演算)の方向に強く偏っています。しかし、この偏りを和らげる、根本的に異なるプログラミング言語も存在します。Section 3.3.5では、そのような言語の例をひとつ見ました。この言語では、計算のオブジェクトは算術的制約でした。制約システムでは、演算の向きと順番はあまりはつきり規定されていません。そのため、計算を実行する際には、シ

ステムは普通の算術演算の場合よりも、より詳細な“どうやるか”の知識を提供する必要がありました。しかしこれは、命令的知識を提供するという責任からユーザが完全に自由になったということではありません。同じ制約の集合を実装する制約ネットワークは数多くあり、ユーザは数学的に等価なネットワークの集合から特定の計算を規定するのに適したネットワークを選ぶ必要があります。

Section 4.3の非決定性プログラマ評価器も、プログラミングとは一方向性関数の計算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は離れています。非決定性言語では、式は二つ以上の値を持つことがあります。その結果として計算は一価関数を扱うというより関係を扱うものとなります。論理プログラミングでは、プログラミングを関係性という視点から見ることと、*unification*（ユニフィケーション）と呼ばれる強力な記号パターンマッチングとを組み合わせて、この考えを拡張しています。⁵⁸

このアプローチは、うまくいけば、プログラムを書くとても強力な方法になります。この力の一部は、単一の“何であるか”という事実が、いくつもの“どうやるか”という構成要素を含むような複数の問題を解くのに使えるこ

⁵⁸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派生したものです。初期の定理証明プログラムにできることは非常に限られていきました。これは、可能な証明空間を網羅的に探索したことによるものです。そのような探索が実用的になったのは、1960 年代早期に *unification algorithm*（ユニフィケーションアルゴリズム）と *resolution principle*（導出原理）（Robinson 1965）の発見といった技術革新があったことによります。例えば、導出は Green and Raphael (1968)（Green 1969も参照）で推論的質問応答システムの基盤として使われています。この時期のほとんどの間、研究者は証明が存在するならその証明を見つけられることが保証されたアルゴリズムに焦点を当てていました。そのようなアルゴリズムは、制御を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制御構造と論理操作システムを融合させる可能性に気づき、それが Section 4.3.1(Footnote 4.47) で述べたような自動探索の研究につながりました。これと同じ時期に、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）を開発していました。彼はこれらのルールを表現するために Prolog というプログラミング言語を発明しました。エディンバラの Kowalski (1973; 1979) は、Prolog プログラムの実行は（線形ホーン節導出という証明テクニックを使って）定理証明として解釈できるということに気づきました。これらの二つの流れが合流して、論理プログラミング運動が起こることになります。このため、論理プログラミングの発展の功労者について考えると、フランスにはマルセイユ大学での Prolog の誕生という得点がある一方で、イギリスはエディンバラ大学での成果というところをアピールできます。MIT の人に言わせると、論理プログラミングは、Hewitt が彼の傑出した、しかし難解な博士論文で何の話をしていたのかをこれらのグループが解明しようとする中で発展したそうです。論理プログラミングの歴史については、Robinson 1983 を参照してください。

ともあるということから来ています。例として、二つのリストを引数として取り、それらの要素を結合してひとつのリストを作る `append` 演算について考えてみましょう。Lisp のような手続き型言語では、Section 2.2.1でやったように、基本的なリストコンストラクタの `cons` を使って `append` を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))))
```

この手続きは、次の二つの規則によって Lisp に翻訳したものと捉えることができます。一つ目の規則は一つ目のリストが空である場合を扱い、二つ目の規則は空でないリスト、つまり二つの部分の `cons` である場合を扱います。

- 任意のリスト y について、空リストと y を `append` すると y になる。
- 任意の u, v, y, z について、 v と y を `append` すると z になるならば、 $(\text{cons } u \ v)$ と y を `append` すると $(\text{cons } u \ z)$ になる。⁵⁹

`append` 手続きを使うと、次のような質問に答えることができます。

$(a \ b)$ と $(c \ d)$ の `append` を求めよ。

しかし、この二つの規則は、そのままで次のような質問に答えるにも十分なものです。これは、この手続きには答えられないものです。

$(a \ b)$ と `append` すると $(a \ b \ c \ d)$ になるようなリスト y を求めよ。

`append` すると $(a \ b \ c \ d)$ になるようなすべての x と y を求めよ。

論理プログラミング言語では、プログラマは上に挙げた `append` についての二つの規則を述べることによって `append`“手続き”を書きます。“どうやって”という知識はインタプリタによって自動的に提供され、この二つの規則だけを使って `append` についての三つのタイプの質問すべてに答えられるようにします。⁶⁰

⁵⁹ 規則と手続きの対応関係を見るには、手続きの x (x が空でない場合) を規則の $(\text{cons } u \ v)$ に対応させます。すると、規則の z は $(\text{cdr } x)$ と y を `append` したものに対応します。

⁶⁰ これはどうやって答えを計算するかという問題全体からユーザを解放してくれるものではないことは確かです。`append` の関係を定式化する数学的に等価な規則は数多くありますが、どちら向きの計算でもできるような有効な手段として使えるものはその一部だけです。それに加えて、“何であるか”という情報が“どうやって”答えを計算するかということについて何の手がかりにもならないこともあります。例えば、 $y^2 = x$ となる y を求めるという問題について考えてみてください。

現代の論理プログラミング言語(ここで実装しているものを含め)には、一般的な“どうやって”の手法が間違った無限ループなどの望ましくないふるまいを引き起こすという点で、本質的な欠点を抱えています。論理プログラミングは計算機科学の活発な研究領域となっています。⁶¹

この章の初めのほうでは、インタプリタの実装技術について掘り下げ、Lisp風言語のインタプリタにとって(というよりも、どんな従来型のインタプリタにとっても)本質的なものとなる要素について記述しました。ここからは、ある論理プログラミング言語のインタプリタについて考えるために、これらの考えを応用していきます。この言語は、言語で表現した*queries*(クエリ)、つまり質問を作ることによってデータベースから情報を検索するのにとても便利なので、*query language*(クエリ言語)と呼ぶことにします。クエリ言語は Lisp とはまったく違うものなのですが、この言語の記述には、これまでずっと使ってきたものと同じ一般的な枠組みがうまく使えるということをこれから見ていきます。その枠組みというのは、基本要素の集合と、単純な要素を組み合わせてより複雑な要素を作る組み合わせ手段と、複雑な要素を単純な概念的単位と捉えることを可能にするような抽象化手段です。論理プログラミング言語のインタプリタは、Lisp のような言語のインタプリタよりもかなり複雑です。それでも、私たちのクエリ言語インタプリタはSection 4.1のインタプリタにあつたものと同じ要素を多く含んでいます。具体的には、式を型によって分類する“eval”部があり、言語の抽象化の仕組み(Lisp の場合は手続きで、論理プログラミングの場合は*rules* 規則)を実装する“apply”部があります。また、中心的な役割を果たすのは、記号とそれに関係づけられた値との対応を決めるフレームデータ構造による実装です。このクエリ言語の実装のもうひとつの面白い側面は、Chapter 3で導入したストリームが大活躍するということです。

⁶¹論理プログラミングに対する関心がピークを迎えたのは 80 年代前半のことです。日本政府が論理プログラミング言語の実行に最適化された超高速コンピュータの作成を目指した野心的なプロジェクトを始めたのはこのころのことです。このようなコンピュータの速度は、通常の FLOPS (Floating-point Operations Per Second) でなく、LIPS(Logical Inferences Per Second) で測られるようになっていました。このプロジェクトは、ハードウェアとソフトウェアを計画通りに開発するという点では成功したのですが、国際的なコンピュータ業界は別の方に進んでいました。日本のこのプロジェクトについての概略的評価についてはFeigenbaum and Shrobe 1993を参照してください。論理プログラミングのコミュニティも、単純なパターンマッチング以外の技術に基づく関係プログラミングを考慮に入れる方向に動きました。そのような技術としては、Section 3.3.5の制約伝播システムで示したような数値制約を取り扱う能力のようなものがあります。

4.4.1 推論的情報検索

論理プログラミングは、情報検索のためのデータベースのインターフェイスを提供するのに優れています。この章で実装するクエリ言語は、そのように使うことを意図して設計されています。

クエリシステムが何をしているかを説明するために、このシステムを Microshaft というボストン地区の活発なハイテク企業で人事記録を管理するのに使うという例を示すことにします。この言語は人事情報に対するパターン主導のアクセスを提供し、また論理的推論を行うための一般的な規則も利用することができます。

サンプルデータベース

Microshaft の人事情報データベースは、従業員に関する *assertions*(表明)を持っています。フルタイムで働くコンピュータウィザード(達人)の Ben Bitdiddle に関する情報は次のようになっています。

```
(address (Bitdiddle Ben) (Slumerville (Ridge Road) 10))
(job (Bitdiddle Ben) (computer wizard))
(salary (Bitdiddle Ben) 60000)
```

それぞれの表明はリスト(この場合は三つ組)で、リストの要素はそれ自体がリストであることもあります。

フルタイムのウィザードとして、Ben は会社のコンピュータ部門の責任者で、プログラマ二人と技術者一人を監督しています。彼らの情報は以下の通りです。

```
(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 が監督する研修生プログラマもいます。

```
(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
(job (Reasoner Louis) (computer programmer trainee))
(salary (Reasoner Louis) 30000)
(supervisor (Reasoner Louis) (Hacker Alyssa P))
```

これらの人々は、職務 (job) 記述の第一項の単語 computer が示すように、みなコンピュータ部門に属しています。

Ben は上級社員で、監督者は会社の重役です。

```
(supervisor (Bitdiddle Ben) (Warbucks Oliver))
(address (Warbucks Oliver) (Swelliesley (Top Heap Road)))
(job (Warbucks Oliver) (administration big wheel))
(salary (Warbucks Oliver) 150000)
```

Ben の監督するコンピュータ部門のほかに、会社には会計主任とアシスタントからなる経理部門もあります。

```
(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))
```

このデータベースは、ほかの職務を持つ人に行える職務についての表明も入っています。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どちらの職務も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には研修生の代わりができます。

```
(can-do-job (computer programmer)  
            (computer programmer trainee))
```

また、次のこともよく知られています。訳注 6

```
(can-do-job (administration secretary)  
            (administration big wheel))
```

単純クエリ

このクエリ言語は、ユーザがシステムのプロンプトにクエリを書くことによって、データベースから情報を検索できるようになっています。例えば、コンピュータプログラマをすべて検索するには、次のようにします。

```
; ; ; Query input:  
(job ?x (computer programmer))
```

システムは、次のように二つの項目を返します。

```
; ; ; Query results:  
(job (Hacker Alyssa P) (computer programmer))  
(job (Fect Cy D) (computer programmer))
```

入力クエリは、ある特定の*pattern*(パターン)にマッチするデータベースのエントリを探しているということを規定しています。この例でパターンが規定しているのは、三つの項目からなるエントリで、その一つ目の項目は文字通りの記号の job、二つ目は任意のもので、三つ目は (computer programmer) という文字通りのリストというものです。マッチングリストの二つ目の項目となるこの“任意のもの”は、*pattern variable*(パターン変数) の ?x によって規定されています。

訳注 6秘書には重役の仕事ができる、つまり重役は実際のところたいした仕事をしていないという皮肉です。

ます。パターン変数の一般的な形式は、変数の名前となる記号の前に ? をつけたものです。`?`を単独でパターンに入れて “任意のもの” を表すのではなく、パターン変数に名前を規定したほうが便利な理由についてはこの先で説明します。システムは、単純クエリに対して、規定されたパターンにマッチするデータベースの全エントリを表示するという応答を返します。

パターンは複数の変数を持つことができます。例えば、以下のクエリは

```
(address ?x ?y)
```

従業員すべての住所を列挙します。

変数を持たないパターンを作ることもできます。その場合、クエリは単純にそのパターンがデータベースのエントリであるかどうかを判断します。もしであれば、マッチがひとつあることになります。なければマッチはありません。

ひとつのクエリで同じパターン変数を複数回使うこともできます。これは、同じ “任意のもの” がそれぞれの場所に現れる必要があるということを規定しています。変数が名前を持つのはこのためです。例えば、

```
(supervisor ?x ?x)
```

このクエリは、自分自身が監督者となっている人々をすべて探します（しかし、ここでのサンプルデータベースにはそのような表明はありません）。

以下のクエリは、

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

三つ目の項目が二要素のリストで、その一つ目の要素が `computer` であるすべての職務エントリにマッチします。

```
(job (Bitdiddle Ben) (computer wizard))
(job (Hacker Alyssa P) (computer programmer))
(job (Fect Cy D) (computer programmer))
(job (Tweakit Lem E) (computer technician))
```

このパターンは、次のものにはマッチ “しません”。

```
(job (Reasoner Louis) (computer programmer trainee))
```

これは、このエントリの三つ目の項目が三要素のリストであるのに対し、パターンの三つ目の項目は二要素であるべきだと規定しているためです。三つ目の項目が `computer` で始まる任意のリストとなるようにパターンを変更するには、以下のように規定します。⁶²

⁶² ここでは、Exercise 2.20で紹介したドット末尾記法を使っています。

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

例えば、

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

このクエリは次のデータにマッチします。

```
(computer programmer trainee)
```

このとき、`?type` はリスト (`programmer trainee`) になります。これは、次のデータにもマッチします。

```
(computer programmer)
```

このとき、`?type` はリスト (`programmer`) になります。また、次のデータにもマッチします。

```
(computer)
```

このとき、`?type` は空リスト () になります。

クエリ言語の簡単なクエリの処理は次のように記述できます。

- システムは、クエリパターン内の変数に対して、パターンを *satisfy* (満たす) すべての代入を探します。これはつまり、パターン変数が値によって *instantiated with* (具体化される) と結果がデータベースに存在するようなすべての値の集合を探すということです。
- システムは、パターンを満たす変数代入によるクエリパターンの具体化をすべて列挙することによってクエリに応答します。

パターンに変数がなければ、クエリはそのパターンがデータベースにあるかどうかを判断するということになります。データベースにある場合、空の代入、つまり変数に値を代入しない代入が、そのデータベースに対してパターンを満たすことになります。

Exercise 4.55: 以下の情報をデータベースから取り出す単純クエリを示せ。

1. Ben Bitdiddle が監督者となっている人すべて
2. 経理部門に属するすべての人の名前と職務
3. Slumerville に住むすべての人の名前と住所

複合クエリ

単純クエリはクエリ言語の基本演算となっています。複合演算を作るために、クエリ言語は組み合わせ手段を提供しています。クエリ言語を論理プログラミング言語といえるものにしている要素のひとつは、組み合わせ手段が論理式を作るのに使う組み合わせ手段とそっくりであるという点です。組み合わせ手段は `and`, `or`, `not` の三つです (ここでの `and`, `or`, `not` は Lisp の基本演算ではなく、クエリ言語の組み込み演算です)。

`and` を次のように使うことによって、コンピュータプログラマの全員の住所を検索することができます。

```
(and (job ?person (computer programmer))
        (address ?person ?where))
```

これの結果は、次のような出力となります。

```
(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⟩)
```

この式が満たされるのは、パターン変数に対するすべての値の集合が、同時に ⟨query₁⟩ ... ⟨query_n⟩ を満たすときです。

単純クエリの場合と同じように、システムが複合クエリを処理する際には、パターン変数に対する代入でそのクエリを満たすものをすべて探し、それらの値によるクエリの具体化をすべて表示します。

複合クエリを構築するもうひとつの手段は `or` によるものです。例えば、

```
(or (supervisor ?x (Bitdiddle Ben))
       (supervisor ?x (Hacker Alyssa P)))
```

この式は、Ben Bitdiddle または Alyssa P. Hacker に監督される従業員をすべて検索します。

```
(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>)
```

この式が満たされるのは、パターン変数に対するすべての値の集合が、<query₁> ... <query_n> のうち少なくともひとつを満たすときです。

複合クエリは、notによって作ることもできます。例えば、

```
(and (supervisor ?x (Bitdiddle Ben))  
      (not (job ?x (computer programmer))))
```

この式は、Ben Bitdiddle に監督され、コンピュータプログラマではないすべての人を検索します。一般的に、

```
(not <query1>)
```

この式が満たされるのは、パターン変数に対するすべての代入が<query₁> を満たさない場合に満たされます。⁶³

最後の組み合わせ形式は lisp-value と呼ばれるものです。lisp-value ががパターンの一つ目の要素であれば、それは次の要素が Lisp の述語であり、残りの(具体化された)要素を引数として適用するということを規定しています。一般的に、

```
(lisp-value <predicate> <arg1> ... <argn>)
```

この式を満たすのは、<predicate> が具体化された <arg₁> ... <arg_n> に適用されたときの値が真になるようなパターン変数への代入です。例えば、給料が 30,000 ドルより多い人をすべて検索する場合は、次のように書くことができます。⁶⁴

⁶³ 実際は、notに対するこの記述が有効なのは簡単な場合だけです。実際の not のふるまいはより複雑です。not の独特なところについては、Section 4.4.2 と Section 4.4.3 で詳しく見ていきます。

⁶⁴ lisp-value は、クエリ言語によって提供されていない演算を行う場合に限って使うべきです。具体的には、等価性のテストや、非等価性のテストに使ってはいけません(等価性については、クエリ言語のマッチがまさにそのために設計されたものだからです。非等価性については、下で示す same によって行うことができます)。

```
(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考えよ。

- a Ben Bitdiddle に監督されるすべての人と、その住所
- b Ben Bitdiddle よりも給料が少ないすべての人と、その給料と、Ben Bitdiddle の給料
- c コンピュータ部門以外の人に監督されているすべての人と、その上司の名前と職務

規則

単純クエリと複合クエリのほかにも、クエリ言語はクエリを抽象化する手段を提供しています。その手段とは*rules(規則)* と呼ばれるものです。以下の規則は、

```
(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))))
```

二人の人が同じ町に住んでいるなら、互いに近くに住んでいる (*lives-near*) とする、ということを規定しています。最後の *not* 節は、この規則によってすべての人がその人自身の近くに住んでいるということになるのを防いでいます。*same* 関係はとても単純な規則によって定義されています。⁶⁵

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

以下の規則は、ある人が誰かを監督する人を監督している場合、その人を組織内での “wheel”(重要人物) であると宣言するものです。

⁶⁵二つのものを同じものであるようにするには *same* は必要ないということに注意してください。その場合は、同じパターン変数をそれぞれに使うだけで十分です—つまり、そもそも二つのものを持たずに、ひとつのものしか持たなければいいということです。例えば、*lives-near* 規則の *?town* や、以下の *wheel* 規則の *?middle-manager* がそうです。*same* が役に立つのは、*lives-near* 規則の *?person-1* と *?person-2* のように、二つのものが違うものであることを強制する場合です。同じパターン変数をクエリの二箇所で使うと、両方の場所で同じ値が現れることを強制するのですが、別々のパターン変数を使っても、別々の値が現れることを強制するわけではありません (別々のパターン変数に代入される値は、同じであることも別々であることもあります)。

```
(rule (wheel ?person)
      (and (supervisor ?middle-manager ?person)
            (supervisor ?x ?middle-manager)))
```

規則の一般形式は次のようにになります。

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

ここで、*<conclusion>* はパターンで、*<body>* は任意のクエリです。⁶⁶ 規則は、大きな(無限であることすらある)表明の集合、つまり規則の本体を満たす変数代入によって規則の結論を具体化したものすべてを表現するものとして考えることができます。単純クエリ(パターン)について記述したときには、具体化されたパターンがデータベースに存在する場合に変数への代入がパターンを満たすと書きました。しかし、パターンは表明としてデータベースに明示的に存在する必要はありません。それは、規則が含意する暗黙的な表明であることもあります。例えば、以下のクエリは、

```
(lives-near ?x (Bitdiddle Ben))
```

次の結果となります。

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

Ben Bitdiddle の近くに住むコンピュータプログラマをすべて検索するには、次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)))
```

複合手続きの場合と同じように、規則はほかの規則の一部として(上記の *lives-near* 規則で見たように)使うことができます。さらに、再帰的な定義も可能です。例えば、次の規則は、

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

⁶⁶*same*のような本体のない規則も認めることにします。そのような規則は、規則の結論は変数の値がどのようなものであっても満たされているものとして解釈します。

ある職員 (staff-person) について、ボス (boss) がその人の上司であるか、(再帰的に) その職員の上司よりそのボスが地位が上 (outranked) であれば、そのボスはその職員より地位が上であるというものです。

Exercise 4.57: 人 1 が人 2 と同じ職務を持っているか、人 1 の職務を持つ人には人 2 の職務もできるという場合で、人 1 と人 2 が同一人物でない場合、人 1 は人 2 を代替 (replace) できるとする規則を定義せよ。この規則を使って、以下の検索を行うクエリを示せ。

- a Cy D. Fect を代替できるすべての人
- b 自分より給料の高い誰かを代替できるすべての人と、その二つの給料

Exercise 4.58: ある人が、自分の勤める部署に勤める監督者がいない場合、その人を “big shot”(重要人物) であるとする規則を定義せよ。

Exercise 4.59: Ben Bitdiddle はある会議を何度も欠席してしまった。この会議を忘れる癖を何とかしないと首になると想って、Ben は対策を取ることを決めた。彼は、以下のような表明として会社の週次ミーティングを Microshaft のデータベースにすべて追加した。

```
(meeting accounting (Monday 9am))
(meeting administration (Monday 10am))
(meeting computer (Wednesday 3pm))
(meeting administration (Friday 1pm))
```

それぞれの表明は部門の全体会議のものだ。それに加えて、Ben は全部門にわたる全社会議のエントリも追加した。この会議には会社の全従業員が参加する。

```
(meeting whole-company (Wednesday 4pm))
```

- a 金曜の朝、Ben はその日にあるすべての会議についてデータベースに問い合わせようと思った。彼はどのようなクエリを使うべきだろうか。

b Alyssa P. Hacker は、このやり方はあまりよくないと思った。彼女は、自分の名前を指定して自分の会議について聞くことができたほうがずっと便利だろうと考えた。そこで、彼女は規則を設計することにした。その規則は、ある人の会議は全社 (whole-company) 会議とその人の部門会議をすべて含むというのだ。Alyssa の規則に本体を補え。

```
(rule (meeting-time ?person ?day-and-time)
      (rule-body))
```

c Alyssa は水曜の朝に職場に着き、その日にどんな会議があるか考えた。上記の規則を定義してあるとして、この検索を行うには、彼女はどのようなクエリを作るべきだろうか。

Exercise 4.60: 以下のクエリによって、

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

自分の近所に住んでいて会社まで車の相乗りができる人を Alyssa P. Hacker は探すことができている。一方、彼女が次のクエリを使って互いに近所に住んでいる人のペアをすべて検索しようとしたとき、

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

互いに近所に住んでいる人のペアがそれぞれ二回ずつ表示されていることに気がついた。例えば、次のようなものだ。

```
(lives-near (Hacker Alyssa P) (Fect Cy D))
(lives-near (Fect Cy D) (Hacker Alyssa P))
```

これはなぜ起こるのだろうか。それぞれのペアが一回ずつしか現れないようにして、互いに近所に住んでいる人のリストを検索する方法はあるだろうか。説明せよ。

プログラムとしての論理

規則は論理的含意の一種と見なすことができます。もしパターン変数に対する値の代入が本体を満たすのであれば、その代入は結論を満たします。したがって、クエリ言語は規則に基づき *logical deductions*(論理的推論) を行う能力

があるものと見なすことができます。例として、Section 4.4の始めに記述した `append` 演算について考えてみましょう。すでに述べたように、`append` は次の二つの規則によって特徴づけられます。

- 任意のリスト `y` について、空リストと `y` を `append` すると `y` になる。
- 任意の `u, v, y, z` について、`v` と `y` を `append` すると `z` になるならば、`(cons u v)` と `y` を `append` すると `(cons u z)` になる。

これを私たちのクエリ言語で表現するために、次の関係に対して二つの規則を定義します。

```
(append-to-form x y z)
```

この関係は、“`x` と `y` を `append` すると `z` になる” ということだと解釈できます。

```
(rule (append-to-form () ?y ?y))
(rule (append-to-form (?u . ?v) ?y (?u . ?z))
      (append-to-form ?v ?y ?z))
```

最初の規則には本体がありません。これは、`?y` がどのような値であっても結論が成り立つということです。二つ目の規則で、リストの `car` と `cdr` に名前をつけるためにドット末尾記法を使っていることに注意してください。

これら二つの規則があれば、二つのリストの `append` を計算するクエリを定式化できます。

```
; ; ; Query input:
(append-to-form (a b) (c d) ?z)
; ; ; Query results:
(append-to-form (a b) (c d) (a b c d))
```

それよりもすごいのは、“`(a b)` に `append` すると `(a b c d)` になるリストはどのようなリストか” と質問するのにも同じ規則が使えるというところです。これは、以下のように実行できます。

```
; ; ; 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)` になるすべてのリストのペアについて質問することも可能です。

```

;;; 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))

```

規則を使って上記のクエリの答えを推論しているという点で、このクエリシステムはかなりの知性を示しているかのように見えるかもしれません。実際のところは、次の節で見ていくように、このシステムは規則を解きほぐすにあたって明確に定義されたアルゴリズムに従っています。残念ながら、このシステムは `append` の場合には見事なぐらいうまくいくのですが、より複雑な場合には、Section 4.4.3で見ていくように、一般的な方法が破綻することもあります。

Exercise 4.61: 次の二つの規則は、リストの隣り合う要素を求める `next-to` 関係を実装している。

```

(rule (?x next-to ?y in (?x ?y . ?u)))
(rule (?x next-to ?y in (?v . ?z))
      (?x next-to ?y in ?z))

```

以下のクエリに対する答えはどのようになるだろうか。

```

(?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` 演算を実装する規則を定義せよ。これは、空でないリストの最後の要素を含むリストを返すものである。`(last-pair (3) ?x)`, `(last-pair (1 2 3) ?x)`, `(last-pair (2 ?x) (3))` のようなクエリに対して、それらの規則をチェックせよ。それらの規則は `(last-pair ?x (3))` のようなクエリに対して正しく動作するだろうか。

Exercise 4.63: 次のデータベース（創世記第4章参照）は、Adaの子孫の家系をたどり、Cainを経て Adamに到るまで遡っている。

```

(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”、“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”といった(聖書時代には現代よりもてはまりやすかったと思われる)規則を定式化せよ。これらによつて、クエリシステムがCainの孫、Lamechの息子、Methushaelの孫を求めることができるようになる(より複雑な関係を推論する規則については、Exercise 4.69参照)。

4.4.2 クエリシステムの仕組み

この先のSection 4.4.4では、クエリインタプリタの実装を手続きの集合という形で示す予定です。この節では、低レベルな実装の詳細によらないシステムの全体的な構造についての概要を見ていくことにします。インタプリタの実装について記述することによって、インタプリタの限界や、クエリ言語の論理演算が数学論理学と微妙に違う点について、理解することができるようになります。

クエリ評価器がクエリをデータベース内の事実や規則に対してマッチさせるためには何らかの探索を行う必要があるということはおわかりかと思います。探索を行う方法のひとつは、Section 4.3のamb評価器を使って、クエリシステムを非決定性プログラムとして実装するというものです(Exercise 4.78参照)。別のやり方としては、ストリームの助けを借りて探索を管理するというものもあります。ここでの実装は、この二つ目のアプローチを取ることにします。

このクエリシステムは、*pattern matching*(パターンマッチング)と*unification*(ユニフィケーション)という二つの中心的な演算をベースに構築します。初めにパターンマッチングについて述べ、フレームストリームという形で構成した情報とパターンマッチングを合わせて使うことによって単純クエリと複合クエリの両方を実装できるということを説明します。その次に、規則を実装するために必要となるパターンマッチングの一般化であるユニフィケーションに

について検討します。最後に、Section 4.1で記述したインタプリタで `eval` が式を分類するのと似たようなやり方で式を分類する手続きによって、クエリインタプリタ全体がうまく組み立てられるということを示します。

パターンマッチング

pattern matcher(パターンマッチャ) とは、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テストするプログラムです。例えば、データリスト `((a b) c (a b))` は、パターン `(?x c ?x)` にマッチし、このときパターン変数 `?x` は `(a b)` に束縛されます。この同じデータリストは、パターン `(?x ?y ?z)` にもマッチし、このとき `?x` と `?z` はどちらも `(a b)` に束縛され、`?y` は `c` に束縛されます。このデータリストは、パターン `((?x ?y) c (?x ?y))` にもマッチし、このとき `?x` は `a` に、`?y` は `b` に束縛されます。しかし、これはパターン `(?x a ?y)` にはマッチしません。このパターンは二つ目の要素が記号 `a` であるリストを指定しているからです。

クエリシステムの使うパターンマッチャは、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規定するものです。パターンマッチャは、フレームにすでに存在する束縛と矛盾しないようにデータがパターンにマッチするかどうかをチェックします。マッチするのであれば、マッチによって決められる束縛がある場合はその束縛を与えられたフレームに追加して、与えられたフレームを返します。マッチしなければ、マッチが失敗したことを伝えます。

例えば、空のフレームが与えられ、パターン `(?x ?y ?x)` を使って `(a b a)` にマッチさせると、`?x` が `a` に、`?y` が `b` に束縛されることを規定するフレームを返します。同じパターン、同じデータと、`?y` が `a` に束縛されることを規定するフレームによってマッチを試みると失敗します。同じパターン、同じデータと、`?y` が `b` に束縛され `?x` が未束縛であるフレームによってマッチを試みると、`?x` の `a` への束縛を与えられたフレームに追加したものが返されます。

規則を含まない単純なクエリを処理するためには、パターンマッチャという仕組みがあれば十分です。例えば、次のクエリを処理するには、

```
(job ?x (computer programmer))
```

データベース内のすべての表明を探索し、初期状態が空であるフレームについてパターンにマッチするものを選択します。見つかったそれぞれのマッチに対して、マッチによって返されたフレームを使って、`?x` の値でパターンを具体化します。

フレームのストリーム

パターンのフレームに対するテストは、ストリームを使うことによって組み立てられています。ひとつのフレームが与えられ、マッチング処理はデータベースのエントリをひとつずつ見ていきます。マッチャは、それぞれのデータベースエントリについて、マッチが失敗したことを示す特別な記号かフレームに対する拡張のどちらかを生成します。全データベースエントリについて結果はストリームに集められ、フィルタを通して失敗が取り除かれます。結果は、データベースの何らかの表明に対するマッチによって与えられたフレームを拡張したフレームすべてのストリームとなります。⁶⁷

私たちのシステムでは、Figure 4.4に示すように、クエリはフレームの入力ストリームを取り、ストリームの各フレームに対して上記のマッチング処理を行います。つまり、クエリは入力ストリームの各フレームに対して、データベース内の表明へのマッチによるそのフレームへの拡張すべてからなる新しいストリームを生成するということです。次に、これらのストリームがすべて組み合わされ、ひとつの巨大なストリームが作られます。このストリームは入力ストリームの各フレームに対するすべ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となります。

単純クエリに答えるには、ひとつの空フレームからなる入力ストリームとともにクエリを使います。結果となる出力ストリームは、空フレームに対するすべての拡張(つまり、クエリに対するすべての答え)を含んでいます。次に、このフレームのストリームを使って、元のクエリパターンのコピーの変数をそれぞれのフレーム内の値によって具体化したもののストリームを生成します。これが最終的に表示されるストリームとなります。

複合クエリ

フレームのストリームという実装の真の美しさがわかるのは複合クエリを扱うときです。複合クエリの処理は、マッチが指定されたフレームと矛盾しな

⁶⁷マッチングは一般的にとても高コストなので、完全なマッチャをデータベースの全要素に適用するのは避けたいところです。通常これは、高速で粗いマッチと最終マッチに処理を分解することによって達成します。粗いマッチはデータベースをフィルタし、最終マッチにかける候補の小さな集合を作ります。うまくやると、粗いマッチのための作業の一部を、候補選択時ではなくデータベース構築時に使うようにデータベースを作ることができます。これをデータベースの*indexing*(インデクシング)といいます。データベースインデクシングの設計に関しては膨大な技術の積み上げがあります。Section 4.4.4で記述するここでの実装は、そのような最適化を単純にしたものを使っています。

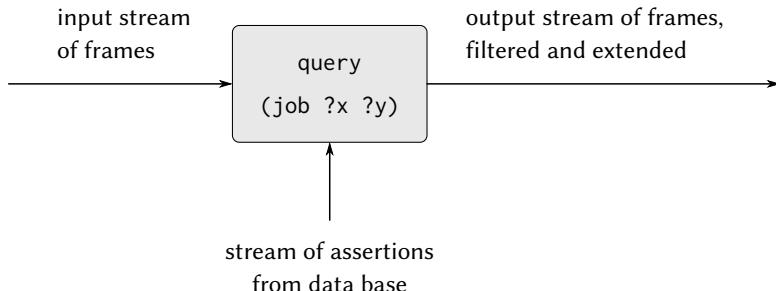


Figure 4.4: フレームのストリームを処理するクエリ

いように要求するというマッチャの能力を利用します。例えば、次のような二つのクエリの `and` を扱うには、

```
(and (can-do-job ?x (computer programmer trainee))
      (job ?person ?x))
```

(わかりやすく言うと、“プログラマ研修生の職務ができる人をすべて見つけよ”) まず次のパターンにマッチするすべてのエントリを探します。

```
(can-do-job ?x (computer programmer trainee))
```

すると、それぞれのフレームが`?x`に対する束縛を持つフレームのストリームができます。次に、ストリームの各フレームについて、与えられた`?x`に対する束縛に矛盾しない形で次のパターンにマッチするエントリをすべて探します。

```
(job ?person ?x)
```

そのようなマッチは、`?x`と`?person`に対する束縛を含むフレームをそれぞれ生成します。二つのクエリの `and` は、Figure 4.5に示すように、部品となる二つのクエリを直列に組み合わせたものとして見ることができます。最初のクエリフィルタを通過したフレームが、二つ目のクエリによってフィルタされ、さらに拡張されるということになります。

Figure 4.6は、似たような考え方で、二つのクエリの `or` を、部品となる二つのクエリを並列に組み合わせたものとして計算するという手法を示したもので、入力となるフレームのストリームは、それぞれのクエリによって別々に拡張されます。それから、結果となる二つのストリームがマージされ、最終的な出力ストリームが生成されます。

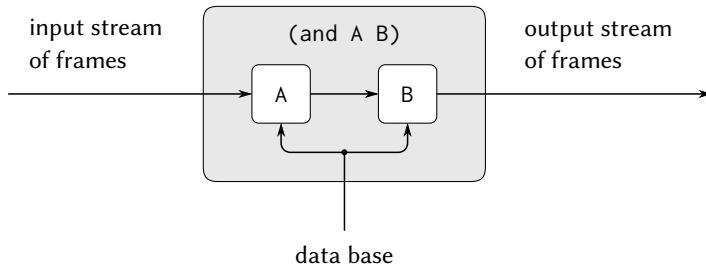


Figure 4.5: 二つのクエリの `and` による組み合わせは、フレームのストリームに対する直列の演算によって生成される

これは高レベルの記述ですが、これだけ見ても複合クエリの処理速度が遅くなりやすいということは明らかです。例えば、クエリはそれぞれの入力フレームについて複数の出力ストリームを生成する可能性があり、`and` は前のクエリから入力フレームを受け取るため、`and` クエリは、最悪の場合、クエリ数に対して指数的な数のマッチを実行する必要があります (Exercise 4.76 参照)⁶⁸。単純クエリだけを扱うシステムはとても実用的なのですが、複合クエリを扱うことは非常に難しいのです。⁶⁹

フレームのストリームという観点からは、クエリの `not` は、クエリが満たされるフレームをすべて取り除くフィルタとしてふるまいます。例えば、次のパターンが与えられたとき、

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

入力ストリームのそれぞれのフレームについて、`(job ?x (computer programmer))` を満たす拡張フレームの生成を試みます。そして、そのような拡張が存在するフレームすべてを入力ストリームから削除します。結果は、フレーム内の`?x` の束縛が `(job ?x (computer programmer))` を満たさないフレームのみからなるストリームとなります。例えば、次のクエリの処理では、

```
(and (supervisor ?x ?y)
```

⁶⁸しかし、このような指数的爆発は `and` クエリではありません。追加条件は、生成されるフレーム数を増やすより減らすことが多いからです。

⁶⁹複合クエリの効果的な処理方法を扱うデータベース管理システムの文献は多数存在します。

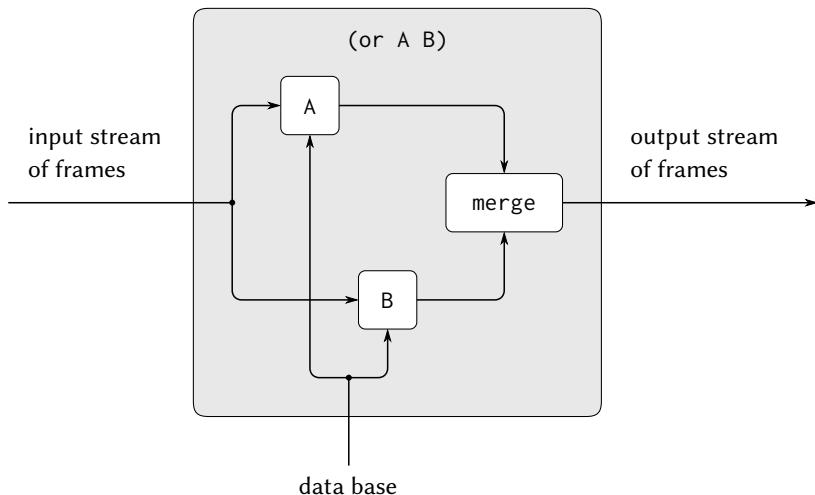


Figure 4.6: 二つのクエリの or による組み合わせは、フレームのストリームに対して並列に演算を行い、その結果をマージすることによって生成される

```
(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、これらのフレームから、 $?x$ 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すべてのフレームを削除することによってフィルタリングします。⁷⁰

`lisp-value` 特殊形式は、フレームのストリームに対する同じようなフィルタとして実装します。ストリームのそれぞれのフレームを使い、パターン内に変数があれば具体化し、それから Lisp 手続きを適用します。述語が失敗するフレームをすべて入力ストリームから削除します。

