Scheme による
プログラミング入門と実習

角川裕次
広島大学工学部第二類 (電気系)

連絡先:
〒 739　広島県東広島市鏡山 1-4-1
広島大学工学部第二類 (電気系)
e-mail: kakugawa@se.hiroshima-u.ac.jp

Copy Free 版
“Scheme によるプログラミング入門と実習”
使用条件書

1996 年 3 月 13 日

角川裕次
広島大学工学部第二類

1. **総則**
   この使用条件書は、 角川裕次と利用者との間での、Copy Free 版 “Scheme によるプログラミング入門と実習” （以下「本件」と呼ぶ）の使用に関する取り決めです。

2. **使用条件**
   以下の条件のいずれかに合致するとき、本件の使用を許可します。
   (2.a) 個人的な使用。
   (2.b) 営利を目的としない教育・研究目的での使用。
   (2.c) 本件の著作権保有者が特別に認めたとき。

3. **配布**
   以下の条件のいずれかに合致するとき、およびそのときのみに限り、媒体を問わず、本件の再配布を許可します。なお、本条件書に違反した者や団体等から本件を入手した場合でも、本条件書に従う限り使用を許可します。
   (3.a) 非営利目的の配布。ただし、接続料金を課すパソコン通信等で、インターネット (The Internet) に接続された計算機での本件の保持は、インターネットに接続された他の組織からの本件の入手が可能であることが条件です。
   (3.b) 著作権保有者が特別に認めたとき。ただしこのときは、配布条件を別途定めるものとします。

4. **改良、変更**
   改良のため、予告なく本件の内容を変更することがあります。利用者による本件の改良・変更は一切禁止します。

5. **無保証**
   本件は無保証です。直接的、間接的を問わず、本件を使用することにより生じた損害、訴訟等に対して、角川裕次および広島大学は一切責任を負いません。

6. **その他**
   (6.a) 本件の著作権は、角川裕次が保有しています。本使用条件書に違反した利用、ならびに本件の一部または全部の無断引用は、著作権侵害となる場合があります。
   (6.b) 本件に関する問い合わせは一切受け付けません。
   (6.c) 本件の営利目的の利用が御希望のときは、別途使用許諾の取り決めが必要です。
   (6.d) 本使用条件書が改定された場合は、利用者は最新の使用条件書に従うものとします。

以上

Scheme によるプログラミング入門と実習　履歴

第 1 版　1993 年 6 月
第 2 版　1994 年 8 月
第 3 版　1995 年 9 月
第 3.1 版　1996 年 3 月
第 3.2 版　1996 年 8 月

目 次

Scheme の規格

本書では、W. Clinger, J. Rees (編) "Revised^4 Report on the Algorithmic Language Scheme" (通称 R4RS) による Scheme の規格を採用しています。R4RS で定義されている Scheme 手続きや構文のうち、以下のものは説明をしていません。

- `force` と `delay`
- マクロ

R4RS で定められている Scheme 手続きや構文は、必須 (essential) なものと非必須なものとに分かれています。必須手続きと必須構文は R4RS 準拠の処理系は必ず備えておかないといけない手続きです。いっぽう非必須手続きと非必須構文は必ずしも用意する必要のないために、処理系によっては用意されていないかも知れません。手続きの説明では、以下の書き方によってそれらを区別しています。(左端の記号に注意して下さい。)

♢ (`scheme-proc` *arg*)
— 必須手続きまたは必須構文を表します。

△ (`scheme-proc` *arg*)
— 非必須手続きまたは非必須構文を表します。

√ (`scheme-proc` *arg*)
— 処理系独自の手続きまたは構文を表します。

本書での表記法

本書では以下の規則に従って、プログラムや Scheme 処理系の出力を書き表しています。(この他の表記法を使う場合は、そのつど説明があります。)

- `Typewrite Font`
  — Scheme のプログラムやデータを表します。
- *`Italic Typewrite Face`*
  — Scheme 処理系による出力 (表示) を表します。

第 1 章
コンピューターとプログラミング

コンピューター (computer) は、事務処理の機械やワードプロセッサとして、オフィスや家庭に広く浸透しています。ビデオゲーム機も、コンピューターのひとつです。そのほかあまり目立たないコンピューターもあります。炊飯器や洗濯機などの家電製品には、超小型のコンピューターが内蔵されていて、いろいろな気のきいた仕事をしてくれます。

ワープロやゲーム機のようなものだけでなく、もっと重要な仕事をしているコンピューターもあります。駅の旅行センターでは、座席の予約システムにコンピューターを使うことで、瞬時に座席の予約ができます。電車会社に設置されている中央のコンピューターは、全国から時々刻々やってくる座席の予約依頼に従って座席の予約をします。銀行のキャッシュディスペンサーは通信回線を通して、本店にある大型コンピューターとつながっています。このコンピューターが銀行口座を管理していて、貯金残高の管理をしています。テレビに出てくる天気予報は、雲の移動や発生などを気象庁のスーパーコンピュータで計算し、(予報官が過去の経験を加味して) 天気の予測をしています。

コンピューターがとても複雑な仕事をあっという間に片付けてゆくからといって、コンピューターを恐がる必要はありません。所詮は人間が作った機械ですし、(故障のない限りは) コンピューターは人間が与えた使命 (**プログラム** (program) といいます) に忠実に従う下僕です。コンピューターに使命を与えるには日本語や英語ではだめで、コンピューター専用の言葉でないといけません。それに加えて、人間がこうしろ、といって与えた使命にちょっと間違いがあっても、何も疑うことなく馬鹿正直にそれに従います。もしコンピューターが変なことすれば、それは主人である人間が間違った命令を与えたからです。ですが、コンピューターに正しい命令を与えると、ちゃんと言う通りに動いてくれます。

この本は Scheme というプログラミング言語 (コンピューターへの命令を書くための言葉) を使って、プログラムの作り方 (プログラミング) を学ぶことを目的としています。コンピューターの内部構造を知らなくても、一応はプログラムを作ることはできます。ですが、コンピューターの内部の仕組みをある程度知っておいた方が、より良いプログラムを作ることができます。この章では典型的なコンピューターの構成と、コンピューターがプ

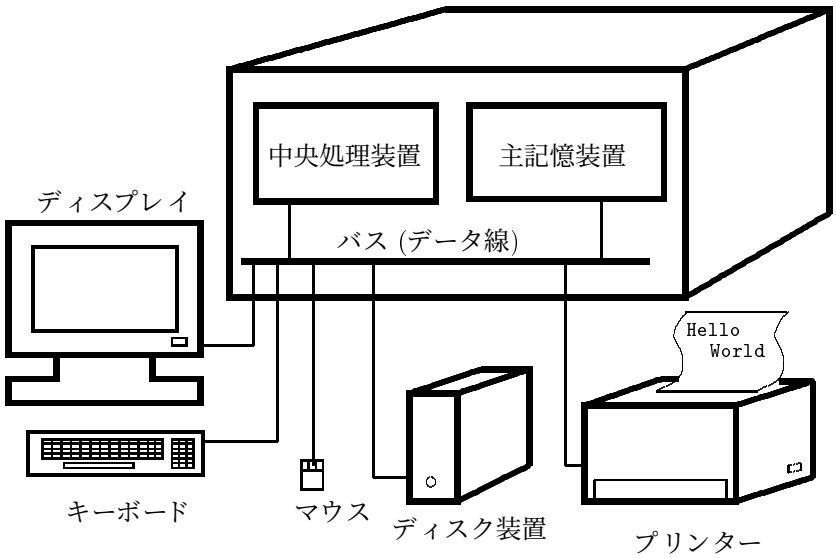


図 1.1: コンピューターシステムの構成図

ログラムを実行してゆく仕組みを簡単に説明します。

1.1 コンピューターの構成

コンピューターがどのようなものから構成されているかを説明します。ワードプロセッサを使って文章を作るときは、次のようにしています。まずキーボードを使って文章を入力します。仮名漢字変換プログラムによって、漢字の入った文章に変換します。最後に、入力した文章をプリンタで印字します。ワードプロセッサは文章を憶えてはいますが、電気が切れると忘れてしまいます。そこで、作った文章はフロッピーディスクやハードディスクに保存します。これらは音楽のカセットテープと同様な原理を使ってデータの記録をしています。

コンピューターには、大まかにいって次のような 3 つの部分より構成されます。(かなりいい加減ですが、括弧の中は人間の体や身の周りのものに相当するものです。)

1. 計算や各種の処理をする部分。(脳みそ)
2. 計算作業をするために、一時的にデータを記憶する場所。(計算用紙、メモ用紙)
3. コンピューターの外の世界との情報のやりとりをする、キーボードなどの装置や、データを記録しておく装置。(目、耳、手足、ノート、帳簿)

これら 3 つは、コンピューターの世界では次のように呼ばれています[1]。(図 1.1参照)

[1]CPU は Central Processing Unit の略で、I/O は Input/Output の略です。

装置名	はたらき
キーボード	キーを押して文章を入力します
CRT ディスプレイ	文字や図形をブラウン管上に表示します
マウス	板の上でマウスを移動させ、 X-Y 座標の入力をします
プリンタ	文字や図形を紙の上に印字します
ディスク	ディスク上に磁気を使ってデータの記録をします
磁気テープ	磁気テープにデータの保存をします
イメージスキャナ	写真や絵をディジタル信号にして読み込みます

図 1.2: 入出力装置の例

1. **中央処理装置** (CPU)
2. **主記憶装置** (main memory)
3. **入出力装置** (I/O device)

先ほど説明したワードプロセッサの例では、仮名漢字変換プログラムやプリンタを制御するプログラムは、中央処理装置によって実行されます。各種プログラムや作成した文章は、主記憶装置に記録されています。フロッピーディスク、プリンタ、キーボードなどは入出力装置です。

入出力装置のうちでも、ディスクにはたくさんののデータを記録することができます。パーソナルコンピュータやワードプロセッサには、フロッピーディスクという装置がついています。これもディスクの仲間です。主記憶装置は電源が切れると記憶内容が失われるので、プログラムやデータはディスクに保存します。データによっては、主記憶装置に入らないくらい大きなものもあります。そのような大きなデータは、ディスクや磁気テープにデータを記録しておき、一部分ずつデータを主記憶装置に読み込んで計算処理を進めます。図 1.2 に入出力装置の例を示します。

最近の主記憶装置は、半導体による IC (Integrated Circuit — 集積化回路) や LSI (Large Scale Integrated Circuit — 大規模集積化回路) で作られています。これらの主記憶装置は、データの読み出しや書き込みを高速に行なうことができます。カセットテープである曲を聞きたいときはテープを回転させて頭出しをするように、入出力装置装置は機械的な動作を必要とします。いっぽう半導体による主記憶装置は電気的な信号を送るだけよく、機械的な動作を伴いません。そのため、より速いデータの読み書きができます。

主記憶装置には、多くの数の**記憶セル** (memory cell) があります。この記憶セルが記憶の単位となって主記憶装置を構成しています。記憶セルは **ビット** (bit) と呼ばれる、さらに小さな記憶単位より構成されています。このビットが最小の記憶の単位で、0 または 1 を覚えておくことができます。(ですが、それだけしか覚えられません。) 記憶セルはビットをいくつか集めたものです。コンピューターの種類にもよりますが、1 つの記憶セルに

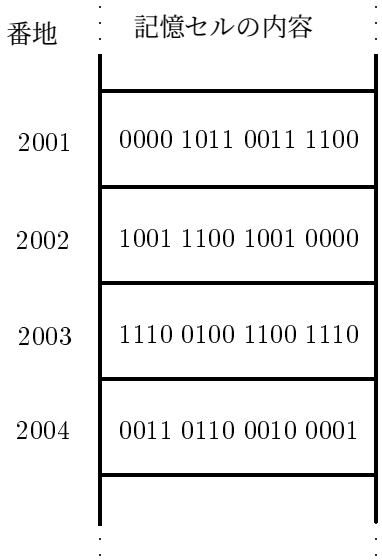


図 1.3: 記憶セルとその内容

は 16 個とか 32 個のビットで構成されているものが多いです。1 つの記憶セルを**語** (word) とも呼んでいます。中央処理装置は記憶セルを 1 つの単位にして、データの読み書きをします。

コンピューターの中では、どのデータも 0 と 1 の組合せで表現されています。では、数の 5 を覚えさせるにはどうすればいいでしょうか? 0 と 1 の組み合わせのそれぞれのパターンがどの数を表すか、ということがあらかじめ決めてあります。たとえば、 記憶セルの内容が 00000101 のときは数の 5 を表す、という具合です。0 と 1 のならびで数を表現する方法は、**二進数** (binary number) と呼ばれています。

文字を覚えさせるには、文字に番号を振り、その番号で記憶させます。文字とその番号を決めている ASCII 文字集合という規格では、アルファベット A に 65 を割り当てています。数 65 は 01000001 と二進数で表すことができ、この 0 と 1 のならびを記憶セルに記憶させます。

主記憶装置には記憶セルがたくさんありますが、記憶セルを使うときには、どの記憶セルを使うかを指定しないといけません。この指定のために、記憶セルには 0,1,2,... と番号が付けられています。この番号は**記憶番地**あるいは**メモリアドレス** と呼ばれています。図 1.3に主記憶装置の概念図を示します。

記憶セルに書かれているだけの内容を見ても、ワードプロセッサの文章の一部なのか、家計簿プログラムでの食費の部分なのかは区別がつきません。というのも、記憶セルには 0 と 1 しか書いてないからです。このために、プログラムを作成するときは、どの番地にどのデータを記憶させているかを把握しておかないといけません。

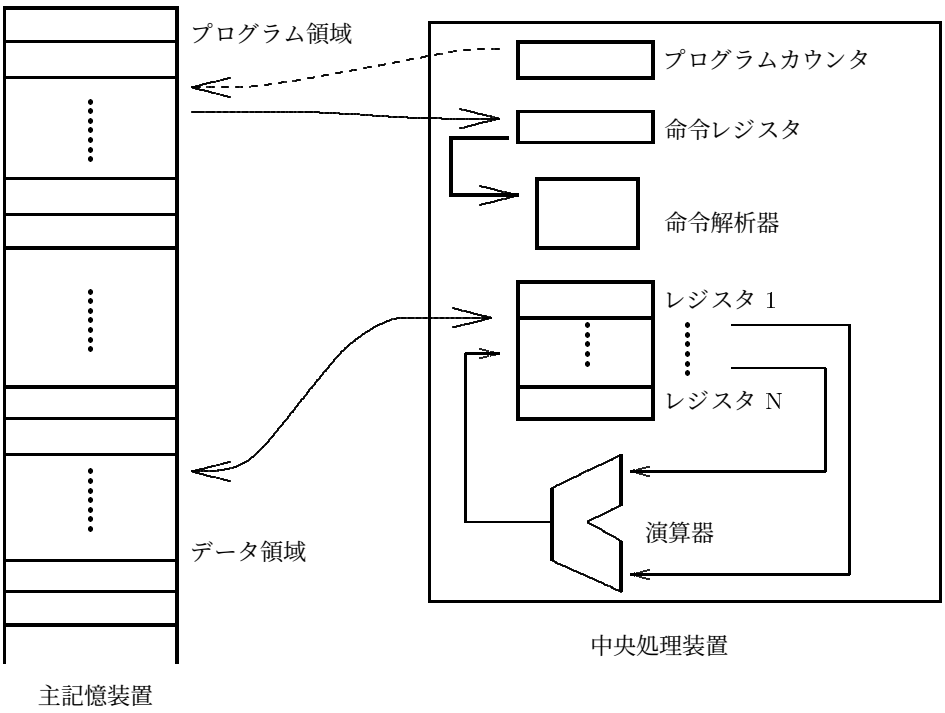


図 1.4: プログラム実行の様子

1.2 プログラムの実行

コンピューターが実行するプログラムは図 1.3のように、主記憶装置の記憶セルに書き込まれます。プログラム全部はひとつの記憶セルには入らないので、複数の記憶セル (連続した場所を使うのが普通です) を使います。コンピューターには、基本的な動作をする命令 (機械語命令) が何種類もあります。プログラムは機械語命令をならべて書いて、望みの動作をコンピューターにさせるものです。

記憶セルに書かれている命令を中央処理装置が実行することを繰り返しすことで、プログラムが実行されます。コンピューターが直接実行できるプログラムは、中央処理装置が直接理解できる二進数 (0 と 1 のならんだもの) で書かれた**機械語** (native language) だけです。Scheme や C や Fortran など、プログラムを書くための言語 (プログラミング言語) が色々な種類があります。しかしそのような言語で書かれたプログラムが、コンピューターで直接実行されるのではありません。特別なプログラムを使って機械語に翻訳し、コンピューターは翻訳された結果の機械語プログラムを実行します。(機械語に翻訳しない形のものもあります。これについては後で述べます。)

コンピューターがどのようにプログラムを実行してゆくかを、もう少し詳しく説明します。(図 1.4を参照)