⁷⁰ フィルタによるこの `not` の実装と、数学論理学での通常の意味の `not` との間には、微妙な違いがあります。Section 4.4.3 参照。

ユニフィケーション

クエリ言語の規則を扱うためには、結果部分が与えられたクエリパターンにマッチする規則を見つけることができなければなりません。規則の結果は表明に似ていますが、変数を持つことがあるというところが違います。そのため、パターンマッチングを一般化したもの—*unification*（ユニフィケーション）と呼ばれます—が必要となります。ユニフィケーションの場合は、“パターン”と“データ”的な両方とも変数を持つ可能性があります。

ユニフィケーションを行うユニファイアは、二つのパターンを引数として取ります。パターンはどちらも定数や変数を含みます。そして、二つのパターンが等しいものとなるような変数への値の割り当てが可能かどうかを決定します。もし可能であれば、そ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、 $?x, ?y, ?z$ のすべてが a に束縛されなければならないというフレームを規定します。一方、($?x ?y a$) と ($?x b ?y$) のユニフィケーションは失敗します。二つのパターンを等しいものとするような $?y$ の値は存在しないからです（両方のパターンの二つ目の要素を等しくするために $?y$ は b でなければなりません。しかし、三つ目の要素を等しくするために $?y$ は a でなければなりません）。クエリシステムで使うユニファイアは、パターンマッチャ同様にフレームを入力として取り、そのフレームと矛盾しないようにユニフィケーションを実行します。

ユニフィケーションのアルゴリズムはクエリシステムの中で技術的に最も難しい部分です。複雑なパターンの場合、ユニフィケーションの実行には推論が必要となるように見えるかもしれません。例えば、($?x ?x$) と (($a ?y c$) ($a b ?z$)) のユニフィケーションを行うには、 $?x$ が $(a b c)$ となり、 $?y$ が b となり、 $?z$ が c となるということを推論しなければなりません。この処理は、パターンの部品同士の方程式を解くというように捉えることができます。一般的に、これらは連立方程式であり、解くにはかなりの量の操作が必要となることもあります。⁷¹ 例えば、($?x ?x$) と (($a ?y c$) ($a b ?z$)) のユニフィケーションは、次の連立方程式を規定しているものとして考えることができます。

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

これらの方程式は、

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

⁷¹ 片側パターンマッチングの場合は、パターン変数を含む等式はすべて明示的で、未知のもの（パターン変数）についてはすでに解かれています。

ということを含意しています。また、これは

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

ということを含意しています。よって、次のようにになります。

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

パターンマッチの場合は、マッチが成功するとパターン変数はすべて束縛され、束縛される値は定数だけを含むようになっています。ここまで見てきたユニフィケーションの例でも、すべてそのことが当てはまります。しかし一般的には、ユニフィケーションが成功した場合でも、変数の値は完全に決定されるとは限りません。変数は未束縛のまま残ることもあり、また変数を含む値に束縛されることもあります。

(?x a) と ((b ?y) ?z) のユニフィケーションについて考えてみましょう。
?x = (b ?y)、a = ?z だということは推論できますが、?x と ?y についてはそれ以上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。 $?x$ と $?y$ に値を割り当てるこ \cdot によって二つのパターンを等しくすることは可能だからです。このマッチは $?y$ の取る値についてはまったく制限していないため、 $?y$ に対する束縛は結果フレームに入れられません。しかし、このマッチは $?x$ の値については制限を加えています。 $?y$ がどのような値であれ、 $?x$ は (b ?y) でなければなりません。そのため、 $?x$ の (b ?y) というパターンへの束縛はフレームに入れられます。もし $?y$ の値が (パターンマッチや、このフレームと矛盾しないことが求められるユニフィケーションによって) 後で決定されてフレームに追加されたら、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。⁷²

規則の適用

ユニフィケーションは、クエリシステム中の規則から推論を行う部分の鍵となっています。その実現方法を見るために、規則の適用が絡むクエリの処理

⁷²ユニフィケーションについてのもうひとつの考え方、二つの入力パターンの特殊化となる最も一般的なパターンを生成するというものです。つまり、(?x a) と ((b ?y) ?z) のユニフィケーションは ((b ?y) a) となり、上で検討した (?x a ?y) と (?y ?z a) のユニフィケーションは (a a a) となるということになります。ここで実装では、ユニフィケーションの結果はパターンとして考えるよりもフレームとして考えたほうが便利です。

について考えてみましょう。例えば、次のようなものです。

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

このクエリを処理するためには、まず上で説明した通常のパターンマッチ手続きを使って、このパターンにマッチする表明がデータベース内にあるかどうかを調べます（この場合にはありません。データベースには、誰が誰の近所に住んでいるかについての直接の表明が含まれていないからです）。次のステップとして、クエリパターンと各規則の結果とのユニフィケーションを試みます。すると、このパターンと、次の規則の結果をユニフィケーションすることによって、

```
(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))))
```

?person-2 が (Hacker Alyssa P) に束縛され、?x が ?person-1 に束縛される (?person-1 と同じ値を持つ) ことを規定するフレームが返されるということがわかります。今度は、このフレームについて、この規則の本体によって与えられる複合クエリを評価します。マッチが成功すると、このフレームは ?person-1 に対する束縛を与えることによって拡張され、結果として ?x の値も決まります。?x の値があれば、それを使って元のクエリパターンを具体化することができます。

一般的に、パターン変数に対する束縛を規定するフレーム内にクエリパターンを位置づけようとするとき、クエリ評価器は以下の手法を使って規則を適用します。

- クエリを規則の結論とユニフィケーションし、成功すれば元のフレームを拡張したものを作る。
- 拡張したフレームについて、規則の本体によって作られるクエリを評価する。

これは、Lisp の eval/apply 評価器での手続き適用手法にそっくりです。

- 手続きの仮引数を引数に束縛し、手続きの元の環境を拡張したフレームを作る。
- 拡張した環境について、手続きの本体によって作られる式を評価する

二つの評価器が似ているということは意外ではないでしょう。手続き定義がLispにおける抽象化の手段であるのと同じように、規則定義はクエリ言語における抽象化の手段です。どちらの場合も、適切な束縛を作成し、それらの束縛について規則や手続きの本体を評価することによって抽象化を解きほぐします。

単純なクエリ

この章では、規則のない単純クエリを評価するやり方をすでに学んでいます。規則の適用方法についてもここまでで学んだため、規則と表明の両方を使って単純クエリを評価するやり方についても記述することができるようになります。

クエリパターンとフレームのストリームが与えられたとき、入力ストリームの各フレームに対して、二つのストリームを生成します。

- (パターンマッチャを使って) データベース内のすべての表明に対してパターンをマッチさせることによって得られる拡張フレームのストリーム
- (ユニファイアを使って) すべての可能な規則を適用することにより得られる拡張したフレームのストリーム⁷³

これら二つのストリームを連結すると、元のフレームと矛盾しないように与えられたパターンを満足するすべてのやり方からなるストリームができます。次に、これらのストリーム(入力ストリームのフレームごとにひとつずつ)がすべて結合され、ひとつの大きなストリームになります。このストリームは、元の入力ストリームのいずれかを拡張して与えられたパターンとのマッチを作るやり方すべてからなるものです。

クエリ評価器とドライバーラープ

基層となるマッチング演算は複雑なものですですが、このシステムは普通の言語の評価器とほぼ同じように構成されます。マッチング演算を司る手続きは `qeval` というもので、Lisp の `eval` 手続きと似たような役割を果たします。`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。出力はフレ

⁷³ユニフィケーションはマッチングの一般化であるため、両方のストリームを生成するのにユニファイアを使うことによってシステムを単純にすることもできるところです。しかし、簡単なケースを単純なマッチャで扱うことによって、(ちゃんとしたユニフィケーションではない)マッチングがそれ自身役に立つということがわかります。

ームのストリームで、それぞれのフレームは入力フレームのいずれかを拡張する成功したマッチに対応しています。これはFigure 4.4で図示した通りです。`eval` 同様、`qevel` はいろいろなタイプの式(クエリ)を分類し、それぞれに対する適切な手続きにディスパッチします。それぞれの特殊形式(`and`, `or`, `not`, `lisp-value`)ごとにひとつ手続きがあり、また単純クエリの手続きもあります。

ドライバループは、この章のほかの評価器の `driver-loop` 手続きと似ていて、端末からクエリを読み込みます。それぞれのクエリについて、ドライバループはそのクエリと空フレームひとつだけからなるストリームを引数として `qevel` を呼び出します。これによって、すべての可能なマッチ(空フレームに対するすべての可能な拡張)のストリームが生成されます。結果となるストリームのそれぞれのフレームについて、ドライバループはフレームにある変数の値を使って元のクエリを具体化します。その後、この具体化されたクエリのストリームが表示されます。⁷⁴

ドライバは、特別コマンド `assert!` についてもチェックします。これは、入力がクエリではなく、データベースに追加する表明または規則であるということを示すものです。次のように使われます。

```
(assert! (job (Bitdiddle Ben)
                 (computer wizard)))
(assert! (rule (wheel ?person)
               (and (supervisor ?middle-manager ?person)
                     (supervisor ?x ?middle-manager))))
```

4.4.3 論理プログラミングは数学論理学か

クエリ言語で使われる組み合わせ手段は、最初は数学論理学の `and`, `or`, `not` 演算と同じように見えるかもしれません。また実際、クエリ言語の規則の適用は、正当な推論の手法によってなされます。⁷⁵ しかし、このようにクエリ言語

⁷⁴ここでフレームの(リストではなく)ストリームを使う理由は、規則を再帰的に適用するとひとつのクエリを満たす無限の数の値が生成されることがあるからです。ストリームに組み込まれた遅延評価がここで重要になります。システムは、応答の数が有限か無限かにかかわらず、応答が生成されるごとにそれらをひとつずつ表示していきます。

⁷⁵ある特定の推論手法が正当であるということは自明な主張ではありません。そのためには、真の前提から始めると真の結論のみが導出されるということを証明する必要があります。規則適用によって表現される推論の手法は、 A が真でありかつ A ならば B が真であるなら B は真であると結論づけるという、*modus ponens*(モーダス・ポネンス) というよく知られた推論手法です。

と数学論理学を同一視するのは、実際には妥当ではありません。クエリ言語は、論理的な命題を手続き的に解釈する *control structure*(制御構造)を持つているためです。この制御構造を活用することはよくあります。例えば、プログラマの上司すべてを探すには、次の二つの論理的に等価な形式のどちらかによってクエリを作ることができます。

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

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

もし会社にプログラマより多くの上司が存在する(それが一般的なケースです)のであれば、二つ目の形式よりも一つ目の形式を使うほうがよいということになります。というのは、*and*の一つ目の節によって生成される中間結果(フレーム)それぞれについてデータベースを走査する必要があるからです。

論理プログラミングが目標としているのは、計算上の問題を、“何を”計算するのかと“どのように”これを計算するのかという二つの別々の問題に分解する技術をプログラマに提供することです。これを達成するために、計算したいものを何でも記述できる程度には強く、かつ制御可能な手続き的解釈が存在する程度には弱くなるように、数学論理学の命題の部分集合を選びます。ここでの意図は、論理プログラミング言語によって規定されるプログラムは、コンピュータによって実行できる効率的なプログラムでなければならないという側面があるということです。制御 (“どのように”計算するか) は、言語の評価の順序を使うことによって達成されるものです。ですので、節の順序や、それぞれの節の中での副目標の順序を調整して、計算が実効的かつ効率的に行われると思われる順序で実行されるようにできる必要があります。それと同時に、計算の結果 (“何を”計算するか) は、論理法則の単純な帰結として見ることができることがあります。

ここでのクエリ言語は、ちょうどそのような、手続き的に解釈可能な数学論理学の部分集合と見なすことができます。表明は、単純な事実(アトミックな命題)を表現します。規則は含意を表現しています。この含意とは、規則の本体が成り立つ場合に規則の結論が成り立つというものです。規則には、自然な手続き的解釈があります。それは、規則の結論を成り立たせるためには、規則の本体を成り立たせるというものです。そのため、規則は計算を規定することになります。しかし、規則は数学的論理の命題と見なすこともできるため、完全に数学的論理の範囲内で操作を行うことで同じ結果が得られると主張することによって、論理プログラムによって行われるどのような“推論”も正当化

することができます。⁷⁶

無限ループ

論理プログラムを手続き的に解釈することによる結果のひとつとして、ある種の問題を解くのにどうしようもないほど非効率的なプログラムを構築してしまうことがあります。非効率の極端なケースは、推論をする中でシステムが無限ループにはまってしまうというものです。単純な例として、有名な夫婦のデータベースを作ることを考えてみましょう。そのデータベースには、次のものが含まれています。

```
(assert! (married Minnie Mickey))
```

ここで、以下の質問をすると、

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

応答は得られません。というのは、「 A が B と結婚している場合、 B は A と結婚している」ということをシステムは知らないからです。そのため、以下の規則を表明します。

```
(assert! (rule (married ?x ?y) (married ?y ?x)))
```

そして、同じ質問をします。

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

残念ながら、こうするとシステムは次のように無限ループに陥ってしまいます。

⁷⁶この文は、「論理プログラムによってなされる“推論”というとき、その計算は停止するものとする」ということについて合意し、その条件をつける必要があります。残念ながら、私たちのクエリ言語の実装では、この条件つきの文でも偽となります(これはProlog や、ほかのほとんど現在の論理プログラミング言語のプログラムについても同じく偽です)。これは、`not` と `lisp-value` を使っていることによります。この先で記述するように、クエリ言語に実装された `not` は数学論理学の `not` と常に一致することは限らず、また `lisp-value` は事情をさらに複雑にします。単純に `not` と `lisp-value` をこの言語から取り除き、プログラムは単純クエリ、`and`、`or`だけで書くということにすれば、数学論理学と一致する言語を実装できるところです。しかし、こうするとこの言語の表現力は大きく制限されます。論理プログラミングの研究における主要な課題のひとつは、表現力を過度に犠牲にすることなく、数学論理学とより一致するようにする方法を見つけるということです。

- システムは `married` 規則が適用可能だということを発見します。これは、規則の結論 (`married ?x ?y`) はクエリパターン (`married Mickey ?who`) とのユニファイに成功し、`?x` が `Mickey`、`?y` が `?who` に束縛されるフレームを生成するということです。そこで、インタプリタはこのフレームの中で規則本体の (`married ?y ?x`) を評価します。つまり、(`married ?who Mickey`) というクエリを処理することになります。
- データベース内には、ひとつの答えが直接的に表明として存在します: (`married Minnie Mickey`)
- それと同時に、`married` 規則も適用可能です。そのため、インタプリタは再度規則本体を評価します。今回は (`married Mickey ?who`) と等価になります。

これでシステムは無限ループになります。システムがループに陥る前に単純な答である (`married Minnie Mickey`) を見つけられるかどうかは、システムがデータベース内の項目をチェックする順番の実装に依存します。これは、起りうるループの非常に単純な例です。相互に関連する規則の集まりの場合、これよりもずっと予測が難しいループを招くことがあります、またループが現れるかどうかは `and` 内の節の順序 (Exercise 4.64 参照) や、システムがクエリを処理する順序に関する低レベルの実装に依存することもあります。⁷⁷

not の問題

クエリシステムのもうひとつの厄介なところは、`not` に関するものです。Section 4.4.1 のデータベースがあるとき、次の二つのクエリについて考えてみましょう。

⁷⁷ これは論理の問題ではなく、インタプリタによって提供される、論理の手続き的解釈の問題です。ここでループに陥らないインタプリタを書くことも一応は可能です。例えば、表敬と規則から導出可能なすべての証明を、深さ優先探索ではなく幅優先探索によって列挙するということもできます。しかし、そのようなシステムでは、プログラム中の推論の順番を利用することが難しくなります。そのようなプログラムで精緻な制御構造を構築する試みが de Kleer et al. 1977 に記述されています。そのような深刻な制御上の問題を引き起こさない別のテクニックとして、特定の種類のループに対する検知器といった、特別な知識を組み込むというものがあります (Exercise 4.67)。しかし、推論を実行する際にシステムが無限の経路をたどり続けることを確実に防ぐ一般的な設計方法はありません。適切に選んだ関数 f についての、“ $P(x)$ 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” という形の悪魔の規則を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))
```

これらの二つのクエリは同じ結果を生成しません。一つ目のクエリは、まずデータベース中の (supervisor ?x ?y) にマッチするすべてのエントリを探し、その次に、返されたフレームの中から ?x の値が (job ?x (computer programmer)) を満たすものを削除します。二つ目のクエリは、まず入力フレームをフィルタし、その中から (job ?x (computer programmer)) を満たすものを削除します。唯一の入力フレームは空フレームなので、データベースから (job ?x (computer programmer)) を満たすパターンがあるかどうかチェックします。普通はこの形のエントリが存在するので、not 節は空フレームを取り除き、空のフレームストリームを返します。その結果、複合クエリ全体は空ストリームを返すことになります。

問題は、not の実装が、実際は変数の値に対するフィルタとして働くように作られているということです。not 節が (上の例での?x のように) 束縛されていないままの変数のあるフレームとともに処理されると、システムは期待しない結果を返します。同じような問題は、lisp-value を使う際にも起こります。Lisp の述語は、未束縛の引数があると動作しません。[Exercise 4.77](#)を参照してください。

クエリ言語の not と数学論理学の not の違いは、これよりもはるかに深刻なところもあります。論理学では、命題 “not P ” を、 P は真ではないという意味に解釈します。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という意味になります。例えば、[Section 4.4.1](#)の社員情報データベースがあるとき、システムはどんな not 命題でも、例えば Ben Bitdiddle は野球ファンではない、外で雨は降っていない、 $2 + 2$ は 4 ではない、といったものでも、喜んで推論してくれるでしょう。⁷⁸

Exercise 4.64: Louis Reasoner は誤って outranked-by 規則 ([Section 4.4.1](#)) をデータベースから削除してしまった。彼がこのこと

⁷⁸(not (baseball-fan (Bitdiddle Ben))) というクエリ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)) がないということを発見し、そのため空フレームはパターンを満たさないことになり、最初のフレームのストリームから取り除かれません。したがって、クエリの結果はその空フレームとなり、それが入力クエリを具体化するために使われて、(not (baseball-fan (Bitdiddle Ben))) が生成されます。

に気づいたとき、すぐにまたその規則を組み込んだ。困ったことには、彼は規則を少し変更してしまい、次のように入力した。

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

Louis がこの情報をシステムに入力してすぐ DeWitt Aull がやってきて、Ben Bitdiddle よりも地位が上の人に誰がいるかを調べようとした。彼は次のクエリを入力した。

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

応答の後、システムは無限ループに陥った。理由を説明せよ。

Exercise 4.65: 組織内での昇進の日を待ち望んでいる Cy D. Fect は、(Section 4.4.1 の規則 `wheel` を使って) 組織の重要人物をすべて検索するクエリを入力してみた。

```
(wheel ?who)
```

彼はシステムからの応答に驚いた。

```
; ; Query results:
(wheel (Warbucks Oliver))
(wheel (Bitdiddle Ben))
(wheel (Warbucks Oliver))
(wheel (Warbucks Oliver))
(wheel (Warbucks Oliver))
```

Oliver Warbucks はなぜ四回表示されたのだろうか。

Exercise 4.66: Ben は、会社についての統計データを出せるようにクエリシステムを一般化する作業をしているところだ。例えば、以下のように入力することで、すべてのコンピュータプログラマの給料の合計を求めることができるようになる。

```
(sum ?amount (and (job ?x (computer programmer))
                     (salary ?x ?amount)))
```

一般的に、Ben の新しいシステムは次の形式の式を受け入れる。

(*accumulation-function* *variable*) *(query pattern)*

ここで、*accumulation-function* は *sum*, *average*, *maximum* のようなものだ。Ben は、この実装は朝飯前だろうと考えた。クエリパターンを単純に *qeval* に与えればいい。そうするとフレームのストリームができる。それから、このストリームを、各フレームから指定された変数の値を取り出しその結果のストリームを集積関数に渡すマップ関数に渡す。Ben が実装を終えて動作確認をしようとしていたところ、Cy がやってきた。Cy は相変わらず Exercise 4.65 の *wheel* クエリの結果に悩んでいた。Cy が Ben にシステムの応答を見せると、Ben は“ああ、ダメだ。これじゃこの単純な集積計画はうまくいかないな”とうめいた。

Ben は何に気づいたのだろうか。状況打開に使える方法の概要を述べよ。

Exercise 4.67: 本文や Exercise 4.64 で示したような種類の単純なループを防ぐために、クエリシステムにループ検知器を組み込む方法を考えよ。考え方の概要是、システムが現在の推論のつながりについて何らかの形で履歴を管理するようにして、すでに実行中のクエリの処理を始めないようにするというものである。この履歴にはどのような種類の情報(パターンとフレーム)が含まれるか、またどのようにチェックを行うかを説明せよ (Section 4.4.4 でクエリシステムの実装の細部を学んでから、システムを修正してループ検知器を組み入れてみるのもいいだろう)。

Exercise 4.68: 与えられたリストと同じ要素を逆順で持つリストを返す、Exercise 2.18 の *reverse* 演算を実装する規則を定義せよ(ヒント : *append-to-form* を使う)。その規則は (*reverse* (1 2 3) ?x) と (*reverse* ?x (1 2 3)) の両方に答えることができるだろうか。

Exercise 4.69: Exercise 4.63 で作ったデータベースと規則をもとに、孫 (grandson) の関係に “great” を追加する規則を考えよ。これによって、システムは Irad が Adam の great-grandson(ひ孫) であることや、Jabal と Jubal が Adam の great-great-great-great-grandsons(ひひひひ孫) であることが推論できるようになる(ヒント : Irad についての事実を、例えば ((great grandson) Adam

Irad) と表現する。また、リストの末尾が単語 grandson であるかどうか決定する規則を書く。これを使って、?rel が grandson で終わるリストであるとき、((great . ?rel) ?x ?y) という関係を導出できる規則を表現する)。これらの規則を、((great grandson) ?g ?ggs) や (?relationship Adam Irad) のようなクエリによって確認せよ。

4.4.4 クエリシステムの実装

Section 4.4.2では、クエリシステムの動作について述べた。ここではシステムの完全な実装を紹介し、細部を埋めていきます。

4.4.4.1 ドライバループと具体化

クエリシステムのドライバループは繰り返し入力式を読み込みます。読み込んだ式がデータベースに追加すべき規則または表明であれば、その情報を追加します。それ以外の場合は、式はクエリと見なされます。ドライバは、ひとつの空フレームからなる初期フレームストリームとともに、このクエリを評価器 qeval に渡します。評価の結果は、データベースに見つかる変数の値でクエリを満たすことによって生成されるフレームストリームです。これらのフレームは、元のクエリのコピーの変数をフレームストリームによって与えられる値によって具体化したものからなる新しいストリームを作るのに使われます。この最後のストリームが端末に表示されます。

```
(define input-prompt ";;; Query input:")
(define output-prompt ";;; Query results:")

(define (query-driver-loop)
  (prompt-for-input input-prompt)
  (let ((q (query-syntax-process (read))))
    (cond ((assertion-to-be-added? q)
           (add-rule-or-assertion! (add-assertion-body q))
           (newline)
           (display "Assertion added to data base."))
          (query-driver-loop)))
    (else
      (newline)))
```

```

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

```

この章のほかの評価器と同じように、ここではクエリ言語の式に抽象構文を使っています。式の構文の実装は、述語 `assertion-to-be-added?` やセレクタ `add-assertion-body` も含め、Section 4.4.4.7 にあります。`add-rule-or-assertion!` はSection 4.4.4.5で定義されます。

入力式の処理を始める前に、ドライバループは入力式をより効率的に処理できるような形に構文的に変換します。この変換にはパターン変数の表現の変更も伴います。クエリが具体化されるときには、未束縛のままの変数があれば、表示前に入力時の表現に再度変換されます。これらの変換は、`query-syntax-process`、`contract-question-mark` という二つの手続きによって行われます (Section 4.4.4.7)。

式を具体化するときは、式をコピーし、中の変数をすべて与えられたフレーム内の値で置き換えます。値自身も変数を含んでいる可能性があるため具体化します (例えば、式の中の`?x` がユニフィケーションの結果`?y` に束縛され、`?y` 自身は 5 に束縛されている場合)。変数が具体化できない場合に取るべき行動は、`instantiate` 手続きの引数として渡されます。

```

(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きはSection 4.4.4.8で定義されます。

4.4.4.2 評価器

`query-driver-loop` により呼ばれる `qevel` 手続きは、クエリシステムの基本的な評価器です。入力としてクエリとフレームのストリームを取り、拡張したフレームのストリームを返します。特殊形式は、`get` と `put` を使ったデータ主導のディスパッチによって識別します。これは、Chapter 2でジェネリック演算を実装したときと同じです。

特殊形式と識別されない任意のクエリは単純クエリと見なされ、`simple-query` によって処理されます。

```
(define (qevel query frame-stream)  
  (let ((qproc (get (type query) 'qevel)))  
    (if qproc  
        (qproc (contents query) frame-stream)  
        (simple-query query frame-stream))))
```

`type` と `contents` はSection 4.4.4.7で定義されるもので、特殊形式の抽象構文を実装します。

単純クエリ

`simple-query` 手続きは単純クエリを扱います。引数として単純クエリ(パターン)とフレームのストリームを取り、それぞれのフレームをクエリのデータベースとのマッチすべてによって拡張することで得られる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))
```

入力ストリームの各フレームについて、[find-assertions\(Section 4.4.4.3\)](#) を使ってパターンをデータベースの表明すべてとマッチさせ、拡張フレームのストリームを生成します。そして、[apply-rules\(Section 4.4.4.4\)](#) を使って可能な規則をすべて適用し、また別の拡張フレームのストリームを生成します。これら二つのストリームは ([stream-append-delayed\(Section 4.4.4.6\)](#) を使って) 組み合わされ、結果として与えられたパターンが元のフレームと矛盾しない形で満たされるすべての方法からなるストリームができます ([Exercise 4.71](#)参照)。個々の入力フレームについてのこのストリームは [stream-flatmap \(Section 4.4.4.6\)](#) によって組み合わされ、元の入力ストリームの任意のフレームを拡張して与えられたパターンとのマッチを作るすべての方法の大きなストリームがひとつできます。

複合クエリ

`and` クエリは、Figure 4.5で図示したように、`conjoin` 手続きによって扱われます。`conjoin` は、入力として連言肢 (`conjunct`) 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。まず、`conjoin` はフレームのストリームを処理して、連言 (`conjunction`) の最初のクエリを満たすフレームの可能な拡張をすべて探しします。次に、これを新しいフレームのストリームとして、クエリの残り部分に対して再帰的に `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` が `and` 形式を見つけたときに `conjoin` にディスパッチするよう設定します。

Figure 4.6に図示したように、`or` クエリも同じように扱います。`or` のそれぞれの選言肢 (`disjunct`) に対する出力ストリームは別々に計算され、[Section 4.4.4.6](#) の `interleave-delayed` 手続きによって結合されます ([Exercise 4.71](#)、[Exercise 4.72](#)参照)。

```
(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave-delayed
        (qevel (first-disjunct disjuncts)
               frame-stream)
        (delay (disjoin (rest-disjuncts disjuncts)
                        frame-stream)))))

(put 'or 'qevel disjoin)
```

連言肢構文と選言肢構文の述語とセレクタはSection 4.4.4.7で示します。

フィルタ

`not` は、Section 4.4.2で概説した方法で扱います。否定対象となるクエリを満たすように入力ストリームの各フレームを拡張することを試み、拡張できない場合のみ与えられたフレームを出力ストリームに入れます。

```
(define (negate operands frame-stream)
  (stream-flatmap
    (lambda (frame)
      (if (stream-null?
            (qevel (negated-query operands)
                  (singleton-stream frame)))
          (singleton-stream frame)
          the-empty-stream))
    frame-stream))
(put 'not 'qevel negate)
```

`lisp-value` は `not` に似たフィルタです。ストリームの各フレームを使ってパターン内の変数を具体化し、指定された述語を適用し、述語が偽を返したフレームを入力ストリームから取り除きます。未束縛のパターン変数がある場合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するのですが、適用する手続きを得るために述語式を `eval` する必要があります。しかし、引数を評価してはいけません。引数はすでに実際の引数であり、(Lisp で) 評価すると引数を生成する式ではないからです。`execute` の実装には基層となる Lisp システムの `eval` と `apply` が使われていることに注意してください。

```

(define (execute exp)
  (apply (eval (predicate exp)
               user-initial-environment)
         (args exp)))

```

特殊形式 `always-true` は、常に満たされているクエリを提供します。`always-true` は中身(通常は空)を無視し、入力ストリームのすべてのフレームをそのまま渡します。`always-true` は、`rule-body` セレクタ (Section 4.4.4.7) が本体なしで定義されている規則(つまり、結論部分が常に満たされる規則)に対して本体を提供するために使われます。

```

(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 で示します。

4.4.4.3 パターンマッチングによる表明の検索

`find-assertions` は、`simple-query` (Section 4.4.4.2) に呼ばれるもので、入力としてパターンとフレームを取ります。返すのはフレームのストリームで、各フレームは与えられたパターンに対するデータベースのマッチによって与えられたフレームを拡張したものです。この手続きは、パターンとフレームに対

するマッチを確認しなければならないデータベース内のすべての表明のストリームを得るために、`fetch-assertions` (Section 4.4.4.5) を使います。ここで `fetch-assertions` を行うのは、マッチが成功する候補のプールの中から多くのエントリを削除してくれる簡単なテストが適用できることがよくあるからです。ここで `fetch-assertions` を省いてデータベース内のすべての表明のストリームをチェックしたとしてもシステムは依然として動作するでしょうが、マッチャの呼び出し回数がずっと多くなるので、計算は非効率になるでしょう。

```
(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 (表明)、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、マッチに失敗した場合は `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、記号 `failed` または与えられたフレームを拡張したものを返します。マッチャの基本的な考え方は、データに対して要素ごとにパターンをチェックし、パターン変数に対する束縛を集積していきます。もしパターンとデータオブジェクトが同じなら、マッチは成功し、そこまで集積された束縛のフレームを返します。同じでなければ、パターンが変数の場合は、フレームにすでに存在する束縛と矛盾しない範囲で、変数をデータに束縛して現在のフレームを拡張します。パターンとデータが両方ともペアである場合は、(再帰的に) パターンの `car` をデータの `car` に対してマッチさせ、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を `cdr` をデータの `cdr` に対してマッチさせます。これらのどれにも当てはまらない場合、マッチは失敗し、記号 `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)))
```

以下は、フレームにすでにある束縛と矛盾しない範囲で新しい束縛を追加してフレームを拡張する手続きです。

```
(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))))
```

対象となる変数についての束縛がフレーム内になければ、変数とデータの束縛を追加します。束縛がある場合は、そのフレームの中で、データをフレーム内の変数の値に対してマッチさせます。格納された値が定数しか持たないのであれば (extend-if-consistent) によって格納されたのであれば必ずそうなっているはずです)、マッチは単純に格納された値と新しい値が同じかどうかテストします。同じであれば、変更されていないフレームを返します。同じでなければ、失敗したことを示すものを返します。しかし実際には、値がユニフィケーションの間に格納されたものである場合、格納された値がパターン変数を含んでいることもあります ([Section 4.4.4.4 参照](#))。格納されたパターンに対する新しいデータの再帰的なマッチは、このパターンに含まれる変数に対する束縛を追加したりチェックしたりすることになります。例えば、 $?x$ が $(f ?y)$ に束縛されていて、 $?y$ が未束縛であるフレームがあるとして、このフレームを $?x$ の $(f b)$ への束縛によって拡張したいとします。この場合、 $?x$ を検索し、それが $(f ?y)$ に束縛されているということを発見します。その結果、 $(f ?y)$ を提案された新しい値の $(f b)$ に対して同じフレームでマッチさせることになります。最終的に、このマッチは $?y$ の b への束縛を追加することによってこのフレームを拡張します。 $?x$ は $(f ?y)$ に束縛されたままです。格納されている束縛を修正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` によって使われる、束縛を操作する手続きはSection 4.4.4.8で定義されています。

ドット末尾を持つパターン

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、Exercise 2.20で述べたドット末尾記法から予想できるように、そのパターン変数はデータリストの(次の要素ではなく)残りにマッチします。ここまでで実装したパターンマッチャはドットを探すことはしませんが、期待通りのふるまいをします。これは、`query-driver-loop` で使っている Lisp の基本手続き `read` が、クエリを読み込んでそれをリスト構造として表現する際に、ドットを特別扱いするためです。

`read` がドットを見つけると、次の項目をリストの次の要素にする (`cons` の `car` にして、その `cdr` がリストの残りになる) のではなく、次の項目をリスト構造の `cdr` にします。例えば、パターン (`computer ?type`) に対する `read` によって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って構築されるものになり、パターン (`computer . ?type`) の場合は式 (`(cons 'computer '?type)`) を評価することによって構築されるものになります。

そのため、`pattern-match` がデータリストとドットを持つパターンの `car` と `cdr` を再帰的に比較していくと、最終的にはドットの次の変数(パターンの `cdr`) がデータリストの部分リストに対してマッチされることになり、変数はそのリストに対して束縛されることになります。例えば、パターン (`computer . ?type`) を (`programmer computer trainee`) にマッチさせると、`?type` をリスト (`programmer trainee`) にマッチさせることになります。

4.4.4.4 規則とユニフィケーション

`apply-rules` は、`find-assertions`(Section 4.4.4.3) の規則版です。入力としてパターンとフレームを取り、データベースの規則を適用することによって拡張したフレームのストリームを作ります。`stream-flatmap` は、(Section 4.4.4.5の `fetch-rules` によって選択された) 適用可能かもしれない規則のストリームに対して `apply-a-rule` をマップし、返されるフレームのストリームを結合します。

```
(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で概説した手法によって規則を適用します。まず、与えられたフレーム内で規則の結論とパターンのユニフィケーションを行うことによって引数のフレームを拡張します。それが成功すると、この新しいフレーム内で規則の本体を評価します。

しかし、これらのことを行なう前に、プログラムは規則内のすべての変数にユニークな名前をつけ直します。この理由は、別々の規則適用の変数が互いに混同されないようにするためです。例えば、二つの規則が両方とも`?x`という名前の変数を使っている場合、規則適用時にどちらも`?x`に対する束縛をフレームに追加するかもしれません。これら二つの`?x`は互いに関係がないので、この二つの束縛に矛盾があるってはならないなどと考えるようになってはいけません。変数名を変えるのではなく、より巧妙な環境構造を考えることもできるところかもしれません。しかし、変数名を変えるというここで選んだやり方は、最も効率のよいものではないにしても、もっとも素直なやり方です (Exercise 4.79 参照)。`apply-a-rule` 手続きは以下のようになります。

```
(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` は規則の部品を取り出すするもので、Section 4.4.4.7 で定義されます。

ユニークな変数名を作る際には、規則適用ごとにユニークな識別子(例えば数字)を関連づけ、その識別子と元の変数名を組み合わせることにします。例えば、規則適用の識別子が 7 なら、規則内の`?x` をすべて`?x-7` に、`?y` をすべて`?y-7` に変更することになります (`make-new-variable` と `new-rule-application-id` は Section 4.4.4.7 の構文手続きに含まれています)。

```
(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のアルゴリズムは、入力として二つのパターンとフレームを取り、拡張されたフレームまたは記号 `failed` を返す手続きとして実装されます。ユニファイアはパターンマッチに似ていますが、対照的だというところが違います — 変数はマッチのどちら側に現れることも可能です。`unify-match` は基本的には `pattern-match` と同じですが、マッチの右側のオブジェクトが変数である場合を扱う（下で “***” をつけた）追加コードがあるという違いがあります。

```
(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では、片側パターンマッチと同じように、既存の束縛と矛盾しないときだけフレーム拡張の提案を受け入れたいところです。ユニフィケーションで使われる手続き `extend-if-possible` は、パターンマッチで使われる `extend-if-consistent` と同じものですが、下記のプログラムで “***” をつけた二箇所の特別なチェックの部分が違います。一つ目のチェックでは、マッ

チさせようとしている変数が未束縛で、それに対してマッチさせようとしている値がそれ自身(別の)変数であるとき、その値が束縛されているかを確認する必要があります。束縛されていればその値をマッチさせる必要があります。マッチの両側がどちらも未束縛であれば、どちらかをもう一方に束縛します。

二つ目のチェックは、ある変数を、その変数を含むパターンに束縛しようとしている場合を扱います。このような状況は、ひとつの変数が両方のパターン内で繰り返し出てくる場合、いつでも起こる可能性があります。例えば、 $?x$ と $?y$ のどちらも未束縛のフレームで、 $(?x ?x)$ と $(?y < ?y \text{ を含む式} >)$ という二つのパターンのユニフィケーションを行う場合について考えてみましょう。まず、 $?x$ は $?y$ にマッチされ、 $?x$ から $?y$ への束縛が作られます。次に、同じ $?x$ が $?y$ を含む与えられた式にマッチされます。 $?x$ はすでに $?y$ に対して束縛されているため、結果として $?y$ をその式にマッチさせることになります。二つのパターンを同じものにするようなパターン変数の値の集合を探すものとしてユニファイアを考えるなら、これらのパターンは $?y$ が $?y$ を含む式に等しくなるような $?y$ を探せという指示を意味することになります。そのような方程式を解く一般的な手法はないため、そのような束縛は却下します。これらのケース

は、述語 `depends-on?` によって識別されます。⁷⁹ 一方、変数をそれ自身に束縛しようとするのは拒否したくないところです。例えば、 $(?x ?x)$ 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。 $?x$ を二回目に $?y$ に束縛しようとすると、 $?y(?x$ に格納された値) と $?y(?x$ の新しい値) をマッチさせることになります。これは `unify-match` の `equal?` 節で処理します。

```
(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
```

⁷⁹一般的に、 $?y$ と、 $?y$ を含む式とのユニフィケーションを行う場合、式 $?y = <?y$ を含む式 $>$ の不動点を見つけることができる必要があります。時には、解に見えるような式を構文的に作ることができます。例えば、 $?y = (f ?y)$ は、不動点 $(f (f (f \dots)))$ を持つように見えます。これは、式 $(f ?y)$ から始めて、 $?y$ を繰り返し $(f ?y)$ に置き換えていくことによって生成できます。残念ながら、そのような式がすべて意味のある不動点を持つわけではありません。ここでの問題は、数学で無限級数を操作する問題と似ています。例えば、私たちは 2 が式 $y = 1 + y/2$ の解であることを知っています。式 $1 + y/2$ から始めて、 y を繰り返し $1 + y/2$ に置き換えていくと、以下のようになります。

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

これは以下の式になります。

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

しかし、式 $y = 1 + 2y$ の解が -1 であるというところから始めると、

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

これは以下の式になります。

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

これらの二つの式を導くのに使った形式的操作は同じものなのに、一つ目の結果は無限級数についての正当な主張になっていて、二つ目はそうなっていません。同じように、ここでのユニフィケーションの結果についても、構文的に構築した勝手な式によって推論しようとすると、誤りを招く可能性があります。

```

(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、パターン変数の値として提案された式がその変数に依存するかどうかをテストする述語です。これは現在のフレームに対して行う必要があります。式は、すでにテスト対象の変数に依存する値を持つ変数を含んでいるかもしれませんからです。`depends-on?` の構造は木に対する簡単な再帰探索となり、その中で必要に応じて変数の値を置き換えていくことになります。

```

(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で重要な問題のひとつは、与えられたパターンをチェックする際に、無関係なデータベースのエントリを検査することができるだけ避けられるようにすることです。ここでのシステムでは、すべての表明をひとつの大きなストリームに格納するのに加えて、`car` が定数記号であるすべての表明を、その記号によって索引づけされたテーブルの中に別々のストリームとして格納します。パターンにマッチするかもしれない表明を取り出すときは、まずパターンの `car` が定数記号かどうかチェックします。

定数記号であれば、格納された表明の中で同じ car を持つものを返します(それをマッチャでテストすることになります)。パターンの car が定数記号でなければ、格納された表明をすべて返します。もっと賢いやり方としては、フレーム内の情報を活用したり、パターンの car が定数記号でない場合にも最適化を試みたりといったことも考えられます。索引づけの基準(carを使うこと、定数記号の場合のみを扱うこと)はプログラム内に組み込まないようにして、代わりにその基準を具体化した述語とセレクタを呼び出すことにします。

```
(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検索します。何も格納されていないければ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)))
```

規則も同じように規則の結論の car によって格納されます。しかし、規則の結論は任意のパターンであるため変数を含むことがあるという点が表明とは違います。car が定数記号であるパターンは、結論が同じ car を持つ規則に加えて、結論が変数で始まる規則にもマッチする可能性があります。そのため、car が定数記号であるパターンにマッチするかもしれない規則を取り出す場合は、結論がそのパターンと同じ car を持つ規則に加えて、結論が変数で始まる規則もすべて取り出します。このために、結論が変数で始まるすべての規則をテーブル内の独立したストリームに格納します。索引には記号?を使います。

```
(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` が表明と規則をデータベースに追加するのに使います。それぞれの項目は、条件を満たせば索引に格納され、それからデータベース内のすべての表明と規則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))
```

実際に表明または規則を格納する際には、索引づけが可能かどうかチェックします。可能であれば、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)))))))

```

以下の三つの手続きは、データベースの索引の使い方を定義するものです。パターン(表明または規則の結論)は、先頭が変数または定数記号であればテーブルに格納されます。

```

(define (indexable? pat)
  (or (constant-symbol? (car pat))
      (var? (car pat))))

```

パターンが格納されるテーブルのキーは、?(パターンの先頭が変数である場合)またはパターン先頭の定数記号です。

```

(define (index-key-of pat)
  (let ((key (car pat)))
    (if (var? key) '? key)))

```

パターンが定数記号で始まるのであれば、パターンにマッチするかもしれない項目を取得するために索引が使われます。

```
(define (use-index? pat) (constant-symbol? (car pat)))
```

Exercise 4.70: 手続き add-assertion! と add-rule! 内の let による束縛の目的は何か。以下に示す add-assertion! の実装はどこが間違っているだろうか。ヒント：Section 3.5.2での1の無限ストリームの定義を思い出そう：(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 ストリーム演算

クエリシステムでは、Chapter 3で紹介していないいくつかのストリーム演算を使っています。

`stream-append-delayed`, `interleave-delayed` は `stream-append`, `interleave` (Section 3.5.3) と似ていますが、(Section 3.5.4 の `integral` のように) 遅延化された引数を取るというところが違います。これはループを先送りすることができます (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、フレームのストリームに手続きをマップして結果となる複数のフレームのストリームを結合するためにクエリ評価器全体を通して使われるのですが、これは Section 2.2.3 で通常のリストに対して導入した `flatmap` 手続きのストリーム版です。しかし、`stream-flatmap` は通常の `flatmap` と違って、単純にストリームを足していくのではなく、結果を互い違いに挟み込むというプロセスによって集積します。

(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))))))

評価器は、次の単純な手続きによって要素ひとつからなるストリームを作るこ
ともあります。

(define (singleton-stream x)
  (cons-stream x the-empty-stream))
```

4.4.4.7 クエリ構文手続き

`type` と `contents` は `qeval`(Section 4.4.4.2) によって使われるもので、ある特殊形式がその `car` の記号によって識別されるということを規定します。これらはエラーメッセージ以外はSection 2.4.2の `type-tag` と `contents` 手続きと同じものです。

```
(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)))
```

以下の二つの手続きはSection 4.4.4.1の `query-driver-loop` によって使われるもので、規則と表明が (`assert! <rule-or-assertion>`) の形式の式によってデータベースに追加されるということを規定します。

```
(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))
```

以下の三つの手続きは規則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-syntax-process を呼び、?symbol という形を持つ式のパターン変数を (? symbol) という内部形式に変形します。つまり、(job ?x ?y) のようなパターンは、実際には内部ではシステムによって (job (? x) (? y)) と表現されているということです。これによってクエリ処理の効率が向上します。こうすると、式がパターン変数かどうかシステムがチェックする際に、式の car が記号? かどうかチェックすればよく、記号から文字を抽出しないですむことになるからです。構文変形は以下の手続きによって実行されます。⁸⁰

⁸⁰ほとんどの Lisp システムでは、ユーザが reader macro character (読み込みマクロ文字) を定義することによって、通常の read 手続きにそのような変形を行わせることができます。クオート式はすでにこのような扱いになっています。読み込み手続きは、評価器が式を見る前に、'expression を自動的に (quote expression) に変形します。同じようにして、?expression が (? expression) と変形されるようにすることもできるところです。しかし、ここではわかりやすさのために変形手続きを明示的に入れました。

expand-question-mark と contract-question-mark は、名前に string を含むいくつかの手続きを使っています。これらは Scheme の基本手続きです。

```

(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)))

```

変数をこのように変形すると、パターン内の変数は?で始まるリストになります。定数記号(データベースの索引づけのために識別する必要がある。Section 4.4.4.5)はただの記号です。

```

(define (var? exp) (tagged-list? exp '?))
(define (constant-symbol? exp) (symbol? exp))

```

規則適用時には、ユニークな変数が構築されます(Section 4.4.4.4)。規則適用のユニーク識別子は数値で、規則適用のたび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))))

```

query-driver-loopが答えを表示するためにクエリを具体化する際には、以下の手続きを使って未束縛のパターン変数をすべて表示用の形式に戻します。

```

(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

フレームは束縛のリストとして表現されます。束縛は変数と値のペアです。

```

(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、simple-query と disjoin の手続き (Section 4.4.4.2) がなぜ以下のような定義ではなく、明示的な delay 演算を用いて実装されているのだろうかと不思議に思っている。

```

(define (simple-query query-pattern frame-stream)
  (stream-flatmap
    (lambda (frame)
      (stream-append
        (find-assertions query-pattern frame)
        (apply-rules query-pattern frame)))
    frame-stream))
(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
    the-empty-stream
    (interleave
      (qevel (first-disjunct disjuncts)
        frame-stream)

```

```
(disjoin (rest-disjuncts disjuncts)
         frame-stream))))
```

これらの簡単な定義を使った場合に望ましくないふるまいを引き起こすクエリの例を挙げよ。

Exercise 4.72: `disjoin` と `stream-flatmap` は、なぜそれらを単純につなげていくのではなく、ストリームを互い違いに挟み込むようしているのか。なぜ相互配置のほうがよりうまくいくのかを示す例を挙げよ。(ヒント: Section 3.5.3ではなぜ `interleave` を使ったのだろうか)

Exercise 4.73: なぜ `flatten-stream` は明示的に `delay` を使っているのだろうか。以下のように定義するとどのような問題があるのだろうか。

```
(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、`negate`, `lisp-value`, `find-assertions` の中で簡単なほうの `stream-flatmap` を使うことを提案した。彼女は、これらの場合にフレームのストリームにマップされる手続きが作るのは常に空ストリームか単一要素のストリームであるため、これらのストリームを組み合わせるのに互い違いに挟み込んでいく必要はないと思った。

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 このような変更を加えると、クエリシステムのふるまいは変わるだろうか。

Exercise 4.75: クエリ言語に `unique` と呼ばれる新しい特殊形式を実装せよ。`unique` は、指定されたクエリを満たす項目がデータベース中にちょうどひとつだけある場合に成功する。例えば、

```
(unique (job ?x (computer wizard)))
```

この式は、一項目からなる次のストリームを表示する。

```
(unique (job (Bitdiddle Ben) (computer wizard)))
```

これは、Ben が唯一のコンピュータウィザードであるからだ。また、

```
(unique (job ?x (computer programmer)))
```

この式は空ストリームを表示する。コンピュータプログラマは二人以上いるからだ。さらに、

```
(and (job ?x ?j) (unique (job ?anyone ?j)))
```

この式は、それをしている人が一人しかいない職位と、それをしている人たちをすべて表示する。

`unique` の実装は二つの部分に分かれる。一つ目の部分はこの特殊形式を扱う手続きを書くためのもので、二つ目の部分は `qeval` がこの手続きにディスパッチするようにするためのものだ。`qeval` はディスパッチをデータ主導によって行うため、二つ目の部分は簡単だ。手続きの名前が `uniquely-asserted` であれば、次のように書けば言い。

```
(put 'unique 'qeval uniquely-asserted)
```

ここで、`qeval` は型 (`car`) が記号 `unique` であるすべてのクエリをこの手続きにディスパッチする。本当の問題は手続き `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)` とフレームのストリームを受け取る。次に、ストリームのそれぞれのフレームに対し、`qeval` を使って与えられたクエリを満たすフレームの拡張すべてのストリームを得る。ちょうどひとつの項目を持っているストリーム以外は取り除く。残ったストリームが返され、それらがひとつの巨大なストリームに集積されて、`unique` クエリの結果になる。これは特殊形式 `not` の実装に似ている。

ちょうど一人の部下を持つ人をすべて列挙するクエリを作り、実装をテストせよ。

Exercise 4.76: ここでのシステムでは、`and` をクエリの直列の組み合わせ (Figure 4.5) として実装している。この実装はエレガントだが、`and` の二つ目のクエリを処理する際に一つ目のクエリが生成したフレームそれぞれについてデータベースを走査しなければならないため非効率だ。データベースが n 個の要素を持っているとして、典型的なクエリが n に比例した数 (n/k 個とする) の出力フレームを生成するとすると、一つ目のクエリによって生成されたフレームそれぞれに対してデータベースを走査するためには n^2/k 回パターンマッチャを呼び出す必要がある。別のアプローチとしては、`and` の二つの節を別々に処理し、互いに矛盾しない出力フレームのペアをすべて探すというものもある。それぞれのクエリが n/k 個の出力フレームを生成するとすると、矛盾テストは n^2/k^2 回実行する必要がある。つまり、必要なマッチの回数が現在の手法よりも k 倍少ないということになる。

この戦略による `and` の実装を考えよ。実装する手続きは入力として二つのフレームを取り、それらのフレームの中の束縛が互いに矛盾しないか確認し、矛盾しないのであれば束縛の二つの集合をマージしたフレームを生成する。この演算はユニフィケーションに似ている。

Exercise 4.77: Section 4.4.3では、変数が束縛されていないフレームに `not` や `lisp-value` といったフィルタリング演算を適用すると、クエリ言語が“間違った”答えを返すことがあるということを学んだ。この欠陥を直す方法を考えよ。ひとつの考え方としては、フィルタリングを“遅延化された”やり方で実行するというものがある。そのためには、フレームにフィルタリングを行うという“約束”を追加し、その約束は演算が可能となる十分な数の変数が束縛されたら実行することにする。フィルタリングの実行は、ほかのすべての演算が実行されてからにすることも考えられる。しかし、効率のために、生成する中間フレームを減らせるように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として実装するのではなく、Section 4.3の評価器によって実装する非決定性プログラムとして再設計せよ。このアプローチでは、それぞれのクエリは(すべての回答のストリームではなく) ひとつの回答を生成し、ユーザは `try-again` を入力することによって回答の続きを見ることがで

きることになる。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含まれていたということに気づくはずだ。しかし、それと同時に、新しいクエリ言語のふるまいがここまでで実装したクエリ言語のふるまいと微妙な違いがあるということにも気づくだろう。その違いを示すような例を見つけることができるだろうか。

Exercise 4.79: Section 4.1で Lisp 評価器を実装したとき、局所環境を使って別々の手続きの引数同士で名前が衝突しないようにする方法を学んだ。例えば、以下を評価する際に、

```
(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` が混同されることはない。それぞれの手続きの本体は、局所変数の束縛を持たせるために特別に構築した環境の中で評価されるからだ。クエリシステムでは、規則適用時の名前衝突を避けるために違う戦略を使った。規則適用のたびに、ユニークであることが保証された新しい名前に変数の名前を変えている。Lisp 評価器で同じような戦略を使うとすると、局所環境をなくし、単純に手続き適用のたびに手続きの本体の中の変数をリネームするということになるだろう。

クエリ言語で、変数のリネームではなく、環境を使う規則適用の手法を実装せよ。その環境構造の上に、ブロック構造手続きの規則版のような、大規模システムを扱うためのクエリ言語の構築物を作ることはできるだろうか。これらの問題と、問題解決の手法として文脈の中で推論を行うという問題（例えば、“もし P が真であると仮定すると、 A と B が推論できる”）とを関連づけることはできるか。（この問題は自由回答だ。うまい解答ができれば博士号が取れるだろう。）

5

レジスタマシンによる計算

私の目的は、天の機械は神聖なものや生きたものなどではなく、一種の時計仕掛けであるということを示すことです（時計が魂を持つと信じる人は、その栄光を時計の作り手ではなく、その仕掛けに帰しているのです）。すべての多様な動きは非常に単純で物質的な力によって引き起こされていて、それは時計のすべての動きがひとつの重りに引き起こされているのと同じことなのです。

—Johannes Kepler (Herwart von Hohenburgへの手紙, 1605)

この本は、プロセスについて考察し、Lisp で書かれた手続きによってプロセスを記述するというところから始めました。これらの手続きの意味を説明するために、いくつかの評価モデルを使いました。[Chapter 1](#)の置換モデル、[Chapter 3](#)の環境モデル、[Chapter 4](#)のメタ循環評価器です。特に、メタ循環評価器を詳しく見ていくことで、Lisp 的言語の解釈についての謎を解くことができました。しかし、メタ循環評価器もひとつの重要な問題に答えていません。これは、この評価器が Lisp システムの制御メカニズムを明らかにしていないことによります。例えば、部分式の評価がどのようにしてその値を使う式に値を返すのか、一般的な再帰関数は再帰プロセスを生成するのにある種の再帰関数は反復プロセスを生成する（つまり、定数空間で評価される）のはどのようにして実現しているのかといったところを、この評価器は説明してくれません。これらの疑問に対する答えがないのは、メタ循環評価器がそれ自身 Lisp プログラム

であり、基層となる Lisp システムの制御構造を引き継いでいるからです。Lisp 評価器の制御構造についてより完全な記述をするためには、Lisp そのものよりも基本的なレベルで見ていく必要があります。

この章では、従来式コンピュータのステップごとの演算によってプロセスを記述していきます。そのようなコンピュータ、別名 *register machine*(レジスタマシン) は、*register*(レジスタ) と呼ばれる一組の固定された記憶素子の中身を操作する *instruction*(命令) を順に実行していきます。典型的なレジスタマシンの命令は、レジスタの中身に基本的な演算を適用し、その結果をほかのレジスタに割り当てるというものです。レジスタマシンにより実行されるプロセスをこのように記述すると、従来式コンピュータの“機械語”に非常によく似ているように見えます。しかし、ここではある特定のコンピュータの機械語を扱うのではなく、いくつかの Lisp 手続きについて調べ、それぞれの手続きを実行するための個別のレジスタマシンを設計していくことにします。そのため、課題に対する私たちのアプローチは、機械語のプログラマというよりも、ハードウェア設計者としての視点によるものとなります。レジスタマシンを設計する際には、再帰のような重要なプログラム構造を実装する仕組みを開発していきます。また、レジスタマシンの設計を記述するための言語も導入します。Section 5.2 では、私たちが設計するマシンをこの言語による記述法を利用してシミュレートする Lisp プログラムを実装します。

私たちのレジスタマシンの基本演算は、ほとんどが非常に単純なものです。例えば、ある演算は二つのレジスタから取ってきた値を足し、答えを出して別のレジスタに格納します。そのような演算は簡単に記述したハードウェアで実行できます。しかし、リスト構造を扱うには、複雑な記憶領域割り当て機構が必要となる `car`, `cdr`, `cons` といったメモリ演算も使います。Section 5.3 では、より初步的な演算によってこれらを実装する方法を学びます。

Section 5.4 では、簡単な手続きをレジスタマシンとして定式化する経験を積んだ後に、Section 4.1 のメタ循環評価器で説明したアルゴリズムを実行するマシンを設計します。これは、評価器の制御メカニズムに対する明示的なモデルを提供し、Scheme の式の解釈がどのように行われるかについての理解のギャップを埋めてくれるでしょう。Section 5.5 では、評価器レジスタマシンのレジスタと演算によって直接実行可能な命令列に Scheme プログラムを翻訳する単純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、まずレジスタマシンの *data path*(データパス) (レジスタと演算) と、それらの演算を正しく並べる *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について示すために、二つの整数の最大公約数 (GCD) を求めるために使われるユークリッドのアルゴリズムについて見ていきましょう。[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よって実行可能です。プロセスは以下の手続きによって規定されます。

```
(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マシンは、 a と b という二つの数値を管理する必要があります。ですので、ここではそれらの数値が a と b という二つのレジスタに格納されているということにします。必要となる基本演算は、レジスタ b の値が 0 かどうかを確認すること、レジスタ a の中身をレジスタ b の中身で割った剰余を求ることの二つです。剰余を求める演算は複雑なプロセスですが、当面は剰余を求める基本装置があると仮定します。GCD アルゴリズムのそれぞれのサイクルでは、レジスタ b の中身によってレジスタ a の中身を置き換えることと、 a の古い中身を b の古い中身で割った剰余によって b の中身を置き換えることが必要になります。これら二つの置換を同時にを行うことができれば都合がいいのですが、私たちのレジスタマシンのモデルでは、各ステップで新しい値を割り当てることができるのはレジスタひとつだけだということになります。二つの置換を行うには、私たちのマシンは三つ目の“一時的”レジスタを使います。これは t という名前にします。(まず、剰余を t に入れます。次に、 b の中身を a に入れます。最後に、 t に格納されている剰余を b に入れます。)

このマシンで必要となるレジスタと演算は、[Figure 5.1](#)のようなデータパス図によって示すことができます。この図では、レジスタ (a, b, t) は長方形で表しています。レジスタに対する値の割り当ては、それぞれ頭の後ろに x のある矢印によって示しています。矢印はデータの代入元からレジスタという方向になっています。この x は、押すと代入元から指定されたレジスタに値を“流す”ボタンとることができます。それぞれのボタンの横にあるラベルは、それぞれのボタンを呼ぶときに使う名前です。この名前は任意で、覚えやすいよう

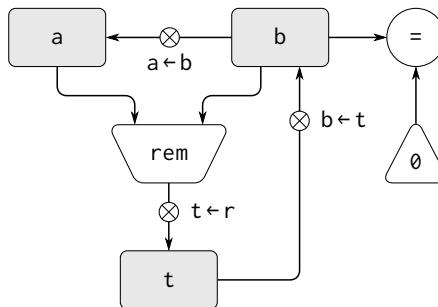


Figure 5.1: GCD マシンのデータパス

に名前をつけることができます(例えば、 $a \leftarrow b$ はボタンを押すとレジスタ b の中身を a に割り当てるということを示しています)。レジスタに対する代入元は($a \leftarrow b$ の代入のように)別のレジスタであったり、($t \leftarrow r$ の代入のように)演算の結果であったり、定数(変更できない組み込みの値で、データパス図では定数を持つ三角形で表します)であることもあります。

定数やレジスタの中身から値を計算する演算は、データパス図では演算の名前を持つ台形で表します。例えば、Figure 5.1で rem というラベルをつけた箱は、その箱に接続された a レジスタと b レジスタの中身の剰余を求める演算を表します。入力レジスタや入力定数から箱に向かって(ボタンのない)矢印があり、また演算の出力値からレジスタに対しても矢印が出ています。テストは、テストの名前を持つ円で表します。例えば、私たちの GCD マシンには、レジスタ b の中身がゼロかどうかをテストする演算があります。テストにも入力レジスタや入力定数からの矢印はありますが、出力の矢印はありません。値は、データパスではなくコントローラによって使われます。全体として、データパス図はこのマシンに必要なレジスタと演算と、またそれらをどう接続するべきかを示しています。矢印を配線と考え、X ボタンをスイッチだと考えると、データパス図は電子部品によって構築できる機械の配線図にそっくりです。

データパスが実際に GCD を求めるには、ボタンが正しい順序で押されなければいけません。この順序については、Figure 5.2 に示すようなコントローラ図によって記述します。コントローラ図の要素は、データパスの部品がどう操作されるべきかを示しています。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特定し、矢印はあるステップから次のステップに移る順

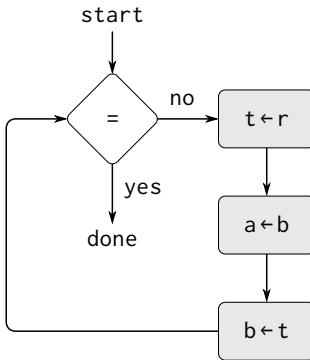


Figure 5.2: GCD マシンのコントローラ

序を示しています。図の中の菱形は判定を表しています。菱形に対応するデータパスのテストの値によって、二つの矢印のどちらかに進むことになります。コントローラは、物理的なアナロジーを使って解釈することができます。コントローラ図をビー玉が転がっている迷路として考えてみましょう。ビー玉が箱に入ると、箱につけられた名前のデータパスのボタンを押します。ビー玉が ($b = 0$ のテストのような) 判定ノードに入ると、そのノードの示すテストの結果によって決まるルートに進みます。データパスとコントローラは、二つ合わせて GCD を計算するマシンを完全に記述していることになります。レジスタ a と b に値をセットしてから、コントローラ (転がるビー玉) を `start` という印の場所から始めます。コントローラが `done` に到達するときには、GCD の値がレジスタ a の中に入っています。

Exercise 5.1: 以下の手続きによって規定されるような、反復アルゴリズムによって階乗を計算す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図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 レジスタマシンの記述言語

データパス図とコントローラ図は、GCD のような単純なマシンを表現するのには向いています。しかし、Lisp インタプリタのような大きなマシンを記述しようと思うと大変です。複雑なマシンを扱えるように、データパス図とコントローラ図のすべての情報をテキスト形式で表現する言語を作ることにします。まずは、図を直接反映した表記から始めます。

マシンのデータパスは、レジスタと演算を記述することによって定義します。レジスタを記述するには、名前をつけ、そのレジスタへの代入を制御するボタンを規定します。これらのボタンそれぞれにも名前をつけ、そのボタンの制御によってレジスタに入るデータの入力元を規定します（入力元はレジスタ、定数、演算です）。演算を記述するには、名前をつけ、その入力元（レジスタまたは定数）を規定します。

マシンのコントローラは、*instruction*（命令）と *label*（ラベル）の列として定義します。ラベルは、列の *entry point*（エントリポイント）を示します。命令は次のうちのいずれかです。

- データパスボタンの名前。その名前のボタンを押してレジスタに値を割り当てる。（これはコントローラ図の箱に対応する。）
- test**（テスト）命令。規定されたテストを実行する。
- コントローララベルの指す場所への条件分岐（branch 命令）。条件分岐は直前のテストの結果に基づいて行われる（テストと分岐は、二つ合わせてコントローラ図の菱形に対応する）。テストが偽であれば、コントローラは命令列の次の命令に進む。真であれば、コントローラはそのラベルの次の命令に進む。
- 無条件分岐（goto 命令）。実行を移すべきコントローララベルを指定する。

マシンはコントローラの命令列の先頭からスタートし、実行が列の終わりにたどり着くと停止します。分岐が制御の流れを変えるとき以外は、命令は並べられている順に実行されます。

Figure 5.3: ↓ GCD マシンの仕様

```
(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、この方法で記述した GCD マシンを示したものです。この例では、この記述方法の一般性を垣間見ることしかできません。GCD マシンはとても単純なケースだからです。このマシンでは、それぞれのレジスタはボタンをひとつしか持たず、それぞれのボタンとテストはコントローラに一回しか使われていません。

残念ながら、このような記述は読むのが大変です。コントローラの命令を理解するには何回もボタン名と命令名の定義に戻る必要があり、ボタンの動作について理解するには演算名の定義を参照する必要がある場合もあります。そのため、データパスの記述とコントローラの記述の情報を組み合わせて一度に見られるように表記法を变形します。

この形式による記述を作るには、適当に決めていたボタンの名前と演算の名前を、ふるまいの定義に基づくものに変更します。つまり、(コントローラで)“ボタン $t <- r$ を押せ”と言ったり、(データパスで)“ボタン $t <- r$ は rem 演算の値をレジスタ t に代入する”、“ rem 演算の入力はレジスタ a とレジスタ b の中身である”と言ったりする代わりに、今後は(コントローラで)“レジスタ a とレジスタ b の中身に対する rem 演算の値をレジスタ 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”と言って、それとは別に(データパスの中で)“= テストはレジスタ b の中身と定数 0 に対して実行する”と言ったりする代わりに、今後は“= テストをレジスタ b の中身と定数 0 に対して実行せよ”と言うことにします。データパスの記述は取り除き、コントローラの命令列だけが残ります。その結果、GCD マシンは以下のような記述になります。

```
(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)
```

この形式による記述は、Figure 5.3で示したものよりも読みやすくなっていますが、欠点もあります。

- この形式は、大きなマシンにはより冗長になります。データパス要素がコントローラの命令列に出てくるたびに、要素の完全な記述が繰り返されることになるからです(これは GCD の例では問題になりません。演算とボタンはそれぞれ一度しか使われないからです)。さらに、データパスの記述を繰り返すことによって、マシンの実際のデータパス構造がわかりにくくなっています。大きなマシンでは、レジスタ、演算、ボタンがそれぞれいくつあって、それらがどのように相互接続されているかすぐにはわかりません。
- マシンの定義に含まれるコントローラの命令は Lisp 式に似ているため、どんな Lisp 式でもいいわけではないということを忘れてしまいがちです。コントローラの命令はマシンの演算として正当なものしか記述できません。例えば、演算が直接扱うことができるのは定数とレジスタの中身だけです。ほかの演算の結果を扱うことはできません。

これらの欠点はありますが、この章を通して、このレジスタマシン用言語を使うことにします。ここでは、データパスの要素や接続よりも、コントローラを理解することに重点を置いています。しかし、実際にマシンを設計する際にはデータパスの設計が非常に重要だということは意識しておく必要があります。

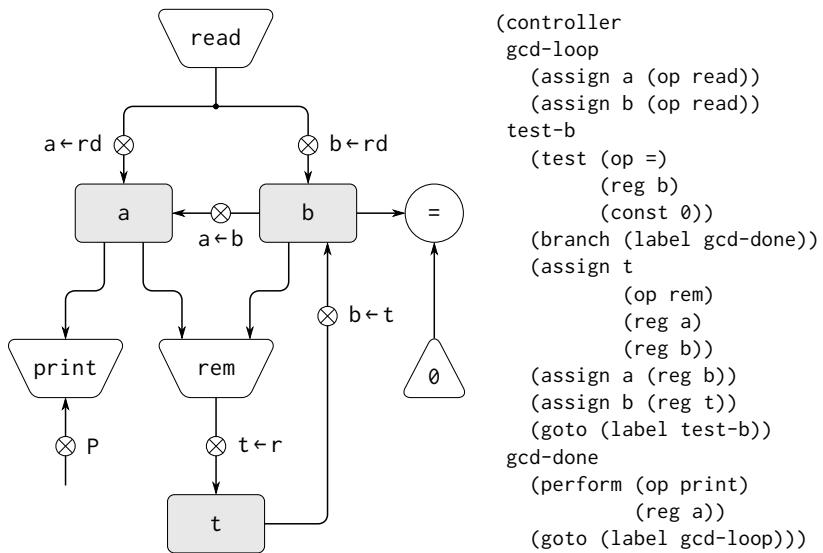


Figure 5.4: 入力を読み込み結果を表示する GCD マシン

Exercise 5.2: レジスタマシン言語を使って Exercise 5.1 の反復階乗マシンを記述せよ。

アクション

GCD マシンを修正して、GCD を求めたい数値の組を入力し、端末に答えが表示されるようにしてみましょう。読み込みや表示の機能を持つマシンの作り方については考えずに、(Scheme で `read` や `display` を使うときのように) それらは基本演算として使えるようになっていると仮定します。¹

`read` は、レジスタに格納できる値を返すという意味で、これまで使ってきた演算に似ています。しかし、`read` はレジスタから入力を取ってくるわけではありません。値は、私たちが設計しているマシンの外側で起きる何かによって

¹ この仮定はかなりの複雑さをごまかしています。Lisp システムでは、読み込みと表示を機能させるために実装のかなりの部分が費やされるのが普通です。

決まります。ここでは、マシンの演算がそのようなふるまいを持つことを許すことにして、図や記述の中では値を計算するほかの演算と同じように `read` を表記します。

一方、`print` はこれまで使ってきた演算と根本的に違います。`print` はレジスタに格納する値を生成しません。この演算は効果を持っていますが、設計しているマシンの一部に対するものではありません。この種の演算は *action*(アクション) と呼ぶことにします。データパス図では、アクションは値を計算する演算と同じように、つまりアクション名を中に含む台形として表現します。アクションの箱には任意の入力から矢印が入っていきます。また、アクションにはボタンも関連づけます。ボタンを押すとアクションが起こります。コントローラにアクションボタンを押させるには、`perform(実行)` という新しい種類の命令を使います。つまり、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は次の命令によって表現されます。

```
(perform (op print) (reg a))
```

新しい GCD マシンのデータパスとコントローラを Figure 5.4 に示します。回答を表示した後は、マシンを停止させるのではなく再開させ、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すようにしています。この構造は、Chapter 4 のインタプリタで使ったドライバループに似ています。

5.1.2 マシン設計における抽象化

これから先で、実際には非常に複雑な“基本”演算を含めたマシンを設計するということをよく行います。例えば、Section 5.4 と Section 5.5 では、Scheme の環境操作を基本演算として扱います。このような抽象化を行うことによって、マシンの一部の細部を無視して設計のほかの側面に集中することができるという意味で、抽象化は役に立ちます。しかし、多くの複雑な部分に蓋をしたからといって、マシンの設計が非現実的だということではありません。複雑な“基本演算”は、より簡単な演算に置き換えることがいつでもできます。

GCD マシンの場合を考えてみましょう。このマシンは、レジスタ `a` とレジスタ `b` の中身の剰余を計算して結果をレジスタ `t` に割り当てる命令を持っています。もし基本演算としての剰余演算を使わずに GCD マシンを構築したければ、例えば引き算のようなより単純な演算によって剰余を計算する方法を規定する必要があります。実際に、この方法で剰余を求める Scheme の手続きは次のように書くことができます。

```
(define (remainder n d)
  (if (< n d) n (remainder (- n d) d)))
```

ですので、GCD マシンのデータパス内の剰余演算は、引き算演算と比較テストで置き換えることができます。Figure 5.5に、そのように詳細化し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

```
(assign t (op rem) (reg a) (reg b))
```

という命令は、Figure 5.6に示すように、ループを含む命令列によって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: Section 1.1.7に示したニュートン法によって平方根を求めるマシンを設計せよ。

```
(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
```

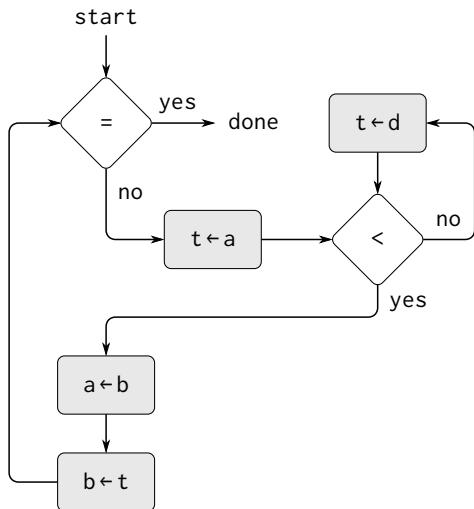
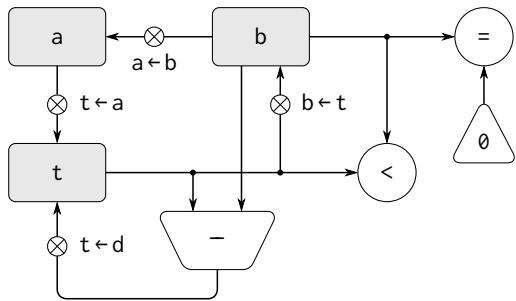


Figure 5.5: 詳細化した GCD マシンのデータパスとコントローラ

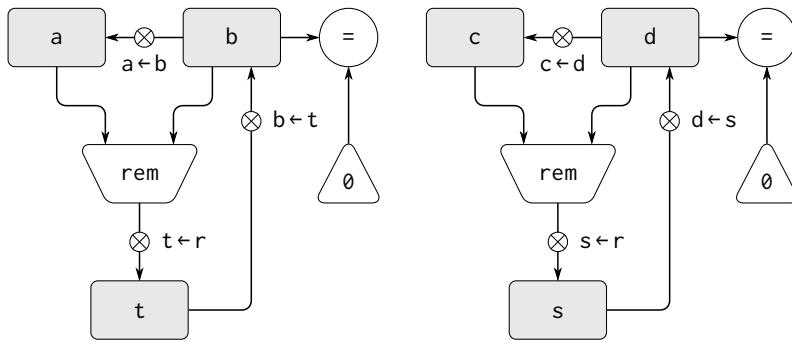
```
(sqrt-iter (improve guess)))  
(sqrt-iter 1.0))
```

最初は `good-enough?` と `improve` 演算が基本命令として使えると仮定して設計せよ。次に、それらを算術演算によって展開する方法を示せ。データパス図を描き、レジスタマシン言語でコントローラ定義を書くことによって、それぞれのバージョンの `sqrt` マシンの設計を記述せよ。

5.1.3 サブルーチン

計算を行うマシンを設計する際には、部品を複製するのではなく、計算のいろいろな部分で共有できるようにしたいということがよくあります。二つの GCD 計算を持つマシンについて考えてみましょう。ひとつはレジスタ `a` とレジスタ `b` の中身の GCD を求めるもので、もうひとつはレジスタ `c` とレジスタ `d` の GCD を求めるものとします。この場合、まず基本演算としての `gcd` 演算があると仮定し、それから二つの `gcd` の実体をより基本的な演算を使って展開するということが考えられます。[Figure 5.7](#) は、その結果としてできるマシンのデータパスの GCD 部分だけを示したものです。それらがマシンの残りの部分とどのように接続されているかは示していません。この図には、マシンのコントローラ命令列の中でそれらに対応する部分も示しています。

このマシンには剰余演算の箱が二つと等価テストの箱が二つあります。重複する部品が、例えば剰余の箱のように複雑なものであれば、これはマシンを構築するのに経済的な方法とはいえないでしょう。外側のマシンの計算に影響しない形でどちらの GCD の計算にも同じ部品を使うようにすれば、データパス部品の重複を避けることができます。もしコントローラが `gcd-2` に来たときにレジスタ `a` とレジスタ `b` の値が必要ないのであれば（または、これらの値をほかのレジスタに退避できるのであれば）、一つ目の GCD を求めるときと同じく、二つ目の GCD を求めるときにも、レジスタ `c` とレジスタ `d` を使う代わりにレジスタ `a` と `b` を使うようにマシンを変更することができます。これをすると、[Figure 5.8](#) に示すようなコントローラ命令列が得られます。



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: 二つの GCD の計算を持つマシンのデータパスとコントローラ命令列の一部

Figure 5.8: ↓ 別々の二つの GCD 計算に同じデータパスの部品を使うマシンのコントローラ命令列の一部

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

```

重複するデータパス部品は削除しました(これでデータパスはまたFigure 5.1のようになります)、コントローラのほうにはエントリポイントのラベルが違うだけの二つの GCD 列があります。これら二つの列は、*gcd subroutine*(サブルーチン) —末尾でメイン命令列の正しい場所に分岐して戻るというもの— というひとつの列への分岐によって置き換えたほうがいいでしょう。これは、次のように実現できます。まず、*gcd* に分岐する前に、*continue* という特別レジスタに(0 や 1 といった)識別するための値を置きます。*gcd* サブルーチンの末尾では、*continue* レジスタの値によって、*after-gcd-1* または *after-gcd-2* に帰ります。Figure 5.9 は、結果としてできるコントローラ命令列の中でこれに関係する部分を示しています。これには *gcd* 命令列のはひとつしかありません。

Figure 5.9: ↓ Figure 5.8でコントローラ命令列の重複を防ぐため
continue レジスタを使う

```

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))

...
;; 最初に必要になる場所から gcd に分岐する前に
;; レジスタ continue に 0 を置く
(assign continue (const 0))
(goto (label gcd))

```

```

after-gcd-1
...
;; 二回目に gcd を使う前にはレジスタ continue に 1 を置く
(assign continue (const 1))
(goto (label gcd))
after-gcd-2