中央処理装置の中には、**レジスタ** (register) がいくつかあります。レジスタは、一時的

に記憶セルの内容や計算結果を記憶する記憶装置です。レジスタの数は少なく、数個から数十個程度しかありません。レジスタは、 記憶セルの内容を保持したり、加算や乗算などの演算結果を保持するのに使われます。レジスタの中に、**プログラムカウンタ** (program counter) と呼ばれる特別なレジスタがあります。このレジスタは、次に実行する命令が書かれている記憶セルの記憶番地を保持するためのものです。

機械語はいくつもの種類の命令の集まりです。たとえば、次のものが典型的な機械語命令です。

- レジスタの内容に対して演算をする。
- 記憶セルの内容をレジスタに読み込む。
- レジスタの内容を記憶セルに書き込む。
- プログラムカウンタの値を変更する。(この命令により、別の場所の機械語命令に実行を移します。)
- レジスタの内容がゼロかどうかを判定する。

どのプログラムも、非常に基本的な動作しかしない機械語命令を組み合わせて、目的とする複雑な計算や情報処理をします。

次に、ひとつの機械語命令を実行する仕組みを説明します。

1. まず中央処理装置は、プログラムカウンタの示している記憶セルの値 (機械語命令) を読み出します。
2. 中央処理装置内の命令解析器という部分で、その値が何をする命令かを割り出します。
3. その命令に従った動作 (レジスタを記憶セルに書き込むとか、レジスタの値を 1 つ減らすなど) を実行します。
4. プログラムカウンタの指す場所を、次の命令のある場所にします。

以上の動作が繰り返されて、プログラムの実行が進められます。

次のプログラムは、 100 番地と 101 番地に書かれている 2 つの数を加え、102 番地に書き込む MC6809 コンピューターの機械語プログラムの例です[2]。

```
10001110
00000000
01100100
10100110
```

[2]MC6809 はモトローラ社の 8 ビットマイクロプロセッサです。記憶セルは 65536 個まで持つことができ、1 語は 8 ビットです。

```
10000100
10101011
00000001
10100111
00000010
```

これを、ほんの少しだけ分かり易い、アセンブリ語と呼ばれる言葉で書きなおすと、次のようになります[3]。

```
LDX   #100
LDA   ,X
ADDA  1,X
STA   2,X
```

このプログラムが何を行なっているかを、簡単に説明します。1 行目の `LDX #100` は、レジスタ X に 100 を読み込む命令です。レジスタ X は 16 ビットのレジスタで、アドレスレジスタとして用いられます。機械語 `10001110`, `00000000`, `01100100` が、これを行なう命令に当たります。`10001110` が `LDX #` を、`00000000` と `01100100` が 100 を表し、3 語よりなる機械語命令です。2 行目の `LDA ,X` は、レジスタ X が指している番地の内容をレジスタ A に読み込む命令です。いまレジスタ X の内容は 100 なので、100 番地の記憶セルの内容をレジスタ A に読み込みます。機械語 `10100110` と `10000100` が、これを行なう命令です。3 行目の `ADDA 1,X` は、レジスタ X の値に 1 加えた番地 (レジスタ X の値は 100 なので 101 番地) の記憶セルの内容とレジスタ A の内容を加え、計算結果をレジスタ A に入れる命令です。機械語 `10101011` と `00000001` が、これを行なう命令です。最後の行は、レジスタ X が指している番地に 2 加えた番地 (レジスタ X の値は 100 なので 102 番地) に、レジスタ A の内容を書き込む命令です。機械語 `10100111` と `00000010` が、これを実行する命令です。

1.3 システムプログラム

コンピューターがあってもそれを使うためのプログラム (ソフトウエア) がなければ、運転手のいないタクシーのように、まったく使いものになりません。コンピューターを利用するためのプログラムとして、ワードプロセッサ、表計算ソフトウエア、ゲームなど、いろいろなものが身の周りにはあります。これらはコンピューターの機能を応用したもので、**応用プログラム** (application program) とか、**アプリケーションソフトウエア**と呼ばれます。

便利な道具としての機能を提供する応用プログラムだけでなく、コンピューターの運営やプログラムの開発のために使われるプログラムもあります。そのようなプログラムは、

[3] 人間に分かりやすいように、機械語を記号にして表わしたしたものがアセンブリ語です。そのため、コンピューターごとにアセンブリ語は違っています。

システムプログラム (system program) と呼ばれています。以下では、プログラミング学習やプログラム開発で必要となる、オペレーティングシステム、言語処理系、エディタについて説明します。

1.3.1 オペレーティングシステム

オペレーティングシステム (operating system) は、入出力機器など、コンピューターシステムすべての装置を管理して、円滑なコンピューターの運営をするためのシステムプログラムです。

たとえば、ワードプロセッサを使う場合を考えてみましょう。オペレーティングシステムのあるコンピューターなら、「ワープロ起動」をキーボードやマウスで指示すれば、ディスクに書かれてあるワードプロセッサのプログラムを自動的に読み込み、起動してくれます。もしこれがオペレーティングシステムがないコンピューターだったらどうでしょうか? まず、キーボードやマウスを制御するプログラムさえないので、キーボードやマウスで「ワープロ起動」の指示すらできません。ディスクの制御や管理するプログラムもないので、プログラムも読み込みもできません。

このように、オペレーティングシステムは縁の下の力持ち的な存在で、コンピューターの利用には必要なものです。

入出力機器の制御の他にも、オペレーティングシステムはいろいろな仕事をします。

- **中央処理装置の管理**
  プログラムを入力するのにどんなに早くキーを叩いても、コンピューターは人間よりも圧倒的に処理速度が早いので、ほとんどの時間は何もせずに遊んでいることになります。ですので、高価なコンピューターをひとりだけで使うのはもったい話です。

  1 台のコンピューターを複数の人が同時に使ったり、一人がいくつものプログラムを同時に実行ができるようにする、**マルチタスクキング** (multitasking) (**多重プログラミング** (multiprogramming)) という技術があります。マルチタスクキングとは、実行するプログラムを 0.01 秒程度の間隔で切替え、見かけ上いくつものプログラムが同時に実行できるようにしたものです。オペレーティングシステムは、中央処理装置の使用の効率が良くなるように、どのプログラムをどういう順に実行するを決めます。

  オペレーティングシステムも、中央処理装置によって実行されるプログラムです。そのため、オペレーティングシステムは自分自身を管理する、といえます。

- **主記憶の管理**
  1 台のコンピューターでいくつものプログラムが同時に走らせるときは、主記憶を適切な区域に分けて、それぞれのプログラムに割り当てる必要があります。書きかけの文章を保持するなど、プログラムの実行には主記憶が必要ですし、プログラム自体

も主記憶に置かれます。もし主記憶が足らなくなれば、新たなプログラムを実行はできません。そのため、主記憶の管理も重要な仕事です。

- **保護**
  オペレーティングシステムは、誤りや故意によるデータの破壊や漏洩から保護します。パーソナルコンピューターはともかく、複数の人が使用するコンピューターでは、ひとつのディスクにいろいろな人の文書などが保持されています。しかし、機密文書や試験問題などは、他の人には知られては困ります。

  オペレーティングシステムは、文書などのデータの所有者を覚えておくと同時に、そのデータを読むことのできる人なども覚えておきます。利用者がデータを読んだり書いたりするとき、必ずオペレーティングシステムは正当な利用者かどうかを調べます。正当な利用者には許可をし、正当でない利用者には許可をしません。

  オペレーティングシステムは、ディスク上のデータだけでなく主記憶上のデータやプログラムも、保護の対象としています。

代表的なオペレーティングシステムには、次のものがあります。

- **Unix**
  マルチタスキングにより、同時に複数の利用者がコンピューターを利用でき、複数のプログラムを同時に実行できるオペレーティングシステムです。1960 年代後半に AT&T (American Telephone & Telegram) のベル研究所で開発され、大学や研究機関を中心に、プログラム開発の基盤として発展しました。個人用のコンピューターからスーパーコンピューターにいたるまで、広範囲に使われています。

- **MS-DOS**
  MS-DOS は Microsoft 社によって開発された、個人でのコンピューター利用を目的としたオペレーティングシステムです。このオペレーティングシステムでは、いちどに 1 人しかコンピューターを使うことができません。個人での利用を想定しているので、保護の概念はほとんどありません。

オペレーティングシステムは、コンピューターと一体になっているわけではありません。コンピューターによっては、数種類のオペレーティングシステムから好みのものを選んで走らせることもできます。さらに進んだシステムでは、ひとつのコンピューターで、いくつものオペレーティングシステムを同時に走らせるものもあります。

1.3.2 プログラミング言語と言語処理系

コンピューターが直接実行できるのは機械語だけですが、0 と 1 の羅列なのでプログラムを書くのは大変です。(たった 1 箇所で 0 と 1 を間違えるだけでも、全然違う動作をしてしまいます!) アセンブリ語を使っても、相当大変な作業です。

```
SUM: LDD   #100
     PSHS  D
     LDD   #0
     PSHS  D
L:   LDD   0,S
     ADDD  2,S
     STD   0,S
     LDD   2,S
     SUBD  #1
     STD   2,S
     BNE   L
     LDD   0,S
     LEAS  4,S
     RTS
```

図 1.5: MC6809 でのプログラム

そこで、人間にとって分かりやすい形で、コンピューターの動作を記述することが考え出されました。動作を記述のための「言葉」は**プログラミング言語** (programming language) と呼ばれていて、多くの種類があります。残念ながら日本語や英語ではコンピューターのプログラムは書けません。そのため、わたしたちが努力をしてプログラミング言語を学ばないといけません。

人間にとって分かりやすいプログラミング言語は**高級言語** (high level language) と呼ばれています。本書で学ぶ Scheme もその 1 つで、他に C, Lisp, Pascal, Fortran, Ada など、いろいろあります。

次の例を見ると、高級言語の利点が良く分かると思います。1 から 100 までの総和を計算するプログラムを作ってみましょう。図 1.5は、 MC6809 マイクロプロセッサの機械語 (正確には MC6809 のアセンブリ語) で書いたものです。図 1.3.2(a), (b) はそれぞれ、Scheme, Pascal で書いたものです。

図 1.5のプログラムは、ちょっと見ただでは何をしようとしているプログラムなのか、さっぱり分かりません。(このように、ちょっと見ただけでは何をしているかが分からないプログラムは、「可読性が低い」 といわれます。) ですが、高級言語で書かれたプログラムは、その言語を知らなくても、さっと眺めればそのプログラム概要は読みとることができます。(このような場合、「可読性が高い」といいます。)

以上のようなことから、機械語やアセンブリ語でプログラムを書くことは、最近ではほとんどありません。オペレーティングシステムの一部や高速性が要求されるプログラムの一部くらいです。

当然ですが、高級言語で書かれたプログラムは機械語ではありません。そのため、作っ

```
(define (sum n)
  (if (= i 0)
      0
      (+ n (sum (- n 1)))))
(sum 100)
```

(a) Scheme でのプログラム

```
PROGRAM Sum(input, output);
  VAR  thesum, i : INTEGER;
BEGIN
  thesum := 0;
  FOR i := 1 TO 100 DO
    thesum := thesum + i;
  WRITELN(thesum)
END.
```

(b) Pascal でのプログラム

たプログラムをそのまま実行はできません。高級言語で書かれたプログラムをコンピューターで実行可能にするために、**言語処理系** (language processor) と呼ばれるシステムプログラムを使う必要があります。

高級言語はたいてい、使うコンピューターやオペレーティングシステムに依存しないように、言語が決められています。というのも、言語処理系をコンピューターごとに用意しておけば、同一のプログラムを変更せずにいろいろなコンピューターで実行できるからです。いいかえれば、言語処理系がコンピューターやオペレーティングシステムの違いを吸収しています。高級言語を使ったプログラム作成では、コンピューターの機種を意識しなくて済みます[4]。

たとえば、NEC 社 PC-9801 用の家計簿プログラムを機械語で自作し、何年もずっと使い続けているとしましょう。ですがある日コンピューターが壊れてしまいました。これを機会に、Apple 社 Macintosh 最新機種に買い替えたいのですが、Macintosh ではその家計簿プログラムは動きません。というのも、Macintosh と PC-9801 では、オペレーティングシステムも機械語も違うからです。もし Macintosh を買えば、家計簿プログラムをゼロから作らないといけません。もしいままでの家計簿プログラムをそのまま使いたければ、PC-9801 を買わないといけません。ですが、家計簿プログラムを高級言語で書いておけば、

[4]ある程度複雑なプログラムでは、使っているコンピューターやオペレーティングシステムに依存する場合もあります。そのような場合でも、高級言語を使うことで、わずかな変更で別のコンピューターで利用可能にできます。(わずかな変更だけで、数種類の CPU と数種類のオペレーティングシステムで動作する、C 言語で書かれたプログラムが実際にあります。)

比較的簡単に別の機種でも動かすようにできます。ですので、安心してコンピューターの買い替えできます[5]。

言語処理系は**コンパイラ** (compiler) によるものと**インタプリタ** (interpreter) によるものとに分類されます。

- **コンパイラ**
  高級言語で書かれたプログラムを、機械語に翻訳 (コンパイル) する言語処理系です。翻訳されたプログラムは、ひとつの独立した機械語プログラムになります。機械語に翻訳されたプログラムは、コンピューターによって直接実行されます。そのために、翻訳されたプログラムの実行のときには、コンパイラは必要ありません。

- **インタプリタ**
  高級言語で書かれたプログラムを、そのつど解釈しながらプログラムの実行をすすめてゆく言語処理系です。インタプリタは高級言語で書かれたプログラムを読んで、どういう動作をすべきかを解釈し、それにしたがった動作をします。プログラムの実行のときに、機械語プログラムとして直接コンピューターに実行されるのはインタプリタです。(利用者が実行していると思っている、高級言語でのプログラムではありません。) そのために、高級言語で書かれたプログラムそのものが実行されているように見えます。

コンパイラとインタプリタには、それぞれ長所と短所があります。コンパイラを使うと、実行するプログラムをは機械語に変換されているために、プログラムの実行速度が速いという長所があります。しかしプログラムの修正をするたびに翻訳をしないといけないので、プログラムを修正してから実行可能になるまでに時間がかかります。一方インタプリタは翻訳作業は必要なく、プログラムの修正をしたらすぐに実行できます。しかしプログラムの一文ごとに解釈しながら実行をするので、プログラムの実行が遅いという欠点があります。プログラムの開発のときはインタプリタを使い、完成したらコンパイラで翻訳して効率のよい実行を得るといった、それぞれの長所を利用する手法も使われます。

1.3.3 エディタ

プログラムを書くには、**エディタ** (editor) と呼ばれるシステムプログラム使います。エディタはプログラム作成が主な目的としています。ワードプロセッサはエディタの仲間ですが、文章の作成を目的としたものです。

プログラムは、ディスク装置に**ファイル** (file) として記録します。ファイルはいくつも作ることができ、それぞれに名前が付けられます。ファイルの名前は**ファイル名**といいます。

[5] まったく違う新しいコンピューターを開発することは、現在ではほとんどありません。処理速度だけを向上した新しいコンピューターの開発がよくされます。これは、いまあるプログラムをそのまま利用でき、しかもより速い計算能力を手に入れることができるからです。

あるファイルを読み書きするときはそのファイル名を指定することで、たくさんのファイルうちのどれを読み書きするかを決めます。

エディタは、ファイルの内容を変更したり、新しくファイルを作る仕事をします。エディタはファイルの内容を画面に表示します。利用者はファイルの内容を見て、どこをどう変更するかなどの編集作業をを決め、エディタに編集命令を指示します。文字の入力とか削除は、基本的な編集命令です。すると、エディタはその指示にしたがってファイルの内容を変更し、変更された結果を画面に表示します。利用者はそれを見て、次の編集作業をします。必要な編集作業が終ればファイルをディスクに書き戻し、エディタを終了します。(4.1では、エディタについてもう少し詳しく説明してあります。)

エディタはプログラムの作成を目的としているので、プログラム作成を支援する機能をいろいろ持っています。たとえば、エディタのなかには次のような機能を持ったものもあります。

- **コンパイラの呼び出し**
  コンパイラを呼び出して、プログラムをコンパイルする機能です。もしプログラムに間違いがあれば、プログラムの間違い部分を表示します。

- **括弧の点滅**
  閉じ括弧を入力すると、それに対応した開き括弧を点滅させて、括弧の対応関係を知らせてくれる機能です。括弧の対応の間違えをコンパイラは検出してくれますが、そのような誤りをプログラムの入力の段階で排除できます。

- **綴りの自動入力**
  プログラムのなかでひんぱんに使われる語を、自動的に入力する機能です。よく使う語のタイプミスを排除できます。

- **自動字下げ**
  プログラムの構造を、一目で分かるようにしたのが字下げです。自動字下げをしたときに意図しない字下げになると、入力ミスでプログラムの構造が変になっていることに気がつきます。