```

このやり方は、小さな問題を扱うにはいいですが、コントローラ命令列の中になんかの GCD 計算があると厄介なことになるでしょう。GCD サブルーチンの後にどこから実行を続けるかを決めるためには、`gcd` を使う場所すべてについて、データパスのテストとコントローラの分岐命令を置かなければなりません。サブルーチン実装のより強力な手法は、サブルーチンが終わった後に実行を続けるべきコントローラ命令列のエントリポインツのラベルを `continue` レジスタに持たせるというものです。この戦略を実装するには、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接続が必要になります。レジスタにコントローラ命令列のラベルを代入し、その値を後から取得して指定されたエントリポインツから実行を継続するのに使うことができるようになる方法が必要です。

この機能を反映させるために、レジスタに値としてコントローラ命令列のラベルを(特別な定数として)代入できるようにレジスタマシン言語の `assign` 命令を拡張することにします。また、`goto` 命令も拡張して、定数ラベルで記述したエントリポインツだけでなく、レジスタの中身として記述されたエントリポインツからも実行を続けられるようにします。これらの新しい機構を使って、`gcd` サブルーチンの末尾には `continue` レジスタに格納された場所への分岐を置くことができるようになります。こうすると、コントローラ列は Figure 5.10 に示すようなものになります。

Figure 5.10: ↓ `continue` レジスタにラベルを代入すると Figure 5.9 の戦略を単純化・一般化できる

```

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
(goto (reg continue))
...
;; gcd を呼ぶ前に、gcd から帰る先のラベルを continue に代入
する
(assign continue (label after-gcd-1))
(goto (label gcd))
after-gcd-1
...
;; 二つ目の gcd 呼び出し、別の継続を持つ
(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であれば、継続レジスタを複数 (gcd-continue, factorial-continue のような) を使うことも考えられますし、すべてのサブルーチンでひとつの continue レジスタを共有するようにすることも考えられます。共有したほうが経済的ですが、別のサブルーチン (sub2) を呼び出すサブルーチン (sub1) がある場合には注意する必要があります。sub1 が sub2 を呼び出すために continue を設定する前に、continue の中身を何かほかのレジスタに保存しておくということをしていなければ、sub1 が終わった時点でどこに行くべきかわからなくなってしまいます。再帰を扱うために次の節で開発するメカニズムは、この入れ子のサブルーチン呼び出しという問題にもよりよい解決法を提供してくれます。

5.1.4 再帰の実装にスタックを使う

ここまでに示した考え方を使えば、プロセスのそれぞれの状態変数に対応するレジスタを持つレジスタマシンを規定することによって、任意の反復プロセスを実装することができます。このマシンはレジスタの中身を変更しながらコントローラループを繰り返し実行し、何らかの終了条件が満たされると終了します。コントローラ命令列のそれぞれの位置で、(反復プロセスの状態を表現する) マシンの状態は、レジスタの中身 (状態変数の値) によって完全に決定されます。

しかし、再帰プロセスを実装するにはメカニズムを追加する必要があり

ます。階乗を計算する以下の再帰的な手法について考えてみましょう。これはSection 1.2.1で詳しく調べたものです。

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

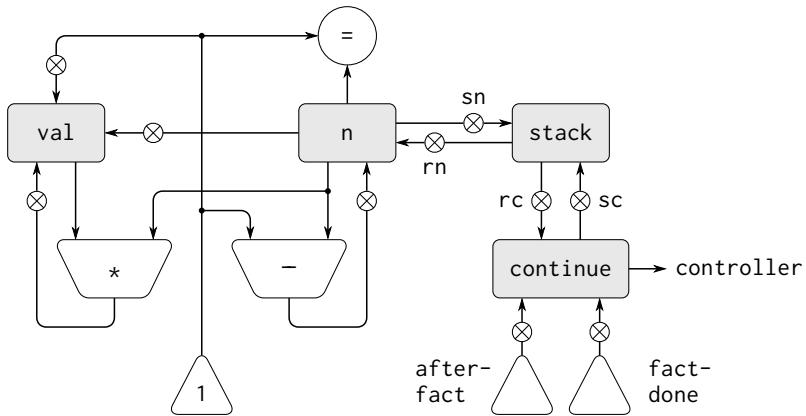
この手続きを見るとわかるように、 $n!$ を計算するには $(n - 1)!$ を計算する必要があります。以下の手続きをモデル化した GCD マシンでも、

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

やはり同じように、別の GCD を計算しなければなりませんでした。しかし、`gcd` 手続きと `factorial` 手続きを重要な違いがあります。`gcd` 手続きを、元の計算を新しい GCD の計算に簡約するのに対し、`factorial` では別の階乗を部分問題として求める必要があります。GCD の場合は、新しい GCD 計算の答えが元の問題の答えになります。次の GCD を計算するには、単純に新しい引数を GCD マシンの入力レジスタに置いて同じコントローラ命令列を実行することによってマシンのデータパスを再利用します。マシンが最後の GCD 問題を解き終わると、計算全体が終わったことになります。

階乗(に限らず、任意の再帰プロセス)では、新しい階乗の部分問題の答えは元の問題の答えではありません。 $(n - 1)!$ について得られた値は、 n をかけて最終的な答えを得る必要があります。GCD の設計の真似をして、レジスタ `n` をデクリメントして階乗マシンを再び動かすことで部分問題を解こうとしても、結果にかけるべき古い `n` の値はもうありません。ですので、部分問題を解くには二つ目の階乗マシンが必要です。この二つ目の階乗計算はそれ自身階乗の部分問題を持っていて、それには三つの階乗マシンが必要です。ずっとこの繰り返しです。それぞれの階乗マシンは中に別の階乗マシンを持っているので、マシン全体は同じようなマシンの無限の入れ子を持つことになり、有限の決まった数の部品からは構築できないということになります。

それでも、入れ子になったそれぞれのマシンに対して同じ部品を使うようにできれば、階乗プロセスをレジスタマシンとして実装することができます。具体的には、 $n!$ を計算するマシンは $(n - 1)!$ を計算する部分問題、 $(n - 2)!$ の部分問題...にも同じ部品を使うようにするということです。これは実現できそうです。階乗プロセスは計算を行うために限りなくたくさんの同じマシンのコピーが必要だということになっていますが、ある特定の時間にアクティブである必要があるのはこれらのコピーのうちのひとつだけだからです。マシンが再帰部分問題にぶつかると、主問題の作業を中断し、同じ物理部品を使い回し



```

(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

て部分問題を処理し、それから中断した計算を続けるというようにすることができます。

部分問題の中では、レジスタの中身が主問題のときと異なるものになります（ここでは、レジスタ n はデクリメントされます）。中断した計算を続けられるようにするには、マシンは部分問題を解いた後に必要になるレジスタの中身を保存して、それらを復元して中断した計算を続けられるようにする必要があります。階乗の場合、デクリメントしたレジスタ n の階乗の計算が終わったときに復元できるように、 n の古い値を保存しておくことになります。²

入れ子の再帰呼び出しの深さにあらかじめ決められた限界はないため、任意の数のレジスタの値を保存することが必要になる可能性があります。それらの値は、保存したのとは逆順に復元しなければなりません。入れ子の再帰では、最後に入った部分問題が最初に終わるからです。このため、レジスタの値の保存には、*stack*(スタック)、つまり「後入れ先出し」のデータ構造を使うことになります。レジスタマシン言語にスタックを含めることは、二つの命令を追加することによって行えます。ひとつは *save* 命令で、これはスタックに値を追加するというものです。もうひとつは *restore* 命令で、これによってスタックから値を復元します。一連の値をスタックに *save* した後に、*restore* 命令を連続して実行すると、値が逆順に取り出されます。³

スタックの助けを借りると、ひとつの階乗マシンのデータパスをそれぞれの階乗部分問題に使い回せるようになります。データパスを操作するコントローラ命令列を再利用する際にも、同じような設計上の問題があります。階乗計算を再実行するには、コントローラは反復プロセスのように単純に最初の場所までループで戻るというわけにはいきません。 $(n - 1)!$ という部分問題を解いた後に、マシンはその結果に n をかける必要があるからです。コントローラは $n!$ の計算を中断し、部分問題 $(n - 1)!$ を解き、それから $n!$ の計算を続けなければなりません。このように階乗演算を見ると、Section 5.1.3 で述べたサブルーチン機構を使うのがよさそうです。サブルーチン機構とは、部分問題を解く部分に移動した後で主問題の残りの部分を続けるために、コントローラがレジスタ *continue* を使うようにするというものでした。この考え方で、*continue* レジスタに格納されたエントリポイントに戻る階乗サブルーチンを作ることが

²ここで古い n を保存する必要はないのではないかと思われるかもしれません。 n をデクリメントして部分問題を解いたら、単純にそれをインクリメントして元の値を復元することもできます。しかし、この戦略は階乗ではうまくいくものの、一般的にはうまくいきません。レジスタの古い値が常に新しい値から計算できるとは限らないからです。

³Section 5.3 では、より基本的な演算を使ってスタックを実装する方法について見てきます。

できます。それぞれのサブルーチン呼び出しの前後では、`n` レジスタと同じように `continue` の保存・復元を行うことにします。これは、階乗計算のそれぞれの“レベル”で、同じ `continue` レジスタを使うことになるからです。つまり、階乗サブルーチンは部分問題を解くために自分自身を呼び出すときには新しい値を `continue` に入れる必要がありますが、部分問題を解くために自分を呼び出した場所に戻るために古い値が必要だということです。

Figure 5.11に、再帰 `factorial` 手続きを実装するマシンのデータパスとコントローラを示します。このマシンは、ひとつのスタックと、`n`, `val`, `continue` という三つのレジスタを持っています。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つけずに、スタック演算ボタン(レジスタを保存する `sc` と `sn`、レジスタを復元する `rc` と `rn`)だけ名前をつけています。マシンを操作するには、階乗を求める数をレジスタ `n` に入れ、マシンをスタートさせます。マシンが `fact-done` に着くと計算は完了で、答えはレジスタ `val` に入っています。コントローラ命令列では、`n` と `continue` がそれぞれの再帰呼び出しの前に保存され、その呼び出しから戻るときに復元されます。呼び出しから戻るのは、`continue` に格納された場所に分岐することによって実現されています。マシンがスタートするときに、最後に帰る場所が `fact-done` になるように `continue` が初期化されます。階乗計算の結果を持つ `val` レジスタは再帰呼び出しの前に保存されません。サブルーチンから帰った後には、`val` の古い中身は役に立たないためです。必要なのは、部分問題によって生成された新しい値だけです。

階乗計算には原理上は無限のマシンが必要ですが、実際にはFigure 5.11のマシンは、無限に大きくなりうるスタック以外は有限です。しかし、スタックを物理的に実装したものはどうやっても有限サイズのものになるため、このマシンで扱える再帰呼び出しの深さはそれによって縛られることになります。この階乗の実装は、スタックという拡張を加えた通常のレジスタマシンによって再帰アルゴリズムを実現する一般的な戦略を示しています。再帰部分問題にぶつかると、部分問題を解いた後に現在の値が必要となるレジスタをスタックに保存し、再帰部分問題を解き、保存したレジスタを復元して主問題の実行を続けます。`continue` レジスタは常に保存する必要があります。保存するべきレジスタがほかにあるかどうかはマシンによります。再帰計算のすべてが、部分問題を解く間に変更されるレジスタの元の値を必要とするわけではないからです (Exercise 5.4 参照)。

二重再帰

より複雑な再帰プロセスである、Section 1.2.2で紹介したフィボナッチ数の木の再帰の計算について考えてみましょう。

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

階乗と同じように、再帰フィボナッチ計算はレジスタ `n`, `val`, `continue` を持つレジスタマシンとして実装できます。このマシンは階乗マシンよりも複雑です。コントローラ命令列の中で再帰呼び出しの実行が必要となる場所が二箇所あるからです。ひとつは $\text{Fib}(n-1)$ を求めるところで、もうひとつは $\text{Fib}(n-2)$ を求めるところです。それぞれの呼び出しの準備として、後で値が必要になるレジスタを保存し、再帰的にフィボナッチ数を求める数 ($n-1$ または $n-2$) を `n` レジスタに設定し、`continue` に戻り先の主命令列のエントリポイント (それぞれ `afterfib-n-1` または `afterfib-n-2`) を代入します。それから `fib-loop` 行きます。再帰呼び出しから帰るときには、答えは `val` に入っています。Figure 5.12にこのマシンのコントローラ命令列を示します。

Figure 5.12: ↓ フィボナッチ数を計算するマシンのコントローラ

```
(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)) ; 再帰呼び出しの実行
  afterfib-n-1 ; リターン時に Fib(n-1) は val に入つて
いる
  (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 Fib( $n - 2$ ) は val に入っている
(assign n (reg val)) ; n には Fib( $n - 2$ ) が入る
	restore val) ; val には Fib( $n - 1$ ) が入る
	restore continue)

(assign val ; Fib( $n - 1$ ) + Fib( $n - 2$ )
      (op +) (reg val) (reg n))
(goto (reg continue)) ; 呼び出し元に戻る、答えは val の中
immediate-answer
(assign val (reg n)) ; 基底の場合: Fib( $n$ ) =  $n$ 
(goto (reg continue))
fib-done)

```

Exercise 5.4: 次のそれぞれの手続きを実装するレジスタマシンを規定せよ。それぞれのマシンについて、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 再帰的指数計算

```

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

```

b 反復的指数計算

```

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

```

```

(expt-iter (- counter 1)
            (* b product))))
(expt-iter n 1))

```

Exercise 5.5: 何らかの自明でない(少なくとも一回は再帰呼び出しの実行が必要となるような)入力を使って、階乗マシンとフィボナッチマシンを手作業でシミュレートせよ。実行中の重要な場所での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命令列に余分な `save` と `restore` があり、取り除くことによってマシンを高速化できることに気がついた。それらの命令はどれか。

5.1.5 命令まとめ

レジスタマシン言語のコントローラ命令は、以下のいずれかの形式をとります。 $\langle input_i \rangle$ はそれぞれ `(reg <register-name>)` か `(const <constant-value>)` のいずれかです。これらの命令は [Section 5.1.1](#) で導入したものです。

```

(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>))

```

ラベルを保存するのにレジスタを使うことについては [Section 5.1.3](#) で導入しました。

```

(assign <register-name> (label <label-name>))
(goto (reg <register-name>))

```

スタックを使う命令は [Section 5.1.4](#) で導入しました。

```

(save <register-name>)
	restore <register-name>)

```

ここまで見た(*constant-value*)の種類は数値だけですが、これから先では、文字列、記号、リストも使うことになります。

```
(const "abc") これは文字列 "abc",
(const abc) これは記号 abc,
(const (a b c)) これはリスト (a b c),
(const ()) これは空リスト
```

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よく理解するには、設計したマシンが期待通りに動くかどうかを見るためにテストを行う必要があります。設計をテストする方法のひとつとして、Exercise 5.5でやったように、コントローラの演算を手作業でシミュレートするというものがあります。しかしこれは、よほど簡単なマシンでもなければ、非常に退屈な作業になります。この章では、レジスタマシン言語で記述されたマシンのシミュレータを構築します。シミュレータは四つのインターフェイス手続きを持つ Scheme のプログラムです。一つ目のインターフェイス手続きは、レジスタマシンの記述を使ってマシンのモデル(それぞれの部品がシミュレート対象のマシンの部品に対応するデータ構造)を構築します。残り三つは、モデルを操作してマシンをシミュレートできるようにするためのものです。

```
(make-machine <register-names> <operations> <controller>)
```

この手続きは、与えられたレジスタ、演算、コントローラを持つマシンのモデルを構築し、それを返します。

```
(set-register-contents! <machine-model>
                      <register-name>
                      <value>)
```

この手続きは、与えられたマシンの(シミュレートした)レジスタに値を格納します。

```
(get-register-contents <machine-model> <register-name>)
```

この手続きは、与えられたマシンの(シミュレートした)レジスタの中身を返します。

```
(start <machine-model>)
```

この手続きは、与えられたマシンの実行をシミュレートします。実行はコントローラ命令列の最初からスタートし、命令列の末尾に着くと止まります。

これらの手続きの使い方の例として、Section 5.1.1 の GCD マシンのモデルとして `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` の一つ目の引数はレジスタ名のリストです。次の引数は、演算名とその演算を実装した(つまり、同じ入力値を与えられると同じ出力値を生成する)Scheme 手続きをペアにしたテーブル(2要素リストのリスト)です。最後の引数は、ラベルとマシン命令のリストとしてコントローラを規定します。これはSection 5.1で示したものと同じです。

このマシンを使って GCD を計算するには、入力レジスタを設定し、マシンをスタートさせ、シミュレーションが終了したら結果を調べます。

```
(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
```

この計算は、Scheme で書いた gcd 手続きよりもずっと遅く動作します。これは、`assign` のような低レベルのマシン命令をずっと複雑な演算によってシミュレートすることになるからです。

Exercise 5.7: シミュレータを使って Exercise 5.4 で設計したマシンをテストせよ。

5.2.1 マシンモデル

`make-machine` で生成するマシンのモデルは、Chapter 3 で開発したメッセージパッキング技術を使って、局所状態を持つ手続きとして表現されます。このモデルを構築するために、`make-machine` はまず `make-new-machine` 手続きを呼んで、すべてのレジスタマシンに共通なマシンモデルの部品を構築します。`make-new-machine` によって構築されるこの基本マシンモデルは、本質的には複数のレジスタとスタックひとつに、コントローラ命令をひとつずつ処理する実行メカニズムをセットにしたもののが入れ物です。

次に、`make-machine` は(メッセージを送って)この基本モデルを拡張して、レジスタ、演算、定義中の特定のマシンのコントローラを持たせます。最初に、与えられたレジスタ名それぞれについて新しいマシンの中にレジスタを割り当て、指定された演算をマシンに組み込みます。次に、`assembler`(アセンブラー)(後ほど Section 5.2.2 で説明します)を使ってコントローラ命令列を新しいマシンの命令に変換し、その命令列をマシンの命令列として組み込みます。`make-machine` は、修正したマシンのモデルを値として返します。

```
(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でやったように、局所状態を持つ手続きとして表現します。手続き `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))
```

以下の二つの手続きがレジスタにアクセスするために使われます。

```
(define (get-contents register) (register 'get))
(define (set-contents! register value)
  ((register 'set) value))
```

スタック

スタックも局所状態を持つ手続きとして表現できます。手続き `make-stack` はスタックを作ります。スタックの局所状態はスタック上の項目のリストです。スタックは、スタックにアイテムを追加する `push`、スタックのトップから項目を取り去りそれを返す `pop`、スタックを空に初期化する `initialize` という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)

```

以下の二つの手続きがスタックへのアクセスに使われます。

```

(define (pop stack) (stack 'pop))
(define (push stack value) ((stack 'push) value))

```

基本マシン

Figure 5.13に示す `make-new-machine` 手続きは、ひとつのスタック、初期値が空の命令列、初期値としてスタックを空にする演算を持つ演算のリスト、初期値として `flag` と `pc`(「プログラムカウンタ」)という名前の二つのレジスタを持つ `register table`(レジスタテーブル)を局所状態として持つオブジェクトを構築します。内部手続き `allocate-register` はレジスタテーブルに新しい項目を追加し、内部手続き `lookup-register` はテーブルからレジスタを検索します。

`flag` レジスタは、シミュレート対象のマシンの分岐を制御するために使います。`test` 命令はテストの結果(真または偽)を `flag` の中身として設定します。`branch` 命令は `flag` の中身を調べて分岐するかどうかを決めます。

`pc` レジスタはマシン動作中の命令の順序を決定します。命令の逐次実行は内部手続き `execute` として実装されています。このシミュレーションモデルでは、それぞれのマシン命令は引数なしの手続きを持つデータ構造で、*instruction execution procedure*(命令実行手続き)と呼ばれます。命令実行手続きを呼ぶと命令の実行をシミュレートします。シミュレーションの実行中、`pc` は次に実行される命令から始まる命令列の場所を指します。`execute` はその命令を取得し、命令実行手続きを呼んでそれを実行し、実行する命令がなくなるまで(つまり、`pc` が命令列の末尾を指すまで)このサイクルを繰り返します。

Figure 5.13: ↓ 基本マシンモデルを実装する `make-new-machine` 手続き

```
(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)))

```

それぞれの命令実行手続きは、演算の一環として、pc を修正して次に実行するべき命令を指すようにします。branch と goto 命令は新しい行き先を指すように pc を変更します。これら以外の命令はすべて、単純に pc を進めて命令列中の次の命令を指すようにします。それぞれの execute の呼び出しが再び execute を呼び出すことに注意してください。これは無限ループにはなりません。命令実行手続きの実行は pc の中身を変更するからです。

`make-new-machine` はメッセージパッキングによるマシンの内部状態へのアクセスを実装した `dispatch` 手続きを返します。マシンのスタートは、命令列の先頭を `pc` に設定し、`execute` を呼ぶことによって行えます。

マシンを扱いやすくするために、マシンの `start` 演算の代替として手続きによるインターフェイスを提供します。ほかに、レジスタの中身の設定と検査の手続きも、Section 5.2 の最初で規定したような形で提供します。

```

(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)

```

これらの手続き（また Section 5.2.2 と Section 5.2.3 の多くの手続き）は、あるマシンのあるレジスタ名のレジスタを探すために次の手続きを使います。

```
(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、あるマシンのコントローラの式の列をそれに対応するマシン命令の列に変換するものです。マシン命令はそれぞれ実行手続きを持ちます。全体として、アセンブラーは Chapter 4で詳しく見てきた評価器によく似ています。どちらも入力言語（この場合はレジスタマシン言語）があり、その言語の式のそれぞれの型に対して適切なアクションを実行する必要があります。

それぞれの命令に対する実行手続きを生成する技術は、Section 4.1.7で実行時の処理から解析を分離して評価器の速度向上をするのに使ったものと同じです。Chapter 4で見てきたように、Scheme 式の有用な解析の多くは、変数の実際の値を知らないでも行うことができます。同じように、レジスタマシン言語の式も、マシンレジスタの実際の中身を知らない今まで多くの有用な解析が行えます。例えば、レジスタへの参照はレジスタオブジェクトへのポインタに置き換えることができますし、ラベルへの参照はそのラベルが指示する命令列の場所へのポインタに置き換えることができます。

アセンブラーは、命令実行手続きを生成する前に、ラベルの参照先をすべて知っておく必要があります。そのため、アセンブラーはまずコントローラのテキストを走査して、命令からラベルを分離します。アセンブラーがテキストを走査する間に、命令リストと、各ラベルとそのリストへのポインタを関連づけるテーブルの両方を構築します。それから、アセンブラーは命令リストのそれぞれの命令に実行手続きを挿入することで命令リストを拡張します。

assemble 手続きはアセンブラーのメインエントリです。コントローラテキストとマシンモデルを引数に取り、モデルに格納すべき命令列を返します。assemble は extract-labels を呼び、与えられたコントローラテキストから最初の命令リストとラベルテーブルを構築します。extract-labels の二つ目の引数は、これらの結果を処理するために呼ぶ手続きです。この手続きは update-insts! を使って命令実行手続きを生成し、それらを命令リストに挿入し、修正し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 `text` というリスト（コントローラ命令式の列）と、`receive` 手続きを取ります。`receive` は、次の二つの値を引数として呼び出されます。(1) 命令データ構造のリスト `insts`。それぞれの命令データ構造は `text` 内の命令を含みます。(2) テーブル `labels`。`text` の各ラベルと、そのラベルが指示するリスト `insts` 内の位置を関連づけます。

```
(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、`text` の要素を順に走査し、`insts` と `labels` を集積していくします。要素が記号（つまりラベル）であれば、`labels` テーブルに適切なエントリが追加されます。そうでなければ、要素は `insts` リストに集積されます。⁴

⁴ここで `receive` 手続きを使っているのは、`extract-labels` に `labels` と `insts` という二つの値を、それらを保持する複合データ構造を明示的に作らずに効率的に返させるためです。明示的に値のペアを返すようにした実装は以下のようになります。

```
(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!` は命令リストを修正します。命令リストは初期状態では命令のテキストしか持っていないませんが、それに対応する実行手続きを追加します。

```
(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)))
```

マシン命令データ構造は、命令テキストと、それに対応する実行手続きを単純にペアにします。実行手続きは `extract-labels` が命令を構築するときにはまだ使用できず、後で `update-insts!` によって挿入されます。

```
(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` からは次のように呼び出すことになります。

```
(define (assemble controller-text machine)
  (let ((result (extract-labels controller-text)))
    (let ((insts (car result)) (labels (cdr result)))
      (update-insts! insts labels machine)
      insts)))
```

ここでの `receive` の使い方は、複数の値を返すエレガントな方法の例と考えていただいてもいいですし、プログラミングトリックをひけらかしたいだけだと考えていただいてもけっこうです。`receive` のような、次に呼び出すべき手続きを渡す引数は、“継続”と呼ばれます。継続はSection 4.3.3で `amb` 評価器のバックトラック制御構造の実装にも使ったことを思い出しましょう。

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

命令テキストはシミュレータには使われませんが、デバッグ用に持っておくと便利です (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のように検索します。

```
(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、ラベル `here` が複数回定義されているため曖昧になっている。

```
start
  (goto (label here))
here
  (assign a (const 3))
  (goto (label there))
here
  (assign a (const 4))
  (goto (label there))
there
```

本文に書いたようなシミュレータでは、制御が `there` に達したとき、レジスタ `a` の中身はどうなっているか。手続き `extract-labels` を修正し、同じラベル名が二つの異なる場所を指すために使われている場合にはエラーを発するようにせよ。

5.2.3 命令の実行手続きの生成

アセンブラーは `make-execution-procedure` を呼んで命令の実行手続きを生成します。Section 4.1.7 の評価器の `analyze` 手続きと同じように、この手続きは適切な実行手続きを生成するために命令の型によってディスパッチを行います。

```
(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))))
```

レジスタマシン言語の命令のそれぞれの型に対して、適切な実行手続きを構築する生成器があります。これらの手続きの詳細が、レジスタマシン言語のそれぞれの命令の構文と意味の両方を決定します。ここでは、データ抽象化を使ってレジスタマシンの式の詳細な構文を全体的な実行メカニズムから分離しています。これは、Section 4.1.2 の評価器で、構文手続きを使って命令のそれぞれの部分を抽出・分類したのと同じです。

assign 命令

`make-assign` 手続きは `assign` 命令を処理します。

```
(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 に対する実行手続き
        (set-contents! target (value-proc))
        (advance-pc pc))))
```

`make-assign` は、`assign` 命令から目標レジスタ名 (命令の二つ目の要素) と値の式 (命令を構成するリストの残りの部分) を次の二つのセレクタを使って抽出します。

```
(define (assign-reg-name assign-instruction)
  (cadr assign-instruction))
(define (assign-value-exp assign-instruction)
  (caddr assign-instruction))
```

レジスタ名は `get-register` によって検索され、目標レジスタオブジェクトを生成します。値の式、値が命令の結果であれば `make-operation-exp` に渡され、そうでなければ `make-primitive-exp` に渡されます。これらの手続き (以下に示します) は、値の式を構文解析し、その値に対する実行手続きを生成します。これは `value-proc` と呼ばれる引数なしの手続きで、シミュレーションの間に評価されてレジスタに代入すべき実際の値を生成します。レジスタ名の検索と値の式の構文解析はアセンブリ時に一回行われるだけで、命令をシミュレーションするたびに毎回行われるわけではないというところに注意してください。この省力化こそが実行手続きを使う理由です。これは、Section 4.1.7 の評価器で、プログラム分析を実行から分離することによって達成した省力化に直接対応します。

`make-assign` によって返される結果は `assign` 命令の実行手続きです。この手続きが (マシンモデルの `execute` 手続きによって) 呼ばれると、`value-proc` 手続きを実行することで得られる結果を目標レジスタの中身に設定します。それから、以下の手続きを実行して `pc` を次の命令に進めます。

```
(define (advance-pc pc)
  (set-contents! pc (cdr (get-contents pc))))
```

advance-pc は、branch と goto 以外のすべての命令の正常終了処理です。

test, branch, goto 命令

make-test は、make-assign と同じように test 命令を処理します。make-test はテスト対象の条件を規定する式を抽出し、それに対する実行手続きを生成します。シミュレーション時には、条件の手続きが呼ばれ、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 命令の実行手続きは、flag レジスタの中身をチェックし、分岐の目的地を pc の中身に設定する(分岐する場合)か、単に pc を進める(分岐しない場合)かのどちらかを行います。branch 命令内で指定される目的値はラベルでなければならず、make-branch 手続きがそれを強制することに注意してください。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たびに毎回検索されるわけではないというところにも注意してください。

```
(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、目的地がラベルまたはレジスタによって規定されるという点と、チェックするべき条件がないというところが違います。`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))

```

その他の命令

スタック命令の `save` と `restore` は、指定されたレジスタについて単純にスタックを使い、`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` が扱う最後の命令型は、実行するべきアクションに対する実行手続きを生成します。シミュレーション時には、このアクション手続きが実行され、`pc` が進められます。

```

(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))

```

部分式の実行手続き

`reg, label, const` 式の値は、レジスタへの代入 (`make-assign`) または演算に対する入力 (下記の `make-operation-exp`) のために必要になることがあります。以下の手続きは、これらの式の値をシミュレーション時に生成する実行手続きを生成します。

```
(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` 式の構文は以下の手続きによって決められています。

```

(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` 命令では、(`reg` と `const` 式によって規定される) オペランドに対する (`op` 式によって規定される) マシンの演算命令の適用を含む可能性があります。以下の手続きは、“演算式”—命令内の演算とオペランド式からなるリスト—に対する実行手続きを生成します。

```

(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)))))


```

演算式の構文は以下の手続きによって決められています。

```
(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))
```

演算式の処理は、Section 4.1.7の評価器の `analyze-application` 手続による手続き適用の処理でそれぞれのオペランドに対して実行手続きを生成したのとそっくりです。シミュレーション時には、オペランドの手続きを呼び、演算をシミュレートする Scheme 手續きをその結果の値に対して適用します。シミュレーション手続きは、演算名でマシンの演算テーブルを検索することによって得られます。

```
(define (lookup-prim symbol operations)
  (let ((val (assoc symbol operations)))
    (if val
        (cadr val)
        (error "Unknown operation: ASSEMBLE"
              symbol))))
```

Exercise 5.9: マシンの演算を上のよう扱うと、定数とレジスタの中身以外に、ラベルに対しても演算が可能になる。式処理手続きを修正し、演算はレジスタと定数だけにしか使えないという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、その新しい構文を使えるようにシミュレータを修正せよ。この節の構文手続き以外のシミュレータの部分を変更せずに新しい構文を実装できるだろうか。

Exercise 5.11: Section 5.1.4で `save` と `restore` を導入したときには、最後に保存したレジスタでないレジスタを復元しようとしたときに何が起こるかを規定していなかった。例えば、次のような命令列の場合である。

```
(save y)  (save x)  (restore y)
```

`restore` の意味として妥当なものはいくつか考えられる。

- a (`restore y`) は、どのレジスタから来た値か関係なしに、スタックに最後に保存された値を `y` に入れる。現在のシミュレータのふるまいはこのようになっている。このふるまいを利用して、Section 5.1.4 のフィボナッチマシン (Figure 5.12) から命令をひとつ削減する方法を示せ。
- b (`restore y`) は、スタックに最後に保存された値が `y` を保存したものである場合に限り、それを `y` に入れる。それ以外の場合にはエラーを発する。シミュレータを修正し、ふるまいをこのように変更せよ。スタックに値のほかにレジスタ名も保存するように `save` を変更する必要がある。
- c (`restore y`) は、最後に `y` から保存した値を `y` に入れる。`y` の後にはかのレジスタの値が保存され、それらが取り出されていない場合にも、この挙動は変わらない。シミュレータを修正し、ふるまいをこのように変更せよ。それぞれのレジスタに対して、別々のスタックを関連づける必要があるだろう。また、`initialize-stack` 演算ではすべてのレジスタスタックを初期化しなければならない。

Exercise 5.12: シミュレータは、あるコントローラを持つマシンを実装するために必要となるデータパスを決めるときの補助として使うことができる。アセンブリを拡張し、以下の情報をマシンモデルに格納するようにせよ。

- 全命令のリスト。重複を削除し、命令の型 (`assign`, `goto` 等) によってソートしたもの。
- エントリポイントを入れるために使われるレジスタの (重複なし) リスト (これらは `goto` 命令によって参照されるレジスタである)。
- `save` または `restore` されるレジスタの (重複なし) リスト。
- それぞれのレジスタに対する代入元の (重複なし) リスト (例えば、Figure 5.11 の階乗マシンでは、レジスタ `val` の代入元は (`const 1`) と ((`op *`) (`reg n`) (`reg val`)) である)。

マシンのメッセージパッキングインターフェイスを拡張し、この新しい情報にアクセスできるようにせよ。この解析器をテストす

るため、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確認せよ。

Exercise 5.13: シミュレータを修正し、`make-machine` の引数としてレジスタリストを要求するのではなく、コントローラ命令列を使ってマシンの持つレジスタを決定するようにせよ。`make-machine` の中にレジスタをあらかじめ割り当てておく代わりに、命令アセンブルの間に初めて現れるたびにレジスタを割り当てるようにしてよい。

5.2.4 マシンパフォーマンスの監視

シミュレーションは、計画中のマシン設計の正しさを検証するのに役に立つだけでなく、マシンのパフォーマンスを計るためにも有用です。例えば、計算中にスタック演算が使われた回数を測る“メーター”をシミュレータに組み込むといったことができます。これを実現するためには、スタックをシミュレートしたものを修正し、スタック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管理するようにします。また、以下に示すように、スタックのインターフェイスに統計情報を表示させるメッセージを追加します。さらに、基本マシンモデルにスタック統計情報を示す演算を追加します。このために、`make-new-machine` の `the-ops` を以下のように初期化します。

```
(list (list 'initialize-stack
            (lambda () (stack 'initialize)))
      (list 'print-stack-statistics
            (lambda () (stack 'print-statistics))))
```

`make-stack` の新しいバージョンは以下のようになります。

```
(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では、レジスタマシンシミュレータに追加できるこのほかの便利な監視・デバッグ機能について示します。

Exercise 5.14: Figure 5.11に示した階乗マシンを使って、いくつかの小さな n の値について $n!$ の計算に必要となる push の回数とスタックの最大の深さを計測せよ。そのデータから、任意の $n > 1$ について、 $n!$ を計算するのに使われる push 演算の総数とスタックの最大の深さを表す n の方程式を決定せよ。いずれも n の線形関数であり、そのため二つの定数によって決定される。統計を表示するため

には、スタックを初期化する命令と統計を表示する命令によって階乗マシンを拡張する必要がある。また、(Figure 5.4で GCD マシンに対して行ったのと同じように) n の値を繰り返し読み込み、階乗を計算し、結果を表示するようにマシンを修正してもいいかもしない。こうすると、`get-register-contents, set-register-contents!, start` を何度も起動しないですむようになる。

Exercise 5.15: レジスタマシンシミュレータに *instruction counting*(命令数カウンタ) を追加せよ。つまり、マシンモデルに実行した命令数を管理させることになる。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してから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*(命令トレース) を追加せよ。つまり、シミュレータが各命令を実行する前に、その命令のテキストを表示するようにするということである。マシンモデルが `trace-on, trace-off` メッセージを受け入れ、それによってトレース機能をオン・オフするようにせよ。

Exercise 5.17: Exercise 5.16 の命令トレース機能を拡張し、シミュレータが命令を表示する前に、コントローラ命令列中でその命令の直前にあるラベルを表示するにせよ。これを行う際に、命令数カウンタ (Exercise 5.15) に干渉しないように注意すること。必要なラベル情報をシミュレータに持たせておく必要があるだろう。

Exercise 5.18: Section 5.2.1 の `make-register` 手続きを修正し、レジスタをトレースできるようにせよ。レジスタはトレースをオン・オフするメッセージを受け入れる。レジスタがトレースされているときには、そのレジスタへの代入が行われると、レジスタの名前・レジスタの古い値・代入する新しい値が表示される。マシンモデルのインターフェイスを拡張し、指定されたマシンレジスタのトレース機能をオン・オフできるようにせよ。

Exercise 5.19: Alyssa P. Hacker は、マシン設計のデバッグに使うために、シミュレータに *breakpoint* (ブレークポイント) 機能がほしくなった。そこで、この機能を組み込むために、彼女はあなたを雇うことになった。彼女は、シミュレータが止まる場所を指定できるようにして、またマシンの状態を調べられるようにしてほしいと思っている。あなたは以下の手続きを実装することになっている。

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

この手続きは、与えられたラベルから n 番目の命令の直前にブレークポイントを設定するというものだ。例えば、

```
(set-breakpoint gcd-machine 'test-b 4)
```

のようにすると gcd-machine でレジスタ a への代入の直前にブレークポイントが設定される。シミュレータがブレークポイントに到達すると、ラベルとブレークポイントのオフセットを表示し、命令の実行を停止する。すると、Alyssa は get-register-contents と set-register-contents! を使ってシミュレートされているマシンの状態を操作することができるようになる。また、Alyssa がその後以下のように入力することによって実行を継続できるようにしなければならない。

```
(proceed-machine <machine>)
```

さらに、以下のようにして特定のブレークポイントを削除したり、

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

以下のようにしてすべてのブレークポイントを削除したりできるようにしなければならない。

```
(cancel-all-breakpoints <machine>)
```

5.3 記憶領域の割り当てとガベージコレクション

Section 5.4では、Scheme 評価器をレジスタマシンとして実装する方法を示します。考察を簡単にするために、このレジスタマシンは *list-structured memory* (リスト構造メモリ) を備えていて、リスト構造のデータを操作する演算は基本演算だとします。そのようなメモリの存在を仮定するというのは、Scheme インタプリタの制御メカニズムに焦点を当てている場合には役に立つ抽象化ですが、これは現代のコンピュータにおける実際の基本データ演算の現実的な見方を反映していません。Lisp システムがどのように動いているかとい

うことについてより全面的な理解を得るために、従来のコンピュータメモリで実現できる形でリスト構造を表現する方法を調査する必要があります。

リスト構造の実装にあたっては、考えるべきことが二つあります。一つ目は、純粹に表現の問題です。Lisp ペアの“箱とポインタ”という構造を、典型的なコンピュータメモリの記憶領域とアドレス機能だけを使って表現するにはどのようにすればよいかということです。二つ目の問題は、計算の進行に伴うメモリ管理に関するものです。Lisp システムの演算は、継続的に新しいデータオブジェクトを作る能力に強く依存しています。継続的に作られるデータオブジェクトには、Lisp の手続きが解釈される間に明示的に作られるもののほかに、環境や引数リストといったインタプリタ自身によって作られる構造も含まれます。継続的に新しいデータオブジェクトを作っていくことは、高速にアクセスできる無限のメモリを持つコンピュータであれば何の問題にもならないところですが、実際にはコンピュータのメモリは有限です(残念ながら)。そのため、Lisp システムは無限メモリという幻想を支えるために、*automatic storage allocation*(自動記憶領域割り当て) という機能を用意しています。データオブジェクトが必要でなくなると、それに割り当てられていたメモリは自動的にリサイクルされ、新しいデータオブジェクトを構築するのに使われます。そのような自動記憶領域割り当てを提供する技術はいくつもあります。この節で検討する手法は *garbage collection*(ガベージコレクション) と呼ばれるものです。

5.3.1 ベクタとしてのメモリ

従来型コンピュータのメモリは、それぞれの枠の中に情報のかけらを入れることができる一列の棚のように考えることができます。それぞれの枠は、*address*(アドレス) や *location*(位置) と呼ばれる一意の名前を持っています。普通のメモリシステムは二つの基本演算を提供します。ひとつは指定された位置に格納されたデータを取ってくる演算で、もうひとつは指定された位置に新しいデータを割り当てる演算です。枠の集合に順にアクセスできるように、メモリアドレスはインクリメントできるようになっています。より一般的に、多くの重要なデータ演算では、メモリアドレスをデータとして扱うことが求められます。メモリアドレスのデータは、メモリ位置に格納したり、マシンのレジスタで操作したりできる必要があります。リスト構造の表現は、そのような *address arithmetic*(アドレス演算) の応用のひとつです。

コンピュータメモリをモデル化するために、ここでは *vector*(ベクタ) と呼ばれる新しい種類のデータ構造を使います。抽象的には、ベクタは整数のイン

デックスによって個々の要素にアクセスでき、またそのアクセス時間がインデックスとは独立であるような複合データ構造です。⁵ メモリ演算を記述するには、ベクタを操作する Scheme の基本手続き二つを使います。

- (`(vector-ref <vector> <n>)`) は、ベクタの n 番目の要素を返します。
- (`(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します。

例えば、`v` がベクタであれば、(`(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。⁶ コンピュータメモリでは、メモリ内でのベクタの開始位置を指定する *base address* (ベースアドレス) 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) をアドレス演算を使って組み合わせることによってこのアクセスを実装することができます。

Lisp データの表現

リスト構造メモリのために必要となる基本的なペア構造は、ベクタを使って実装することができます。コンピュータメモリが `the-cars` と `the-cdrs` という二つのベクタに分かれていると想像してみましょう。リスト構造は、以下のように表現することにします。まず、ペアへのポインタはこの二つのベクタのインデックスとします。ペアの `car` は `the-cars` の指定されたインデックスの項目で、ペアの `cdr` は `the-cdrs` の指定されたインデックスの項目です。また、ペア以外のオブジェクト (数値や記号など) の表現と、それぞれのデータの種類を判別する方法も必要になります。これを達成する手法はいろいろありますが、いずれも結局は *typed pointers* (型つきポインタ) を使うというところに帰着します。型つきポインタというのは、“ポインタ”の概念を拡張し、データ型の情報を持たせるようにするということです。⁷ データ型があると、ペア (“ペア”というデータ型と、メモリベクタのインデックスからなる) のポインタと、

⁵ メモリを項目のリストとして表現することも可能ですが。しかし、その場合アクセス時間はインデックスとは独立でなくなってしまいます。リストの n 番目の要素にアクセスするには、 $n - 1$ 回の `cdr` 演算が必要だからです。

⁶ ベクタを完全なものにするためには、ベクタを構築する `make-vector` 演算を規定しなければならないところです。しかし、ここでのアプリケーションでは、コンピュータメモリの決まった部分をモデル化するだけにしか使いません。

⁷ これは、Chapter 2 でジェネリック演算を扱うために導入した “タグ付きデータ” とまったく同じ考え方です。しかし、ここではデータ型をリストを使って構築するのではなく、マシンの基本レベルで含めるようにしています。

ほかの種類のデータ（ほかの何らかのデータ型と、その型を表現するのに使われる何らかのもの）のポインタをシステムが判別できるようになります。二つのデータオブジェクトは、それらのポインタが同一である場合に同じ (eq?) であると見なされます。⁸ Figure 5.14は、この手法によってリスト ((1 2) 3 4) を表現した図で、箱とポインタの図も同時に示しています。ここでは、前につけた文字によってデータ型情報を示しています。つまり、インデックス 5 を持つペアへのポインタは p5、空リストはポインタ e0、数値 4 へのポインタは n4 によって示されます。箱とポインタの図では、それぞれのペアの左下に、そのペアの car と cdr が格納される位置を示すベクタのインデックスを記しています。the-cars と the-cdrs の空白の場所には、（ここで扱っていない）ほかのリスト構造の一部などが入っている可能性もあります。

数値へのポインタ、例えば n4 は、数値データを示す型と、数値 4 を実際に表現したものからなることもあります。⁹ ひとつのポインタに割り当てられる固定長のメモリでは表現できないほど大きい数値を扱うには、別に *bignum* (ビッグナム) データ型を使うことが考えられます。この場合、ポインタは数値の各部分を格納したリストを指すことになります。¹⁰

記号は、記号の文字表現を構成する文字の列を指す型つきポインタとして表現することが考えられます。この列は、入力の中で文字列が最初に出てきたときに Lisp の読み込み器によって構築されます。ここで、ひとつの記号の二つ

⁸ 型情報は Lisp システムを実装するマシンがどのようなものであるかによっていろいろな符号化方法が考えられます。Lisp プログラムの実行効率は、この選択をどれだけうまくやるかに強く依存します。しかし、どのような選択をするのがよいかという一般的な設計規則を定式化することは簡単ではありません。型つきポインタを実装する最も素直なやり方は、それぞれのポインタに固定長のビット列を割り当てて、それをデータ型を符号化する *type field* (型フィールド) とするというものです。このような表現を設計するにあたっては、考慮すべき重要な問題がいくつかあります。型ビットは何ビット必要か、ベクタのインデックスはどれだけ必要か、ポインタの型フィールドを操作するのにマシンの基本命令がどれだけ効率的に使えるか、といったものです。型フィールドを効率よく扱うための特別なハードウェアを持つマシンは *tagged architectures* (タグアーキテクチャ) を持つというように言われます。

⁹ ここで数値をどう表現することにするかによって、ポインタの等価性をテストする eq? を数値の等価性のテストに使えるかどうかが決まります。ポインタが数値そのものを含んでいれば、等しい数値は同じポインタを持つことになります。しかし、ポインタが数値が格納されている位置へのインデックスを持っているのであれば、同じ数を決して異なる位置に格納しないよう注意する場合に限り、等しい数値が同じポインタになることが保証されることになります。

¹⁰ これは数値を数字の列として書くのと似ていますが、この場合、各“桁”は 0 からひとつずつのポインタに格納できる最大値の間の数値となります。

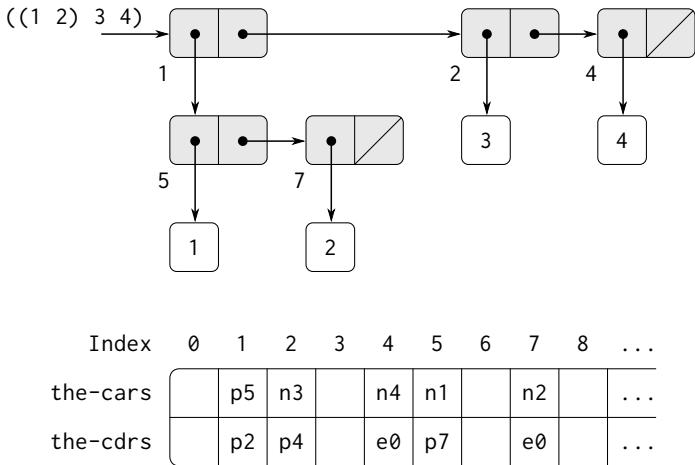


Figure 5.14: リスト $((1\ 2)\ 3\ 4)$ の箱とポインタ表現と
モリベクタ表現

の実体があるとき、それらは `eq?` によって“同じ”記号として認識されるよう
にしたいところです。それと同時に、`eq?` はポインタの等価性の簡単なテスト
としておきたいところです。そのため、読み込み器が同じ文字列を二回見た場合
には、その二回の出現両方に対して同じポインタ（同じ文字列を指すポイン
タ）を使うということを保証する必要があります。これを達成するために、読み
込み器は出てきた記号すべてのテーブルを管理します。このテーブルは伝統的
に *obarray*（オブアレイ）と呼ばれます。読み込み器が文字列にぶつかって記
号を構築しようとするとき、同じ文字列がそれまでに出てきていないかオブア
レイをチェックします。出てきていないれば、その文字列を使って新しい記号
(新しい文字列を指す型つきポインタ) を構築し、そのポインタをオブアレイに
入れます。読み込み器がその文字列をすでに見たことがあれば、読み込み器は
オブアレイに格納されている記号ポインタを返します。文字列を一意なポイン
タで置き換えるこの処理は、記号の *interning*（閉じ込め）と呼ばれます。

基本リスト演算の実装

ここまで述べたような表現法を使うと、レジスタマシンのそれぞれの“基本”リスト命令をいくつかの基本ベクタ演算で置き換えられるようになります。ここではメモリベクタを指す二つのレジスタ `the-cars` と `the-cdrs` を使い、`vector-ref` と `vector-set!` は基本演算として使えると仮定します。また、ポインタ演算（ポインタのインクリメント、ペアポインタによるベクタの位置指定、二つの数値の加算など）は、型つきポインタのインデックス部だけを使うものと仮定します。

例えば、以下の命令をサポートするレジスタマシンは、

```
(assign <reg1> (op car) (reg <reg2>))
(assign <reg1> (op cdr) (reg <reg2>))
```

それぞれ次のように実装することによって作ることができます。

```
(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>))
```

次のように実装できます。

```
(perform
  (op vector-set!) (reg the-cars) (reg <reg1>) (reg <reg2>))
(perform
  (op vector-set!) (reg the-cdrs) (reg <reg1>) (reg <reg2>))
```

`cons` は、未使用のインデックスを割り当て、`the-cars` と `the-cdrs` の中のそのインデックスが指すベクタ位置に `cons` の二つの引数を格納することによって実行できます。ここでは、次に使用可能なインデックスの入ったペアポインタを持つ `free` という特別なレジスタが存在し、そのポインタのインデックス部をインクリメントすることで次の空き位置を見つけられると仮定します。¹¹ 例えれば、次の命令は、

¹¹ 空き領域を探す方法はほかにもあります。例えば、未使用のペアをすべて `free list`（空きリスト）として連結するということも可能です。Section 5.3.2で見ていくように、ここでは圧縮 GC を使うため、空き位置は連続的（そのため、ポインタのインクリメントによってアクセス可能）です。

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

以下のようなベクタ命令の列として実装できます。¹²

```
(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))
```

以下の eq? 演算は、

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

単純にレジスタのすべてのフィールドの等価性をテストします。pair?, null?, symbol?, number? のような述語は、型フィールドをチェックするだけで十分です。

スタックの実装

私たちのレジスタマシンはスタックを使いますが、スタックはリストによってモデル化できるため、ここでは特別なことをする必要はありません。スタックは保存した値のリストとして実装することができます。the-stack という特別なレジスタがスタックを指すものとします。そのため、(save <reg>) は次のように実装できます。

```
(assign the-stack (op cons) (reg <reg>) (reg the-stack))
```

同じように、(restore <reg>) は次のように実装できます。

```
(assign <reg> (op car) (reg the-stack))  
(assign the-stack (op cdr) (reg the-stack))
```

また、(perform (op initialize-stack)) は次のように実装できます。

```
(assign the-stack (const ()))
```

¹²これは、本質的にはSection 3.3.1で述べた set-car! と set-cdr! による cons の実装と同じです。そのときの実装で使った get-new-pair 演算は、ここでは free ポインタによって実現されています。

これらの演算は、上で示したベクタ演算によってさらに展開することができます。しかし、従来型コンピューターアーキテクチャでは、スタックを別のベクタとして割り当てるほうが通常は有利です。その場合、プッシュとポップはそのベクタのインデックスのインクリメントとデクリメントとして実現できます。

Exercise 5.20: free ポインタの初期値を p1 として、次の式によつて生成されるリスト構造について、(Figure 5.14のように) 箱とポインタ表現とメモリベクタ表現を描け。

```
(define x (cons 1 2))
(define y (list x x))
```

free の最終的な値は何になるか。どのポインタが x と y の値を表現しているだろうか。

Exercise 5.21: 以下の手続きに対応するレジスタマシンを実装せよ。リスト構造メモリ演算はマシンの基本演算として使えると仮定する。

a 再帰的 count-leaves:

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

b 明示的にカウンタを持つ再帰的 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では、二つのリストを足してひとつの新しいリストにする `append` 手続きと、二つのリストを継ぎ合わせる `append!` 手続きを紹介した。これらの手続きそれぞれを実装するレジスタマシンを設計せよ。リスト構造メモリ演算は基本演算として使えると仮定する。

5.3.2 無限メモリ幻想の維持

Section 5.3.1で概観した表現手法は、無限の容量のメモリさえあるなら、リスト構造の実装という問題の解決になっています。しかし、実際のコンピュータでは、新しいペアを構築するための空き領域をそのうち使い果たしてしまいます。¹³しかし、典型的な計算で生成されるペアのほとんどは中間結果を保存するためだけに使われるものです。結果にアクセスされた後は、それらのペアはもう必要なくなります—つまり、*garbage* (ゴミ) になるということです。例えば、次の計算は、

```
(accumulate + 0 (filter odd? (enumerate-interval 0 n)))
```

列挙リストと、列挙リストをフィルタリングした結果のリストという二つを構築します。集積が完了すると、これらのリストはもう必要なく、割り当てたメモリは再利用できます。すべてのゴミを定期的に回収する仕組みを作り、それによってメモリを再利用する速度が新しいペアを構築するのと同じ速度であれば、無限の容量のメモリがあるという幻想を維持できたことになります。

ペアを再利用するには、どの割り当て済みペアが(ペアの中身が将来の計算に影響することがないという意味で)不要かを決定する方法が必要です。これを達成するためにここで検討する手法は、*garbage collection*(ガベージコレクション)として知られるものです。ガベージコレクションの基本となる事実

¹³これが当てはまらなくなるときが来る可能性はあります。コンピュータの耐用期間中に空き領域を使い切れないほどメモリが大きくなるということも考えられるからです。例えば、一年は $3 \cdot 10^{13}$ マイクロ秒なので、1マイクロ秒に1回 `cons` を行うとすると、メモリを使い切ることなしに30年間運用可能なコンピュータを作るのには約 10^{15} のメモリセルが必要ということになります。それだけの容量のメモリというのは、今の尺度で考えると馬鹿馬鹿しいほど巨大なものに思えますが、物理的に不可能というわけではありません。一方、プロセッサは速くなっていますし、未来のコンピュータはひとつのメモリに対して並列に演算を行う多数のプロセッサを持つようになるかも知れないのです、ここでの想定よりもずっと速くメモリを使い果たすようになることも考えられます。

は、Lisp の逐次実行のどの時点でも、将来の計算に影響を与えるオブジェクトは、現在マシンのレジスタ内にあるポインタから `car` と `cdr` を繰り返して到達できるものだけだということです。¹⁴ そのようなアクセスが可能でないメモリセルは再利用できます。

ガベージコレクションを実行する方法はいろいろあります。ここで検討する手法は *stop-and-copy* (ストップ & コピー) と呼ばるもので、基本的な考え方では、メモリを“ワーキングメモリ”と“フリーメモリ”的に分割するというものです。`cons` がペアを構築するときは、ペアをワーキングメモリに割り当てます。ワーキングメモリがいっぱいになるとガベージコレクションを行います。ガベージコレクションは、ワーキングメモリ内の有用なペアをすべて探し、それらをフリーメモリの連続した位置にコピーすることによって行います。(有用なペアは、マシンレジスタからすべての `car` と `cdr` のポインタをたどっていくことによって探します)。ゴミはコピーしないので、新しいペアを割り当てるのに使える余分のフリーメモリがおそらくできているはずです。それに加えて、ワーキングメモリの中身はすべて必要ありません。有用なペアはすべてコピー済みだからです。つまり、ワーキングメモリとフリーメモリの役割を交換すると、処理を続けられることになります。新しいペアは新しいワーキングメモリ(元フリーメモリ)に割り当てられ、それがいっぱいになったなら、有用なペアを新しいフリーメモリ(元ワーキングメモリ)にコピーすることができます。¹⁵

¹⁴ ここでは、スタックはSection 5.3.1で書いたようにリストとして表現されていると仮定します。そのため、スタック上の項目はスタックレジスタのポインタを経由してアクセス可能ということになります。

¹⁵ この考え方方は Minsky によって発明され、実装されたものです。Minsky の実装は、MIT 研究所の電子工学研究所にあった PDP-1 で動かす Lisp 実装の一部でした。これは、Multics 時分割システムの Lisp 実装で使うために、Fenichel and Yochelson (1969) によってさらに開発が進められました。のちに Baker (1978) は、ガベージコレクションの間に計算を止める必要がない、この手法の“リアルタイム”版を開発しました。Baker の考え方方は Hewitt, Lieberman, Moon によって拡張され (Lieberman and Hewitt 1983 参照)、データ構造には揮発性のものもあれば永続的なものもあるということを利用するようになりました。

このほかによく使われるガベージコレクション技術としては、*mark-sweep* (マーク & スイープ) 手法があります。これは、マシンレジスタからアクセス可能な構造をすべてたどり、到達できたペアそれぞれに印をつけるというものです。それからメモリ全部を走査し、印のついていないメモリをすべてゴミとして“掃き出し”、再使用可能にします。マーク & スイープについては、Allen 1978 で詳細に検討されています。

Minsky-Fenichel-Yochelson アルゴリズムはメモリの有用な部分のみを調査するため、大きなメモリを持つシステムでは支配的なものとなっています。これは、スイープの段

ストップ & コピーガベージコレクタの実装

ここからは、レジスタマシン言語を使ってストップ & コピーアルゴリズムをさらに詳しく見ていきます。ここでは、`root` というレジスタがあり、そのレジスタはある構造へのポインタを含んでいて、アクセス可能なデータはそのポインタからすべてたどれると仮定します。この条件を整えるには、ガベージコレクションを行う直前にすべてのマシンレジスタの中身を事前に割り当てたりストに格納し、`root` がそれを指すようにします。¹⁶ さらに、現在のワーキングメモリのほかに、有用なデータをコピーできるフリーメモリがあると仮定します。現在のワーキングメモリは `the-cars` と `the-cdrs` というレジスタにベースアドレスが入っているペクタからなり、フリーメモリは `new-cars` と `new-cdrs` というレジスタにベースアドレスが入っているペクタからなります。

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なくなったときに起動されます。つまり、`cons` 演算がメモリペクタの終端を越えて `free` ポインタをインクリメントしようとしたときということです。ガベージコレクション処理が終わったときには、`root` ポインタは新しいメモリを指し、`root` からアクセス可能なオブジェクトはすべて新しいメモリに移動されています。また、`free` ポインタは新しいメモリの中で新しいペアを割り当てができる次の位置を示しています。それに加えて、ワーキングメモリと新しいメモリの役割が交換されています。新しいペアは `free` の指す位置から始まる新しいメモリの中に構築され、(以前の) ワーキングメモリは次のガベージコレクションのための新しいメモリとして利用可能になっています。Figure 5.15はガベージコレクション直前と直後のメモリの配置を示したものです。

ガベージコレクション処理の状態は、`free` と `scan` という二つのポインタを管理することによってコントロールされています。これらのレジスタは、新しいメモリの先頭を指すように初期化されます。まず、このアルゴリズムは `root` の指すペアを新しいメモリの先頭に再配置します。そのペアはコピーされ、`root` ポインタはインクリメントされます。それに加えて、ペアの古い位

階でメモリすべてをチェックする必要があるマーク & スイープとは対照的です。ストップ & コピーのもうひとつの利点は *compacting*(圧縮) ガベージコレクタであるということです。これは、ガベージコレクション段階が終わったときには有用なデータは連続したメモリ位置に移動させられていて、ゴミとなるペアは排除されているということです。このことは、広く分散したメモリアドレスにアクセスすると余計なページング処理が必要となる仮想メモリを持つ機械では、非常に重要なパフォーマンス上の考慮点になります。

¹⁶ このレジスタリストは、記憶領域割当システムが使うレジスタ — `root`, `the-cars`, `the-cdrs` や、この節で紹介するほかのレジスタは含みません。

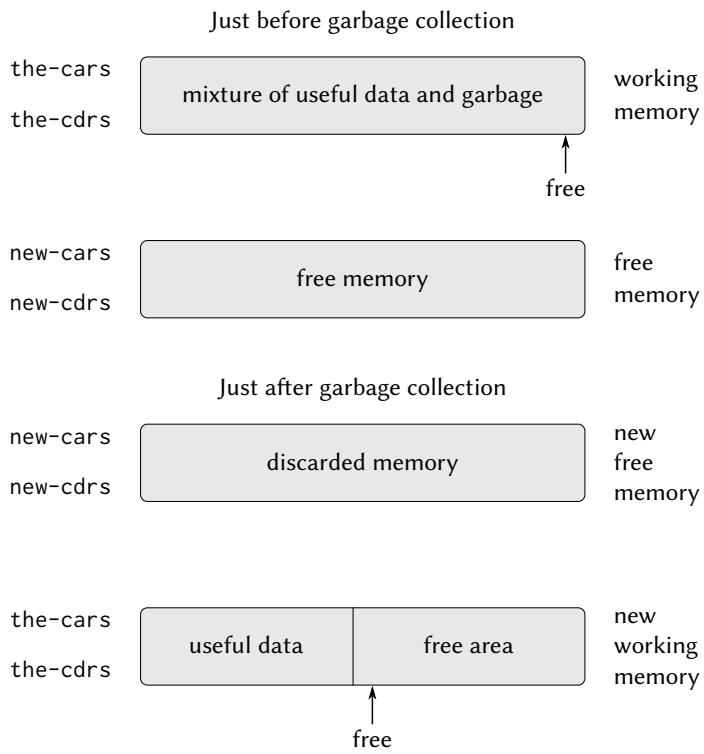


Figure 5.15: ガベージコレクション処理によるメモリの再構成

置にはその中身が移動されたことを示す印をつけます。これは次のように行います。`car` の位置には、これが移動済み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*(失恋ペア)と呼ばれます)¹⁷。`cdr` の位置には、オブジェクトの移動先の位置を指す *forwarding address* (移転先アドレス) を置きます。

`root` を再配置した後は、ガベージコレクタは基本サイクルに入ります。アルゴリズムの各ステップで、`scan` ポインタ (最初は再配置後の `root` を指す) は、新しいメモリに移動されたものの `car` ポインタと `cdr` のポインタが依然として古いメモリ内のオブジェクトを参照しているペアを指しています。それらのオブジェクトはそれぞれ再配置され、`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` ポインタの指すオブジェクト) を再配置するときは、そのオブジェクトが移動済みかどうか (そのオブジェクトの `car` の位置に失恋タグがあるかどうか) を確認します。オブジェクトが移動済みでなければ、`free` によって示される位置にそのオブジェクトをコピーして、`free` を更新し、オブジェクトの古い位置に失恋ペアを設定し、そのオブジェクトへのポインタ (この例では、走査中のペアの `car` ポインタ) がオブジェクトの新しい位置を指すように更新します。オブジェクトが移動済みである場合は、走査中のペアのポインタを、(失恋ペアの `cdr` の位置にある) 移転先に置き換えます。最終的に、アクセス可能なオブジェクトすべての移動と走査が終わります。そのとき `scan` ポインタは `free` ポインタに追いつき、処理は停止します。

トップ & コピーアルゴリズムはレジスタマシンの命令列として規定できます。オブジェクト再配置の基本ステップは `relocate-old-result-in-new` と呼ばれるサブルーチンによって実行されます。このサブルーチンは、再配置対象オブジェクトのポインタという引数を `old` という名前のレジスタから取得します。サブルーチンは指定されたオブジェクトを再配置し (その過程で `free` をインクリメントし)、再配置済みオブジェクトへのポインタを `new` と呼ばれるレジスタに入れてリターンします。リターンは、`relocate-continue` レジスタに格納されたエントリポイントに分岐することによって行います。ガベージコレクションを始めるためには、`free` と `scan` を初期化してからこのサブルーチンを起動して `root` ポインタを再配置します。`root` の再配置が終わったら、`new` ポインタを新しい `root` として設定し、ガベージコレクタのメインループに入ります。

¹⁷失恋ペアという用語は David Cressey による造語です。彼は 1970 年代初期に MIT で開発された Lisp 方言である MDL のガベージコレクタを書きました。

```

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、走査するべきオブジェクトがまだあるかどうか決定する必要があります。そのためには、`scan` ポインタと `free` ポインタが同じ位置にあるかどうかをテストします。ポインタが等しければ、アクセス可能なオブジェクトはすべて再配置が終わっていることになるので、`gc-flip` に分岐します。`gc-flip` では、中断された計算を継続できるように後片付けを行います。まだ走査するべきペアが残っていれば、(`car` ポインタを `old` に入れることによって) 再配置サブルーチンを呼び出して次のペアの `car` を再配置します。サブルーチンからリターンした後に `car` ポインタを更新するように `relocate-continue` レジスタを設定しておきます。

```

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` では、走査中のペアの `car` ポインタを修正し、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` にリターンします。再配置と `cdr` の更新が終わったらそのペアの走査は終わりなので、メインループを続けます。

```

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、次のようにしてオブジェクトを再配置します。まず、(old の指す) 再配置対象の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そのまま (new の中で) 返します (例えば、`car` が数値の 4 であるペアを走査しているとします。ここで、Section 5.3.1 で記述したようにその `car` を `n4` として表現しているとすると、“再配置済み”の `car` のポインタも、そのまま `n4` であってほしいところです)。ペアであれば、再配置を行う必要があります。もし再配置対象のペアの `car` の位置に失恋ペアのタグがあるなら、そのペアは実際にはもう移動済みなので、(失恋ペアの `cdr` の位置から) 移転先を取得し、それを `new` に入れて返します。`old` 内のポインタが移動済みでないペアを指している場合は、そのペアを (`free` の指す) 新しいメモリの最初の空きセルに移動させ、元の位置には失恋タグと移動先を格納して失恋ペアを設定します。`relocate-old-result-in-new` は、`old` の指すオブジェクトの `car` または `cdr` を入れるためにレジスタ `oldcr` を使います。¹⁸

```

relocate-old-result-in-new
  (test (op pointer-to-pair?) (reg old))
  (branch (label pair))
  (assign new (reg old))
  (goto (reg relocate-continue))

```

¹⁸ ガベージコレクタでは、リスト構造の `pair?` 演算ではなく、低レベル述語の `pointer-to-pair?` を使います。これは、実際のシステムではガベージコレクションという目的ではペアとして扱われるものがいろいろとあるかもしれないからです。例えば、IEEE 標準に準拠した Scheme システムでは、手続きオブジェクトは述語 `pair?` を満たさない特別な種類の“ペア”として実装することが許されています。ここでのシミュレーションでは、`pointer-to-pair?` は `pair?` として実装しても問題ありません。

```

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
  (perform (op vector-set!)
    (reg new-cars) (reg new) (reg oldcr))
  (assign oldcr (op vector-ref) (reg the-cdrs) (reg old))
  (perform (op vector-set!)
    (reg new-cdrs) (reg new) (reg oldcr))
  ;; 失恋ペアの構築
  (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を終える前に、ポインタを交換して古いメモリと新しいメモリの役割を交換します。交換するのは `the-cars` と `new-cars`, `the-cdrs` と `new-cdrs` です。これで、次にメモリを使い切ったときにまたガベージコレクションを実行するための準備ができたことになります。

```

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 明示制御評価器

Section 5.1では、簡単な Scheme プログラムをレジスタマシンの記述に変形する方法を見てきました。ここでは、この変形をより複雑なプログラムであるSection 4.1.1–Section 4.1.4のメタ循環評価器について行います。メタ循環評価器は、Scheme インタプリタのふるまいを `eval` 手続きと `apply` 手続きによって記述する方法を示しています。この節で開発する *explicit-control evaluator*(明示制御評価器) は、評価処理の基層として使われている手続き呼び出しと引数渡しのメカニズムをレジスタとスタックの演算によって記述する方法を示します。さらに、明示制御評価器は従来型コンピュータの機械語によく似た言語で書かれた Scheme インタプリタの実装もあります。この評価器はSection 5.2のレジスタマシンシミュレータで実行できます。Scheme 評価器の機械語による実装を作るための最初の一歩としてこの評価器を使うこともできますし、Scheme の式を評価する特殊用途マシンを作るうえでの足がかりとすることさえ可能です。Figure 5.16にそのようなハードウェアによる実装を示します。これは、Scheme の評価器として動作するシリコンチップです。チップの設計者は、この節で記述する評価器に似たレジスタマシンのデータパスとコントローラの仕様からスタートし、集積回路のレイアウトの構築には設計自動化プログラムを使っています。¹⁹

レジスタと演算

明示制御評価器を設計するにあたっては、レジスタマシンで使う演算を規定する必要があります。メタ循環評価器は `quoted?` や `make-procedure` のような手続きを使って抽象構文によって記述しました。レジスタマシンを実装する際には、これらの手続きを基本的なリスト構造メモリ演算の列として展開し、これらの演算をレジスタマシン上で実装することも可能です。しかし、そうすると評価器はとても長くなり、基本的な構造が実装の細部に紛れてわかりにくくなってしまいます。見た目をすっきりさせるために、Section 4.1.2で出てきた構文手続きや、Section 4.1.3とSection 4.1.4で出てきた環境その他の実行時データを表現する手続きはレジスタマシンの基本演算に含めることにします。低レベルの機械語によるプログラムやハードウェアでの実装が可能な評価器を完全に規定するためには、Section 5.3に記述したリスト構造の実装を使って、より基本的な演算によってこれらの演算を置き換えることになります。

¹⁹ このチップとその設計手法についての詳細はBatali et al. 1982を参照。

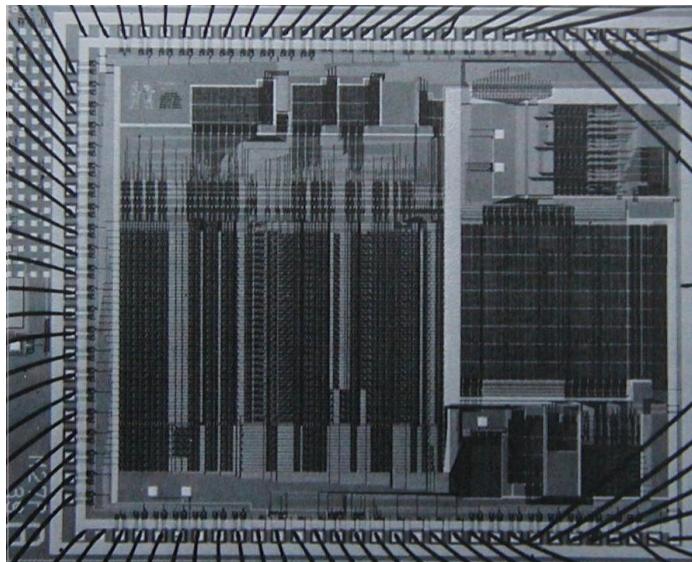


Figure 5.16: Scheme 評価器のシリコンチップ実装

ここで作る Scheme 評価器レジスタマシンは、ひとつのスタックと七個のレジスタ、`exp`, `env`, `val`, `continue`, `proc`, `argl`, `unev`を持っています。`exp`は評価対象の式を保持し、`env`は評価を行う環境を保持します。評価終了時には、指定された環境で式を評価することによって得られた結果が `val` に入ります。`continue` レジスタは、Section 5.1.4で説明したように再帰の実装に使います（評価器は自分自身を再帰的に呼び出す必要があります。式の評価の際には、その部分式を評価することが必要になるからです）。レジスタ `proc`, `argl`, `unev` は組み合わせの評価に使います。

ここでは、評価器のレジスタと演算の接続をデータパス図を描いて示すことはしません。また、マシンの演算の完全なリストも提供しません。それらは評価器のコントローラから間接的にわかるからです。コントローラについて詳しく解説します。

5.4.1 明示制御評価器のコア

評価器の中心となる要素は、`eval-dispatch` から始まる命令列です。これは Section 4.1.1 で記述したメタ循環評価器の `eval` 手続きに対応するものです。コントローラが `eval-dispatch` からスタートすると、コントローラは `exp` によって指定される式を `env` によって指定される環境の中で評価します。評価が完了すると、コントローラは `continue` に格納されたエントリポイントに行きます。そのとき、`val` レジスタには式の値が入っています。メタ循環評価器の `eval` と同じように、`eval-dispatch` は、評価対象の式の構文型に対する場合分けという構造になっています。²⁰

```
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))
```

²⁰ このコントローラでは、ディスパッチは `test` と `branch` の命令列として書かれています。別のやり方として、テストを連続して実行することを避けるため、また新しい式の型の定義をやりやすくするために、ディスパッチをデータ主導スタイルで書くことも考えられます（また、実際のシステムではおそらくそうなっています）。Lisp 実行のために設計されたマシンは、そのようなデータ主導ディスパッチを効率的に実行するための `dispatch-on-type` 演算を持っているでしょう。

単純な式の評価

(自己評価の) 数値と文字列、変数、クオート式、lambda 式には、評価するべき部分式はありません。これらに対しては、評価器は単純に正しい値を val レジスタに入れ、`continue` によって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コードによって実行されます。

```
ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))
ev-variable
  (assign
    val (op lookup-variable-value) (reg exp) (reg env))
  (goto (reg continue))
ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))
ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  (assign val (op make-procedure)
    (reg unev) (reg exp) (reg env))
  (goto (reg continue))
```

ここで、`ev-lambda` は `unev` と `exp` レジスタを使ってラムダ式の引数と本体を保持し、`env` 中の環境とともに `make-procedure` 演算に渡せるようにしています。

手続き適用の評価

手続き適用は、演算子とオペランドを持つ組み合わせによって規定されます。演算子は、その値が手続きとなる部分式で、オペランドは、その値が手続き適用対象の引数となる部分式です。メタ循環評価器の `eval` は、自分自身を再帰的に呼んで組み合わせの各要素を評価し、それから結果を `apply` に渡します。`apply` が実際の手続き適用を実行します。明示制御評価器も同じことを行います。再帰呼び出しは、`goto` 命令と、再帰呼び出しから戻ったときに復元できるようにスタックを使ってレジスタを保存することによって実装します。そ

それぞれの呼び出しの前には、どのレジスタを保存するべきか注意深く見極める必要があります（値が後で必要になるからです）。²¹

適用の評価では、まず演算子を評価して手続きを生成します。これがのちに、評価済みのオペランドに適用されることになります。演算子を評価するには、演算子を `exp` レジスタに移し、`eval-dispatch` に進みます。`env` レジスタに入っている環境は、そのままでも演算子を評価する適切な環境になっていますが、後でオペランドの評価にもこの環境が必要になるので、`env` を保存しておきます。また、オペランドを `unev` に展開し、それをスタックに保存します。`continue` は、演算子の評価が終わった後に `eval-dispatch` が `ev-appl-did-operator` から再開できるように設定します。ですがその前に、最初の部分で、適用処理が終わった後に継続するべき場所をコントローラに伝える `continue` の古い値を保存しておきます。

```
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))
```

演算子の部分式の評価から戻ると、組み合わせのオペランドを評価し、返される引数を `argl` に保存されるリストに集積する処理に進みます。まず、評価前のオペランドと環境を復元します。`argl` は空リストに初期化します。それから、演算子を評価することによって生成された手続きを `proc` レジスタに代入します。オペランドがない場合は、直接 `apply-dispatch` に進みます。オペランドがある場合は、`proc` をスタックに保存し、引数評価ループを開始します。

²¹ この点は、Lisp のような手続き型言語からレジスタマシン言語にアルゴリズムを変換するにあたって重要なところであり、また難しいところもあります。必要なものだけを保存するというやり方以外に、再帰呼び出しの前に (val 以外の) すべてのレジスタを保存するというやり方も考えられます。これは、*framed-stack*(スタックフレーム) という呼ばれる方式です。この方式は問題なく動作しますが、必要以上のレジスタを保存することにもなります。スタック演算が高価なシステムでは、これは重要な考慮点になります。後で使う中身の必要なないレジスタを保存すると、本来ガベージコレクションして領域を解放・再利用できたはずの不要なデータを持ち続けることにもつながります。

```

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)

```

引数評価ループのそれぞれのサイクルでは、`unev` 内のリストからオペランドをひとつ取得してそれを評価し、その結果を `argl` に集積していきます。オペランドを評価するときは、オペランドを `exp` レジスタに入れ、引数集積フェーズから実行を再開できるように `continue` を設定してから `eval-dispatch` に進みます。しかし、まずはそれまでに集積した引数 (`argl`)、環境 (`env`)、未評価の残りのオペランド (`unev`) を保存します。最後のオペランドの評価は特別扱いとなり、`ev-appl-last-arg` で処理します。

```

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))

```

²²Section 4.1.3 の評価器データ構造の手続きのほかに、引数リストを走査する次の二つの手続きを追加します。

```

(define (empty-arglist) '())
(define (adjoin-arg arg arglist) (append arglist (list arg)))

```

さらに、組み合わせの最後のオペランドかどうかのテストを行う追加の構文手続きも使います。

```
(define (last-operand? ops) (null? (cdr ops)))
```