- **キーボード マクロ**
  一連のキー入力をエディタが覚えておき、覚えておいた一連のキー入力を何度でも実行できるようにしたものです。決まりきった一連の編集作業を、何度も繰り返すときに便利です。

- **機能の拡張**
  基本的な編集命令を組み合わせて、新しい編集命令を定義する機能です。エディタが用意している専用のプログラミング言語を使い、新しい編集命令を「プログラミン

グ」します。新しい編集命令は、あらかじめ用意されている編集命令と同じように使うことができます。

- **オンラインマニュアル**
  エディタの使い方のほか言語処理系の使い方などのマニュアルを、エディタ上で参照できるようにしたものです。

プログラムを書く人にとって、エディタは使用時間のもっとも多い道具といえるでしょう。そのために、エディタを十分マスターしているかどうかは、プログラム作成の効率に大きく関わることです。がんばってエディタの使い方をマスターしましょう。

1.4 ソフトウエア開発

以上では、エディタでプログラムを書き、実行のために言語処理系を使用することを学びました。簡単なプログラムの作成の場合はこれでいいのですが、商用の実用的なソフトウエアの作成にはこれらの概念だけでは不十分です。

実用的なソフトウエアは大きなもので、一人の手で作成されることはなく、多くの人の共同作業によって作られます。趣味でプログラムを作る場合は利用者と開発者は同じ人ですが、商用のソフトウエアの場合では利用者と開発者が違います。そのために開発者は、利用者が必要としているものは何かとか、どれだけの性能が必要なのか、などを十分に調べ、ソフトウエアを作る必要があります。さらに、出来上がったソフトウエアを使用する際には、利用者用のマニュアルを作りますし、さらには利用者を集めて講習会を開くこともあります。

ソフトウエアの開発過程のモデル化の一つである**ウォーターフォールモデル** (waterfall model) では、次の順番でソフトウエアの開発を進めてゆきます。それぞれの段階で不十分と分かれば前の段階に戻り、必要な事項を変更や追加をし、次の段階に進んでゆきます。

1. **要求分析**
   どのような機能と性能が必要となっているかを調べます。ソフトウエアの依頼者にも、開発に必要となる明確な形で要求が表せないことがあります。開発者は依頼者の要求を聞き、要求を明確にしてゆきます。納期や開発費用などを考えながら、実現可能かを検討します。

2. **プログラム設計**
   作成するプログラムの構造を、要求分析に従って決定します。ひとつのシステムをサブシステムやモジュールといった、より細かな単位に分解します。また、それらがどのような機能を持つかとか、互いにどのように関連するかなどを決定します。

3. **コーディング**
   プログラム設計で決定された機能や動作を実現する、サブシステムやモジュールを、プログラム言語の形で書き下します。

4. **テスト**
   コーディングで作られたプログラムが、設計通りの動作をするかどうか検査をします。もし誤りがみつかれば、プログラムを修正をします。利用者の要求を満たせないプログラムが出来上がる場合があります。そのときはプログラムの設計を修正することになります。

5. **運用**
   依頼者にソフトウエアを納入し、運用をします。利用者に操作法を教えたり、トラブル時には対策を施すなど、円滑な利用ができるようにします。納入の後でもプログラムに誤りを修正したり、性能を上げるためにプログラムを改良したりします。

このように、ソフトウエアの開発には、プログラミング以外に部分に非常に多くの作業が必要となります。数人のグループが半年かけて作成する小規模のソフトウエア開発プロジェクトですら、人件費や開発機材等の必要経費を考えると、慎重なプロジェクトの計画と運営が必要となります。

まとめ

以上でコンピューターシステムの基礎を説明しました。本書を理解するには、以上のことでとりあえずは十分です。さらに必要なことは、必要になった時点で説明します。

ですが、本書に出てくるものよりも複雑なプログラムを書くには、コンピューターに関してもう少し詳しく知っておく方がいいと思います。そのようなときのために、参考書をいくつか紹介します。

コンピューター科学入門の教科書

- Alan W. Biermann (和田 英一 監訳), “やさしいコンピューター科学,” アスキー出版局, 1993.

- M. A. Arbib, A. J. Kfoury, R. N. Moll (甘利 俊一, 金谷 健一, 嶋田 晋 訳), “計算機科学入門,” Information&Computing-1, サイエンス社, 1984.

コンピューターの歴史について

- P. A. Kidwell, P. E. Ceruzzi (渡邊 了介 訳), “目で見るディジタル計算の歴史,” ジャストシステム, 1995.

パーソナルコンピューターの歴史とこれからの流れについて

- Bill Gates (西 和彦 訳), “ビル･ゲイツ 未来を語る,” アスキー出版局, 1995.

コンピューターシステムの内部構造やシステムプログラムについて

- 所 真理雄, “計算システム入門,” 岩波講座ソフトウエア科学第 1 巻, 岩波書店, 1988 年.
- Ullman (浦 昭二 他訳), “プログラミングシステムの基礎”, 培風館.
- 前川 守, “オペーレーティングシステム,” 岩波講座ソフトウエア科学第 6 巻, 岩波書店, 1988 年.
- J. L. Peterson, A. Silberschatz (宇津井 孝一, 福田 晃 訳), “オペレーティングシステムの概念 原書第二版,” 培風館, 1987 年.
- G. J. Nutt, (小幡 すぎ子 訳), “集中/分散オペレーティングシステム,” トッパン, 1992 年.
- 佐々 政孝, “プログラミング言語処理系,” 岩波講座ソフトウエア科学第 5 巻, 岩波書店, 1989 年.
- 武市 正人, “プログラミング言語,” 岩波講座ソフトウエア科学第 4 巻, 岩波書店, 1994 年.
- N. Wirth (筧 捷彦 訳), “翻訳系構成法序論,” 近代科学社, 1986 年.

ソフトウエアの開発と、プログラムの誤りに関して

- すずきひろのぶ, かとうみつあき, “The BUG,” オーム社, 1995 年.
- F. P. Brooks Jr. (山内 正彌 訳), “ソフトウエア開発の神話,” 企画センター, 1977.

電氣の仕掛けはよく損じた。近所の蒔田といふ電気器具商の主人が來て修繕した。彼女はその修繕するところに附纏つて、めずらしさうに見てゐるうちに、彼女にいくらかの電気知識が攝り入れられた。

岡本かの子 「老妓抄」
『鮨』収録 昭和十六年 改造社刊

第 2 章

Scheme 処理系の使い方

Scheme の言語処理系の基本的な使い方を、NGSCM を例にして説明します[1]。Scheme では、プログラムもその実行も「式」の形をしています。ここでは Scheme 処理系の起動方法と、簡単な式を入力して Scheme 処理系に実行させる方法を学びます。

Unix オペレーティングシステムを使う場合は、ログインとログアウトが必要になります。Unix は 1 つのコンピューターを複数の利用者が同時に使うことができます。そのために、利用者が誰かをコンピューターに知らせ、利用者を区別しなければいけません。利用者の登録名をコンピューターに知らせることを、**ログイン** (または **ログオン**) といいます。利用の終了をコンピューターに知らせることは、**ログアウト** といいます。

注意: MS-DOS などの単一ユーザーのオペレーティングシステムを使う場合は、ログインとログアウトは不要です。

重要: Unix オペレーティングシステムを使っているコンピューターは、通常 24 時間年中無休で運用している場合が多いです。このため、利用者みずからコンピューターの電源を入れることはまずありません。もし電源が入っていなければ、無断で電源を入れずに、そのコンピューターの管理者の指示を得るようにしましょう。

2.1 ログイン

コンピューターの画面を見ると、次のような表示がされていると思います。

```
login: ▉
```

画面でチカチカ点滅しているものは**カーソル** (cursor) と呼ばれ、キー入力をする場所を示しています。

[1] NGSCM は、テキストエディタと Scheme 処理系を一体化したシステムです。NGSCM の入手法は、Appendix D を参照して下さい。

コンピューターによっては `login:` ではなくて、たとえば `owl login:` というようになっているかもしれません。もし画面に何も表示されていなければ、1,2 回リターンキーを押してみてください。それでも何も表示されない場合は故障の可能性がありますので、そのコンピューターの管理者に知らせて指示を仰いでください。

ではここで、あなたの利用者名 (アカウント名、あるいはログイン名) を入力します。 たとえば利用者名が pooh なら、次のように入力します。

```
login: pooh RET
```

RET はリターンキーを押すことを表しています。 すると、

```
login: pooh
Password: ▉
```

のように、パスワードの入力が求められます。**パスワードは、入力しても画面には表示されません。**

パスワードは、コンピューターを使おうとしている人が正しい利用者であることを証明するものです。名前を名乗るだけなら誰でもできますが、本当にその人であるかどうかを、その人が合い言葉を知っているかどうかで確認します。これはキャッシュカードを拾っても、その暗証番号も分からないことにはお金を引き出せないことに似ています。アリババのように合い言葉が人にばれないよう、パスワードは人に知られないようにしましょう。

パスワードの入力が終ると、リターンキーを押します。ログインが成功すると、そのことを示すメッセージが画面に表示されるでしょう。[2]

```
Last login: Thu Mar 16 20:32:25 from gull
SunOS Release 4.1.2-JLE1.1.2 (GENERIC) #3: Fri Mar 5 15:34:42 JST 1993

% ▉
```

カーソルの前に表示されている % は**プロンプト** (prompt) と呼ばれ、いまその場所で入力を受け付けていますよ、という印です。あなたの使っているシステムでは % の代わりに、> あるいは他の文字がプロンプトに使われているかもしれません。

もし間違ったパスワードを入力をしてしまうと、ログインに失敗します。このときは次のようなメッセージが出るでしょう。

```
login: pooh
Password:
Login incorrect
login: ▉
```

失敗したときは、利用者名の入力からやり直してください。

[2] この例は Sun マイクロシステムズ社の Unix ワークステーション SparcStation ELC で、オペレーティングシステムが SunOS 4.1.2 JLE 1.1.2 の場合です。

2.2 Scheme 処理系 (NGSCM) の起動と終了

Scheme 処理系の起動方法を説明します。ログインが終了したので、コンピューターを利用できるようになりました。ここでは NGSCM の起動方法を説明します。

次のように入力して下さい。

% ngscm RET

すると、図 2.1のような画面になるでしょう。

```
NGSCM Version 3.3.2 of 01/20/96 on owl (FreeBSD2.0.5-950622-SNAP)
Copyright (C) 1992,1993,1994,1995,1996 Hirotsugu Kakugawa.
NGSCM is based on Ng editor by Shigeki Yoshida et al.
Type C-h for help. (‘C-’ means use CTRL key.)

NGSCM comes with ABSOLUTELY NO WARRANTY.
You may give out copies of NGSCM; type C-h C-c to see the conditions
Type C-h t for a tutorial on using NGSCM.
Type C-h T for a Japanese tutorial on using NGSCM.
Type C-j or C-c C-e to evaluate the expression before point.
Type C-g to abort evaluation.

SCM version 4e1, Copyright (C) 1990, 1991, 1992, 1993, 1994 Aubrey Jaffer.
SCM comes with ABSOLUTELY NO WARRANTY; for details type ‘(terms)’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘(terms)’ for details.
;loading "/usr/local/lib/scm/Transcen.scm"
;done loading "/usr/local/lib/scm/Transcen.scm"
;Evaluation took 119 mSec (0 in gc) 9993 cells work, 12800 bytes other
> 

--**-NGSCM: *scheme*                    (-EE:fundamental-Scheme_Interaction)--
```

図 2.1: ngscm の起動時の画面

これで NGSCM が利用できるようになりました。(もしかしたら、あなたか実際に使用しているシステムと上の図とは少し違うかもしれませんが、あまり気にしなくても結構です。)

使い方の説明の前に、終了方法を説明しておきます。 NGSCM の終了には C-x C-c を入力します。ここで C-x C-c は、まず最初にコントロールキーを押しながら x を押し、次にコントロールキーを押しながら c を入力することを表しています。すなわち C-とは、「コントロールキーを押しながら」、ということを表しています。

C-x C-c を入力しても終了できない場合があるかもしれません。このときの原因として、以下のものが考えられます。

1. C-x C-c を入力する前に、 ESC キーなどが押されていたとき:
   1,2 度 C-g を入力してから、C-x C-c を入力してみてください。

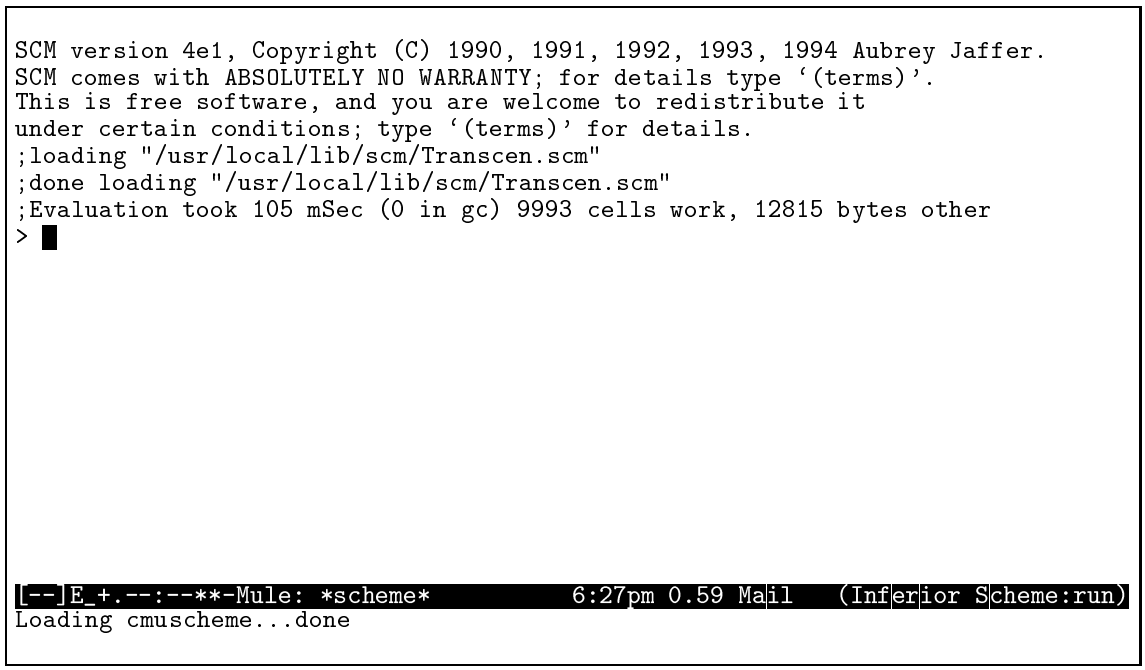


図 2.2: Mule での Scheme 処理系の実行

2. 書換えたファイルを書き込まずに、終了しようとしているとき:
   画面の一番下の行で、

   ```
   Save file /home/pooh/honey-jar/lunch.scm? (y or n)
   ```

   となっていれば、この場合です。 y を押せば終了できます。

詳しい説明をすると長くなりますので、とりあえず現時点ではそのようにして使ってください。詳しい説明は後の章で説明します。

2.3 その他の Scheme 処理系の起動と終了

2.3.1 Mule の場合

まず Mule を起動します。

% mule RET

次に処理系を起動します。Mule で ESC x run-scheme RET と入力して下さい。すると、図 2.2のような画面になります。(Mule から呼び出す Scheme 処理系を SCM に設定した場合です。他の処理系を呼び出すようにしてあるときは、違う表示になります。)

Mule での Scheme 処理系の使い方は、 NGSCM に対する場合とほぼ同じですが、式の入力の終りには RET を押します。Mule を終了する方法は NGSCM と同じで、C-x C-c を入力します。

Scheme 処理系の起動でエラーが起きたときは、設定が正しくできていない可能性があります。あなたの .emacs ファイルに次のものを書き加えて、もう一度試して下さい。

```
(autoload 'run-scheme "cmuscheme"
    "Run an inferior Scheme process." t)
(setq scheme-program-name "scm")
```

(使う Scheme 処理系が SCM 以外の場合は、"scm" の代わりにその処理系のプログラム名を指定します。) これでもうまくゆかないときは、コンピューターの管理者に相談して解決して下さい。

2.3.2 その他の場合

上で説明した処理系の以外の場合は、使用する Scheme 処理系のコマンド名を入力して、リターンキーを押します。(これは Scheme 処理系 SCM の場合です。)

```
% scm RET
```

Scheme 処理系 SCM, MIT Scheme, ELK, では、C-d の入力で処理系の終了ができます。そのほかの場合でも、たぶん C-d の入力で処理系の終了ができるとおもいます。正確なことは、使用する処理系のマニュアルを読んで下さい。

2.4 Scheme 処理系の使い方の基礎

Scheme システムでは、

1. プログラムを入力する
2. それを実行する
3. 実行結果を表示する

の 3 つのステップが繰り返されています。図 2.1でカーソルのある行をみると、

```
> 
```

となっています。この表示での > は、 Scheme 処理系のプロンプトです。ここで Scheme プログラムを入力すると、その結果が計算され、画面に表示されます。

Scheme のプログラムは、すべて式を単位にして記述されます。プログラムの実行も、式の形で処理系に与えます。

2.4.1 手入力による実行

例として 30 + 26 を計算する式を実行してみましょう。Scheme でこれを表す式は (+ 30 26) となります。

Scheme でプログラムを実行するときは、最初に開き括弧 (を書きます。つぎに、実行するプログラムの名前を書きます。プログラムの名前の後には、そのプログラムに与えるデータを書きます。そして最後に、閉じ括弧) を書きます。

今の例ですと、プログラム名は + (足し算をするプログラム) で、20 と 26 がプログラムに与えるデータ (それぞれ数の 20 と 26) です。

では、(+ 30 26) を入力してください。もしタイプミスをしたときは、DEL キーで文字の消去できます。入力が終れば、C-j (コントロールキーを押しながら j を押します) を入力してください。なお C-j は、入力した式の実行を始めなさい、という合図です。(Scheme 処理系によっては、DEL の代わりに BS または ⇐ を使うものもあります。また、C-j の代わりに RET を使うものもあります。)

```
> (+ 30 26)C-j
;Evaluation took 0 mSec (0 in gc) 0 cons work
56
```

表示について説明します。あなたの入力した部分が (+ 30 26) で、その次の行に表示されている *;Evaluation took 0 mSec (0 in gc) 0 cons work* は、その計算 (式の実行) にかかった時間などを表しています。その次の行の表示 *56* (= 30 + 26) が実行結果です。

実行結果の表示の次の行には、再びプロンプト > が表示されます:

```
> (+ 30 26)
;Evaluation took 0 mSec (0 in gc) 0 cons work
56
> 
```

さらに続けて別の式の実行ができます。次の例では、4 × 5 の計算をする式を実行させています。

```
> (* 4 5) C-j
;Evaluation took 0 mSec (0 in gc) 0 cons work
20
> 
```

上の例では、足し算やかけ算をするプログラムに数そのものを与えただけでした。数の代わりに別の式を書くこともできます。

```
> (+ 3 (* 2 4)) C-j
;Evaluation took 0 mSec (0 in gc) 0 cons work
11
```

この例は、$3+(2\times4)$ を計算する式です。まず最初に (* 2 4) が計算されて 8 となり、この結果が置き換えられます。その次に $3+8$ が計算され、結果 11 を得ます。

以上により、簡単な式の計算を Scheme 処理系を使ってできるようになりました。

2.4.2 プログラムファイルの読み込み

プログラムが長いと、そのつど手で入力するのは大変です。プログラムをファイルに書いておき、それを Scheme 処理系に読み込ませることもできます。(なお、ファイルにプログラムを書く方法は第 7章で学びます。)

プログラムファイルを読み込ませることを**ロード** (load) といいます。その方法は、次のように proc を使います。

```
> (load "sum.scm") C-j
;loading "sum.scm"
;done loading "sum.scm"
;Evaluation took 7 mSec (0 in gc) 77 cells work, 87 bytes other
#<unspecified>
>
```

ここで "sum.scm" は、プログラムファイルのファイル名です。

2.5 ログアウト

今度はコンピューターの使用を終了する方法を説明します。このことを **ログアウト** (logout) といいます。ログアウトをするには、次のように入力してください。

```
% logout RET
```

すると画面には、ログイン前の表示

```
login:
```

になるはずです。これで計算機の使用が終了し、すべてが済んだことになります。

注意: Scheme 処理系を終了した後、ログアウトするのを忘れないようにしましょう。

練習問題

下にいくつかの Scheme の式があります。それぞれ実行すると、どのような計算結果になるか考えなさい。そしてそれが正しいか、実行して確かめなさい。

1. `(+ 1 2 3)`
2. `(* 4 6)`
3. `(- 10 2)`
4. `(+ (* 2 3) 4)`
5. `(* (+ 1 2) (+ 3 4))`
6. `(/ 100 (+ 1 1))`

「實は僕たちも、あなた一人をあてにして、さつきからここでお待ちしてゐたのです。
きっとあなたも招待されてゐると思ひましたから。」
「いや、そんなにあてにされると僕も少し困るのだが。」
私たち三人は、力無く笑つた。

太宰治 「不審庵」
『黄村先生言行録』収録 昭和二十二年 日本出版株式会社刊

第 3 章

Scheme 入門 (初級編)

本章ではプログラミング言語 Scheme の文法の基礎を学習します。出てきた例を Scheme 処理系で実際に実行し、動作を確認するようにしてください。例をいろいろ変更して試してみると、より深く身につきます。

Scheme プログラムには括弧がたくさんあるので、戸惑うかもしれません。ですが Scheme を学んでプログラムを書いてゆくうちに、括弧に対するアレルギーが取れてゆきます。(NGSCM や Mule のように、閉じ括弧を入力すると、対応する開き括弧が点滅する機能を持つテキストエディタを使えば、括弧を恐れることはありません。)

3.1 入力-評価-表示ループと式

Scheme 処理系が起動されると、次のようにプロンプトが出ています。(Scheme 処理系によっては、プロンプトが違う場合があります。)

```
> ▉
```

ここで 2001 を入力し、最後に C-j (コントロールキーを押しながら、j を押す) を入力します。(別の Scheme 処理系の場合は、 RET キーの場合があります。)

```
> 2001 C-j
```

すると、 次のような表示になります。

```
> 2001
;Evaluation took 33 mSec (0 in gc) 45 cons work
2001
> ▉
```

ここで *2001* のように斜体で印刷されている部分が、Scheme 処理系のの出力 (Scheme の計算結果) です。

```
;Evaluation took 33 mSec (0 in gc) 45 cons work
```

の行は NGSCM 特有の表示で、計算に要した時間などを表しています。(別の Scheme 処理系ではこの行は表示されなかったり、別のものが表示されているかもしれません。) 以降ではこの行は省略します。

注意: 実際のシステムでは上記のように斜体で表示はされません。利用者の入力と Scheme の出力とが区別がつくように、本書では Scheme の出力を斜体で書いています。また、これまでは利用者の入力する部分には下線を引き、最後に C-j を書いていましたが、これ以降はそれらを省略します。

こんどは、$3+5$ の計算をしてみましょう。Scheme への入力は、(+ 3 5) です。前の章でも説明しましたが、Scheme では $3+5$ を表すのに (+ 3 5) と、呼び出す実行する手続き名 (プログラム名) + を書き、手続きに与える引数(ひきすう)(パラメーターとも呼ばれます) 3 5 を書き、それら全体を括弧でくくる、という書き方をします。なお、+ は足し算を計算する手続きの名前です。

```
> (+ 3 5)
8
```

これが実行され、計算結果 8 が表示されています。

Scheme 処理系は、

1. 入力を受けとり
2. それを計算し
3. その結果を表示する

という動作を繰り返しています。入力は**式** (expression) が単位となっています。たとえば 2001 は (数ですがそれ自体で) 式ですし、(+ 3 5) も式です。

こんどは足し算とかけ算を組み合わせた例を見てみましょう。$3\times4+2\times9$ は、次のようにすれば計算できます。(* はかけ算を計算する手続きです。)

```
> (+ (* 3 4) (* 2 9))
30
```

なぜ式 (+ (* 3 4) (* 2 9)) が $3\times4+2\times9$ を計算になっているのかを、以前にも説明しましたが、もういちど説明します。まず (* 3 4) と (* 2 9) は、それぞれ 3×4 と 2×9 を計算する式です。最初にこれらの式が計算され、その結果はそれぞれ 12 と 18 です。次に、これらを置きかえて (+ 12 18) の計算をします。最後にこの計算がされて、計算結果 30 を得ます。

もう少し複雑な例を見てみましょう。

```
> (+ (* (- 5 4) (- 3 1)) 10)
12
```

これは次の過程で計算されています。

```
(+ (* (- 5 4) (- 3 1)) 10)
  = (+ (* 1 2) 10)
  = (+ 2 10)
  = 12
```

これまでの例で想像がつくと思いますが、Scheme は次の規則によって式の計算をします。(Scheme の世界では、「計算」という言葉よりも「**評価** (evaluation)」という言葉を使っていますので、これ以降は「評価」という言葉を使うことにします。)

- データが基本的なデータ (数など) のとき。
  評価結果はそのデータそのものです。たとえば 10 を評価すると、その結果も 10 です。(この後では変数について学びます。データが変数のときは、変数値が評価結果となります。)

- データが $(a_0\ a_1\ a_2\ \cdots\ a_n)$ の形をしているとき。
  まず a_0,...,a_nを評価し、その結果が b_0,...,b_n であるとします。$b_1, ..., b_n$ を引数として、手続き b_0 を呼び出します。式の評価結果は、手続き b_0 が返す値です[1]。

3.2 変数

これまでに学んだことを使えば、簡単な計算ができるようになりました。でもこれだけでは、安電卓と同じ程度のことしかできません。ここで学ぶ**変数** (variable) を使うことで計算結果を保持し、その値をあとで利用することができます。(これでメモリー付きの電卓と同じことができるようになります。)

変数とその値を定義するには、define を使います。

◇ (define ⟨ 変数 ⟩ ⟨ 式 ⟩)
— 変数を定義します。⟨ 変数 ⟩ の値は ⟨ 式 ⟩ を評価した結果となります。

たとえば、250000 を値とする変数 income を定義するには、次のようにします。

[1] a_0 自身も評価することが不思議に思えるかもしれません。Scheme では + も式のひとつで、その値は「足し算をする手続き」となっています。+ を評価して「足し算をする手続き」を得て、そしてその手続きにデータを渡します。それによって、足し算がおこなわれるのです。

```
> (define income 250000)      — 変数の定義
#<unspecified>
> income                      — 変数値の参照
250000
> (+ income 10000)            — 変数値を使った足し算
260000
```

最初の行で #<unspecified> という表示がされていますが、気にしなくてかまいません[2]。

以前に説明した式の評価の方法では、引数の部分を先に評価すると説明しました。ですが define の場合は少し特殊です。最初の引数 (上の例だと income) は評価しないようになっています。次の例を見れば、このことがよく分かります。

```
> (define expense (+ 10000 46000))
#<unspecified>
> expense
56000
```

式 (define expense (+ 10000 46000)) の評価をするのに、もしすべての引数が評価されたらどうなるか、考えてみましょう。まず expense と (+ 10000 46000) が評価されますが、この時点ではまだ expense は値をもっていないので、エラーとなってしまいます。もし expense の値を持っていて、10000 だった場合を考えてみましょう。そると引数が評価された後に、式 (define 10000 56000) の評価をすることになります。すると define は 10000 の値を 56000 にするという、意味不明なことになってしまいます。

define は第一引数を評価せず、第二引数だけを評価します。Scheme には、手続き呼び出しのような形をしていても、引数を評価しないものがいくつかあります。このように引数をすべて評価せず、手続き呼び出しとは違ったことをするものを**特殊形式** (special form) と呼び、手続き呼び出しと区別しています。

次は変数の値を書き換える方法 (代入) について説明します。代入のために、特殊形式 set! が用意されています。

◇ (set! 〈変数〉〈式〉)
— 〈変数〉の値を、〈式〉の評価結果に置き換えます。

注意しないといけないことは、**すでに define 定義されている変数に対してしか set! は使えない、**ということです。

では例を見てみましょう。最初に変数 tax が未定義であることを確認します。

[2] Scheme では、式を読み込み、評価し、それの結果を表示する、といいました。(define income 250000) もひとつの式ですが、define の返す値が NGSCM の場合には #<unspecified> となっています。もし他の Scheme 処理系を使った場合は、別の値 (たとえば、変数名だったり保持される値だったりします。)

```
> tax
Error: unbound variable:  tax      — tax は未定義
```

次に未定義変数である tax に対して set! を使って、値を与えてみます。

```
> (set! tax 15000)
Error: unbound variable:  tax
```

... が、エラーになりました。では define によって変数を定義した後に、set! で変数値を変更してみましょう。

```
> (define tax 10000)
#<unspecified>
> tax
10000
> (set! tax 15000)
#<unspecified>
> tax
15000
```

こんどはうまくゆきました。

重要: 変数の定義のとき

```
(define 'age 17)
```

のように、定義しようとする変数名に引用符をつけるのは間違いです。'age は (quote age) の省略系なので、

```
(define (quote age) 17)
```

となります。これは後に説明する手続きの定義の形をしていて、quote という手続きを定義する形になっています。それ以降で 'age のような引用符を使おうとしても、手続き quote の呼び出しとなってしまい、おかしな動作となってしまいます。(quote を別の意味に変えようとすると、エラーとする Scheme 処理系もあります。)

3.3 記号

Scheme は数だけでなく、記号やそれらをつなげてリストにしたデータも取り扱うことができます。変数には、数だけでなく**記号** (symbol) も値として持たせることができます。

```
> (define princess 'masako)
#<unspecified>
> princess
masako
```

こんどは、記号の**リスト** (list) を変数に保持してみましょう。

```
> (define friends '(yumiko mayuko hiroko noriko))
#<unspecified>
> friends
(yumiko mayuko hiroko noriko)
```

上の 2 つの例で、データの前に ' がついていることに注意してください。これは**引用符** (quote) と呼ばれ、その直後の式を評価しないようにするための特別な記法です。式 s に対して 's は、(quote s) の省略記法です。quote は特殊形式で、第一引数を評価せずに第一引数をそのまま値として返します。

次の例を見れば、この引用符が重要なはたらきをすることがわかると思います。

```
> (define masako 'owada)          — 引用符がある
#<unspecified>
> masako
owada
> (define princess masako)        — 引用符がない
#<unspecified>
> princess
owada
```

どうして変数 princess の値が owada になったかを説明します。式 (define princess masako) を評価するとき、まず masako が評価され、その値は owada です。これが変数 princess の値として定義されたのです。(もし引用符をつけていれば masako が評価されないので、princess の値は masako になります。)

上の例にでてきた変数 friends の定義で、もし引用符を付けなかったらどうなるでしょうか? いままでの例に習って考えると、まず式 (yumiko mayuko hiroko noriko) を評価することになります。ですが、これは手続き yumiko を、引数 mayuko hiroko noriko で呼び出すことになります。ですが、この手続き呼び出しをするときの引数の評価で、mayuko hiroko noriko は未定義な変数なのでエラーとなりますし、手続き yumiko も未定義なので、未定義な手続きの呼び出しとなってエラーとなるでしょう。このような理由で、引用符が重要なはたらきをすることが分ると思います。

重要: ' や , は特別の目的のために使われるので、これらを記号の一部として使わないようにして下さい。(詳細は 10.3を参照して下さい。)

ここまでの一連の例での princess や friends は、変数として使われていました。これらは**記号** (symbol) と呼ばれるデータ型の Scheme データです。記号には、次のようなものも使えます。

```
lambda                  FOOBAR                  ZAP!!
number->string          Okey?                   Mar21
aCb3Wdj                 $5                      *the-state*
this-identifyer-is-very-long-but-its-ok
```

記号はアルファベット文字や数字あるいは拡張アルファベット文字 (+ - . * / < = > ! ? : $ %など) を連ねたものです。(ですが、最初の文字が数字だったらいけません[3] 。

与えられたデータが記号であるかを判定するには、手続き symbol?を使います。

◇ (symbol? ⟨変数⟩)
— ⟨変数⟩の値が記号なら真 #t を、そうでなければ偽 #f を返します。

真理値については後に詳しく説明しますが、#t の時は yes を、#f のときは no を表しています。

例をみてみましょう。

```
> (symbol? 'mayuko)
#t
> (symbol? 10)
#f
```

3.4 リストと対

リスト (list) とは、(yumiko mayuko hiroko noriko) のように、括弧で囲まれたいくつのデータ (記号のみとは限りません) のならびをいいます。リストの構成要素がまたリストであっても構いません。リストの例を以下に示します:

```
()
(a)
(NISHIDA HIKARU)
(1 a 2 b)
((pi 3.1415) (e 2.71828))
(a (ba bb) (ca (cba cbb) cc) d)
```

[3] この規則に従うと -A も記号となります。

最初の行の () は**空リスト** (empty list) という、「何もないリスト」を表すデータです。空リストは、リストの終りを表すのにも使います。空リストをプログラム中に書くときは、引用符を忘れずに '() と書いて下さい。空リストを画面に表示すれば () となるだけですが、プログラム入力のときは '() とします[4]。

与えられたデータが空リストかどうかを判定するには、手続き null?を使います。

◇ (null? 〈変数〉)
— 〈変数〉の値が空リストなら真 #t を、そうでなければ偽 #f を返します。

```
> (null? '(yumiko mayuko))
#f
> (null? '(yumiko))
#f
> (null? '())
#t
> (null? 'miho)
#f
```

リストの長さには制限はなく、好きな長さのリストを作ることができます[5]。数を値として持っていた変数に、記号を代入してこれを新しい値にすることも可能です。このように、Scheme ではデータの取り扱いにおいて非常に柔軟性があります。C や Pascal や Fortran などの言語ですと、変数はどのデータ型のものであるかをあらかじめ宣言しておき、宣言された型のみの代入しか許していません[6]。