オペランドが評価されると、値が `argl` 内のリストに集積されます。次に、そのオペランドは `unev` 内の未評価オペランドのリストから削除され、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処理になります。最後の引数を評価した後は必要がなくなるため、`eval-dispatch` に進む前に環境や未評価オペランドリストを保存する必要はありません。そのため、評価が終わると、`ev-appl-accum-last-arg` という特別なエントリポイントに帰ります。この手続きは引数リストを復元し、新しい引数を集積し、保存した手続きを復元し、適用実行処理に移動します。²³

```
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))
```

組み合わせの中のオペランドをインタプリタが評価する順番(左から右、あるいは右から左など。[Exercise 3.8](#) 参照)は、引数評価ループの実装によって決定されます。メタ循環評価器では、実装に使った基層の Scheme から制御構造を継承しているため、評価順はメタ循環評価器によって決定されるわけではありません。

²³ 最後のオペランドの処理を特別扱いするこの最適化は、*evals 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扱いすると、引数評価ループをもう少し効率よくすることができます。その場合、`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、`argl` を保存することを避けられます。[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` 手続きと比較してください)。

ません。²⁴ (`ev-appl-operand-loop` で一続きのオペランドを `unev` から抽出するために使う) `first-operand` セレクタは `car` として実装されていて、`rest-operands` は `cdr` として実装されているので、明示制御評価器は組み合わせのオペランドを左から右という順番で評価します。

手続き適用

エントリポイン `apply-dispatch` は、メタ循環評価器の `apply` 手続きに対応するものです。`apply-dispatch` に着くときには、`proc` レジスタには適用する手続きが入っていて、`argl` にはその手続きの適用対象となる評価済み引数のリストが入っています。保存された `continue` の値(元は `eval-dispatch` に渡され、それが `ev-application` で保存されたもの)がスタック上にあり、これが手続き適用の結果を持って帰る先の場所を示しています。適用が終わると、コントローラは保存された `continue` の指すエントリポインに移動します。このとき、`val` には適用の結果が入っています。メタ循環評価器の `apply` と同じように、二つの場合を考える必要があります。適用する手続きが基本手続きである場合と、複合手続きである場合です。

```
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))
```

基本手続きはそれぞれ、引数を `argl` から取得し、結果を `val` に入れるよう実装されていると想定します。マシンの基本手続きの扱いを規定するには、それぞれの basic手続きを実装するコントローラ命令列を提供し、`primitive-apply` が `proc` の中身によって基本手続きを特定し、その手続きの命令列にディスパッチするようにする必要があるところです。ここで見たいのは評価処理の構造であって、基本手続きの細かい実装ではないため、その代わりに、単純に `proc` 内の手続きを `argl` 内の引数に適用する `apply-primitive-procedure` 演算を使うことにします。[Section 5.2](#) のシミュレータを使って評価器のシミュレーションを行うという目的のために、基層の Scheme システムを呼び出し適用を実行

²⁴ メタ循環評価器では、オペランドの評価順は、[Section 4.1.1](#) の手続き `list-of-values` 内の `cons` に対する引数の評価順によって決定されます ([Exercise 4.1](#) 参照)。

する `apply-primitive-procedure` を使うのですが、これはSection 4.1.4のメタ循環評価器で行ったことと同じです。

基本手続きの適用の値を計算してから、`continue` を復元して、指定されたエントリポイントに移ります。

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

複合手続きの適用はメタ循環評価器と同じような進め方になります。手続きの仮引数を引数に束縛するフレームを構築し、手続きの持つ環境をそのフレームによって拡張し、その拡張した環境の中で手続きの本体となる式の列を評価します。列の評価は、この先Section 5.4.2で説明する `ev-sequence` で処理します。

```
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` レジスタに新しい値が割り当てられるインタプリタ内で唯一の場所です。メタ循環評価器の場合と同じように、新しい環境は、手続きの持っている環境と、引数リストと、それに対応する束縛する変数のリストから構築します。

5.4.2 列の評価と末尾再帰

明示制御評価器の `ev-sequence` の部分は、メタ循環評価器の `eval-sequence` 手続きと構造が似ています。この手続きが扱うのは、手続き本体の式の列と、明示的に `begin` 式によって表される式の列です。

明示的な `begin` 式は、`unev` に評価する式の列を入れ、`continue` をスタッカに保存し、`ev-sequence` にジャンプすることによって評価します。

```
ev-begin
```

```
(assign unev (op begin-actions) (reg exp))
(save continue)
(goto (label ev-sequence))
```

手続き本体の暗黙的な列は、compound-apply から ev-sequence にジャンプすることによって処理します。このとき、continue はすでに ev-application で保存されたものがスタック上にあります。

ev-sequence と ev-sequence-continue のエントリポイントは、順番に列のそれぞれの式を評価するループになっています。未評価の式のリストは unev に保持されています。それぞれの式を評価する前に、評価するべき式がまだ列にあるかどうか確認します。まだあるなら、未評価の式の残り (unev に保持されている) と、それらを評価する環境 (env に保持されている) を保存してから、eval-dispatch を呼んで式を評価します。この評価から戻ると、ev-sequence-continue で保存した二つのレジスタが復元されます。

列の最後の式は特別扱いとなり、ev-sequence-last-exp で処理します。その式の後にはもう評価するべき式はないため、eval-dispatch に行く前に unev と env を保存する必要はありません。最後の式の値が列全体の値となるため、最後の式の評価が終わった後にするべきことは、現時点でスタックにあるエントリポイント (これは ev-application または ev-begin で保存されたものです) から継続することだけです。eval-dispatch がこの場所に帰ってくるように continue を設定して、その後スタックから continue の値を復元して、そのエントリポイントから継続するということをする代わりに、eval-dispatch に行く前にスタックから continue を復元します。こうすると、eval-dispatch が式を評価した後にそのエントリポイントから継続します。

```
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](#)で、次のような手続きによって記述されるようなプロセスは反復プロセスであると書きました。

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

この手続きは構文的には再帰的（自分自身を使って定義されている）なのですが、ひとつの `sqrt-iter` 呼び出しから次の呼び出しに移る際に評価器が情報を保存する論理的必要性はありません。²⁵ `sqrt-iter` のような手続きが自分自身を呼び出し続ける間に記憶領域を増やさずに実行できる評価器は、*tail-recursive*（末尾再帰）評価器と呼ばれます。[Chapter 4](#)のメタ循環評価器の実装は、評価器が末尾再帰であるかどうかを規定しません。これは、この評価器が状態を保存するメカニズムが基層の Scheme から受け継いだものだからです。しかし、明示制御評価器の場合は、評価プロセスをたどって、手続き呼び出しがスタック上に情報の最終的な集積を行うタイミングを調べることができます。

私たちの評価器は末尾再帰です。列の最後の式を評価する際に、スタックに何も情報を保存せずに直接 `eval-dispatch` に移動するからです。そのため、列の最後の式を評価しても—それが手続き呼び出しであっても（例えば、`sqrt-iter` では手続き本体の最後の式である `if` 式が `sqrt-iter` の呼び出しに簡約されます）—スタックに何か情報が積まれることはありません。²⁶

²⁵ [Section 5.1](#)では、そのようなプロセスをスタックを持たないレジスタマシンで実装する方法を学びました。この場合、プロセスの状態は固定のレジスタ集合に格納されます。

²⁶ `ev-sequence` のこの末尾再帰の実装は、多くのコンパイラで使われる有名な最適化テクニックの一種です。手続き呼び出しで終わる手続きをコンパイルする際には、その呼び出しを呼び出される手続きのエントリポイントへのジャンプによって置き換えるこ

この場合に情報を保存する必要はないということを利用することを考えていなければ、`eval-sequence` の実装で、列内の式をすべて同じ扱いにしていたかもしれません。つまり、レジスタを保存し、式を評価し、戻ってレジスタを復元するということをすべての式の評価が終わるまで繰り返すということです。²⁷

```
ev-sequence
  (test (op no-more-exp?) (reg unev))
  (branch (label ev-sequence-end))
  (assign exp (op first-exp) (reg unev))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exp) (reg unev))
  (goto (label ev-sequence))
ev-sequence-end
  (restore continue)
  (goto (reg continue))
```

これは以前の列評価のコードに対するちょっとした変更のように見えるかもしれません。違うのは、列の最後の式で、ほかの式の場合と同じように、保存・復元サイクルを通るという点だけです。しかし、これは末尾再帰の実装に対しては致命的な変更となります。こうしてしまうと、列の最後の式を評価した後に、(不要な) 保存レジスタを復元するために戻らなければならなくなるからです。こうやって余分に保存する分は、手続き呼び出しのネストの間に蓄積されていきます。結果として、`sqrt-iter` のようなプロセスが必要とする領域は一定ではなくなり、繰り返し回数に比例するようになってしまいます。この違い

とができます。この節でやったように、この戦略をインタプリタに組み入れると、言語全体で一律に最適化が有効になります。

²⁷`no-more-exp?` は以下のように定義できます。

```
(define (no-more-exp? seq) (null? seq))
```

が大きな影響を及ぼすこともあります。例えば、末尾再帰があれば、無限ループは手続き呼び出しのメカニズムだけで表現できます。

```
(define (count n)
  (newline) (display n) (count (+ n 1)))
```

末尾再帰がなければ、このような手続きは最終的にはスタック領域を使い尽くしてしまうことになります。真の反復を表現しようと思うと、手続き呼び出し以外の何らかの制御メカニズムが必要になるでしょう。

5.4.3 条件文、代入、定義

メタ循環評価器と同じように、特殊形式は式の一部を選択的に評価することによって処理します。if 式の場合は、まず述語を評価して、その値をもとに結果部と代替部のどちらを評価するか決定します。

述語を評価する前に、後で結果部または代替部を抽出するできるように if 式自身を保存しておきます。さらに、環境も保存します。これは、結果部または代替部を評価する際に必要となります。それに加えて、continue も保存します。これは、if の値を受け取る式の評価に戻るために後で必要となります。

```
ev-if
  (save exp) ; 後で使うために式を保存
  (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
```

述語の評価から戻るとき、述語が真であったか偽であったかをテストし、eval-dispatch に飛ぶ前にその結果によって結果部か代替部を exp に入れます。ここで env と continue を復元することによって、eval-dispatch が正しい環境を持つようにし、また if 式の値を受け取る正しい場所から継続するようにしていることに注意してください。

```
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` で処理します。ここに到達するのは、`eval-dispatch` で `exp` に代入式が入っているときです。`ev-assignment` のコードは、まず式の値部分を評価し、それから新しい値を環境に組み込みます。`set-variable-value!` はマシン演算として利用可能であると仮定します。

```

ev-assignment
  (assign unev (op assignment-variable) (reg exp))
  (save unev) ; 後で使うために変数を保存
  (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
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)))

```

定義も似たような方法で処理します。

```

ev-definition
  (assign unev (op definition-variable) (reg exp))
  (save unev) ; 後で使うために変数を保存
  (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
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: 評価器を拡張して、`cond` や `let` などの派生式を扱えるようにせよ (Section 4.1.2)。“する”をして、`cond->if` のような構文変換器がマシン演算として利用可能であると仮定してもよい。²⁸

Exercise 5.24: `cond` を `if` に簡約せずに新しい基本特殊形式として実装せよ。真である述語を見つけるまで連続した `cond` 節の述語をテストしていくループを構築し、それから `ev-sequence` を使ってその節のアクションを評価する必要があるだろう。

Exercise 5.25: Section 4.2 の遅延評価器をもとに評価器を修正して、正規順序評価を使うようにせよ。

5.4.4 評価器の実行

明示制御評価器の実装で、Chapter 1 から続いてきた開発は終わりを迎えることになります。この開発を通して、私たちは評価プロセスのモデルを精緻化していくことを目指してきました。最初はそれほど厳密でない置換モデルからスタートし、それから Chapter 3 でこれを環境モデルに拡張しました。環境モデルでは状態と変更が扱えるようになりました。Chapter 4 のメタ循環評価器では、式の評価の間に構築される環境構造をより明確にするための言語として

²⁸ これは、実際にはするではありません。実際に評価器を一から実装するとしても、実行の前の構文フェーズで、明示制御評価器を使って `cond->if` のようなソースレベルの変換を行う Scheme プログラムを解釈させることになるでしょう。

Scheme そのものを使いました。この章では、レジスタマシンを導入することで、評価器のメモリ管理、引数渡し、制御のメカニズムを詳しく見てきました。新しいレベルの記述に取り組むたびに、それ以前の、そこまで正確ではない評価処理では明らかになっていなかった問題を提起したり、曖昧性を解決したりする必要がありました。明示制御評価器のふるまいを理解するにあたっては、それをシミュレートしてそのパフォーマンスを監視するということが可能です。

ここでは、この評価器にドライバループを組み込みます。これは、Section 4.1.4 の `driver-loop` 手続きにあたる役割を果たします。評価器はプロンプトを表示し、式を読み込み、`eval-dispatch` に行って式を評価し、結果を表示するということを繰り返します。以下に示す命令列が、明示制御評価器のコントローラ命令列の先頭部分になります。²⁹

```
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))
```

手続きの中で (`apply-dispatch` の示す “未知の手続き型エラー” のような) エラーにぶつかると、エラーメッセージを表示してドライバループに戻ります。

²⁹ ここでは、`read` やさまざまな表示命令が基本マシン演算として利用可能であると仮定しています。これはシミュレーションには便利ですが、実際にはまったく非現実的です。これらは本当は非常に複雑な演算です。実際には、これらの演算は、デバイスとの間で個別の文字をやりとりするような低レベルの入出力演算を使って実装することになるでしょう。

`get-global-environment` 演算をサポートするには、以下のように定義します。

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

```

unknown-expression-type
  (assign val (const unknown-expression-type-error))
  (goto (label signal-error))
unknown-procedure-type
  (restore continue)      ;(apply-dispatchの) スタックをクリアする
  (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ために、ドライバループを通るたびにスタックを初期化します。(未定義変数のような) エラーによって評価が中断された後は、スタックが空でない可能性があるためです。³¹

Section 5.4.1からSection 5.4.4までで提示したコード片をすべてくっつけると、Section 5.2のレジスタマシンシミュレータを使って実行できる評価器マシンの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
      (ここまでで提示したマシンコントローラ全体) )))

```

このほか、評価器が使う基本演算をシミュレートするために Scheme 手続きを定義する必要があります。これらはSection 4.1のメタ循環評価器で使ったのと同じ手続きです。それに加えて、Section 5.4を通して脚注で定義してきたいいくつかの手続きがあります。

```

(define eceval-operations
  (list (list 'self-evaluating? self-evaluating)

```

³⁰ インタプリタに扱わせたいエラーはほかにもありますが、それらはそれほど単純ではありません。Exercise 5.30参照。

³¹ エラーが起こったときだけスタックの初期化を行うことも可能ですが。しかし、ここから先で見ていくように、ドライバループの中で初期化するようにすると評価器のパフォーマンスを監視するのに便利です。

⟨明示制御評価器の全演算リスト⟩))

これで、グローバル環境を初期化して評価器を実行できるようになります。

```
(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、この方法で式を評価すると、Scheme に直接入力する場合よりもずっと時間がかかります。この方式では何段ものシミュレーションがあるからです。式は明示制御評価器マシンに評価されるのですが、それは Scheme プログラムによってシミュレートされたもので、そのプログラムはそれ自身 Scheme インタプリタに評価されることになります。

評価器のパフォーマンス監視

シミュレーションは評価器の実装を導いてくれる強力なツールとして使えます。また、シミュレーションはレジスタマシンの設計をいろいろ試すのに便利なだけでなく、シミュレーション対象の評価器のパフォーマンスを監視するのにも使えます。例えば、パフォーマンスの重要な要因のひとつとして、評価器がスタックをどれだけ効率的に使えるかということがあります。さまざまな式を評価するうえで必要となるスタック演算の数は、スタック使用の統計を取るバージョンのシミュレータ (Section 5.2.4) によって評価器のレジスタマシンを定義して、さらに評価器の print-result エントリポイントにその統計情報を表示する命令を追加することによって、観察することができます。

```
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
```

こうすると、評価器とのやりとりは以下のようになります。

```
;;; 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
```

評価器のドライバループはやりとりのたびにスタックを再初期化するため、表示される統計情報は直前の式の評価に使われたスタック演算のみについてのものになります。

Exercise 5.26: 監視機能つきスタックを使って評価器の末尾再帰の特性 (Section 5.4.2) を調査せよ。評価器をスタートさせ、Section 1.2.1の反復 factorial 手続きを定義せよ。

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

いくつかの小さな n の値を使ってこの手続きを実行せよ。それらの値の $n!$ を求めるのに必要となった最大スタック深度とプッシュ回数を記録せよ。

a $n!$ を評価するのに必要な最大深度が n から独立していることがわかるだろう。その深さはいくつか。

b 記録したデータから、任意の $n \geq 1$ について $n!$ を評価するのに使われる push 演算の総数を表す n の方程式を決定せよ。使われる演算の数は n の線形関数であるため、定数二つによって決定される。

Exercise 5.27: Exercise 5.26との比較のために、階乗を再帰的に求める以下の手続きのふるまいを調査せよ。

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

この手続きを監視機能つきのスタックを使って実行し、任意の $n \geq 1$ に対して $n!$ を評価するのに使われるスタックの最大深度とプッシュの総数を n の関数として決定せよ（この二つの関数はまた線形になる）。以下の表に適切な n の式を埋めて実験結果をまとめよ。

	最大深度	push の総数
階乗 (再帰)		
階乗 (反復)		

最大深度は、計算の実行の際に評価器が使う記憶領域の量を表している。プッシュの総数は必要な時間とよく相関している。

Exercise 5.28: eval-sequence を Section 5.4.2 で記述したように変更して評価器の定義を修正し、評価器が末尾再帰でなくなるようにせよ。Exercise 5.26 と Exercise 5.27 の実験を再実行し、factorial 手続きのどちらのバージョンも、入力に対して線形に増加する記憶領域を必要とするようになっていることを示せ。

Exercise 5.29: 木の再帰によるフィボナッチ数計算のスタック演算を監視せよ。

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2))))))
```

a $n \geq 2$ について $\text{Fib}(n)$ を計算するのに必要なスタックの最大深度を n の方程式で表せ。ヒント：Section 1.2.2 では、このプロセスの使用する記憶領域は n に対して線形に増加することを論じた。

- b $n \geq 2$ について $\text{Fib}(n)$ を計算するのに使われるプッシュの総数を方程式で表せ。プッシュの総数（これは消費時間とよく相関する）が n に対して指数的に増加することがわかるだろう。ヒント： $S(n)$ を $\text{Fib}(n)$ を計算するのに使われるプッシュの総数とする。すると、 $S(n-1)$, $S(n-2)$, それに加えて n と独立した定数 k という固定の“オーバヘッド”を使って $S(n)$ を表す方程式があると主張できるだろう。この方程式を示し、また k の値を求めよ。それから、 $S(n)$ が $a \cdot \text{Fib}(n+1) + b$ として表せることを示し、 a と b の値を与えるよ。

Exercise 5.30: 今のところ、評価器は二種類のエラー—未知の型と未知の手続きの型—だけキャッチして知らせるようになっている。ほかのエラーがあると評価器の REPL から抜けてしまう。この評価器をレジスタマシンシミュレータを使って実行している場合は、これらのエラーは基層の Scheme システムにキャッチされる。これは、ユーザプログラムがエラーを起こしたときにコンピュータがクラッシュするというのに相当する。³² 実際のエラーシステムを動かすというのは一大プロジェクトになるが、エラーシステムにはどういったことが必要なのかを理解するということは、その労力に十分見合うものだ。

- a 評価プロセスで起こるエラー（未束縛変数へのアクセスの試行など）は、探索演算に修正を加えて、特別な状態コードを返すようにすることでキャッチできる。状態コードはユーザ変数の取り得る値であってはならない。評価器はこの状態コードをテストし、`signal-error` に行くために必要なことをを行う。評価器の中でそのような変更が必要となる箇所をすべて探して修正せよ。これは大変な作業だ。
- b もっと大変なのは、基本手続き適用によって発せられるエラーの扱いという問題だ。そのようなエラーには、ゼロ除算の試行や、記号から `car` を取り出そうとすることなどがある。プロによって書かれた高品質システムでは、基本手続きの適用は、その基本手続きの一部として、安全チェックが行

³² 残念ながら、C 言語のような従来のコンパイラベースの言語システムではこれが普通の状況です。UNIX(tm) ではシステムは“コアをダンプ”し、DOS/Windows(tm) ではフリーズします。Macintosh(tm) は、爆発する爆弾の絵を表示して、コンピュータを再起動する機会を与えてくれます—運がよければですが。

われている。例えば、`car` の呼び出しでは、引数がペアであるかをまずチェックするようにできるだろう。引数がペアでなければ、その適用は評価器に特別な状態コードを返し、それを受けて評価器が失敗を報告する。レジスタマシンシミュレータでは、それぞれの基本手続きが適用可能性チェックを行い、失敗時には特別な状態コードを適切に返すようにすることによって、これが実現できる。こうすると、評価器の `primitive-apply` コードは、状態コードをチェックしたうえで必要であれば `signal-error` に行くようにすることができる。この構造を構築し、動くようにせよ。これは一大プロジェクトだ。

5.5 コンパイル

Section 5.4 の明示制御評価器は、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、コントローラが Scheme インタプリタではないレジスタマシンで Scheme プログラムを動かす方法について見てていきます。

明示制御評価器は万能です。この評価器は、Scheme で記述可能な計算プロセスであればどんなものでも実行できます。評価器のコントローラは、望み通りの計算を実行するために、評価器のデータパスの使い方を指揮します。このため、評価器のデータパスは万能です。適切なコントローラがあれば、望み通りの計算を十分実行できるのです。³³

商用の汎用コンピュータは、効率的で便利な万能データパスを構成するレジスタと演算の集合を中心として構成されているレジスタマシンです。汎用マシンのコントローラは、ここまで使ってきたようなレジスタマシン言語のインタプリタです。この言語はマシンの *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、そのマシンのデータパスを使う命令の列です。例えば、明示制御評価器の命令列は、特化したインタプリタマシンのコントローラとして考えるのでは

³³ これは理論的にはそうだということです。この評価器のデータパスが汎用コンピュータのデータパスとして特に便利であるとか効率的であるとか主張しているわけではありません。例えば、このデータパスは高性能浮動小数点計算やビットベクタを集中的に走査するような計算を実装するのにはあまり向いていません。

なく、汎用目的のコンピュータの機械語プログラムとして考えることができます。

高級言語とレジスタマシン言語の橋渡しをする戦略として一般的なものは二つあります。明示制御評価器が示しているのは解釈という戦略です。機械のネイティブ言語で書かれたインタプリタ(解釈器)は、評価を実行するマシンのネイティブ言語とは異なる言語(*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マシンを設定します。ソース言語の基本手続きは、マシンのネイティブ言語によって書かれたサブルーチンライブラリとして実装されます。解釈対象のプログラム(*source program*(ソースプログラム))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たどりながらソースプログラムの構造を解析していきます。その過程で、インタプリタはライブラリから適切な基本手続きのサブルーチンを呼ぶことによってソースプログラムの意図されたふるまいをシミュレートします。

この節で見ていくのは、*compilation*(コンパイル)というもうひとつの戦略です。あるソース言語とマシンに対するコンパイラは、ソースプログラムをそのマシンのネイティブ言語で書かれた等価なプログラム(*object program*(オブジェクトプログラム))に変換します。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制御評価器のデータバスを使って実行する命令列に変換するものです。³⁴

解釈という戦略に比べると、コンパイルという戦略はプログラム実行の効率性を大きく向上させることができます。このことはのちにコンパイラの概観のところで説明します。一方、インタプリタは対話的にプログラムを開発したりデバッグしたりするためのずっと強力な環境を提供してくれます。実行されているソースプログラムを実行時に検査したり修正したりできるからです。それに加えて、基本手続きのライブラリ全体があるため、デバッグ中に新しいプログラムを構築してシステムに追加するということも可能です。

コンパイルと解釈には互いに補う利点があるという観点から、現代のプログラム開発環境は混合戦略を目指しています。Lisp インタプリタは、一般的に解釈される手続きとコンパイル済みの手続きが互いに呼び出せるように構成さ

³⁴ 実際は、コンパイルされたコードを実行するマシンはインタプリタのマシンよりも単純にできます。コードを実行するマシンでは `exp` レジスタと `unev` レジスタを使わないからです。インタプリタはこれらを使って未評価の式の一部を保持していました。しかし、コンパイラの場合はそれらの式はレジスタマシンが実行するコンパイル済みコードの中に組み込まれることになります。それと同じ理由で、式の構文を扱う機械語演算も必要なくなります。しかし、コンパイル済みのコードは、明示制御評価器にはなかった(コンパイル済み手続きオブジェクトを表現するための)機械語演算を追加でいくつか使います。

れています。そのおかげで、プログラマはデバッグ済みと想定されるプログラムの部分をコンパイルして効率性という利点を得ると同時に、対話的開発とデバッグという流れの中にあるプログラムの部分の実行をインタプリタモードのままにしておくことができます。コンパイラを実装した後のSection 5.5.7では、コンパイラとインタプリタと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作る方法を示します。

コンパイラの概要

ここで作るコンパイラは、構造と実行する機能の両面で、ここまで作ったインタプリタによく似ています。そのため、コンパイラが式を解析するメカニズムはインタプリタのものと似たものになります。さらに、コンパイル済みコードと解釈されるコードとの接続を簡単にするため、出力するコードがインタプリタのレジスタの用法と同じ約束に従うものになるようにコンパイラを設計します。つまり、環境は `env` レジスタに保持し、引数リストは `arg1` に集積し、適用する手続きは `proc` に入れ、手続きは答えを `val` に入れて戻り、手続きが戻るべき場所は `continue` に入れるようにするということです。全体として、コンパイラがソースプログラムをオブジェクトプログラムに翻訳するとき、インタプリタが同じソースプログラムを評価する際に行われるのと本質的に同じレジスタ演算をオブジェクトプログラムが行うように翻訳を行います。

このように考えると、基本的なコンパイラを実装する戦略が見えてきます。まず、インタプリタと同じように式をたどっていきます。そして、インタプリタであれば式の評価中に行うであろうレジスタ命令にぶつかると、その命令の実行はせずに、その代わりにそれを列に集積していきます。その結果としてできる命令列がオブジェクトコードになります。コンパイラーが解釈に対して性能面での利点があるということがわかるでしょうか。インタプリタが式—例えば (`f 84 96`)—を評価するたびに、式の分類(これが手続き適用であるということを知る)とオペランドリストの終端テスト(オペランドが二個であるということを知る)という仕事を行います。コンパイラの場合、式はコンパイル時に命令列が生成されるときに一度解析されるだけです。コンパイラに生成されたオブジェクトコードは、演算子と二つのオペランドを評価し、引数リストを組み立て、(`proc` にある)手続きを (`arg1` にある) 引数に適用するという命令しか含んでいません。

これは、Section 4.1.7 の解析評価器で実装したものと同じ種類の最適化です。しかし、コンパイル済みコードでは、さらに効率を上げられる機会があります。インタプリタの実行時には、言語のどんな式に対しても使える必要があ

るプロセスに従って動作しています。それに対して、コンパイル済みコードのある一部は、ある特定の式を実行することが想定されています。これは大きな違いになることもあります。例えば、スタックを使ってレジスタを保存する場合です。インタプリタが式を評価する際には、インタプリタはどんな事態に対しても備えておく必要があります。部分式を評価する前には、インタプリタは後で必要になるレジスタをすべて保存します。これは、部分式がどんな評価を必要とするかわからないからです。一方、コンパイラの場合は、処理している特定の式の構造を利用して、不必要的なスタック演算を行わないコードを生成することができます。

一例として、組み合わせ (`f 84 96`) について考えてみましょう。インタプリタが組み合わせの演算子を評価する前に、オペランドと環境を持つレジスタを保存してこの評価に備えます。レジスタの値は後で必要になります。次に、インタプリタは演算子を評価して、その結果を `val` に取得し、保存したレジスタを復元し、最後に結果を `val` から `proc` に移します。しかし、いま扱っているこの特定の式では、演算子は記号 `f` であり、その評価は機械語演算の `lookup-variable-value` によって行われます。この演算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ことを利用して、次の命令によって演算子を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しないでませるほかに、検索した値を直接 `proc` に割り当てるということもしています。インタプリタであれば、結果を `val` の中に取得し、その後 `proc` に移すところです。

コンパイラは環境へのアクセスも最適化することができます。コードを解析した状態では、コンパイラは多くの場合どのフレームに特定の変数があるかを知っていて、`lookup-variable-value` による検索を実行せずに、そのフレームに直接アクセスできます。そのような変数アクセスの実装方法については [Section 5.5.6](#) で検討します。しかし、それまでの間は、上で記述したようなレジスタとスタックの最適化に焦点を合わせることにします。コンパイラが行える最適化はほかにも数多くあります。その中には、`apply` メカニズム ([Exercise 5.38](#) 参照) を使わずには、基本演算を“インライン”でコードするといったものがあります。しかし、ここでは深入りしないことにしましょう。この章の主な目的は、単純な (それでいて面白さを失わない) 文脈でコンパイル処理を説明するというところにあります。

5.5.1 コンパイラの構造

Section 4.1.7では、元のメタ循環インタプリタに修正を加えて、分析と実行を分離しました。それぞれの式を分析し、環境を引数として必要な計算を行う実行手続きを生成するようにしました。ここでのコンパイラも、本質的に同じ解析を行います。しかし、実行手続きを生成するのではなく、レジスタマシンで動かす命令列を生成することになります。

コンパイラのトップレベルのディスパッチは、`compile`という手続きです。これは、Section 4.1.1の`eval`手続きや、Section 4.1.7の`analyze`手続き、Section 5.4.1の明示制御評価器の`eval-dispatch`エントリポインツに対応するものです。このコンパイラは、インタプリタと同じように、Section 4.1.2で定義した式構文手続きを使います。³⁵ `compile`は、コンパイル対象の式の構文型について場合分けを行い、それぞれの式の型を、それらに専門化した`code generator`(コード生成器)にディスパッチします。

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

³⁵ このコンパイラは Scheme プログラムであり、式を操作するのに使う構文手続きはメタ循環評価器を使った実際の Scheme 手続きです。それに対して、明示制御評価器のときには、それらと等価な構文演算がレジスタマシンの演算として利用可能であると仮定していました(もちろん、レジスタマシンを Scheme でシミュレートしたときには、レジスタマシンのシミュレーションに実際の Scheme 手続きをいました)。

```
((application? exp)
  (compile-application exp target linkage))
(else
  (error "Unknown expression type: COMPILE" exp))))
```

ターゲットとリンク

`compile` と、`compile` が呼ぶコード生成器は、コンパイル対象の式に加えて二つの引数を取ります。ひとつは *target*(ターゲット) で、コンパイル済みコードが式の値を返すレジスタを指定します。もうひとつは *linkage descriptor*(リンク記述子) で、式をコンパイルした結果のコードが実行を終えたときにどのような動作をするか指定します。リンク記述子は、以下の三つのうちのひとつの動作をするようコードに要求します。

- 列の次の命令から実行を続ける(これはリンク記述子 `next` によって規定します)
- コンパイルしている手続きから戻る(これはリンク記述子 `return` によって規定します)
- 名前つきエントリポイントに飛ぶ(これは指定したラベルをリンク記述子として使うことによって規定します)

例えば、ターゲットをレジスタ `val`、リンク記述子を `next` として、(自己評価の) 式 `5` をコンパイルすると、以下の命令が生成されます。

```
(assign val (const 5))
```

同じ式を、リンク記述子を `return` としてコンパイルすると、以下の命令が生成されます。

```
(assign val (const 5))
(goto (reg continue))
```

一つ目の場合は、実行は列の中の次の命令から続行されます。二つ目の場合は、手続き呼び出しから戻ることになります。いずれの場合も、式の値はターゲットレジスタ `val` に入れられます。

命令列とスタックの使用

それぞれのコード生成器は、式に対して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って達成されます。これは、複合式の評価が部分式を評価することによって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` という手続きです。これは順に実行する任意の数の命令列を引数として取り、それらを連結し、組み合わせた列を返します。つまり、 $\langle seq_1 \rangle$ と $\langle seq_2 \rangle$ が命令列であるとすると、以下を評価すると、

```
(append-instruction-sequences <seq1> <seq2>)
```

次の列が生成されることになります。

```
<seq1>  
<seq2>
```

レジスタを保存する必要があるかもしれないときは、コンパイラのコード生成器は `preserving` を使います。これは、命令列の組み立てを行うためのより込み入った手法です。`preserving` は三つの引数を取ります。ひとつのレジスタの集合と、二つの命令列です。この手続きは、もし二つ目の命令列の実行に必要であれば、ひとつ目の命令列を実行する際に集合内のレジスタの中身が保存されるようなやり方で、列を連結します。これはつまり、もし一つ目の命令列がレジスタを書き換え、二つ目の命令列が実際にそのレジスタの元の中身を必要とするのであれば、`preserving` は列を連結する前に最初の列をそのレジスタの `save` と `restore` によって囲うということです。そうでなければ、`preserving` は単純に連結した命令列を返します。よって、例えば `(preserving (list <reg1> <reg2>) <seq1> <seq2>)` は、 $\langle seq_1 \rangle$ と $\langle seq_2 \rangle$ が $\langle reg_1 \rangle$ と $\langle reg_2 \rangle$ をどのように使うかによって、以下の四つの命令列のうちのひとつを生成します。

$\langle seq_1 \rangle$	$\langle save \langle reg_1 \rangle \rangle$	$\langle save \langle reg_2 \rangle \rangle$	$\langle save \langle reg_2 \rangle \rangle$
$\langle seq_2 \rangle$	$\langle seq_1 \rangle$	$\langle seq_1 \rangle$	$\langle save \langle reg_1 \rangle \rangle$
	$\langle restore \langle reg_1 \rangle \rangle$	$\langle restore \langle reg_2 \rangle \rangle$	$\langle seq_1 \rangle$
	$\langle seq_2 \rangle$	$\langle seq_2 \rangle$	$\langle restore \langle reg_1 \rangle \rangle$

`preserving` を使って命令列を組み合わせることによって、コンパイラは不要なスタック命令を避けることができます。また、こうすることによって `save` と `restore` という命令を生成するかどうかという実装上の詳細を `preserving` 手続きの中に分離し、これらの問題を個別のコード生成器を書くときに起こる考慮点から切り離すことができます。実際に、`save` と `restore` という命令をコード生成器が明示的に生成することはできません。

原則として、命令列は単純な命令のリストとして表現できるはずです。そうすると、`append-instruction-sequences` は通常のリストの `append` を実行することによって命令列の組み立てができるところです。しかし、その場合、`preserving` は複雑な演算になってしまいます。それぞれの命令列を解析して、命令列がどのようにレジスタを使うのか調べなければならないからです。`preserving` は、複雑なうえに非効率なものになります。というのは、引数の命令列はそれ自身 `preserving` の呼び出しによって構築されたものかもしれませんし、その場合はそれらの部品はすでに解析されているはずなのですが、それをまた解析しなければならないからです。そのような繰り返しの多い解析を避けるために、それぞれの命令列に、レジスタ使用情報を関連づけることにします。基本的な命令列を構築する際にこの情報を明示的に提供するようにして、命令列を組み合わせる手続きは、部品となる命令列に関連づけられた情報から組み合せた命令列のレジスタ使用情報を導き出します。

命令列は三つの情報を持つことになります。

- 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が *needed*（必要とする）と言います）
- 列内の命令によって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）とも言います）

命令列は、三つの部品のリストとして表現します。よって、命令列のコンストラクタは以下のようになります。

```
(define (make-instruction-sequence
           needs modifies statements)
  (list needs modifies statements))
```

例えば、現在の環境内で変数 `x` の値を検索し、その結果を `val` に割り当て、それから戻るという二命令の列は、レジスタ `env` と `continue` が初期化されている必要があり、またレジスタ `val` を変更します。したがって、この列は以下のように構築されることになります。

```
(make-instruction-sequence
  '(env continue)
  '(val)
  '((assign val
             (op lookup-variable-value) (const x) (reg env))
    (goto (reg continue))))
```

場合によっては、命令文を持たない命令列を構築することが必要になることもあります。

```
(define (empty-instruction-sequence)
  (make-instruction-sequence '() '() '()))
```

命令列を組み立てる手続きはSection 5.5.4で示します。

Exercise 5.31: 手続き適用を評価する際、明示制御評価器は常に演算子の評価の前後で `env` レジスタの保存と復元を行い、(最後の 1 つを除いて) 各オペランドの評価の前後で `env` の保存と復元を行い、各オペランドの評価の前後で `arg1` の保存と復元を行い、オペランド列の評価の前後で `proc` の保存と復元を行う。以下のそれぞれの組み合わせに対して、これらの `save` と `restore` 命令のうち、余分であり、そのためコンパイラの `preserving` メカニズムによって取り除くことができるものを答えよ。

```
(f 'x 'y)
((f) 'x 'y)
(f (g 'x) y)
(f (g 'x) 'y)
```

Exercise 5.32: `preserving` メカニズムを使うと、組み合わせの演算子が記号である場合、コンパイラは演算子の評価の前後で `env` の保存と復元を回避することになる。このような最適化は、評価器にも組み込むことができるものだ。実際に、Section 5.4 の明示制御評価器は、オペランドのない組み合わせを特別扱いすることによって、すでに似たような最適化を行っている。

- a 明示制御評価器を拡張し、演算子が記号である組み合わせを独立したクラスの式として認識するようにし、そのような式を評価する際にその事実を利用するようにせよ。

- b Alyssa P. Hacker は、評価器が認識する特別ケースをどんどん増やしていくべき、コンパイラの最適化をすべて組み込むことができ、そうするとコンパイルをすることの利点は完全になくなるのではないかという考えを述べた。あなたはこの考え方についてどう思うか。

5.5.2 式のコンパイル

この節と次の節では、`compile` 手続きのディスパッチ先となるコード生成器を実装していきます。

リンクコードのコンパイル

一般的に、それぞれのコード生成器の出力は、要求されたリンクを実装する命令で終わります。これは手続き `compile-linkage` によって生成されます。リンクが `return` であれば、命令 (`goto (reg continue)`) を生成しなければなりません。これは `continue` レジスタを必要とし、どのレジスタも変更しません。リンク記述子が `next` であれば、命令を追加する必要はありません。これら以外の場合、リンク記述子はラベル、そのラベルへの `goto` を生成します。この命令はどのレジスタも必要とせず、変更もしません。³⁶