リストは、実は**対** (pair) と呼ばれる、より基本的なデータを組み合わせて作られます。対の説明をする前に、与えられたリストに含まれている構成要素を調べる方法を説明します。

リストは、要素がいくつかならんだものでした。最初の要素を返す手続きが car で、最初の要素を取り除いたリストを返す手続きが cdr です[7]。なお、car は「カー」と、cdr は「クダー」と発音します。

---

[4] というのも、単に () とすると「手続き名を書き忘れた、手続き呼び出し」とみなされるからです。(Scheme 処理系によっては () という書き方を許しています。しかしこれは Scheme の言語仕様では許されていない書式なので、そのような書き方はしないで下さい。)

[5] もっとも主記憶装置の容量には限りがありますので、その範囲内で、という意味です。

[6] これにも長短があります。Scheme のように変数の宣言がないと、プログラムを作るときに変数名のつづりを間違っても、実際に走らせてみなければそれに気がつきません。Pascal などの変数宣言をする言語ですと、プログラムを機械語へ翻訳するとき (プログラム作成時) に変数名のチェック (つづりやデータ型) をおこない、実行時にはそれをおこないません。もし実行時に型エラー (記号と数を加えようとしてしまう間違いなど) が起き、プログラムが中断されては重大な事態になってしまうことがあります。(とくに、原子炉の制御や集中治療室の制御などでは人命に関わる事態です。) そのような場合で使われるプログラムは、前もって変数の名前や型をチェックする必要があります。

[7] 1950 年台後半のころ、IBM 社の IBM 7090 という計算機で、世界で最初の Lisp (Scheme ご先祖にあたる言語) の処理系が作られたことに由来しています。IBM 7090 の機械語命令にはアドレス (address) 部とデクリメント (decrement) 部があって、car は Contents of Address part of Register の、cdr は Contents of Decrement part of Register の略です。いまや IBM 7090 は死に絶えてしまっているのに、car や cdr はこんにちでも使われ続けています。

```
> (define idols '(hikaru miho hiroko noriko))
#<unspecified>
> (car idols)
hikaru
> (cdr idols)
(miho hiroko noriko))
```

car ではリストの最初の要素しか取り出せませんが、car と cdr を組合せることで、2 番めの要素や 3 番めの要素も取り出すことができます。

```
> (car (cdr idols))
miho
> (car (cdr (cdr idols)))
hiroko
> (car (cdr (cdr (cdr idols))))
noriko
```

3 番目や 4 番目の要素を取り出すのに car や cdr を何度も書くのはめんどくさいので、car と cdr を組み合わせた手続きが用意されています。たとえば (cadr S) は (car (cdr S)) と同じで、(cadar S) は (car (cdr (car S))) と同じです。このほかにも car と cdr を 4 つまで組み合わせた手続きが用意されています。(たとえば、cadr, caddr, cadddr など。)

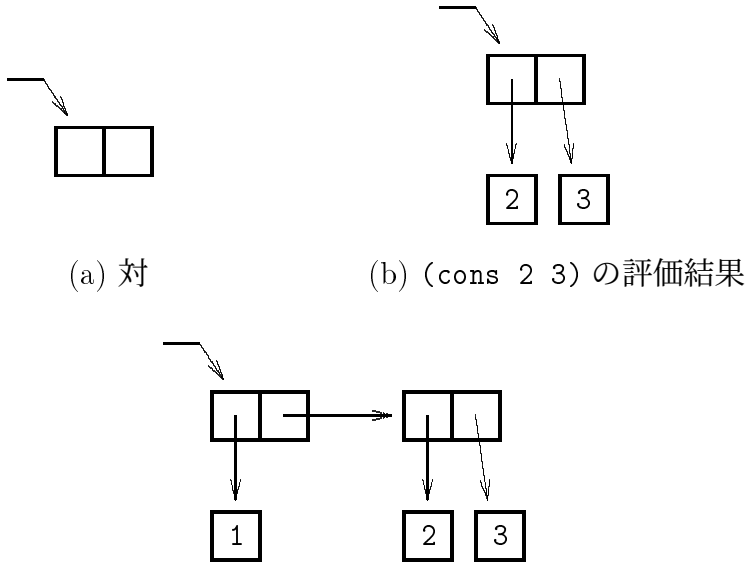
では、上の例と同じことをやってみましょう。

```
> (cadr idols)
mami
> (caddr idols)
hiroko
> (cadddr idols)
noriko
```

前で説明したように、リストは Scheme の基本的なデータ型ではありません。リストは **対** (pair) と呼ばれる基本データ型を組み合わせたものです。図 3.1(a) のように、対は 2 つの Scheme データを収めるハコのようなものです。(図 3.1(c) には、3 つの対があります。)

対は cons という手続きで作られます[8]。手続き cons は、2 つの引数を持ちます。式 (cons a b) を評価するときには、まず新しいハコ (対) が用意され、ハコの前の部分に a (の評価値) を、ハコの後ろの部分に b (の評価値) を入れます。そうしてこのハコが、(cons

[8] cons は construct (英語で「構成する」、の意) にちなんで付けられた名前で、リストを「構成する」のでこの名前がつけられました。



(a) 対　　(b) (cons 2 3) の評価結果

(c) (cons 1 (cons 2 3)) の評価結果

図 3.1: 対

a b) の評価値 (実行結果) として返されます。この理由のため、対の前の部分は **car 部**、後ろの部分は **cdr 部** と、それぞれ呼ばれます。

以上のことをまとめると次のようになります。

◇ (car ⟨対⟩)
— ⟨対⟩ の car 部を取り出します。

◇ (cdr ⟨対⟩)
— ⟨対⟩ の cdr 部を取り出します。

対は (a . b) のように、要素を . で区切り、括弧 (と) で囲むことで表されます。たとえば (cons 1 2) を評価すると、対 (1 . 2) が作られます。(図 3.1(b) を参照してください。) まんなかの点 . は、対を構成する 2 つの要素を区切るためのもので、小数点ではありません。(. の前後に空白文字がありますので、注意深く見れば区別がつきます。)

ではもう少し複雑な例を見てみましょう。式 (cons 1 (cons 2 3)) を評価すると、上で学んだことから考えると (1 . (2 . 3)) が得れることがわかります。このように . を使った書き方を、**ドット記法** (dotted notation) と呼んでいます。

普通はドット記法 (a . (b . ())) のように書かずに、(a b) と書きます。より一般的にいえば、(a1 . (a2 . ···(an . ()) ···)) を (a1 a2 ···an) と書き表します。このような書き方を **リスト記法** (list notation) と呼んでいます。

式 (cons 1 (cons 2 3)) を評価してみましょう。

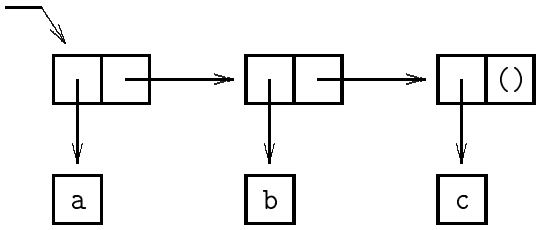


図 3.2: リスト (a b c)

```
> (cons 1 (cons 2 3))
(1 2 . 3)
```

結果は (1 . (2 . 3)) ではなく、(1 2 . 3) と表示されました。ほとんどすべての Scheme 処理系では、ドット記法でなく改良リスト記法を使ってデータを表示します[9]。というのも、ドット記法を使うと、長いリストの時は括弧をたくさん表示しないといけないからです。

リスト (list) とは対が cdr 部で連なったもので、しかも最後の対の cdr 部が空リストであるものをいいます。図 3.2のように、最後の対の cdr 部が空リストでないときはリストとは呼びません。

与えられたデータが対かどうか、あるいはリストかどうかを判定するために、手続き pair? と手続き list? が用意されています。

◇ (pair? ⟨データ⟩)
— ⟨データ⟩ が対なら真 #t を、そうでなければ偽 #f を返します。

◇ (list? ⟨データ⟩)
— ⟨データ⟩ がリストなら真 #t を、そうでなければ偽 #f を返します。

手続き pair? は、与えられたデータがハコかどうか調べるだけです。(ハコの car 部や cdr 部がどのような値を持っているかは、一切調べません。) いっぽう手続き list? は、まず与えられたデータがハコかどうか調べます。次にそのハコの cdr 部もさらにハコになっているかを繰り返し調べます。最後に空リストにたどり着けばそのデータはリストである、と判定します。これが pair? と list? の違いです。

```
> (pair? '(a b))
#t
> (pair? '())          — 空リストは対ではない!
```

[9]改良リスト記法とは、できる限りリスト記法を使うのですが、リスト記法が使えない時のみドット記法を使う方法です。この方法では、(a1 . (a2 . ···(am (an . ())) ···)) は、(a1 a2 ···am . an) と表記されます。

```
#f
> (pair? '(1 . 2))
#t
> (list? '(a b c))
#t
> (list? '())
#t
> (list? '(1 . 2))
#f
> (list? '(0 1 . 2))
#f
```

list?, pair?, null? の関係を、以下に例示します。

	データ			
	'()	(pooh)	(pooh piglet owl)	(kanga . roo)
list?	#t	#t	#t	#f
pair?	#f	#t	#t	#t
null?	#t	#f	#f	#f

3.5 リスト操作関数

前節で学んだ car, cdr, cons を使うことで、リストや対に対する操作ができるようになりました。ここではリストに関する便利な手続きを学びます。

リスト (1 2 3 4) は、(cons 1 (cons 2 (cons 3 (cons 4 '())))) を評価すれば作ることができます。もしリストの要素の値があらかじめ決まっている時は '(1 2 3 4) というように、引用符を使うこともできます。ですが変数 a1, a2, a3, a4 の値を要素とするリストを作るには、引用符を使う方法は使えません:

```
> '(a1 a2 a3 a4)
(a1 a2 a3 a4)
```

かといって、(cons a1 (cons a2 (cons a3 (cons a4 '())))) と入力するのも長くて大変です。

手続き list は、新たなリストを作る手続きです。

◇ (list ⟨データ$_1$⟩ ⟨データ$_2$⟩ ···)
— ⟨データ$_1$⟩ ⟨データ$_2$⟩ ···を要素とするリスト (⟨データ$_1$⟩ ⟨データ$_2$⟩ ···) を新たに作ります。

例を見てみましょう。

```
> (define a 100)
#<unspecified>
> (define b 200)
#<unspecified>
> (list 1 2 3 4)
(1 2 3 4)
> (list a b)
(100 200)
> (list a '(a b) b)
(100 (a b) 200)
> (list a '(a b) 'b)
(100 (a b) b)
```

リストに関する手続きとして、以下のものが用意されています。

◇ (length ⟨ リスト ⟩)
— ⟨ リスト ⟩ の要素数を返します。

◇ (append ⟨ リスト$_1$⟩ ⟨ リスト$_2$⟩ ···)
— ⟨ リスト$_1$⟩ ⟨ リスト$_2$⟩ ···をつなげたリストを返します。

◇ (reverse ⟨ リスト ⟩)
— ⟨ リスト ⟩ の要素を逆順にしたリスト返します。

◇ (list-ref ⟨ リスト ⟩ i)
— ⟨ リスト ⟩ の i 番目の要素を返します。

△ (list-tail ⟨ リスト ⟩)
— ⟨ リスト ⟩ の最後の要素を返します。

実行例を見てみましょう。

```
> (length '(a b c d))
4
> (length '(a b ))
2
> (length '())
0
> (length '((a) (b c d)))
2
> (length '((a) (b c d) e))
```

```
3
> (append (list 'a 'b) (list 'c 'd))
(a b c d)
> (append (list 'a '(b c)) (list 'd 'e))
(a (b c) d e)
> (reverse '(a b c d e))
(e d c b a)
> (reverse '(a (b c) ((d)) e))
(e ((d)) (b c) a)
> (reverse '())
()
> (list-ref '(a b c d e) 0)
a
> (list-ref '(a b c d e) 1)
b
> (list-ref '(a b c d e) 4)
e
```

3.6 さまざまなデータ型 (その 1)

これまででは記号と数の 2 つのデータ型だけを使ってきましたが、Scheme ではこの他にもいろいろなデータ型が用意されています。ここでは、ブール型、 数 (整数と実数のみ)、文字列について説明します[10]。

3.6.1 ブール型

ブール型 (boolean) は真偽を表すデータ型です。真を表すデータは `#t` で、 偽を表すデータは `#f` です。 ブール型のデータは後で説明する条件式 `if` や `cond` の条件判定部などで、ある条件が成立するかを調べるのに使われます[11]。

ブール型のデータは評価されてもそれ自身を値として持ちます。そのためプログラムに中に書くときは、引用符をつけないてすみます。このようなデータは **自己評価的** (self-evaluating) である、と呼ばれています。

```
> #t
#t
```

[10] この他にも、文字型、ベクトル、手続き、継続などがあります。

[11] 厳密なことをいえば、Scheme の条件式では `#f` のみが条件の偽として取り扱われ、 それ以外の場合は真とみなされます。

```
> #f
#f
> '#t
#t
```

与えられたデータがブール型のデータかどうかを判定するには、手続き boolean? を使います。

◇ (boolean? 〈データ〉)
— 〈データ〉がブール型のデータなら真 #t を、そうでなければ偽 #f を返します。

```
> (boolean? #t)
#t
> (boolean? #f)
#t
> (boolean? 10)
#f
> (boolean? '(1 2 3))
#f
> (boolean? '())
#f
```

ブール値の真と偽を反転させるには、手続き not を使います。

◇ (not 〈データ〉)
— 〈データ〉が #f なら #t を、そうでなければ #f を返します。(〈データ〉は必ずしもブール型のデータでなくてもかまいません。)

```
> (not #t)
#f
> (not #f)
#t
> (not 10)
#f
> (not (cons 1 2))
#f
> (not '())
#f
```

3.6.2 数

Scheme では **数** (numbers) を取り扱うことができます。これまでの例では整数しか使いませんでしたが、Scheme では**整数** (integer)、**有理数** (rational)、**実数** (real)、**複素数** (complex) の 4 つの部分型が用意されています。(Scheme 処理系によっては用意されていない型があるかもしれません。) ここでは整数と実数について説明します。なお数のデータ型は演算結果に応じて変化します。例えば 4 は整数型ですが、これに 1.5 をかければその結果は実数型の 6.0 になります。

数に対する演算には、次のものが用意されています。

◇ `(+` x_1 x_2 $\cdots$`)`
— $x_1 + x_2 + \cdots$

◇ `(-` x`)`
— $-x$

◇ `(-` x_1 x_2`)`
— $x_1 - x_2$

△ `(-` x_1 x_2 x_3 $\cdots$`)`
— $x_1 - x_2 - x_3 - \cdots$ (処理系によってはこの形式での使い方ができないことがあるので注意して下さい。)

◇ `(*` x_1 x_2 $\cdots$`)`
— $x_1 \cdot x_2 \cdot \cdots$

◇ `(/` x`)`
— $1/x$

◇ `(/` x_1 x_2`)`
— x_1/x_2

△ `(/` x_1 x_2 x_3 $\cdots$`)`
— $x_1/(x_2 \cdot x_3 \cdots)$ (処理系によってはこの形式での使い方ができないことがあるので注意して下さい。)

◇ `(min` x_1 x_2 $\cdots$`)`
— $x_1, x_2, \cdots$の中の最小値

◇ `(max` x_1 x_2 $\cdots$`)`
— $x_1, x_2 \cdots$の中の最大値

◇ (abs x)
— x の絶対値

◇ (floor x)
— x よりも大きくない最大の整数

◇ (ceiling x)
— x よりも小さくない最小の整数

◇ (truncate x)
— 小数点以下を切捨てる

◇ (round x)
— 小数点以下を四捨五入する

例を見てみましょう。

```
> (+ 2 4)
6          — 演算結果は整数 6
> (+ 2 4.0)
+6.0       — 演算結果は実数 6.0
> (- 2 3)
-1
> (- 3)
-3
> (* 3.0 4.5)
+13.5
> (/ 10 5)
2
> (/ 10 5.0)
+2.0
> (max 1 3 4 5 10 23)
23
> (min 4 2 8 10)
2
> (abs -8)
8
> (floor 10.3)
10
> (floor -10.3)
```

```
-11
> (ceiling 10.3)
11
> (ceiling -10.3)
-10
> (round -10.3)
-10
```

整数に関する演算手続きとして、次のものがあります。

◇ (gcd n_1 n_2)
— 最大公約数

◇ (lcm n_1 n_2)
— 最小公倍数

◇ (quotient n_1 n_2)
— 商

◇ (remainder n_1 n_2)
— 剰余

◇ (modulo n_1 n_2)
— 剰余

gcd と lcm は、それぞれ最大公約数と最小公倍数を求める手続きです。quotient, remainder, modulo は、それぞれ次の値を返します。

- (quotient n_1 n_2) $\Rightarrow$ n_3
- (remainder n_1 n_2) $\Rightarrow$ n_4
- (modulo n_1 n_2) $\Rightarrow$ n_4

ただし、$n_1 = n_2 n_3 + n_4$ で、しかも $0 \leq n_4 < n_2$ です。(n_3 は商を、n_4は剰余を表しています。) 引数が負のときは、remainder と modulo は違う値を返します。

```
> (remainder 17 3)
2
> (modulo 17 3)
2
> (remainder -17 3)
-2
```

```
> (modulo -17 3)
1
```

数に関する述語[12]として、次の手続きが用意されています。

◇ (= x_1 x_2 $\cdots$)
— $x_1 = x_2 = \cdots$か?