³⁶ この手続きは、*backquote(バッククオート)*(または*quasiquote(擬似クオート)*) というリスト構築に便利な Lisp の機能を使ってています。リストの前にバッククオート記号を置くというのはクオートにとても似ていますが、リストの中でカンマによって印をつけられたものは評価されるというところが違います。

例えば、`linkage` の値が記号 `branch25` である場合、以下の式を評価すると

```
`((goto (label ,linkage)))
```

次のリストになります。

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

同じように、`x` の値がリスト `(a b c)` である場合、以下の式を評価すると

```
`(1 2 ,(car x))
```

次のリストになります。

```
(1 2 a)
```

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

リンクのコードは、`preserving` によって `continue` レジスタを保存したうえで追加されます。これは、`return` リンクが `continue` レジスタを必要とするからです。与えられた命令列が `continue` を変更するものであり、またリンクのコードが `continue` を必要とする場合、`continue` は保存と復元が行われます。

```
(define (end-with-linkage linkage instruction-sequence)
  (preserving '(continue)
    instruction-sequence
    (compile-linkage linkage)))
```

単純な式のコンパイル

自己評価式、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レジスタに割り当て、リンク記述子によって規定された通りに進む命令列を構築します。

```
(define (compile-self-evaluating exp target linkage)
  (end-with-linkage linkage
    (make-instruction-sequence '() (list target)
      `((assign ,target (const ,exp)))))

(define (compile-quoted exp target linkage)
  (end-with-linkage linkage
    (make-instruction-sequence '() (list target)
      `((assign ,target (const ,(text-of-quotation exp)))))))

(define (compile-variable exp target linkage)
  (end-with-linkage linkage
    (make-instruction-sequence '(env) (list target)
      `((assign ,target
```

```
(op lookup-variable-value)
  (const ,exp)
  (reg env))))))

```

これらの代入命令はすべてターゲットレジスタを変更します。また、変数の検索を行う命令は `env` レジスタを必要とします。

代入と定義は、インタプリタとだいたい同じように扱われます。変数に割り当てる値を求めるコードを再帰的に生成し、それに二命令からなる列を追加します。この命令列は、実際に変数の設定や定義を行い、式全体の値（記号 `ok`）をターゲットレジスタに代入します。再帰的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、コードは結果を `val` に入れ、その後に追加されるコードから続行します。追加は `env` を保存して行われます。これは、変数の設定や定義には環境が必要であり、また変数の値のコードはレジスタを任意に変更するような複雑な式のコンパイルとなるかもしれませんからです。

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

```

追加される二命令の列は、`env` と `val` を必要とし、ターゲットを変更します。この命令列のために `env` は保存するのに、`val` は保存しないということに気をつけてください。これは、この命令列が使うために、`get-value-code` が結果を明示的に `val` に入れるよう設計されていることによります。(実際、もし `val` を保存するとバグになります。こうしてしまうと、`get-value-code` の実行直後に `val` の以前の中身が復元されるようになってしまうからです。)

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でコンパイルされた `if` 式のコードは以下の形式になります。

```

<ターゲット val, リンク記述子 next での述語のコンパイル>
(test (op false?) (reg val))
(branch (label false-branch))
true-branch
<与えられたターゲットと、与えられたリンク記述子または after-if での結果部のコンパイル>
false-branch
<与えられたターゲットとリンク記述子での代替部のコンパイル>
after-if

```

このコードを生成するには、述語、結果部、代替部をコンパイルし、その結果できるコードを、述語の結果をテストする命令、真と偽の分岐と条件文の終わりの印となる新しく生成されたラベルと組み合わせます。³⁷ このコードの配置

³⁷ ここで、`true-branch`, `false-branch`, `after-if` というラベルを上で示したようにそのまま使うことはできません。プログラム中に `if` 文は複数あるかもしれませんからです。`make-label` は、記号を引数に取り、与えられた記号で始まる新しい記号を返します。例えば、`(make-label 'a)` を連続して呼び出すと、`a1, a2, ...` を返します。`make-label` は、クエリ言語で一意の変数名を生成したときと同じように、以下のように実装できます。

```

(define label-counter 0)
(define (new-label-number)

```

では、テストが偽であれば、真の分岐を飛び越す必要があります。ここでひとつだけ注意するべきところは、真の分岐のリンクをどう扱うかということです。条件文のリンクが `return` またはラベルの場合、真と偽の分岐は両方ともこの同じリンクを使います。しかし、リンクが `next` の場合、真の分岐は、偽の分岐を飛び越し条件文の最後に飛ぶコードで終わることになります。

```
(define (compile-if exp target linkage)
  (let ((t-branch (make-label 'true-branch))
        (f-branch (make-label 'false-branch))
        (after-if (make-label 'after-if)))
    (let ((consequent-linkage
           (if (eq? linkage 'next) after-if linkage)))
      (let ((p-code (compile (if-predicate exp) 'val 'next))
            (c-code
              (compile
                (if-consequent exp) target
                consequent-linkage)))
        (a-code
          (compile (if-alternative exp) target linkage)))
      (preserving '(env continue)
        p-code
        (append-instruction-sequences
          (make-instruction-sequence '(val) '()
            `((test (op false?) (reg val))
               (branch (label ,f-branch))))))
        (parallel-instruction-sequences
          (append-instruction-sequences t-branch c-code)
          (append-instruction-sequences f-branch a-code)))
        after-if)))))



---


(set! label-counter (+ 1 label-counter))
label-counter)

(define (make-label name)
  (string->symbol
    (string-append (symbol->string name)
                  (number->string (new-label-number)))))
```

`env` は、真と偽の分岐で必要とされるかもしれない、述語コードの前後で保存されます。`continue` は、真と偽の分岐に含まれるリンクコードで必要とされるかもしれないで保存されます。(順に実行されるわけではない) 真と偽の分岐のコードは、Section 5.5.4で記述する `parallel-instruction-sequences` という専用の結合器を使って連結されます。

`cond` は派生式なので、その処理のためにコンパイラがするのは、(Section 4.1.2の) `cond->if` 変換器を適用し、結果の `if` 式をコンパイルすることだけです。

列のコンパイル

列(手続きの本体または明示的な `begin` 式)のコンパイルは、列の評価と同じような構造です。列のそれぞれの式がコンパイルされるとき、最後の式は列全体に対して指定されたリンクでコンパイルされ、ほかの式は(列の残りを実行するために)`next` リンクでコンパイルされます。個別の式の命令列は、(列の残りのために必要な)`env` と(列末尾のリンクコードで必要となるかもしれない)`continue` を保存しながら、ひとつの命令列に連結されます。

```
(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` 式は手続きを構築します。`lambda` 式のオブジェクトコードは以下の形式に従う必要があります。

〈手続きオブジェクトを構築し
それをターゲットレジスタに割り当てる〉
〈リンク〉

`lambda` 式をコンパイルするときには、手続き本体のコードも生成します。本体は手続き構築時に実行されるわけではないですが、これをオブジェクトコードの `lambda` 式のコードの直後に挿入しておくと便利です。`lambda` 式のリンク

がラベルまたは `return` の場合、これは問題になりません。しかし、リンクが `next` の場合、手続き本体の後ろに挿入されたラベルまでジャンプするリンクを使って、手続き本体のコードを飛び越す必要があります。よって、オブジェクトコードは以下の形式になります。

```
<手続きオブジェクトを構築し  
それをターゲットレジスタに割り当てる>  
<与えられたリンクのコード> または (goto (label after-lambda))  
<手続き本体をコンパイルしたもの>
```

`after-lambda`

`compile-lambda` は、手続きオブジェクトに手続き本体のコードが続くものを構築するコードを生成します。手続きオブジェクトは、現在の環境 (`lambda` 式定義時の環境) とコンパイル済みの手続き本体を指すエントリポイント (新しく生成されたラベル) を引数として、実行時に構築されることになります。³⁸

```
(define (compile-lambda exp target linkage)
  (let ((proc-entry (make-label 'entry))
        (after-lambda (make-label 'after-lambda)))
    (let ((lambda-linkage
           (if (eq? linkage 'next) after-lambda linkage)))
      (append-instruction-sequences
       (tack-on-instruction-sequence
        (end-with-linkage lambda-linkage
                           (make-instruction-sequence '(env) (list target)
                                         `((assign ,target
                                                   (op make-compiled-procedure)
                                         (label ,proc-entry)
```

³⁸ ここで、コンパイル済み手続きを表現するデータ構造を実装する機械語演算がいくつか必要になります。データ構造は、Section 4.1.3で記述した複合手続きの構造と同じようなものになります。

```
(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))
```

```

        (reg env))))))
  (compile-lambda-body exp proc-entry))
  after-lambda))))
```

`compile-lambda` が手続きの本体を `lambda` 式のコードに連結する際には、`append-instruction-sequences`(Section 5.5.4) ではなく、特別な結合器 `tack-on-instruction-sequence` を使います。手続き本体は、組み立てた列に入ったときに実行される命令列の一部ではないからです。手続き本体が列の中にあるのは、単純に置く場所として便利だからというだけです。

`compile-lambda-body` は手続き本体のコードを構築します。このコードは、エントリーポイントのラベルから始まります。その次に、手続き本体を評価する正しい環境、つまり手続きの定義環境に仮引数と手続き呼び出しの引数との束縛を加えて拡張した環境に実行時の評価環境を切り替えさせる命令が続きます。その後には、手続き本体となる式の列のコードが続きます。列は、手続きの結果が `val` に入った状態で最後に手続きからのリターンで終わるように、リンク `return` とターゲット `val` でコンパイルされます。

```

(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

コンパイル処理の核心部分は手続き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で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。

<ターゲット *proc*, リンク *next* で演算子をコンパイル>
 <オペランドを評価し、*argl* に引数リストを構築>
 <与えられたターゲットとリンクで
 手続き呼び出しをコンパイル>

レジスタ *env*, *proc*, *argl* は、演算子とオペランドを評価する間に保存と復元を行なう必要があるかもしれません。このコンパイラで、*val* 以外のターゲットが指定される場所はここだけです。

必要なコードは *compile-application* によって生成されます。これは、演算子をコンパイルして適用すべき手続きを *proc* に入れるコードを生成し、オペランドをコンパイルして適用の個々のオペランドを評価するコードを生成するということを行なっています。オペランドの命令列は、(*construct-arglist* によって)*argl* に引数リストを構築するコードと組み合わされ、結果となる引数リストのコードは手続きコードと (*compile-procedure-call* によって生成される) 手続き呼び出しを実行するコードと組み合わされます。コード列を連結するにあたって、*env* レジスタは演算子の評価の前後で保存する必要があります（演算子の評価が *env* を変更するかもしれません、また *env* はオペランドの評価で必要となるため）。また、*proc* レジスタは引数リスト構築の前後で保存する必要があります（オペランドを評価する際に、実際の手続き適用に必要な *proc* レジスタを変更するかもしれません）。*continue* も、手続き呼び出しのリンクに必要となるため、全体を通して保存しなければなりません。

```

(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コードは、それぞれのオペランドを評価して *val* に入れ、それからその値を *argl* に集積中の引数リストに *cons* します。*argl* に順に引数を *cons* していくので、結果となるリストの中で引数が最初から最後という順番になるように、最後の引数から始めて最初のもので終わるようにする必要

があります。この一連の評価のために `argl` を空リストに初期化して一命令を無駄にする代わりに、最初のコード列に `argl` の初期値を構築せらるるようにします。そのため、引数リスト構築の一般的な形式は以下のようになります。

```
(val をターゲットとして最後のオペランドをコンパイル)
(assign argl (op list) (reg val))
(val をターゲットとして次のオペランドをコンパイル)
(assign argl (op cons) (reg val) (reg argl))
...
(val をターゲットとして最初のオペランドをコンパイル)
(assign argl (op cons) (reg val) (reg argl))
```

`argl` は、(ここまで集積した引数が失われないように) 最初ひとつ以外の各オペランドの評価の前後で保存する必要があります。また、`env` は(残りのオペランド評価で使うために) 最後のひとつ以外の各オペランドの評価の前後で保存する必要があります。

この引数コードのコンパイルには少し注意が必要です。評価対象の最初のオペランドを特別扱いすることになり、また `argl` と `env` を別々の位置で保存する必要があるからです。`construct-arglist` 手続きは個々のオペランドを評価するコードを引数として取ります。オペランドがなければ、単純に以下の命令を作ります。

```
(assign argl (const ()))
```

オペランドがある場合は、`construct-arglist` は最後の引数で `argl` を初期化するコードを作り、引数の残りを評価してそれらを順次 `argl` につなげるコードを連結していきます。引数を最後から最初という順番で処理するために、オペランドのコード列のリストを `compile-application` によって渡された順番の逆順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)))))))

```

手続きの適用

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` 内の手続きを `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](#) のメタ循環評価器の `apply` 手続きや、[Section 5.4.1](#) の明示制御評価器の `apply-dispatch` エントリポイントと同じディスパッチを実行します。まず、適用する手続きが基本手続きか複合手続きか確認します。基本手続きには `apply-primitive-procedure` を使います。これがどのようにコンパイルされた手続きを扱うかについては、すぐ後で見ていくことにします。手続き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でコンパイル済み手続きを適用するコード>

```

```

primitive-branch
  (assign <target>
    (op apply-primitive-procedure)
    (reg proc)
    (reg argl))
  ⟨リンク⟩
after-call

```

コンパイル済み手続きの分岐 (compiled-branch) は基本手続きの分岐 (primitive-branch) をスキップする必要があります。そのため、元の手続き呼出のリンクが next であった場合、複合手続きの分岐は基本手続きの分岐の後に挿入されたラベルにジャンプするリンクを使う必要があります (これは、compile-if の中で真の分岐に使われるリンクと似ています。)

```

(define (compile-procedure-call target linkage)
  (let ((primitive-branch (make-label 'primitive-branch))
        (compiled-branch (make-label 'compiled-branch))
        (after-call (make-label 'after-call)))

  (let ((compiled-linkage
         (if (eq? linkage 'next) after-call linkage)))
    (append-instruction-sequences
      (make-instruction-sequence '(proc) '()
        `((test (op primitive-procedure?) (reg proc)
                  (branch (label ,primitive-branch))))
      (parallel-instruction-sequences
        (append-instruction-sequences
          compiled-branch
          (compile-proc-appl target compiled-linkage))
        (append-instruction-sequences
          primitive-branch
          (end-with-linkage linkage
            (make-instruction-sequence '(proc argl)
              (list target)
              `((assign ,target
                        (op apply-primitive-procedure)
                        (reg proc)

```

```
(reg arg1))))))  
after-call))))
```

基本手続きと複合手続きの分岐は、compile-if での真と偽の分岐と同じように、通常の append-instruction-sequences ではなく parallel-instruction-sequences を使って追加されます。これらは順番に実行されるわけではないからです。

コンパイル済み手続きの適用

手続きの適用を扱うコードは、生成する命令列はとても短いのですが、コンパイラの中で最も難しい部分です。(compile-lambda に構築された) コンパイル済み手続きは、手続きのコードが始まる場所を指定するラベルであるエントリポイントを持っています。このエントリポイントのコードは結果を計算して val に入れ、(goto (reg continue)) という命令を実行して戻ります。そのため、与えられたターゲットとリンクによるコンパイル済み手続きの適用コード(これは compile-proc-appl によって生成されます)は、リンクがラベルであれば以下のようになり、

```
(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 でない場合に挿入  
される  
(goto (label <リンク>)) ; リンクコード
```

リンクが return であれば以下になると期待できます。

```
(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 でない場合に挿入  
される  
(restore continue)  
(goto (reg continue)) ; リンクコード
```

このコードは、手続きが `proc-return` に戻るよう `continue` を設定し、手続きのエントリポイントにジャンプします。`proc-return` のコードは、(必要であれば) 手続きの結果を `val` からターゲットレジスタに転送し、それからリンクによって指定された位置にジャンプします(リンクは常に `return` かラベルです。`compile-procedure-call` は、複合手続きの分岐の `next` リンクを `after-call` ラベルに置き換えるからです)。

実際、ターゲットが `val` でない場合、これがまさにコンパイラが生成するコードとなります。³⁹ しかし、通常はターゲットは `val` であるため(コンパイラが別のレジスタを指定するのは演算子の評価のターゲットを `proc` にすることだけです)、手続きの結果は直接ターゲットレジスタに入れられることになるので、それをコピーする特別な場所に戻る必要はありません。そのため、その場所に戻る代わりに、呼び出し元のリンクで指定された場所に手続きが直接“戻る”ように `continue` を設定して、コードを単純化します。

(`continue` にリンクを設定)

```
(assign val (op compiled-procedure-entry) (reg proc))  
(goto (reg val))
```

リンクがラベルであれば、手続きがそのラベルに戻るように `continue` を設定します(つまり、手続きの終端の `(goto (reg continue))` が、上記の `proc-return` の `(goto (label <linkage>))` と等価になります)。

```
(assign continue (label <リンク>))
```

```
(assign val (op compiled-procedure-entry) (reg proc))  
(goto (reg val))
```

もしリンクが `return` なら、`continue` を設定する必要はありません。`continue` にはすでに望ましい位置が入っています(これは、手続きの終端の `(goto (reg continue))` が、`proc-return` の `(goto (reg continue))` が行くはずの場所に直接行くということです)。

```
(assign val (op compiled-procedure-entry) (reg proc))  
(goto (reg val))
```

この `return` リンクの実装で、コンパイラは末尾再帰のコードを生成します。手続きの本体の最後のステップとして手続き呼び出しを行うと、スタックに何の情報も保存せず、直接移動を行うことになります。

³⁹ 実際のところ、ターゲットが `val` でなく、リンクが `return` である場合にはエラーを発生させます。`return` リンクを要求する場所は手続きのコンパイルの中だけで、手続きは値を `val` に入れる約束だからです。

このようにする代わりに、リンクが `return` でターゲットが `val` である場合を上に示した `val` 以外のターゲットと同じように扱うと仮定してみましょう。こうすると、末尾再帰を壊してしまいます。このシステムは、どんな式に対しても相変わらず同じ値を返すでしょう。しかし、手続きを呼ぶたびに `continue` を保存することになり、呼び出しから帰ってきて(必要のない)保存の取り消しをすることになります。この余分な保存は、入れ子の手続き呼び出しの間に蓄積されていきます。⁴⁰

`compile-proc-appl` は、呼出のターゲットが `val` かどうか、リンクが `return` かどうかによって、四つの場合を考慮して上記の手続き適用コードを生成します。命令列は、すべてのレジスタを変更するように宣言されています。これは、手続きの本体を実行すると、どのレジスタがどう変更されるかわからないからです。⁴¹ また、ターゲットが `val` で、リンクが `return` の場合のコード列は `continue` を必要とすると宣言されていることに気をつけてください。`continue` は二命令の列では明示的に使われていませんが、コンパイル済み手続きに入るときには `continue` が正しい値を持っているということを保証する必要があります。

```
(define (compile-proc-appl target linkage)
  (cond ((and (eq? target 'val) (not (eq? linkage 'return)))
         (make-instruction-sequence '(proc) all-reg
```

⁴⁰ コンパイラに末尾再帰のコードを生成させるというのは素直な考え方のように思えるかもしれません。しかし、C 言語や Pascal を含め、一般的な言語ではこれを行わず、そのためこれらの言語では反復プロセスを手続き呼び出しだけで表現することはできません。これらの言語で末尾再帰が難しいのは、それらの実装ではスタッќをリターンアドレスを格納するのに使うだけでなく、手続きの引数や局所変数を格納するためにも使っているからです。この本で記述されている Scheme の実装は、引数と変数をメモリに入れ、ガベージコレクションの対象にしています。変数と引数にスタッќを使う理由は、ほかのところでガベージコレクションを使わない言語では、そうすることによってガベージコレクションが必要なくなり、またそれがより効率的だと一般に信じられているということです。実際のところ、高機能な Lisp コンパイラは、末尾再帰を壊さずに引数のためにスタッќを使うことができます(この記述については Hanson 1990 を参照してください)。また、そもそもスタッќ割り当てはガベージコレクションより効率的なのかなというところにも議論がありますが、この問題の詳細はコンピューターアーキテクチャの細部によるようです(この問題に反対の立場からの意見については、Appel 1987 と Miller and Rozas 1994 を参照してください。)

⁴¹ 変数 `all-reg` は、全レジスタの名前のリストに束縛されています。

```
(define all-reg ' (env proc val argl continue))
```

```

`((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、命令列の表現方法と組み合わせ方の詳細を述べます。命令列は必要とされるレジスタ、変更されるレジスタ、実際の命令というリストとして表現されるというSection 5.5.1の記述を思い出してください。このほかに、ラベル(記号)についても、レジスタを必要としたり変更したりしない単純化した命令列を考えることにします。よって、命令列によって必要とされたり修正されたりするレジスタを決定するには、以下のセレクタを使います。

```
(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したり変更したりするかを判断するには、以下の述語を使います。

```
(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使って、コンパイラを全体を通して使用される、命令列に対するさまざまな結合器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、順に実行される任意の数の命令列を引数として取り、すべての命令列の命令文を連結した命令文を持つ命令列を返します。難しいところは、結果となる命令列が必要とするレジスタと変更するレジスタを決定するというところです。命令列のいずれかが変更するレジスタは、変更するレジスタとなります。必要とするレジスタは、最初の命令列が実行される前に初期化されなければならないレジスタ（最初の命令列に必要とされるレジスタ）と、それに加えて、そのほかの命令列に必要とされるレジスタのうちそれ以前の命令列によって初期化（変更）されていないものです。

命令列は `append-2-sequences` によって一度に二つが接続されます。これは二つの命令列 `seq1` と `seq2` を取り、`seq1` の命令文の後に `seq2` の命令文が置かれた命令文を持つ命令列を返します。この命令列の変更するレジスタは、`seq1` か `seq2` のいずれかが変更するレジスタになります。また、この命令列の必要とするレジスタは、`seq1` によって必要とされるレジスタに、`seq2` で必要とされ、かつ `seq1` で変更されないレジスタを加えたものです。（集合演算で考えると、必要なレジスタの新しい集合は、`seq1` に必要とされるレジスタの集合と、`seq2` に必要とされるレジスタと `seq1` に変更されるレジスタの差集合との、和集合ということになります。）

```
(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))

```

この手続きは、リストとして表現された集合を操作する簡単な演算をいくつか使います。この表現は、Section 2.3.3で述べた（順序なし）集合表現に似ています。

```

(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)))))


```

二番目に重要な命令列結合器である `preserving` は、レジスタリスト `regs` と、順に実行する二つの命令列 `seq1` と `seq2` を取ります。この結合器の返す命令列の持つ命令文は、`seq1` の命令文に `seq2` の命令文が続くものですが、`regs` のうち `seq1` に変更されるが `seq2` に必要とされるレジスタを守るために、`seq1` の前後には適切な `save` と `restore` が追加されます。これを達成するために、`preserving` はまず、必要な `save`、`seq1` の命令文、必要な `restore` を持つ命令列を作ります。この命令列は、`seq1` によって必要とされるレジスタに加えて、保存と復元が行われるレジスタを必要とし、`seq1` で変更されるレジスタのう

ち、保存と復元が行われるレジスタを除いたものを変更します。次に、この拡張した命令列と seq2 が通常の方法で連結されます。次の手続きはこの戦略を再帰的に実装したもので、保存するべきレジスタのリストを下っていきます。

⁴²

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

命令列結合器 tack-on-instruction-sequence は、compile-lambda が手続きの本体をほかの命令列に連結するために使うものです。手続きの本体が“埋め込まれている”のは、組み合わせた命令列の一部として実行するためではないので、そのレジスタ使用は埋め込む命令列のレジスタ使用に影響しません。そのため、手続きの本体をほかの命令列にくっつけるときには、それが必要とするレジスタと変更するレジスタの集合を無視します。

```
(define (tack-on-instruction-sequence seq body-seq)
  (make-instruction-sequence
    (registers-needed seq)
    (registers-modified seq)
    (append (statements seq)
```

⁴²preserving が append を三つの引数で呼び出していることに注意してください。この本で示した append の定義は二つの引数しか受け入れませんが、Scheme は任意の数の引数を取る append 手続きを標準的に提供しています。

```
(statements body-seq))))
```

`compile-if` と `compile-procedure-call` は、テストに続く二つの選択肢の分岐を追加するために、`parallel-instruction-sequences` という特別な結合器を使います。二つの分岐が順に実行されることは決してありません。テストの評価がどのようなものであっても、どちらかひとつの分岐に入ることになります。このため、二つ目の分岐に必要とされるレジスタは、それが一つ目の分岐に修正されるものであっても、組み合わせ後の命令列で必要とされることになります。

```
(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 コンパイル済みコードの例

ここまでで、コンパイラのすべての要素について見てきたことになります。ここでは、コンパイル済みコードの例を詳しく調べて、要素がどのように組み合わされているのか見てみましょう。ここでは、`compile` を呼ぶことによって、再帰 `factorial` 手続きの定義をコンパイルすることにします。

```
(compile
  '(define (factorial n)
     (if (= n 1)
         1
         (* (factorial (- n 1)) n)))
  'val
  'next)
```

`define` 式の値はレジスタ `val` に入れるよう規定しています。`define` を実行した後にコンパイル済みコードが何をするかには興味がないので、リンクとして `next` を選んでいることに特に意味はありません。

`compile` は、式が定義であると判断し、`compile-definition` を呼び出して、前から順に (ターゲット `val` に) 割り当てる値を求めるコード、定義を組み込むコード、`define` の値 (記号 `ok`) をターゲットレジスタに入れるコード、リンクコードという並びをコンパイルします。`env` は定義を組み込むのに必要なので、値の計算の前後で保存されます。リンクは `next` なので、この場合リンクコードはありません。よって、コンパイル済みコードの骨組みは次のようにになります。

```
<値を計算するコードで変更されるなら env を保存>
  <ターゲット val、リンク next での定義の値のコンパイル>
  <上で保存していたら env を復元>
  (perform (op define-variable!)
    (const factorial)
    (reg val)
    (reg env))
  (assign val (const ok))
```

変数 `factorial` の値を作るためにコンパイルする式は `lambda` 式で、その値は階乗を計算する手続きです。`compile` は `compile-lambda` を呼んでこれを処理します。`compile-lambda` は手続きの本体をコンパイルし、それに新しいエントリポイントとしてラベルをつけ、新しいエントリポイントにある手続き本体を実行時環境と組み合わせて結果を `val` に割り当てる命令を生成します。次に、命令列はこの時点で挿入されるコンパイル済みコードをスキップします。手続きのコード自身は、まず手続きの定義環境を、仮引数 `n` を手続きの引数に束縛するフレームによって拡張します。それから実際の手続き本体が続きます。変数の値のためのこのコードは `env` レジスタを変更ないので、上で示したオプションの `save` と `restore` は生成されません (`entry2` にある手続きのコードはこの時点では実行されません。そのため、それが `env` を使うかどうかは影響しません)。よって、コンパイル済みコードの骨組みは次のようにになります。

```
(assign val
  (op make-compiled-procedure)
  (label entry2)
  (reg env))
(goto (label after-lambda1))
entry2
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env
```

```

(op extend-environment)
  (const (n))
  (reg arg1)
  (reg env))
<手続き本体のコンパイル>
after-lambda1
  (perform (op define-variable!)
    (const factorial)
    (reg val)
    (reg env))
  (assign val (const ok))

```

手続きの本体は、常にターゲット `val` とリンク `return` を持つ命令列として (`compile-lambda-body` によって) コンパイルされます。今回の場合、命令列はひとつの `if` 式からなります。

```

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

```

`compile-if` が生成するコードは、まず最初に述語を (ターゲットを `val` として) 計算し、それからその結果を確認して、述語が偽であれば真の分岐をスキップします。`env` と `continue` は述語コードの前後で保存されます。これらは `if` 式の残りの部分で必要となる可能性があるためです。この `if` 式は手続きの本体を構成する命令列内の最後の (そして唯一の) 式であり、ターゲットは `val` で、リンクは `return` であるため、真と偽の分岐はどちらもターゲット `val` とリンク `return` でコンパイルされます。(つまり、条件文の値が手続きの値となるということです。条件文の値はいずれかの分岐で計算される値です。)

<述語により変更され、分岐により必要とされるのであれば `continue`, `env` を保存する>

```

<ターゲット val, リンク next での述語のコンパイル>
<上で保存していれば continue, env を復元する>
  (test (op false?) (reg val))
  (branch (label false-branch4))

true-branch5
  <ターゲット val, リンク return での真の分岐のコンパイル>
false-branch4

```

〈ターゲット *val*, リンク *return* での偽の分岐のコンパイル〉
after-if3

述語 ($= n 1$) は手続き呼び出します。これは演算子 (記号 $=$) を検索し、その値を *proc* に入れます。次に、引数である *1* と、変数 *n* の値を *arg1* に集めます。それから、*proc* に入っているものが基本手続きか複合手続きかをテストし、それに従って基本手続きの分岐か複合手続きの分岐にディスパッチします。どちらの分岐もラベル *after-call* から再開します。演算子とオペランドの評価の前後ではレジスタを保存する必要がありますが、今回の場合はこれらの評価は問題となるレジスタを変更しないため、結果的にレジスタの保存は行われません。

```
(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 であり、これを (ターゲット *val* とリンク *return* で) コンパイルすると、以下のようになります。

```
(assign val (const 1))
(goto (reg continue))
```

偽の分岐のコードはまた別の手続き呼び出します。手続きは記号 * の値で、引数は n と、また別の手続き呼び出し (`factorial` の呼び出し) の結果です。これらの呼び出しは、それぞれ `proc` と `arg1` と、その手続き自身の基本手続きと複合手続きの分岐を準備します。[Figure 5.17](#)は、手続き `factorial` の定義の完全なコンパイル結果を示したものです。述語の前後で、生成される可能性のある `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ください。これらのレジスタは述語内の手続き呼び出しで変更され、分岐内の手続き呼び出しと `return` のリンクに必要とされるためです。

Exercise 5.33: 次の階乗手続きの定義について考えよ。これは上に挙げたものとわずかに異なっている。

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

この手続きをコンパイルし、その結果のコードを `factorial` に対して生成されるコードと比較せよ。見つけた違いについて説明せよ。二つのプログラムの実行効率に違いはあるだろうか。

Exercise 5.34: 反復階乗手続きをコンパイルせよ。

```
(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つけ、`factorial` の反復版と再帰版のコードで、片方がスタック空間を積み上げていきもう片方が一定のスタック空間で動作するという違いを生む、本質的な違いを示せ。

Figure 5.17: ↓ `factorial` 手続き定義のコンパイル結果

```
; 手続きを構築し、手続き本体のコードをスキップする
(assign val
        (op make-compiled-procedure)
        (label entry2))
```

```

        (reg env))
(goto (label after-lambda1))
entry2      ; factorialへの呼び出しはここから入ることになる
(assign env (op compiled-procedure-env) (reg proc))
(assign env
        (op extend-environment)
        (const (n))
        (reg argl)
        (reg env))
;; 実際の手続き本体開始
(save continue)
(save env)
;; (= n 1) を計算
(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 ; val には (= n 1) の結果が入っている
	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) を計算して返す
(assign proc
        (op lookup-variable-value)
        (const *)
        (reg env))
(save continue)
(save proc) ;* 手続きを保存する
(assign val
        (op lookup-variable-value)
        (const n)
        (reg env))
(assign argl (op list) (reg val))
(save argl) ;* の部分引数リストを保存
;; * のもうひとつの引数である (factorial (- n 1)) を計算
(assign proc
        (op lookup-variable-value)
        (const factorial)
        (reg env))
(save proc) ; factorial 手続きを保存
;; factorial の引数である (- n 1) を計算
(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-call16 ; val には (- n 1) の結果が入っている
(assign argl (op list) (reg val))
	restore proc ; factorial を復元
;; factorial を適用
(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 ; val には (factorial (- n 1)) の結果が入っている
(* restore argl) ; * の部分引数リストを復元
(assign argl (op cons) (reg val) (reg argl))
(* restore proc) ; * を復元
(* restore continue)
;; * を適用し、その値を返す
(test (op primitive-procedure?) (reg proc))
(branch (label primitive-branch14))
compiled-branch13
;; この複合手続き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
;; 手続きを変数 factorial に割り当てる
(perform (op define-variable!)
          (const factorial))

```

```

  (reg val)
  (reg env))
(assign val (const ok))

```

Exercise 5.35: Figure 5.18に示すコードは、どのような式をコンパイルして生成されたものか。

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: 我々のコンパイラは、組み合わせのオペランドに対してどのような評価順を生成するだろうか。左から右だろうか、右から左だろうか、またはほかの順序だろうか。この順序はコンパイラのどこで決定されるだろうか。コンパイラを修正し、別の評価順を生成するようにせよ (Section 5.4.1での明示制御評価器の評価順の議論参照)。オペランドの評価順を変えると、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だろうか。

Exercise 5.37: スタック使用を最適化するためのコンパイラの *preserving* のメカニズムを理解する方法のひとつとして、この考えを使わなかったとしたどのような演算が余分に生成されていったかを見るというものがある。*preserving* を修正し、*save* と *restore* 演算を常に生成するようにせよ。簡単な式をいくつかコンパイルし、生成される不必要なスタック演算を特定せよ。生成されたコードと、*preserving* メカニズムをそのままにした状態で生成されるコードとを比較せよ。

Exercise 5.38: 我々のコンパイラは、不必要的スタック演算を避けることに関してはうまくやっているが、言語の基本手続きの呼び出しをマシンによって提供される基本演算を使ってコンパイルすることに関しては、まったくうまくやっていない。例えば、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よう。このコードは、引数リストを *argl* に準備

し、(環境内で記号`+`を探すことによって見つける) 基本加算手続きを`proc`に入れ、この手続きが基本手続きか複合手続きかテストする。コンパイラは、このテストを行うコードに加えて、基本手続きと複合手続きの分岐(そのうちひとつしか実行されない)も常に生成する。コントローラの中で基本手続きを実装する部分は示さなかったが、それらの命令はマシンのデータパスの基本算術演算を使うものと仮定する。ここで、もしコンパイラが基本手続きを`open-code`(オープンコード)できたら—つまり、これらの基本マシン演算を直接使うコードを生成できたら—生成されるコードがどれだけ少なくなるか考えてみよう。式`(+ a 1)`は、以下のような単純なものにコンパイルされることになるかもしれない。⁴³

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

この課題では、我々のコンパイラを拡張し、選ばれた基本手続きのオープンコードをサポートするようにする。それらの基本手続きの呼び出しに対しても、一般的な手続き適用のコードではなく、特別目的のコードが生成される。これをサポートするために、マシンには`arg1`と`arg2`という特別な引数レジスタを追加する。マシンの基本数値演算は、入力を`arg1`と`arg2`から取る。その結果は、`val, arg1, arg2`のいずれかに入れる。

コンパイラは、ソースプログラム内のオープンコードな基本手続きの適用を認識できる必要がある。`compile`手続きにディスパッチを追加し、いま認識できる予約語(特殊形式)に加えてそれらの名前も認識できるようにする。⁴⁴ 特殊形式それぞれに対して、私たちのコンパイラはコード生成器を持っています。この練習問題では、オープンコードの基本手続きに対して、コード生成器を構築する。

⁴³ ここでは、`+`という同じ記号をソース言語の手続きとマシン演算の両方を示すために使っています。一般的には、ソース言語の基本手続きとマシンの基本演算の間に一対一の対応はありません。

⁴⁴ 基本手続きを予約語に入れるというのは一般的にはよくない考えです。そうしてしまうと、ユーザがそれらの名前を別の手続きに束縛し直すことができなくなるからです。そのうえ、いま使われているコンパイラに予約語を追加すると、それらの名前で手続きを定義している既存のプログラムが動作しなくなります。この問題を避けるための考え方についてはExercise 5.44を参照してください。

- a オープンコードの基本手続きはすべて、特殊形式とは違つて、オペランドの評価を必要とする。オープンコードのコード生成器すべてによって使われるコード生成器 `spread-arguments` を書け。`spread-arguments` はオペランドのリストを取り、連続した引数レジスタをターゲットとして与えられたオペランドをコンパイルする。オペランドはオープンコードの基本手続きの呼び出しを含む可能性があるため、オペランド評価の間に引数レジスタを保存しなければならないことに注意せよ。
- b 基本手続き `=`, `*`, `-`, `+` のそれぞれについてコード生成器を書け。コード生成器は、演算子を含む組み合わせ、ターゲット、リンクを引数に取り、引数をレジスタに入れ与えられたターゲットと与えられたリンクで演算を実行するコードを生成する。二つのオペランドを持つ式が扱えればそれでよい。`compile` がこれらのコード生成器にディスパッチを行うようにせよ。
- c 新しいコンパイラを `factorial` の例で試せ。結果のコードをオープンコードな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ドを持つ式を扱えるようにせよ。三つ以上のオペランドを持つ式は、それぞれが二つだけの入力を持つ演算の列にコンパイルする必要があるだろう。

5.5.6 レキシカルアドレッシング

コンパイラによって一般的に行われる最適化のひとつとして、変数検索の最適化があります。ここまで実装してきたコンパイラは、評価器の `lookup-variable-value` 演算を使うコードを生成します。この演算は、実行時環境を外に向かってフレームごとに見ていきながら、現在束縛されているそれぞれの変数と比較することによって変数を検索します。この検索は、フレームの入れ子が深かつたり、変数が多かつたりすると、高コストになります。例えば、以下の式によって返される手続きの適用で、式 `(* x y z)` を評価する途中に `x` の値を検索するという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 組み合わせのシンタクスシュガーに過ぎないため、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 を検索するたびに、記号 x が(一番目のフレームでは) y, z や(二番目のフレームでは) a, b, c, d, e と eq? でないことを判定する必要があります。ここではさしあたり、プログラムは define を使わず、変数は lambda によってのみ束縛されるものと仮定します。私たちの言語はレキシカルスコープを持つ言語なので、式の実行時環境は、その式があるプログラムのレキシカルな構造を反映した構造を持ちます。⁴⁵ そのため、コンパイラが上の式を分析するとき、手続きが適用されるときはいつでも、($* x y z$) の変数 x が現在のフレームから二つ外のフレームで見つかるということ、またそのフレームの最初の変数であるということを知ることができます。

lexical-address-lookup という新しい種類の変数検索演算を作ることで、この事実を活用することができます。この演算は、環境と、二つの数値からなる lexical address (レキシカルアドレス) を引数として取ります。二つの数値は、フレームをいくつパスするかを指定する frame number(フレーム数) と、そのフレーム内で変数をいくつパスするかを指定する displacement number(変位数) です。lexical-address-lookup は、現在のフレームに対してその相対的なレキシカルアドレスの位置に格納されている変数の値を返します。私たちのマシンに lexical-address-lookup 演算を追加すると、lookup-variable-value を使う代わりにこの演算を使って変数を参照するコードをコンパイラに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! 演算を使うことができます。