◇ (< x_1 x_2 $\cdots$)
— $x_1 < x_2 < \cdots$か?

◇ (> x_1 x_2 $\cdots$)
— $x_1 > x_2 > \cdots$か?

◇ (<= x_1 x_2 $\cdots$)
— $x_1 \leq x_2 \leq \cdots$か?

◇ (>= x_1 $x_2 \cdots$)
— $x_1 \geq x_2 \geq \cdots$か?

◇ (zero? x)
— x は零か?

◇ (positive? x)
— x は正か?

◇ (negative? x)
— x は負か?

◇ (odd? n)
— n は奇数か?

◇ (even? n)
— n は偶数か?

これらの実行例を見てみましょう。

```
> (= 3 3 3)
#t
> (= 1 3 3)
#f
```

[12]述語(じゅつご)とは、ある性質が成り立つかどうかを調べるものをいいます。

```
> (< 3 4 5)
#t
> (<= 3 3 4 4 6)
#t
> (zero? 2)
#f
> (zero? 0)
#t
> (positive? 2)
#t
> (negative? -2)
#t
> (odd? 3)
#t
> (odd? 2)
#f
> (even? 2)
#t
```

データ型を調べるための手続きも用意されています。

◇ (number? x)
— x が数 (整数/有理数/実数/複素数のいずれか) なら #t が返され、そうでなければ #f が返されます。

◇ (integer? x)
— x が整数なら #t が返され、そうでなければ #f が返されます。例外として、x が実数でも、 小数部分が 0 のときは #t が返されます。

◇ (real? x)
— x が実数なら #t が返され、そうでなければ #f が返されます。(整数は実数の特別な場合みなせますので、x が整数のときも #t が返されます。)

これらを試してみましょう。

```
> (number? 'foobar)
#f
> (number? 8.2)
#t
> (number? 2)
```

```
#t
> (integer? 8.2)
#f
> (real? 8.2)
#t
> (integer? 2)
#t
> (integer? 2.0)
#t
> (real? 100)
#t
```

計算機の内部では、実数と整数とでは表現形式がまったく違います。取り扱える範囲を越えない限り、整数の演算は誤差なくできます。ですが、実数はコンピューター内部では近似値で保持されるので、どうしても誤差が生じてしまいます。(ほとんどの計算機は、浮動小数点(ふどうしょうすうてん)と呼ばれる形式で実数を計算機内部に保持しています。) たとえば 10 割る 3 を計算し、コンピューター内部に保持することを考えてみましょう。計算結果は $3.3333\cdots$ と、無限に 3 が続きます。そのために、この結果を保持するには無限に多くの記憶領域が必要になります。ですがそれは不可能なので、あるところで切って値を保持せざるを得ません。3.33333 として保持したとすれば、本当の値とは少し違っていることが分かると思います。保持する桁数を多くすれば本当の値に近くなりますが、それでも本当の値とはわずかに違っています。

手続き `real?` は整数 `1` に対して `#t` を返しますし、手続き `integer?` は実数 `1.00` に対しても `#t` を返します。ですがさきほど説明した誤差の理由で、整数なのか実数なのかを厳密に調べたいことがあります。そのため手続き `integer?` と `real?` では、数がコンピューター内部で整数として保持されているのか、それとも実数として保持されているのかを知ることができません。

Scheme ではこの概念を**厳密性** (exactness) という言葉で表現しています。コンピューター内部で誤差が生じない形で保持されているものを**厳密数** (exact number) と呼び、そうでないものを**非厳密数** (inexact number) と呼んでいます。

数が厳密数か非厳密数かを調べるのには、次の手続きを使います。

◇ `(exact?` x`)`
— x の値が厳密数なら真 `#t` を、そうでなければ偽 `#f` を返します。

◇ `(inexact?` x`)`
— x の値が非厳密数なら真 `#t` を、そうでなければ偽 `#f` を返します。

△ `(inexact->exact` n`)`
— 非厳密数 n を厳密数に変換します。

△ (exact->inexact n)
— 厳密数 n を非厳密数に変換します。

これらを試してみましょう。

```
> (exact? 1)
#t
> (integer? 1.0)
#t
> (exact? 1.0)
#f
> (real? 1)
#t
> (inexact? 1)
#f
> (inexact? 1.0)
#t
> (inexact->exact 4.4)
4
> (inexact->exact 4.5)
5
> (exact->inexact 4)
4.0
```

この他に、三角関数や指数関数なども用意されています[13]。

△ (exp x)
— 指数関数 e^x (e は自然対数の底)

△ (log x)
— 対数関数 $\log_e x$

△ (sin x)
— 正弦関数 $\sin x$

△ (cos x)
— 余弦関数 $\cos x$

△ (tan x)
— 正接関数 $\tan x$

[13] これらは Scheme に必須の手続きと定められていません。そのため、処理系によっては用意されていない場合があります。

△ (asin x)
— 逆正弦関数 $\sin^{-1} x$

△ (acos x)
— 逆余弦関数 $\cos^{-1} x$

△ (atan x)
— 逆正接関数 $\tan^{-1} x$

△ (atan y x)
— 逆正接関数 $\tan^{-1} y/x$

△ (sqrt x)
— 平方根 $\sqrt{x}$

△ (expt x y)
— 指数関数 x^y

3.6.3 文字列

文字列 (string) とは、文字のつらなりであるデータです。プログラム中では、文字列は文字がいくつかならんだものを二重引用符 " で囲むことで表されます。たとえば、

```
"honey"
"Pooh and Piglet"
""
```

は、いずれも文字列です。文字列の中に空白文字があってもかまいませんし、まったく文字がなくてもかまいません。まったく文字が含まれない文字列 "" は、**空文字列** (null string) と呼ばれる文字列です。

文字列は自己評価的なデータなので、引用符をつける必要はありません。

```
> '"This is a string"
"This is a string"
> "This is a string"
"This is a string"
```

文字列の中に二重引用符を持つ文字列を作るには、二重引用符の前バックスラッシュ \ を付けて \" とします[14]。

[14]使用するキーボードによっては \ がないことがあります。そのときは代わりに、円記号 ¥ を使ってください。そのような場合では、表示は \ の代わりに ¥ となります。

```
> "He said, \"Are you sure?\""
"He said, \"Are you sure?\""
> (display "He said \"Are you sure?\"")
He said "Are you sure?"#<unspecified>
```

手続き display は、引数に与えられたデータを表示し、手続きの実行結果として #<unspecified> を返します。上の例で #<unspecified> と表示されているのは、手続き display の返した値が表示されたものです。(この値は処理系によって違うので、注意して下さい。)

データが文字列かどうかを判定するには、手続き string? を使います。

◇ (string? 〈データ〉)
— 〈データ〉が文字列なら真 #t を、そうでなければ偽 #f を返します。

実行例は次のようになります。

```
> (string? 'apple)
#f
> (string? "apple")
#t
> (string? 1924)
#f
```

そのほか文字列に関して、次の手続きが用意されています。

◇ (string-length 〈文字列〉)
— 〈文字列〉の長さ (文字数) を返します。

◇ (substring 〈文字列〉 x y)
— 〈文字列〉の x番目から y番目までの部分文字列を返します。(最初の文字は 0 番目です。) 新たに作られる文字列には y 番目の文字は含まれず、x と y は 0 以上 (length 〈文字列〉) 以下でなくてはいけません。なお x と y は整数でないといけません。

◇ (string-append 〈文字列$_1$〉〈文字列$_2$〉 $\cdots$
— 〈文字列$_1$〉〈文字列$_2$〉$\cdots$を順につないだ文字列を返します。

ではこれらを試してみましょう。

```
> (string-length "Pooh and Piglet")
15
> (string-length "")
```

```
0
> (substring "Pooh, Owl, Piglet" 6 9)
"Owl"
> (substring "Pooh" 0 0)
""
> (string-append "Winnie" "-" "the" "-" "Pooh")
"Winnie-the-Pooh"
```

文字列の大小を比較するために、次のような手続きがあります。

◇ (string=? ⟨文字列$_1$⟩ ⟨文字列$_2$⟩)
— ⟨文字列$_1$⟩と⟨文字列$_2$⟩が等しいか?

◇ (string-ci=? ⟨文字列$_1$⟩ ⟨文字列$_2$⟩)
— ⟨文字列$_1$⟩と⟨文字列$_2$⟩が等しいか?

◇ (string<? ⟨文字列$_1$⟩ ⟨文字列$_2$⟩)
— ⟨文字列$_1$⟩ は ⟨文字列$_2$⟩ より小さいか?

◇ (string-ci<? ⟨文字列$_1$⟩ ⟨文字列$_2$⟩)
— ⟨文字列$_1$⟩ は ⟨文字列$_2$⟩ より小さいか?

◇ (string>? ⟨文字列$_1$⟩ ⟨文字列$_2$⟩)
— ⟨文字列$_1$⟩ は ⟨文字列$_2$⟩ より大きいか?

◇ (string-ci>? ⟨文字列$_1$⟩ ⟨文字列$_2$⟩)
— ⟨文字列$_1$⟩ は ⟨文字列$_2$⟩ より大きいか?

◇ (string<=? ⟨文字列$_1$⟩ ⟨文字列$_2$⟩)
— ⟨文字列$_1$⟩ は ⟨文字列$_2$⟩ より小さいまたは等しいか?

◇ (string-ci<=? ⟨文字列$_1$⟩ ⟨文字列$_2$⟩)
— ⟨文字列$_1$⟩ は ⟨文字列$_2$⟩ より小さいまたは等しいか?

◇ (string>=? ⟨文字列$_1$⟩ ⟨文字列$_2$⟩)
— ⟨文字列$_1$⟩ は ⟨文字列$_2$⟩ より大きいまたは等しいか?

◇ (string-ci>=? ⟨文字列$_1$⟩ ⟨文字列$_2$⟩)
— ⟨文字列$_1$⟩ は ⟨文字列$_2$⟩ より大きいまたは等しいか?

これらの手続きのうちで名前に -ci のついているものは、大文字と小文字の違いを無視して比較します。すなわち、 a と A は同じものとみなして比較します。文字列の大小は、辞書式順序で決められます。たとえば "b" < "ba" < "baa" < "bb" < "c" です。

例を見てみましょう。

```
> (string=? "a b" "a b")
#t
> (string=? "a b" "a b c")
#f
> (string-ci=? "a b" "A B")
#t
> (string<? "aa" "ab")
#t
> (string<? "aa" "a")
#f
> (string>? "aba" "aaa")
#t
```

なお、 それぞれの文字についての大小関係は次の通りに定められています。

- 大文字: "A"<"B"< ... <"Z"
- 小文字: "a"<"b"< ... <"z"
- 数字: "0"<"1"< ... <"9"
- 大文字と小文字: "A"<"a", "B"<"b" ,...
- 各数字はどの大文字より小さい[15]。
- 各数字はどの小文字より小さい[16]。

3.7 等価性判定

2 つのデータが同じかどうかを判定する手続きを紹介します。2 つの数が等しいかを判定する手続き = はすでに紹介しましたが、= は数の比較だけのためのものです。数のほかにも、2 つのリストが同じかどうかの比較も必要となります。比較のために、以下の手続きがあります。

◇ (eqv? 〈データ$_1$〉〈データ$_2$〉)

◇ (eq? 〈データ$_1$〉〈データ$_2$〉)

◇ (equal? 〈データ$_1$〉〈データ$_2$〉)

[15,16]Scheme の言語仕様書では、 この場合または、これのまったく逆 (各数字はどの小文字より大きい) のどちらかであると定めています。

これらの手続きはいずれも、2 つの引数 ⟨データ$_1$⟩ ⟨データ$_2$⟩ が (それぞれの手続きの意味で) 同じであれば真 #t を、そうでなければ偽 #f を返します。

もし与えられた 2 つの引数が同一であれば、どの手続きも #t を返します。ここで「同一」とは、2 つの引数の実体が計算機の記憶領域の中で同一の場所にある、という意味です。2 つのものが同じ記憶番地にあれば、2 つは同一のものです。画面に表示させるとまったく同じでも記憶番地が違えば、もし一方が書き換えられるとそれらは違うものになってしまいます。

```
> (define qwe '(have fun))
#<unspecified>
> (eqv? qwe qwe)
#t
> (eq? qwe qwe)
#t
> (equal? qwe qwe)
#t
```

この例では同一のものを比較していますので、比較結果はいずれも #t になっています。もし 2 つの引数の型が違うと、比較結果は必ず #f になります。

```
> (eqv? 'abc "2")
#f
> (eq? 'abc 3.14)
#f
> (equal? "3.14" 3.14)
#f
```

手続き eqv? は、次のいずれかの場合に #t を返します。

- 2 つの引数がともに #t、 またはともに #f のとき。
- 2 つの引数がともに記号で、 同じ文字のならびであるとき。
- 2 つの引数がともに数で、しかも数値として同じであり、さらにともに整数もしくはともに実数のとき。
- 2 つの引数がともに空リストのとき。
- 2 つの引数がともに対、ベクトル、あるいは文字列で、同一の記憶番地にあるとき。
- 2 つの引数がともに手続きで、同一の記憶番地にあるとき。(手続きについては、次節で説明します。)

例を見てみましょう。

```
> (eqv? 'abc 'abc)
#t
> (eqv? 'foo 'bar)
#f
> (eqv? '() '())
#t
> (eqv? 16 16)
#t
> (eqv? 16 (* 2 8))
#t
> (eqv? 16 16.0)
#f
> (eqv? (list 'pooh 'piglet) (list 'pooh 'piglet))
#f
> (define animal (list 'pooh 'piglet))
#<unspecified>
> (eqv? animal animal)
#t
```

eqv? は、文字列やリストの比較のときは、メモリアドレスが同一かどうかで等価性の判定をします。例として (eqv? "A" "A") のように定数データの比較の場合を考えましょう。2 つのデータが同一のメモリアドレスのものとなるどうかは処理系に依存しています。これらは定数なので、ある処理系はあるメモリアドレスに文字列を "A" を置き、eqv? の 2 つの引数に同じものを与えるあるかもしれませんし、別の処理系ではそうしないかも知れません。このために、定数データ同士の比較は処理系に依存するので避けるべきです。同じ理由により、(eqv? '(1 2) '(1 2)) や (eqv? '(b) (cdr '(a b))) のような使い方もしないでください。

手続き eq? は、2 つの引数が同一 (同一のメモリアドレス) にあるときに#t を返します。eq? は eqv? と似ています。ですが、数値的に同じでも記憶場所の違う数の比較のときに #t を返すかどうかは Scheme の言語仕様では規定されていないので、そういった使い方はしないでください。数の比較には eq? は使わず、数同士の比較をする手続き=, eqv? あるいは後述の equal? を使ってください。

```
> (eq? 'a 'a)
#t
> (eq? 'a 'b)
#f
```

```
> (eq? (list 'pooh 'piglet) (list 'pooh 'piglet))
#f
> (define animal (list 'pooh 'piglet))
#<unspecified>
> (eqv? animal animal)
#t
> (eq? '() '())
#t
```

手続き equal? は、対やベクトルなどの比較のときには、要素それぞれを比較しますが、さらに対やベクトルがあればそれらの要素についても、さらに等しいかどうかを調べます。(ベクトルは、6.1.2 にて学びます。) その他 (数や記号など) に対しては、eqv?と同じ方法で同じかどうかを調べます。(直観的な説明ですが、2 つのデータが同じに表示されるときは、equal? は #t を返す場合が多いです。)

```
> (equal? '(a b) '(a b))
#t
> (equal? (list 'a 'b) (list 'a 'b))
#t
> (equal? '(a b (c d)) '(a b (c d)))
#t
> (equal? "my home" "my home")
#t
> (equal? 13 13)
#t
> (equal? 3.14 3.14)
#t
> (equal? 1 1.0)
#f
```

3.8 手続き

利用者が自分で新しい手続きを定義し、あらかじめ用意してある手続きと同じように呼び出して使うことができます。あらかじめ用意してある手続きのことを**基本手続き** (primitive procedure) と呼び、利用者が定義した手続きのことを**複合手続き** (compound procedure) と呼んでいます。

以前でも説明しましたが、手続きを定義するのには特殊形式 define を使います[17]。次

[17] こうして定義された手続きは、**複合手続き** (compound procedure) と呼ばれています。

の例では引数に 1 を加えた数を返す手続き add1 を定義しています。

```
> (define (add1 n) (+ n 1))
#<unspecified>        — define が返す値
> (add1 1)
2
> (add1 99)
100
```

手続き呼び出しで与えられた引数(ひきすう) のことを、**実引数**(じつひきすう)(actual argument) といいます。上での (add1 1) の場合だと 1 が実引数です。手続き add1 の定義の中での n は、add1 が呼び出されたときの実引数の値が入れられ、add1 の本体部分である (+ n 1) が実行されます。n は手続き add1 の**仮引数**(かりひきすう)(formal argument) と呼ばれます。

n が実引数の値を持つのは、手続き add1 の内部だけです。add1 の外で n が値を持っていても add1 の中では実引数の値が優先され、外での n の値は隠されてます。n が実引数の値を持つのは手続き add1 の内部だけで、add1 の実行が終了したら元の値に戻ります。

```
> (define n 10)
#<unspecified>
> n
10
> (add1 1)
2
> n
10
```

この例では、まず変数 n の値を 10 にしています。そして手続き add1 を呼び出していますが、n の値を 10 にしたことは影響していません。add1 の実行が終了したのちに変数 n の値を調べていますが、add1 の実行によって値が 1 になっておらず、前のままの 10 であることがわかります。

次は、2 つの引数を持った手続きの例です。

```
> (define (distance x y) (sqrt (+ (* x x) (* y y))))
#<unspecified>
> (distance 1 1)
+1.414213562373095
> (distance 2 4)
+4.47213595499958
```

一般的に手続き定義は次の形をしています。

(define (⟨ 手続き名 ⟩ ⟨ 仮引数のならび ⟩)
  ⟨ 式$_1$⟩
  ⟨ 式$_2$⟩
  ⋮
  ⟨ 式$_n$⟩)

ここで ⟨ 手続き名 ⟩ は変数で、⟨ 仮引数のならび ⟩ は変数を任意個ならべたものです。仮引数のならびには、同じ名前の変数が現れてはいけません。

手続き呼び出しは、 一般的には次の形をしています。

(⟨ 手続き名 ⟩ ⟨ 実引数のならび ⟩)

以前に出てきた変数の定義では (define count 0) という書き方をしていました。手続き定義の場合では、定義する手続き名と仮引数が括弧で囲まれていて、同じ define なのに変数のときと使い方が違っています。上で紹介した手続き定義の方法は、実は簡略記法です。簡略でない記法で add1 の定義を書くと次のようになります。

```
(define add1
  (lambda (n) (+ n 1)))
```

ここで (lambda ···) は、**ラムダ式** (λ expression) と呼ばれるものです。これが評価されると **手続き** (procedure) という型のデータが返されます。これでわかるように、手続き定義は変数定義とまったく同じです。定義する変数の値が手続きデータであるというだけです。

◇ (lambda ⟨ 仮引数 ⟩ ⟨ 本体 ⟩)
— ⟨ 仮引数 ⟩ をパラメータとして持ち、本体の式のならびが⟨ 本体 ⟩ である手続きを作ります。

◇ (procedure? ⟨ データ ⟩)
— ⟨ データ ⟩ が手続きデータなら真 #t を、そうでなければ偽 #f を返します。

ラムダ式は名前のない手続きともみなせます。たとえば与えられた数に 1 を加える手続きは

```
(lambda (n) (+ n 1))
```

と表せます。ですので 1 を加えた数を求めるのに、define を使って新しい手続きを定義する必要はありません。

```
> ((lambda (n) (+ n 1)) 10)
11
```

この例ではまず (lambda (n) (+ n 1)) が評価されて、「引数に 1 を加える手続きデータ」が作られます。次に引数が評価されて 10 を得ます。最後に 10 が手続きデータに渡されて、実行されます。

こんどは次の場合を考えてみましょう。

```
> (add1 10)
11
```

この場合も上の場合と同じです。まず add1 が評価されます。この変数の値は、「引数に 1 を加える手続きデータ」です。次に引数が評価されて 10 を得ます。最後に 10 が手続きデータに渡されて実行されます。

手続きもひとつのデータなので、手続きを他の手続きに引数として与えることもできます。

```
> (define (calc x op y) (op x y))
#<unspecified>
> (calc 3 + 4)
7
> (calc 2 - 8)
-6
> (calc 3 * 6)
18
```

+, -, * はそれぞれ、「加算をする手続き」、「減算をする手続き」、「乗算をする手続き」を値として持っています。手続き calc の中で op は、引数で与えられた手続きデータを値として持っています。式 (op x y) の評価は、 op が値として持っている手続きデータに x と y を渡して実行するようになっていて、その結果が calc の値となっています。

これまで説明したラムダ式の引数は、(x op y) のような変数のリストで、取り得る「引数の数」は決まっていました。そのために、式 (calc 2) を評価しようとすると、引数の数が異なるためにエラーとなります:

```
> (calc 2)


Error: Wrong number of args to ...
in excpression: (... calc 2)
in top level environment.
; Evaluation took 50msec ...
```

たとえば足し算をする手続き + は、任意の数の引数をとることができます。このようにあるときの呼び出しでは 2 つの引数で、またあるときの呼び出しでは 4 つの引数で、というように、手続き呼び出しのたびに引数の数を変える方法を紹介します。

任意の数の引数を手続きに渡すのに、(foo (list a b c d e)) として引数をすべて 1 つのリスト入れるのもひとつの方法です。すると手続き foo は、1 引数の手続きとして作ればよいことになります。ですがいちいちリストにまとめるのも大変です。

手続きが可変数の引数を受けとるには、次のようにします。

```
(define 〈手続き名〉
  (lambda 〈引数名〉
    〈式のならび〉))
```

ここで〈引数名〉は記号のリストではなく、ひとつの記号であることに注意して下さい。

たとえば

```
(define foo
  (lambda s
    (if (null? s)
        '()
        (list-ref s (- (length s) 1)))))
```

は、foo の最後の引数を返す手続きです。(foo 1 2 3 4) の評価において仮引数 s は、値 (1 2 3 4) を持つことになります。

```
> (foo 1 2)
2
> (foo 1 2 3 4 5)
5
> (foo)
()
```

3.9 局所変数

プログラムが少し複雑になってくると、計算の途中結果を一時的に保持する変数を、手続きの内部で使いたくなることがあります。たとえば関数 $f(x) = x^4 + g(x)x^2$ を計算する手続き f1 を作る場合を考えてみましょう。($g(x)$ を計算する手続き g は、すでに定義されていると仮定します。)

```
(define (f1 x)
  (+ (* x x x x)  (* (g x) x x)))
```

この定義では、 $x \cdot x \cdot x \cdot x + g(x) \cdot x \cdot x$ を求めることで、関数 f の計算をしています。ですが私たちが計算するときはこのような計算はせず、ひとまず $y = x^2$ を計算してから、$y \cdot y + y$ を計算します。ではこの方針に従って f1 を改良した手続き f2 を作ります:

```
(define y 0)
(define (f2 x)
  (set! y (* x x))
  (+ (* y y) (* (g x) y)))
```

(最初の行の (define y 0) は単に変数 y を用意しているだけで、y の値を 0 にすること自体は重要でなく、値は何でもかまいません。) この方法では確かにかけ算の回数は減っています。ですが f1 が呼び出している手続き g が、一時的な値の保持に y を使っているかもしれません。もしそうならば g が y の値を書き換えてしまい、予期しない結果を得るでしょう。ですのでこの方法は完全ではありません。

以上の問題は、f2 の内部だけでしか使われない変数を用意してやることで解決できます。このように使える範囲を一部だけに限定した変数を**局所変数** (local variable) といいます。

Scheme では局所変数を用意する特殊形式として let, let*, letrec が用意されています。ここでは let と let* について説明します。letrec は 6.4で紹介します。

3.9.1 let による局所変数

局所変数を使えば、関数 f を計算する手続き f は次のようになります。

```
(define (f x)
  (let ((y (* x x)))
    (+ (* y y) (* (g x) y))))
```

let の一般的な形は

◇ (let ((⟨ 変数$_1$⟩ ⟨ 式$_1$⟩)
(⟨ 変数$_2$⟩ ⟨ 式$_2$⟩)
⋮
(⟨ 変数$_n$⟩ ⟨ 式$_n$⟩))
⟨ 本体 ⟩)

です。なお ⟨ 本体 ⟩は、式のならびです。

let の動作は以下の通りです。まず変数⟨ 変数$_1$⟩, ⟨ 変数$_2$⟩,..., ⟨ 変数$_n$⟩の値を保持する場所を確保し、⟨ 式$_1$⟩, ⟨ 式$_2$⟩,..., ⟨ 式$_n$⟩を評価します。そしてそれぞれの値を、 新たに確保された⟨ 変数$_1$⟩, ⟨ 変数$_2$⟩,...,⟨ 変数$_n$⟩の値を保持する記憶場所に代入します。これで局所変数の初期値が決められます。

⟨ 本体 ⟩ の評価のときに変数⟨ 変数$_1$⟩,⟨ 変数$_2$⟩,...,⟨ 変数$_n$⟩が現れれば、それらの値は新しく用意された場所に蓄えられている値が使われます。⟨ 本体 ⟩の一連の式の評価が終ると局所変数とその値の関係は捨てられ、⟨ 本体 ⟩の最後の式の評価結果を let の値とします。

同じ名前を持つ変数が let の外にあっても、同じ名前の変数を影響を与えずに let の中で使うことができます。例を見てみましょう。

```
> (define x 10)
#<unspecified>
> x
10
> (let ((x 7)) (set! x (* x x)) x)
49
> x
10
```

上の例では let の中で x の値を set! によって 49 にしていますが、トップレベル定義の x に影響していない事がわかります。

ほかにもいくつかの例を見てみましょう。

```
> (define x 1)
#<unspecified>
> (define y 3)
#<unspecified>
> (let ((a (+ y 1)) (b (* x 2)))  (+ a b))
6
> (let ((x (+ y 1)) (y (* x 2)))  (+ x y))
6
```

最後の例での局所変数の初期値を決める式 (+ y 1) と (* x 2) の評価では、変数 x と y は let の外での値が使われます。(それらの値はそれぞれの 4 と 2 でです。) このように let は、局所変数の初期値の計算をするのに let の外で定義されている変数の値を使います。

3.9.2 let*による局所変数

let* の一般的な形は

△ (let* ((⟨ 変数$_1$⟩ ⟨ 式$_1$⟩)
　　(⟨ 変数$_2$⟩ ⟨ 式$_2$⟩)
　　⋮
　　(⟨ 変数$_n$⟩ ⟨ 式$_n$⟩))
　　⟨ 本体 ⟩)

です[18]。

[18] この形式での使い方は Scheme に必須な機能とは定められていません。そのために、処理系によっては使えない場合がありますが、多くの処理系では用意されていることが多いようです。

let* は let と似て局所変数を用意します。最初の局所変数の初期値を求めると、その値を次の局所変数の初期値を求めるときに使うことができる点が、let と違うところです。例を見てみましょう。

```
> (define x 1)
#<unspecified>
> (define y 3)
#<unspecified>
> (let* ((x (+ y 1)) (y (* x 2))) (+ x y))
12
```

(+ y 1) を評価すると 4 なので、これが x の初期値となります。次は (* x 2) を評価して y の初期値を決めますが、ここでの x は最初の局所変数が使われます。ですので、y の初期値は 4 となります。

let や let* は入れ子にして使うこともできます。次の式を評価すると、どういう結果が得られるでしょうか?

```
(let ((x 0))
  (let ((x 1))
    x))
```

答えは 1 です。変数は、文面上で一番近い定義による束縛による値が使われますので、(let ((x 1) ··· での定義が使われます。このため、 上の式の評価結果は 1 になります。

3.10 制御構造 (その 1)

これまでの例で学んだ方法では、手続きに単純な計算をおこなわせることしかできません。ここでは場合分けをする方法や、 繰り返し実行をする方法を学びます。

3.10.1 if による場合分け

if はある条件式が真か偽かによって、2 通りの場合分けをする特殊形式です。

次に示す 手続き foo は、与えられたデータ n が 0 ならば記号 zero を、そうでなければ記号 non-zero を返します。

```
(define (foo n)
  (if (zero? n)
      'zero
      'non-zero))
```

実行例は次の通りです。

```
> (foo -1)
non-zero
> (foo 0)
zero
> (foo 68000)
non-zero
```

このように if は

◇ (if ⟨ 条件式 ⟩
⟨ 式$_1$⟩
⟨ 式$_2$⟩)

の形をしています。⟨ 条件式 ⟩ を評価した結果が真 (厳密いうと、#f 以外) なら⟨ 式$_1$⟩ を評価し、その値が if の値になります。そうでなければ ⟨ 式$_2$⟩ を評価し、その値が if の値になります。
⟨ 式$_2$⟩ のない

△ (if ⟨ 条件式 ⟩
⟨ 式 ⟩)

の形も使うことができます[19]。

この場合は ⟨ 条件式 ⟩ を評価した結果が真 (厳密いうと、#f 以外) ならば ⟨ 式 ⟩ を評価し、その値が if の値になります。そうでなければ if 返される値は定められていません[20]。

3.10.2 cond による場合分け

特殊形式 cond は、複数の条件式をならべておき、真であるものに対応した式が選ばれ評価され、その値が返されます。

次の例を見てください。

```
(define (score n)
  (cond ((<= n 50) 'fail)
        ((<  n 70) 'poor)
        ((<  n 80) 'fair)
        (else      'excellent)))
```

[19] この形式での使い方は Scheme に必須な機能とは定められていません。そのために、処理系によってはこの使い方ができない可能性もありますが、ほとんどの処理系では用意されています。

[20] どのような値が返されるかは処理系に依存します。

この手続き score は、引数 n が

- 50 以下なら fail を、
- 51 以上 70 未満なら poor を、
- 70 以上 80 未満なら fair を、
- 80 以上であれば excellent を

返す手続きです。

実行例を見てみましょう:

```
> (score 30)
fail
> (score 67)
poor
> (score 79)
fair
> (score 96)
excellent
```

一般に cond は、以下の形をしています。

◇ (cond ⟨ 節$_1$⟩
⟨ 節$_2$⟩
⋮
⟨ 節$_n$⟩)

つまり cond は、**cond-節** (cond-clause) と呼ばれる ⟨ 節 ⟩ のならびです。それぞれの ⟨ 節 ⟩は (⟨ 条件$_1$⟩ ⟨ 式$_1$⟩ ···) の形をしています。

上から順に節の条件部 ⟨ 条件 ⟩ が評価され、その結果が真 (厳密には #f 以外) ならその節の⟨ 式 ⟩··· が評価され、最後の式の評価結果が cond の値になります。もし述語部が #f であれば次の節に移り、上と同様なことを続けます。

一番最後の節の ⟨ 条件$_n$⟩ には、 else を使うこともできます。どの述語も偽であった場合に、この節の式 ⟨ 式$_n$⟩ ··· を評価し、この最後の式の評価結果を cond の評価結果とします。

cond では、節が書かれている順番に節の条件部が評価される、ということに十分注意してください。たとえば、上の例の節の順番を変えた手続き score2 を見てください。

```
(define (score2 n)
  (cond ((<  n 80) 'fair)
```

```
        ((<  n 70) 'poor)
        ((<= n 50) 'fail)
        (else      'excellent)))
```

では、これを実行してみます。

```
> (score2 30)
fair
```

テストで 30 点をとってしまったにも関わらず成績は fair になり、上での場合と結果が違っています。これは最初の節の条件が真なのでこういう結果になりました。

3.10.3 begin による式の逐次評価

特殊形式 begin は、ならんだ複数の式を順に評価します。begin の評価結果は、最後の式の評価結果となります。たとえば次のようになります。

```
> (define x 10)
#<unspecified>
> (define y 3)
#<unspecified>
> (begin (display "10 * 3 = ") (display (* 10 3)) (newline))
10 * 3 = 30
#<unspecified>
```

となります。ここで手続き display は、引数を表示する手続きです。また手続き newline は、カーソル位置を次の行に移す (改行する) 手続きです。

一般に begin は、

◇ (begin ⟨ 式$_1$⟩
⟨ 式$_2$⟩
⋮
⟨ 式$_n$⟩)

の形をしていて、⟨ 式$_1$⟩,⟨ 式$_2$⟩ ···⟨ 式$_n$⟩の順番で式が評価されます。そして⟨ 式$_n$⟩ の評価結果が begin の値として返されます。

このように begin は、その本体にならんでいる式をすべて評価しますが、最後の式の評価結果以外は使われません。その本体にならんでいる式の評価結果そのものよりも、

- 画面にデータを表示する
- 変数に値を代入する

などをする式を順次実行するために使うのが主な目的です。式の評価で得られる結果の以外の動作を**副作用** (side effect) といいます。たとえば `set!` はその式が返す値が目的ではなく、変数の値を書き換えるという動作を目的として使っていますし、手続き `display` は画面への表示を目的として使っています。