そのようなコードを生成するためには、変数の参照をコンパイルしようとすると、コンパイラはその変数のレキシカルアドレスを確定できなければな

⁴⁵ 内部定義を許すのであれば、それらを取り除かない限り、これは成り立ちません。Exercise 5.43 参照。

りません。プログラム中の変数のレキシカルアドレスは、それがコードのどこにあるのかに依存します。例えば、以下のプログラムで、式 $\langle e1 \rangle$ の中では、 x アドレスは $(2, 0)$ 、つまり 2 フレーム後ろで、そのフレームの最初の変数ということになります。その位置では、 y はアドレス $(0, 0)$ であり、 c はアドレス $(1, 2)$ です。式 $\langle e2 \rangle$ では、 x は $(1, 0)$ 、 y は $(1, 1)$ 、 c は $(0, 2)$ です。

```
((lambda (x y)
  (lambda (a b c d e)
    ((lambda (y z) ⟨e1⟩)
     ⟨e2⟩
     (+ c d x))))  
3  
4)
```

コンパイラがレキシカルアドレッシングを使うコードを生成するひとつの方
法として、*compile-time environment*(コンパイル時環境) と呼ばれるデータ構
造を管理するというものがあります。このデータ構造は、特定のアクセス演算
が実行されるときにどの変数が実行時環境のどのフレームのどの位置にある
かということを記録するものです。コンパイル時環境はフレームのリストで、
それぞれが変数リストを持っています(もちろん、変数に束縛される値はあり
ません。コンパイル時には値は計算されないからです)。コンパイル時環境は
`compile` の追加の引数になり、それぞれのコード生成器に渡されます。トッ
ップレベルでの `compile` への呼び出しには、空のコンパイル時環境を使います。
`lambda` の本体がコンパイルされるときは、`compile-lambda-body` は手続きの
仮引数を持つフレームによってコンパイル時環境を拡張し、本体を構成する命
令列がその拡張された環境を引数としてコンパイルされるようにします。コン
パイルの各時点で、`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み
込むために、このレキシカルアドレッシング戦略の概要をどのように完成させ
るべきかを述べたものです。 **Exercise 5.44** では、コンパイル時環境の別の使い
方を記述します。

Exercise 5.39: 新しい検索演算を実装する `lexical-address-lookup`
手続きを書け。引数はレキシカルアドレスと実行時環境の二つ
で、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。
`lexical-address-lookup` は、変数の値が記号 `*unassigned*` であ

ればエラーを発生させる。⁴⁶ 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演算を実装する手続き `lexical-address-set!` を書け。

Exercise 5.40: コンパイラを修正し、上で述べたコンパイル時環境を維持するようにせよ。つまり、`compile` とさまざまなコード生成器に `compile-time-environment` という引数を追加し、`compile-lambda-body` の中でそれを拡張するようにせよ。

Exercise 5.41: 引数として変数とコンパイル時環境を取り、その環境に対するその変数のレキシカルアドレスを返す手続き `find-variable` を書け。例えば、上で示されたプログラムの断片では、式`(e1)` 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))` となる。`find-variable` は以下のようない結果を返す。

```
(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` を使って `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(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、以前と同じように、コード生成器が環境命令を使って束縛を検索するようにする。(コンパイル時に見つからない変数が存在しうる位置はグローバル環境のみである。グローバル環境は実行環境の一部だが、コンパイル時環境の一部ではない。⁴⁷ その

⁴⁶ これは、内部定義を取り除くための走査手法を実装するのであれば必要となる、変数検索に対する修正です (Exercise 5.43)。レキシカルアドレッシングを動作させるためには、内部定義を取り除く必要があります。

⁴⁷ レキシカルアドレスはグローバル環境内の変数にアクセスするためには使えません。グローバル環境の名前はいつでも対話的に定義・再定義ができるからです。Exercise 5.43 のように内部定義を取り除くと、コンパイラから見える定義はグローバル環境に影響するトップレベルのものだけになります。定義をコンパイルしても、定義した名前がコンパイル時環境に入ることはできません。

ため、やろうと思えば、評価器の演算に `env` にある実行時環境をすべて検索させるのではなく、直接グローバル環境を見に行くようになることもできる。グローバル環境は演算 (`op get-global-environment`) によって取得できる。) 修正したコンパイラを、この節の最初に出てきた入れ子の `lambda` の組み合わせのような、いくつかの簡単な例でテストせよ。

Exercise 5.43: Section 4.1.6では、ブロック構造の内部定義は“本物の”`define` と考えるべきでなく、手続きの本体は、定義されている内部変数が `set!` を使って正しい値に初期化された通常の `lambda` 変数であるかのように解釈されるべきだと主張した。Section 4.1.6とExercise 4.16では、メタ循環インタプリタを修正して内部定義を走査して削除することによってこれを達成する方法を示した。コンパイラを修正し、手続きの本体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にコンパイル時環境を使うことに焦点を当てた。しかし、コンパイル時環境にはそのほかの使い方もある。例えば、Exercise 5.38では基本手続きをオープンコード化することによってコンパイル済みコードの効率を向上させた。我々の実装では、オープンコードの手続きの名前を予約後として扱った。もしプログラムがそのような名前を再束縛しようとすると、Exercise 5.38で記述したメカニズムは、新しい束縛を無視して、依然としてそれを基本手続きとしてオープンコード化するだろう。例えば、`x` と `y` の線形結合を求める次の手続きについて考える。

```
(lambda (+ * a b x y)
  (+ (* a x) (* b y)))
```

これは、例えば `+matrix`, `*matrix`, 4つの行列といった引数で呼ぶことができる。しかし、オープンコードのコンパイラは、`(+ (* a x) (* b y))` 内の `+` と `*` を依然として基本手続きの `+` と `*` としてオープンコード化してしまうだろう。オープンコードのコンパイラを変更し、基本手続きの名前を含む式を正しいコードにコンパイルするためにコンパイル時環境を参照するようにせよ(このコードは、プログラムがこれらの名前に `define` や `set!` を行わない限り正しく動作する)。

5.5.7 コンパイル済みコードと評価器の接続

ここまで、コンパイル済みコードを評価器にロードしたり、実行したりする方法については触れてきませんでした。ここでは、明示制御評価器がSection 5.4.4の通りに定義されており、Footnote 38で規定した追加命令も含んでいると仮定し、この評価器に `compile-and-go` という手続きを実装します。この手続きは、Scheme 式をコンパイルし、その結果できるオブジェクトコードを評価器マシンにロードし、評価器にそのコードを評価器のグローバル環境の中で実行させ、結果を表示し、評価器のライブループに入るというものです。また、解釈される式が、解釈される手続きだけでなく、コンパイル済み手続きも呼び出せるように評価器を修正します。こうすると、コンパイル済み手続きをマシンに入れて、評価器にそれを呼び出せることができます。

```
(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
```

評価器がコンパイル済み手続きを扱えるようにする(例えば、上記の `factorial` の呼び出しを評価する)ためには、`apply-dispatch`(Section 5.4.1) のコードを変更して、コンパイル済み手続きを(複合手続きや基本手続きと別のものとして)識別し、コンパイル済みコードのエントリポイントに制御を直接移動させる必要があります。⁴⁸

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
```

⁴⁸もちろん、コンパイル済み手続きは解釈される手続きと同じく複合手続き(非基本手続き)です。明示制御評価器で使った用語との一貫性のために、この節では“複合手続き”を(コンパイル済みではなく)解釈される手続きを指すために使います。

```

(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 `continue` を復元していることに注意してください。評価器は、`apply-dispatch` の時点で継続がスタックの一番上になるようになっています。一方、コンパイル済み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、コンパイル済みコードが実行される前に `continue` を復元する必要があります。

評価器の開始時にコンパイル済みコードを実行できるようにするために、評価器の最初に `branch` 命令を追加します。この命令は、`flag` レジスタが設定されていれば、マシンを新しいエントリポインツに行かせるというものです。

⁴⁹

```

(branch (label external-entry)) ; flag が設定されていれば分岐する
read-eval-print-loop
  (perform (op initialize-stack))
  ...

```

`external-entry` は、`val` に結果を入れ、`(goto (reg continue))` で終わる命令列の位置が `val` に入っている状態でマシンが開始されると仮定します。このエントリポインツから開始すると、`val` に指定された位置にジャンプしますが、実行が `print-result` に戻るよう最初に `continue` を設定します。`print-`

⁴⁹ 評価器が `branch` から始まるようになったので、評価器を開始する前には常に `flag` レジスタを初期化する必要があります。マシンを通常の REPL から開始させるためには、以下のコードを使います。

```

(define (start-eceval)
  (set! the-global-environment (setup-environment))
  (set-register-contents! eceval 'flag false)
  (start eceval))

```

`result` は `val` の値を表示し、それから評価器の REPL の最初に移動します。

50

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

これで、次の手続きを使うと、手続き定義をコンパイルし、コンパイル済みコードを実行し、REPL を実行して手続きを試してみることができます。コンパイル済みコードには、結果を `val` に入れて `continue` の位置に戻ってきてほしいので、式はターゲット `val` とリンク `return` でコンパイルします。コンパイラによって生成されるオブジェクトコードを評価器レジスタマシンで実行可能な命令に変換するために、レジスタマシンシミュレータ (Section 5.2.2) の手続き `assemble` を使います。それから `val` レジスタを命令リストを指すように初期化し、`flag` を評価器が `external-entry` に行くように設定して、評価器を開始します。

```
(define (compile-and-go expression)
  (let ((instructions
         (assemble
           (statements
             (compile expression 'val 'return))
           eval))))
```

50 コンパイル済み手続きはシステムが表示を試みるかもしれないオブジェクトなので、システムの表示演算 `user-print` (Section 4.1.4) も修正し、コンパイル済み手続きの部品を表示させないようにします。

```
(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))))
```

```
(set! the-global-environment (setup-environment))
(set-register-contents! eceval 'val instructions)
(set-register-contents! eceval 'flag true)
(start eceval)))
```

Section 5.4.4 の終わりでやったようにスタック監視を設定すると、コンパイル済みコードのスタック使用を調査できます。

```
(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
```

この例と、Section 5.4.4 の終わりに示した同じ手続きの解釈版を使った (factorial 5) の評価を比べてみてください。解釈版は、プッシュ 144 回と、最大スタック深度 28 を必要としていました。これは、コンパイル戦略による最適化のよい例となっています。

解釈とコンパイル

この節のプログラムを使うと、解釈とコンパイルという実行戦略の選択肢を実験することができるようになります。⁵¹ インタプリタはマシンをユーザプログラムのレベルに上げるもので、コンパイラはユーザプログラムを機械語のレベルに落とすものです。Scheme 言語は (Scheme 言語だけでなく、どんなプログラミング言語でも)、機械語の上に建てられた一貫性のある抽象化の一族と見なすことができます。インタプリタは、プログラム実行のステップがこれ

⁵¹ コンパイラを拡張して、コンパイルされたコードが解釈される手続きを呼び出せるようにすると、さらによくなります。Exercise 5.47 参照。

らの抽象化によって構成されていて、そのことによってプログラマに理解しやすくなっているため、対話的プログラム開発やデバッグに向いています。コンパイル済みコードは、プログラム実行のステップが機械語によって構成されており、またコンパイラは高レベルの抽象化の壁を越えて最適化を自由に行えるため、実行速度が速くなります。⁵²

解釈とコンパイルという二つの選択肢は、言語を新しいコンピュータに移植する際にもそれぞれ異なった戦略につながります。例えば、Lisp を新しいマシンに実装したいとしましょう。ひとつの戦略は、Section 5.4 の明示制御評価器をベースにして、その命令を新しいマシンの命令に翻訳するというものです。もうひとつの戦略は、コンパイラをベースにして、新しいマシンのコードを生成するようにコード生成器を作り替えるというものです。二つ目の戦略を取る場合、プログラムをまず元の Lisp システムで動くコンパイラでコンパイルして、それをコンパイル済みの実行時ライブラリとリンクすることで、どんな Lisp プログラムでも実行できるようになります。⁵³ それより良いのは、コ

⁵² ユーザープログラムの実行時にエラーにぶつかったとき、システムを止めたり間違った答えを出したりすることを許すのではなく、エラーを検出してシグナルを発生させることを求める、実行戦略を問わず、かなりのオーバーヘッドを引き起こすことになります。例えば、範囲外の配列参照は、実行前に参照の有効性をチェックすることによって検出できます。しかし、チェックのオーバーヘッドは配列参照そのもののコストの何倍にもなる可能性があるので、そのようなチェックが望ましいのかどうかを決める際にはプログラムは速度と安全性をはかりにかける必要があります。良いコンパイラは、そのようなチェックのあるコードを生成する能力を持ち、冗長なチェックを避け、コンパイル済みコード中のエラーチェックの程度や種類をプログラマが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、実行ができるだけ高速にするために、エラーチェック演算を実行コードにほとんど入れません。その結果、明示的にエラーチェックを行うことがプログラマに求められます。しかし、残念ながら、スピードが制約とならない重要なアプリケーションですら、これがおろそかにされることがよくあります。例えば、1988 年にインターネットを麻痺させた悪名高い“ワーム”は、finger デーモンで入力バッファがオーバーフローしたかどうかというチェックがされていないという UNIX(tm) オペレーティングシステムの欠陥を利用していました (Spafford 1989 参照)。

⁵³もちろん、解釈とコンパイルのどちらの戦略を取る場合でも、メモリ割り当て、出入力その他のこれまで評価器やコンパイラの議論の中で“基本的”としてきたさまざまな演算も実装する必要があります。ここで、できるだけ作業量を減らすひとつの戦略として、これらの演算を可能な限り Lisp で書いて、それらを新しいマシン用にコンパイルするというものがあります。究極的には、新しいマシンのために手書きするものは、(ガベージコレクションや実際のマシンの基本手続きを適用するメカニズムなどの) 小さなカーネルにすべてが集約されることになります。

ンパイラ自身をコンパイルして、それを新しいマシンで動かしてほかの Lisp プログラムをコンパイルするということです。⁵⁴ また、Section 4.1 のインタプリタのひとつをコンパイルして、新しいマシンで動くインタプリタを作るということも可能です。

Exercise 5.45: コンパイル済みコードと評価器が同じ計算を行う際に使うスタック演算を比較することで、コンパイラがスタック使用の最適化をどの程度行っているかを、速度(スタック演算の総数を減らすこと)と空間(スタックの最大深度を減らすこと)の両面から判断できる。この最適化されたスタック使用と、それと同じ計算をするための特別目的マシンの性能を比較すると、コンパイラの品質についてのある種の指標が得られる。

- a. Exercise 5.27では、上記の再帰階乗手続きを使って $n!$ を計算するのに評価器が必要とするプッシュの数と最大スタック深度を n の関数として表すよう求めた。Exercise 5.14では、Figure 5.11に示した特別目的階乗マシンについて、同じ測定を行うよう求めた。ここでは、コンパイルした `factorial` 手続きについて同じ分析を行う。

コンパイル版のプッシュの数と解釈版のプッシュの数の比を求めよ。また、スタック深度についても同じことを行え。 $n!$ を計算するのに使われる演算数とスタック深度は n に線形なので、 n が大きくなるにつれてこれらの比は定数に近づくはずだ。それらの定数を答えよ。同様に、特別目的マシンのスタック使用と解釈版のスタック使用の比を求めよ。

特定目的マシンのコードに対する解釈されるコードの比率と、コンパイル済みコードに対する解釈されるコードの比率を比較せよ。特定目的マシンがコンパイル済みコードよりもずっと性能がいいことがわかるだろう。手書きによるコントローラのコードは、我々の初步的な汎用目的コンパイラに生成されるものよりずっと優れているからだ。

⁵⁴ この戦略を取る場合、コンパイルしたコンパイラを新しいマシンで動かしたものによってプログラムをコンパイルした結果が元の Lisp システムでコンパイルしたものと同一になるかといった、コンパイラの正確性についての面白いテストをすることになります。違いの原因を突き止めるのは楽しい作業なのですが、イライラすることもよくあります。結果はほんのわずかな詳細に大きく影響されるからです。

- b. 手書き版により近い性能のコードを生成するようコンパイラを改良するアイデアを提案できるか。

Exercise 5.46: 木の再帰による以下のフィボナッチ手続きについて、

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
          (fib (- n 2)))))
```

これをコンパイルしたときの性能と、Figure 5.12の特別目的フィボナッチマシンを使う場合の性能とを比較し、Exercise 5.45のような分析を実行せよ（解釈版のパフォーマンス測定については Exercise 5.29 参照）。フィボナッチ数の計算では、使われる時間的リソースは n に対して線形ではないため、スタック演算の比は n と独立な極限値には近づかない。

Exercise 5.47: この節では、解釈されるコードがコンパイル済み手続きを呼び出せるように明示制御評価器を修正する方法について記述した。コンパイル済み手続きが、基本手続きとコンパイル済み手続きだけでなく、解釈される手続きも呼べるようにコンパイラを修正する方法を示せ。このためには、`compile-procedure-call` を修正し、複合（解釈される）手続きの場合を扱うようにする必要がある。`compile-proc-appl` と同じ `target` と `linkage` の組み合わせをすべて扱うよう気をつけること。実際の手続き適用を行うには、コードは評価器の `compound-apply` エントリポイントにジャンプしなければならない。このラベルはオブジェクトコードからは直接参照できない（アセンブル中のコードによって参照されるラベルはすべてそのコード内で定義したものであるようアセンブラーが要求することによる）。そのため、`compapp` というレジスタを評価器マシンに追加してそのエントリポイントを持たせるようにして、このレジスタを初期化する命令を追加する。

```
(assign compapp (label compound-apply))
  (branch (label external-entry)) ;flag が設定されていれば分岐する
read-eval-print-loop ...
```

コードをテストするには、まず手続き `g` を呼ぶ手続き `f` を定義する。それから `compile-and-go` を使って `f` の定義をコンパイルし、評価器をスタートさせる。そこで、評価器に対し入力を行って `g` を定義し、`f` の呼び出しを試みる。

Exercise 5.48: この節で実装した `compile-and-go` インタフェイスは、一回（評価器開始時）しか呼ぶことができないため、扱いにくい。明示制御評価器から次のようにしてコンパイラを呼び出すことができる `compile-and-run` という基本手続きを追加して、コンパイラ-インタプリタ間のインターフェイスを拡張せよ。

```
; ; 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: 明示制御評価器の REPL の代替手段として、REPL を行うレジスタマシンを設計せよ。つまり、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、結果を出力するというループを走らせるということになる。これは、シミュレーション構成の中では簡単に実行できる。手続き `compile` と `assemble` を“レジスタマシン演算”として呼ぶようにすればよいからだ。

Exercise 5.50: Section 4.1 のメタ循環評価器をコンパイラを使ってコンパイルし、レジスタマシンシミュレータを使ってそ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には、それらの定義を `begin` の中にまとめればよい）。その結果できるインタプリタは、複数レベルで解釈が行われるため、実行速度はとても遅くなるだろう。しかし、そのすべての詳細をうまく動かせるようにすることは学びの多い練習となる。

Exercise 5.51: Section 5.4の明示制御評価器を翻訳することによって、C 言語（または好きな低レベル言語）による Scheme の基本的な実装を開発せよ。このコードを動かすためには、適切なメモリ割り当てルーチンや、その他の実行時サポートを用意する必要があるだろう。

Exercise 5.52: Exercise 5.51と対応するものとして、コンパイラを修正して Scheme の手続きを C 言語の命令列にコンパイルするようせよ。Section 4.1のメタ循環評価器をコンパイルして、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 revised⁴ 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.

練習問題リスト

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

この索引にの間違いがあるとしたら、コンピュータの助けを借りて索引を作ったことによるものだろう。

—Donald E. Knuth, *Fundamental Algorithms*
(Volume 1 of *The Art of Computer Programming*)

- k*-term finite continued fraction, 76
- k* 項有限連分数, 76
- n*-fold smoothed function, 83
- n* 重平滑化関数, 83
- abstract models, 97
- abstract syntax, 402
- abstraction barriers, 88, 94
- accumulator, 124, 243
- acquire, 342
- action, 551
- additive, 195
- additively, 89, 184
- address, 589
- address arithmetic, 589
- agenda, 307
- algebraic specification, 98
- aliasing, 253
- and-gate, 299
- AND ゲート, 299
- applicative-order, 439
- arbiter, 345
- arguments, 7
- assembler, 568
- assertions, 486
- assignment operator, 237
- atomically, 344
- automatic storage allocation, 589
- average damping, 76
- B-trees, 171
- backbone, 291
- backquote, 634
- backtrack, 457
- balanced, 120, 170
- barrier synchronization, 347
- base address, 590
- Bertrand's hypothesis, 364
- bignum, 591
- bind, 29
- bindings, 256
- binomial coefficients, 44
- block structure, 31
- bound variable, 29

box-and-pointer notation, 104
breakpoint, 587
broken heart, 600
bugs, 2
B 木, 171

cache-coherence, 328
call-by-name, 442
call-by-need, 442
call-by-name, 357
call-by-name thunks, 357
call-by-need, 357
call-by-need thunks, 357
capturing, 29
Carmichael numbers, 56
case analysis, 17
cell, 343
chronological backtracking, 457
Church numerals, 100
Church-Turing thesis, 424
clause, 18
closure, 88
code generator, 629
coerce, 217
coercion, 212
combinations, 6
comments, 134
compacting, 598
compilation, 626
compile-time environment, 667
composition, 83
compound data, 86
compound data object, 86
compound procedure, 13
computability, 424
computational process, 1
concurrently, 326
congruent modulo, 54
connectors, 313
consequent expression, 18
constraint networks, 313
constructors, 89

continuation procedures, 470
continued fraction, 76
control structure, 510
controller, 544
conventional interfaces, 88, 123
current time, 310

data, 1, 97
data abstraction, 86, 89
data path, 544
data-directed, 185
data-directed programming, 89, 195
deadlock, 346
deadlock-recovery, 346
debug, 2
deep binding, 419
deferred operations, 35
delayed argument, 384
delayed evaluation, 236, 349
delayed object, 352
dense, 226
dependency-directed backtracking, 458
depth-first search, 457
deque, 291
derived expressions, 411
digital signals, 299
dispatching on type, 194
displacement number, 666
dotted-tail notation, 112
driver loop, 421

empty list, 108
encapsulated, 240
enclosing environment, 256
entry point, 547
enumerator, 124
environment, 9
environment model, 236
environments, 256
Euclid's Algorithm, 51
Euclidean ring, 231

evaluation, 6
evaluator, 398
event-driven simulation, 299
evlis tail recursion, 610
execution procedure, 434
explicit-control evaluator, 604
expression, 6

failure continuation, 470
FIFO, 286
filter, 66, 124
first-class, 82
fixed point, 74
fixed-length, 175
forcing, 442
forwarding address, 600
frame, 500
frame coordinate map, 146
frame number, 666
framed-stack, 608
frames, 256
free, 29
free list, 593
front, 285
full-adder, 302
function boxes, 299
functional programming, 249
functional programming languages, 393

garbage, 596
garbage collection, 589, 596
garbage collector, 274
garbage-collect, 446
generic operations, 88
generic procedures, 180, 185
glitches, 2
global, 32, 256
global environment, 9
golden ratio, 39
grammar, 464

half-interval method, 72
half-adder, 300
Halting Theorem, 427
headed list, 292
hiding principle, 240
hierarchical, 106
hierarchy of types, 214
higher-order procedures, 61
Horner's rule, 129

imperative programming, 254
indeterminates, 220
index, 590
indexing, 501
instantiated with, 490
instruction, 543, 547
instruction counting, 587
instruction execution procedure, 570
instruction sequence, 631
instruction tracing, 587
integerizing factor, 232
integers, 5
integrator, 378
interning, 592
interpreter, 3, 398
invariant quantity, 48
inverter, 299
iterative improvement, 84
iterative process, 35

key, 174

label, 547
lazy evaluation, 439
lexical address, 666
lexical addressing, 419
lexical scoping, 31
linear iterative process, 35
linear recursive process, 35
linkage descriptor, 630
list, 107
list structure, 107

list-structured, 92
list-structured memory, 588
local evolution, 32
local state variables, 237
location, 589
logic-programming, 399
logical and, 299
logical deductions, 496
logical or, 300

machine language, 625
macro, 411
map, 124
mark-sweep, 597
memoization, 42, 298
memoize, 442
merge, 394
message passing, 99, 203
message-passing, 242
metacircular, 400
Metalinguistic abstraction, 397
Miller-Rabin test, 59
modular, 235
modulo, 54
modus ponens, 509
moments in time, 326
Monte Carlo integration, 247
Monte Carlo simulation, 245
mutable data objects, 273
mutators, 273
mutex, 342
mutual exclusion, 342

native language, 625
needed, 632
network, 397
Newton's method, 79
nil, 108
non-computable, 427
non-strict, 440
nondeterministic, 332
nondeterministic choice point, 457

nondeterministic computing, 399, 453
normal-order, 439
normal-order evaluation, 399

obarray, 592
object program, 626
objects, 236
open-code, 664
operand, 7
operator, 7, 432
or-gate, 300
order of growth, 44
ordinary, 205
OR ゲート, 300
output prompt, 421

package, 196
painter, 137
pair, 91, 92
parse, 463
Pascal's triangle, 44
pattern, 488
pattern matcher, 500
pattern matching, 499
pattern variable, 488
pipelining, 326
pointer, 104
poly, 221
power series, 366
predicate, 18
prefix, 176
prefix code, 176
prefix notation, 7
pretty-printing, 8
primitive constraints, 313
probabilistic algorithms, 57
procedural epistemology, xxiii
procedure, 4
procedure definitions, 12
program, 1
programming languages, 2
prompt, 421

pseudo-random, 244
pseudodivision, 232
pseudoremainder, 232

quasiquote, 634
queries, 485
query language, 485
queue, 285
quote, 154

Ramanujan numbers, 377
rational functions, 230
RC circuit, 378
RC 回路, 378
read-eval-print loop, 8
reader macro character, 535
real numbers, 5
rear, 285
recursion equations, 2
Recursion theory, 425
recursive, 10, 26
recursive process, 35
red-black trees, 171
referentially transparent, 252
register, 543
register machine, 543
register table, 570
release, 342
resolution principle, 483
ripple-carry adder, 304
robust, 153
RSA アルゴリズム, 57
rules, 485, 493

satisfy, 490
scope, 29
selectors, 89
semaphore, 342
separator code, 176
sequence, 106
sequence accelerator, 370
sequences, 65

serializer, 333
serializers, 335
series RLC circuit, 386
shadow, 257
shared, 280
side-effect bugs, 253
sieve of Eratosthenes, 360
smoothing, 83
source language, 626
source program, 626
sparse, 226
stack, 36, 561
state variables, 35, 237
statements, 632
stop-and-copy, 597
stratified design, 152
stream, 349
streams, 236, 348
strict, 440
subroutine, 556
substitution model, 15
subtype, 214
success continuation, 470
summation of a series, 62
summer, 378
supertype, 214
symbolic expressions, 88
syntactic sugar, 12
syntax, 401
system, 397
systematically search, 457

tableau, 371
tabulation, 42, 298
tagged architectures, 591
tail-recursive, 36, 614
target, 630
thrashing, ix
thunk, 441
time, 326
time segment, 310
tower, 214

tree accumulation, 10
tree recursion, 38
trees, 117
truth maintenance, 458
Turing machine, 424
type field, 591
type tag, 190
type tags, 185
type-inferencing, 389
typed pointers, 590

unbound, 256
unification, 483, 499, 505
unification algorithm, 483
univariate polynomials, 220
universal machine, 424
upward-compatible extension, 450

value, 8
value of a variable, 256
variable, 8
variable-length, 175
vector, 589

width, 102
wires, 299
wishful thinking, 90

Y コンビネータ, 432

zero crossings, 380

アクション, 551
アセンブラー, 568
アトミック, 344
アドレス, 589
アドレス演算, 589
アービタ, 345
イベント駆動シミュレーション, 299
インタプリタ, 3, 398
インデクシング, 501
インデックス, 590

インバータ, 299
エイリアス, 253
エブリス末尾再帰, 610
エラトステネスのふるい, 360
エントリポイント, 547
オブアレイ, 592
オブジェクト, 236
オブジェクトプログラム, 626
オペランド, 7
オペレータ, 7
オープンコード, 664
カプセル化, 240
カーマイケル数, 56
ガベージコレクション, 446, 589, 596
ガベージコレクタ, 274
キャッシュコヒーレンス, 328
キャプチャ, 29
キュー, 285
キー, 174
クエリ, 485
クエリ言語, 485
クオート, 154
クロージャ, 88
グリッチ, 2
グローバル, 32, 256
グローバル環境, 9
コネクタ, 313
コメント, 134
コントローラ, 544
コンパイル, 626
コンパイル時環境, 667
コード生成器, 629
ゴミ, 596
サブタイプ, 214
サブルーチン, 556
サンク, 441
システム, 397
シリアルライザ, 335
シンタックスシュガー, 12
ジェネリック手続き, 180, 185
スコープ, 29
スタック, 36, 561

スタックフレーム, 608
ストップ & コピー, 597
ストリーム, 236, 348, 349
ストリーム処理, 17
スラッシング, ix
スーパーイフ, 214
セマフォ, 342
セル, 343
ゼロ交差, 380
ソースプログラム, 626
ソース言語, 626
タイプタグ, 185, 190
タグアーキテクチャ, 591
タブロー, 371
タワー, 214
ターゲット, 630
チャーチ・チューリングのテーゼ, 424
チャーチ数, 100
チューリングマシン, 424
テーブル化, 42, 298
デジタル信号, 299
デッドロック, 346
デッドロック回復, 346
デバッグ, 2
データ, 1, 97
データバス, 544
データ主導, 185
データ主導プログラミング, 89, 195
データ抽象化, 86, 89
ドット末尾記法, 112
ドライバープ, 421
ニュートン法, 79
ネイティブ言語, 625
ネットワーク, 397
バグ, 2
バッククオート, 634
バックトラック, 457
バランスが取れている, 120
バランスの取れた, 170
バリア同期, 347
パイプライン, 326
パスカルの三角形, 44
パターン, 488
パターンマッチャ, 500
パターンマッチング, 499
パターン変数, 488
パッケージ, 196
ピッグナム, 591
ファーストクラス, 82
フィルタ, 66, 124
フレーム, 256, 500
フレーム数, 666
ブレークポイント, 587
ブロック構造, 31
プリティプリント, 8
プログラミング言語, 2
プログラム, 1
プロセス, 36
プロンプト, 421
ベクタ, 589
ベルトランの仮説, 364
ベースアドレス, 590
ペア, 91, 92
ペインタ, 137
ホーナー法, 129
ポインタ, 104
マクロ, 411
マップ, 124
マーク & スイープ, 597
マージ, 394
ミューテックス, 342
ミューテータ, 273
メタ循環, 400
メタ言語抽象化, 397
メッセージパッキング, 99, 203, 242
メモ化, 42, 298, 442
モジュール式, 235
モンテカルロシミュレーション, 245
モンテカルロ積分, 247
モーダス・ポネンス, 509
ユニフィケーション, 483, 499, 505
ユニフィケーションアルゴリズム, 483
ユークリッドの互除法, 51
ユークリッド環, 231

- ラベル, 547
 ラマスジャン数, 377
 リスト, 107, 112
 リスト構造, 92, 107
 リスト構造メモリ, 588
 リンク記述子, 630
 レキシカルアドレス, 666
 レキシカルアドレッシング, 419
 レキシカルスコーピング, 31
 レジスタ, 543
 レジスタテーブル, 570
 レジスタマシン, 543
 レプレル, 8
 一元多項式, 220
 万能機械, 424
 上位互換性のある拡張, 450
 不動点, 74
 不变量, 48
 不定元, 220
 両端キュー, 291
 並行, 326
 予定表, 307
 代入演算子, 237
 代数的仕様, 98
 位置, 589
 体系的に探索, 457
 依存主導バックトラック, 458
 値, 8, 154
 停止性問題, 427
 先入れ先出し, 286
 先端, 285
 全加算器, 302
 共有, 280
 具体化される, 490
 再帰, 10
 再帰プロセス, 35
 再帰方程式, 2
 再帰理論, 425
 再帰的, 26
 幕級数, 366
- 出力プロンプト, 421
 分離符号, 176
 列, 65, 106
 列加速, 370
 列挙, 124
 制御構造, 510
 制約ネットワーク, 313
 前置記法, 7
 副作用バグ, 253
 加法的, 89, 184, 195
 加算器, 378
 区間二分法, 72
 半加算器, 300
 厳密, 440
 参照透明, 252
 反復プロセス, 35
 反復改良法, 84
 可変データオブジェクト, 273
 可変長, 175
 合成, 83
 名前呼び, 357, 442
 名前呼びサンク, 357
 命令, 543, 547
 命令トレース, 587
 命令列, 631
 命令型プログラミング, 254
 命令実行手続き, 570
 命令数カウンタ, 587
 命令文, 632
 固定長, 175
 圧縮, 598
 型つきポインタ, 590
 型によるディスパッチ, 194
 型の階層, 214
 型フィールド, 591
 型推論, 389
 基本制約, 313
 場合分け, 17
 増加オーダー, 44
 変位数, 666
 変数, 8
 変数の値, 256

- 外側の環境, 256
 失敗継続, 470
 実数, 5
 実行手続き, 434
 密, 226
 導出原理, 483
 局所展開, 32
 局所状態変数, 237
 希望的思考, 90
 幅, 102
 平均緩和法, 76
 平滑化, 83
 式, 6
 引数(ひきすう), 7
 強制, 442
 強制型変換, 212, 217
 必要とする, 632
 必要呼び, 357, 442
 必要呼びサンク, 357
 成功継続, 470
 手続き, 4, 36
 手手続きの定義, 12
 手手続き的認識論, xxiii
 抽象モデル, 97
 抽象化の壁, 88, 94
 抽象手続き, 28
 抽象構文, 402
 接頭符号, 176
 接頭辞, 176
 擬似クオート, 634
 擬似乱数, 244
 擬剩余, 232
 擬除算, 232
 整数, 5
 整数化因子, 232
 文法, 464
 明示制御評価器, 604
 時間, 326
 時間区分, 310
 時間的バックトラック, 457
 最大公約数, 230
 有理関数, 230
 木, 117
 木の再帰, 38
 木の集積, 10
 未束縛, 256
 末尾再帰, 36, 614
 束縛, 29, 256
 束縛変数, 29
 枠座標マップ, 146
 構文, 401
 構文解析, 463
 標準インターフェイス, 123
 機械語, 625
 正規順序, 439
 正規順序評価, 17, 399
 法 n に関して合同, 54
 法 n に関する a の剩余, 54
 派生式, 411
 深い束縛, 419
 深さ優先探索, 457
 満たす, 490
 演算プロセス, 1
 演算子, 7
 特殊形式, 11
 状態変数, 35, 237
 獲得, 342
 現在時刻, 310
 環境, 9, 256
 環境モデル, 236
 疎, 226
 直列 RLC 回路, 386
 直列変換器, 333
 相互排除, 342
 真理維持, 458
 瞬間, 326
 確率的アルゴリズム, 57
 確立されたインターフェイス, 88
 移転先アドレス, 600
 積分器, 378
 空きリスト, 593
 空リスト, 108

- 箱とポインタ記法, 104
節, 18
級数の総和, 62
終端, 285
組み合わせ, 6
結果式, 18
絶対値, 190
継続手続き, 470
総称演算, 88
線, 299
線形再帰プロセス, 35
線形反復プロセス, 35
繰り上がり伝播加算器, 304
置換, 16
置換モデル, 15

背骨, 292
自動記憶領域割り当て, 589
自由, 29
表明, 486
複合データ, 86
複合データオブジェクト, 86
複合手続き, 13
規則, 485, 493
解放, 342
計算不可能, 427
計算可能性, 424
記号式, 88
評価, 6
評価器, 398
読み込みマクロ文字, 535
論理プログラミング, 399

論理和, 300
論理的推論, 496
論理積, 299
赤黒木, 171
述語, 18

通常の, 205
連分数, 76
遅延オブジェクト, 352
遅延引数, 384
遅延演算, 35
遅延評価, 236, 349, 439
適用順序, 439
適用順序評価, 17
閉じ込め, 592
閉包性, 105
関数型プログラミング, 249
関数型プログラミング言語, 393
関数箱, 299
階層, 106
階層化設計, 152
隠蔽, 257
隠蔽原理, 240
集積機, 124
非厳密, 440
非決定性計算, 399, 453
非決定性選択点, 457
非決定的, 332
頑健, 153
頭つきリスト, 292
高階手続き, 61
黄金比, 39

奥付

表紙に描かれているのは、Agostino Ramelli によって 1588 年に作られたブックホイール機構です。これは、初期のハイパーテキストナビゲーション支援といえるでしょう。この版画の画像は [New Gottland](#) の J. E. Johnson によってホストされています。

本文のフォントは Linux Libertine で、見出しは Linux Biolinum です。どちらも Philipp H. Poll によるものです。等幅フォントは、Raph Levien によって作られた Inconsolata を Dimosthenis Kaponis と Takashi Tanigawa が Inconsolata LGC という形で補完したものを使っています。^{誤注 7}

グラフィックデザインとタイプグラフィは Andres Raba の手によります。Texinfo のソースは Perl スクリプトで 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 で描かれたものです。

^{誤注 7}日本語版では漢字に IPA フォントを使っています。