3.10.4 and と or による式の逐次評価

特殊形式 `and` と `or` は式のならびを順に評価してゆき、ある条件が成立すると途中で終了します。

特殊形式 `and` は一般に、

◇ (and ⟨式$_1$⟩
⟨式$_2$⟩
⋮
⟨式$_n$⟩)

の形をしています。この特殊形式の実行は次のようにしておこなわれます。まず ⟨式$_1$⟩ を評価します。もし式の評価結果が真 (厳密には `#f` 以外) ならば、次の式 ⟨式$_2$⟩ の評価をおこないます。これを、式の評価結果が偽になるまで続けます。

もし最後の ⟨式$_n$⟩ まで評価結果が真ならば、最後の式の評価結果が `and` の値として返されます。もし途中で評価結果が偽のものがあると、そこで `and` の実行を終了して、`#f` が `and` の値として返されます。

例を見てみましょう。

```
> (define x 10)
#<unspecified>
> (and (number? x) (> x 0) (+ x 1))
11
```

この例では (`number? x`) を評価すると真で、次の (`> x 0`) の評価結果も真です。最後に (`+ x 1`) が評価され、その結果は `11` となり、これは偽ではないのでその値が `and` の値として返されています。

では別の例を見てみましょう。

```
> (and (number? x) (> x 100) (+ x 1))
#f
```

この場合では (`> x 100`) が偽なので、その時点で `and` の実行が打ち切られます。そして `#f` が `and` の値なっています。

`and` が特殊形式であることに注意してください。(単に論理積を計算するのではありません。) もし特殊形式でなくて普通の手続きと同じように、実行の前にすべての引数が評価

された場合を考えましょう。上の例で変数 x が数ではなくて記号だったら、引数評価のとき、(+ x 1) の評価をしようとして、記号に 1 を加えることになってエラーになります。

特殊形式 or は一般に、

◇ (or ⟨式$_1$⟩
⟨式$_2$⟩
⋮
⟨式$_n$⟩)

の形をしています。特殊形式 or の実行は次のようになります。まず ⟨式$_1$⟩ を評価します。もし評価結果が偽 #f ならば、次の式 ⟨式$_2$⟩を評価します。同様にして、評価結果が偽 #f ならばさらに次の式を評価します。これを評価結果が偽である限り続けてゆきます。もし最後まで評価結果が偽なら、or の値として偽が返されます。もし途中で評価結果が偽以外のものがあれば or の実行を終了し、その評価結果が or の値として返されます。or も単なる論理和を計算するのではないことに注意しましょう。

例を見てみましょう。

```
> (define x -20)
#<unspecified>
> (if (or (< x 0) (> x 100))  "x < 0 or 100 < x" "else")
"x < 0 or 100 < x"
```

変数 x の値は -20 なので、この例では (< x 0) が真となり、この値が or の値として返されます。

```
> (if (or (= x 0) (> x 0))  "x >= 0" "x < 0")
"x < 0"
```

この例では、 or に現れるどの式も評価結果が偽なので or の値は#f となり、if により "x < 0" が返されています。

3.11 再帰

以上までは手続きの始めから終りの方に向かって、式を順々に実行していました。もっと複雑な計算をするには、「繰り返し」 をする方法が必要です。Scheme で繰り返しをする基本的な方法は**再帰** (recursion) です。

注意: Pascal, C, Fortran などのプログラミング言語をすでに習得している人にとっては、再帰による繰り返し実行はなじみ難いかも知れません。Pascal などの FOR 文や Fortran

の DO 文に似た、ループのための構文が Scheme にも用意されています。興味がある人は 10.1.1 を参照して下さい。

再帰 (recursion) は、Scheme での繰り返し実行のための基本的な概念です。次に示す階乗の計算をする手続き `fact` を見てください。

```
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

階乗は 1 から n までの全ての整数をかけあわせたものです。すなわち、 定義は $n! = 1 \cdot 2 \cdots (n-1) \cdot n$ となっています。`(fact 4)` が上の手続きによってどのように計算されるか、計算過程を見てみましょう。

```
(fact 4)
⇒ (* 4 (fact 3))
⇒ (* 4 (* 3 (fact 2)))
⇒ (* 4 (* 3 (* 2 (fact 1))))
⇒ (* 4 (* 3 (* 2 (* 1 (fact 0)))))
⇒ (* 4 (* 3 (* 2 (* 1 1))))
⇒ (* 4 (* 3 (* 2 1)))
⇒ (* 4 (* 3 2))
⇒ (* 4 6)
⇒ 24
```

これをわかりやすい形で書けば、

- `(fact 4)` は 4 に `(fact 3)` をかけたもの.
- `(fact 3)` は 3 に `(fact 2)` をかけたもの.
- `(fact 2)` は 2 に `(fact 1)` をかけたもの.
- `(fact 1)` は 1 に `(fact 0)` をかけたもの.
- `(fact 0)` は 1 である.
- よって `(fact 1)` は $1 \cdot 1 = 1$ である.
- よって `(fact 2)` は $2 \cdot 1 = 2$ である.
- よって `(fact 3)` は $3 \cdot 2 = 6$ である.

- よって (fact 4) は $4 \cdot 6 = 24$ である.

という過程で計算されています。

このようにある手続きが計算するのに、さらにその手続き自身を呼び出す方法を**再帰** (recursion) といいます[21]。

別の例として、リストの長さを求める手続きを作ってみましょう。リスト l の長さは、次のようにして定義されます。

- l が空リストなら、lの長さは 0 です。
- そうでなければ、(cdr l) の長さに 1 加えたものが、l の長さです。

これを手続きにすると、次のようになります。

```
(define (len l)
  (if (null? l)
     0
     (+ (len (cdr l)) 1)))
```

この定義によると、l が空リストでないときは (cdr l) を引数として、再帰的に呼び出しをしています。ですが再帰呼び出しのたびに l は短いリストになってゆきますので、いつかは空リストになり、(null? l) の条件が満たされます。そのため、再帰呼び出しが永遠に繰り返されることはありません。

ではもういちど階乗を計算する手続き fact を見てみましょう。この計算法で (fact n) の計算をするのに、(fact $n-1$) の計算を終了した後にその計算結果と n をかけるようになっています。ですので、(fact $n-1$) の計算した後にまた戻ってきて n をかけないといけない、ということを覚えておく必要があります。この計算法では再帰呼び出しをするたびに、戻ってきた後にすべき仕事を記憶する必要があります。(fact 5) の計算過程で括弧の入れ子が深くなってゆくのはこのためです。このことは、実行のために多くの記憶領域を必要とすることを意味します。

階乗の計算は単に 1 から n までを順々にかけ合わせればいいので、あとで n をかける、という「戻り情報」は覚えなくてもよいはずです。この考えに基づいた計算方法による階乗計算をする手続き fact2 の定義を示します。

```
(define (fact2 n)
  (define (fact-sub n f)
    (if (= n 0)
      f
      (fact-sub (- n 1) (* f n))))
  (fact-sub n 1))
```

[21]手続き f が直接それ自身を呼び出さずに、別の手続きを g を呼び出し、g が f を呼び出す場合も再帰といいます。このような場合は特に、**相互再帰** (mutual recursion) といいます。

(fact2 4) の計算過程は、次のようになります。

```
(anonther-fact 4)
⇒ (fact-sub 4 1)
⇒ (fact-sub 3 4)
⇒ (fact-sub 2 12)
⇒ (fact-sub 1 24)
⇒ (fact-sub 0 24)
⇒ 24
```

fact2 の内部で定義されている手続き fact-sub では、その第 1 引数 n が 0 以外であれば、再帰的に fact-sub を呼び出しています。ですが呼び出し側は、再帰呼び出しで得られた値を別の手続きの引数にすることはせず、呼び出した手続きの値をそのまま計算結果として返しています。このために、再帰呼び出しされた手続きは計算結果を持って呼出し側に戻る必要はありません。Scheme では、このような場合には「呼び出し」の代わりに「分岐」を使い、戻り情報を保持しないようにしています。

fact2 のように、プログラムの文面上では手続きの再帰呼び出しでも、実際の手続き呼び出しの際には「分岐」がおこなわれます。そのため、戻り情報を記憶しないので、n の値に関わらず、一定の記憶領域しか必要としません。

手続き呼び出しがあるたびにプログラムの戻り場所を**スタック** (stack) と呼ばれる記憶場所に蓄えます。手続きの実行が終了したら戻る場所をスタックから取り出し、計算結果を持って戻り場所に分岐します。そのため手続きの中でさらに手続きを呼び出すと、スタックにはどんどん戻り場所情報が蓄積されます。これは記憶領域をどんどん消費してゆくことを意味します。ですがある手続き f がその中で手続き g を呼び出していて、g の返す値をそのまま f の値として返す場合を考えましょう。g を呼び出すときに戻り場所をスタックに蓄えず、実行を g に移します。g が終了するときにスタックから取り出される戻り場所は、f の呼び出しに対する戻り場所となります。このため g が終了すると f に戻ることはなく、f の呼び出しに対する戻り場所に戻ることになります。しかも得られた値は、戻り情報をすべてスタックに蓄えていたときとまったく同じです。このように他の手続きを呼び出しその計算結果をそのまま返す場合は、戻り情報を保持しないことでスタックの使用 (記憶領域の消費) をしなくて済みます。

このため手続き fact の実行には、引数 n に比例した数の戻り情報をスタックに保持する必要があります。(再帰の入れ子が一番深い時にもっとも多くのスタックを使います。) いっぽう手続き fact2 では、評価のときに使う作業領域は引数の値とは関係なく、一定の領域しか使いません。このような性質は**末尾再帰的** (tail recursive) であると呼ばれます。

このことを分かりやすい例で説明すれば、次のようになります。あなたはある町工場で働いている社員だとしましょう。A 社から依頼されたある機械の製作を、下請けの B 社に依頼したとします。ある日、B 社から機械が完成したとの依頼を受けました。製品検査や機械の調整が必要なら、B 社に機械を受け取りに行き、自分の会社に運び、そして A 社に納品することになります。ところがあなたが B 社の技術を信用していて製品検査は必要ないと思えば、わざわざ自分の会社に運ぶ必要はありません。B 社に機械を受け取りに行き、その足で直接 A 社に納品すればいいでしょう。営業担当の人が、得意先廻りで夜になったときに、特に何もなければ会社に戻らずそのまま自

宅に帰ったりします。ここでも無意識のうちに「末尾再帰」の考えが使われています。

3.12 プログラム作成に関連した手続き

Scheme プログラムをテキストエディタで書いても、それは単なるディスクファイルでしかありません。それを実行するには、Scheme 処理系に読み込ませ、実行したい手続きを評価することが必要です。(ファイルに書かれているプログラムを読み込むことを、**ロード** (load) といい、すでに学びました。) この他にも、プログラムを作成するときに便利な機能が用意されています。

◇ (load ⟨ ファイル ⟩)
— ⟨ ファイル ⟩ で指定された Scheme プログラムが書かれているファイルを読み込みます。⟨ ファイル ⟩ は文字列でないといけません。

△ (transcript-on ⟨ ファイル ⟩)
— この式が評価されると、それ以降の画面への出力が、⟨ ファイル ⟩ にも書き込まれます。⟨ ファイル ⟩ は文字列でないといけません。

△ (transcript-off)
— transcript-on によって開始された、画面表示のファイルへの書き込みを終了します。

Scheme 処理系によっては transcript-on と transcript-off が用意されていない場合があるので、注意して下さい。

load は、ファイルに書かれている式をひとつづつ読み込み、それを評価してゆきます。これは ⟨ ファイル ⟩ に書かれている内容を、手で入力したのと同じ効果を持ちます。もしファイルの中に Scheme プログラムの間違いがあるとエラーとなります。

ファイルに書かれている式をひとつづつ評価してゆくことを利用して、ロードすると自動的にプログラムを実行させることもできます。これは単に、一連の手続きを書いた後に、それらを呼び出す式をロードするファイルに書くだけです。

次のものが、ファイル sin-table.scm に書かれているとします。

```
;; sqrt-table.scm
;; --- prints values of square root of 1...10.
(define (print-sqrt-table from to)
  (print-sqrt-table2 from to))
(define (print-sqrt-table2 i to)
  (if (> i to)
    'done
    (begin
      (display i) (display "   ") (display (sqrt i))
      (newline)
      (print-sqrt-table2 (+ i 1) to))))
(print-sqrt-table 1 10)
```

これをロードすると (print-sqrt-table 1 10) の評価によって、次のような出力を得ます。

```
> (load "sqrt-table.scm")
;loading "prog/sqrt-table.scm"
1   1.0
2   1.4142135623731
3   1.73205080756888
4   2.0
5   2.23606797749979
6   2.44948974278318
7   2.64575131106459
8   2.82842712474619
9   3.0
10   3.16227766016838
;done loading "sqrt-table.scm"
;Evaluation took 14 mSec (0 in gc) 216 cells work, 212 bytes other
#<unspecified>
>
```

手続き transcript-on を実行し、いくつかの式を評価した後に、手続き transcript-off を実行してみます。

```
> (transcript-on "LOG-FILE")
;Evaluation took 0 mSec (0 in gc) 6 cells work, 37 bytes other
#<unspecified>
> (+ 1 2 3)
;Evaluation took 0 mSec (0 in gc) 11 cells work, 37 bytes other
6
> (* 1 2 3 4 5 6 7 8 9 010)
;Evaluation took 0 mSec (0 in gc) 32 cells work, 51 bytes other
3628800
> (transcript-off)
#<unspecified>
```

ファイル LOG-FILE には、transcript-on から transcript-off までの間の入力と表示が書き込まれています:

```
;Evaluation took 0 mSec (0 in gc) 6 cells work, 37 bytes other
#<unspecified>
> (+ 1 2 3)
;Evaluation took 0 mSec (0 in gc) 11 cells work, 37 bytes other
6
> (* 1 2 3 4 5 6 7 8 9 010)
;Evaluation took 0 mSec (0 in gc) 32 cells work, 51 bytes other
3628800
> (transcript-off)
```

このように、transcript-on と transcript-off を使えば、Scheme 処理系との対話がファイルに保存されます。プログラムの実行結果を人に見せる必要があるとき (たとえばプログラミング実習で、実行結果をレポートと一緒に提出する時など) に重宝します。

まとめ

本章では、Scheme の基本を学びました。基本的とはいっても、これだけでもかなり複雑なプログラムが作成できます。Scheme は、不合理な制約がなく、覚えるべき約束事も他のプログラミング言語よりも少ないプログラミング言語です。これまででは、一部の特殊形式 (構文) や手続きしか紹介しませんでしたが、第 6章では、もう少し高度な Scheme 文法を説明しています。また、Appendix B の Scheme の言語仕様書で定義されている特殊形式と手続きの表も参考にして下さい。

しかしいづれにしても異人の言葉を覺えなくてはいけないと腹を定め、宿直の水夫たちを師匠にして口のききかたを教わつた。年のせゐか、傳藏が一ばん物覺えが惡く、萬次郎は一人とびぬけてよく覺えた。

井伏鱒二 「ジョン万次郎漂流記」
『ジョン万次郎漂流記』収録 昭和二十二年 文學界社刊

第 4 章

NGSCM の基礎的な使い方

この章ではエディタについての基礎概念を最初に学びます。ここで書かれていることは Mule でもほぼ同じです。それをマスターすれば、基本的なプログラム作成と実行ができます。操作に慣れないうちは、 Appendix にある NGSCM の編集コマンドの一覧をコピーして手元に置いておくと便利でしょう。

基本的な操作法を完全にマスターした人のために、さらに高度な使用法を第 7章 と第 8 章で紹介していますので、参考にして下さい。

4.1 エディタの概念

ここでは, **エディタ** (editor) についての基礎知識を学びます。第 1章で学んだように、文書やプログラムなどは、ファイルとしてディスクに記録します。ファイルの内容を変更したりするソフトウエアがエディタです。

毎日のように文書作成やプログラム作成をする人にとっては、ワードプロセッサやエディタを使っている時間は相当なものです。エディタの上手な使い方や便利な機能をマスターすれば、毎日の作業がずいぶんと楽になることは想像つくと思います。

ではエディタの基本概念の説明に入ります。ここで説明することはほとんどすべてのエディタにあてはまります。図 4.1 を参照してください。編集の対象となるディスク上のファイルを、主記憶装置の一部にコピーします。ファイルの内容を保持している部分を、**編集バッファ**(editing buffer) と呼びます。(編集バッファは、**バッファ**(buffer) とも呼ばれます。以降、本章ではバッファという言葉を使います。) バッファに保持されているプログラムや文書を総称して、**テキスト** (text) とも呼びます。

書き換えや削除などの編集作業は、このバッファに蓄えられているファイルのコピーに対して行ないます。そのため、編集作業をすることにより、ファイルの内容とバッファの内容が違うことになります。ファイルを書き換えた結果を記録するには、変更したバッファをファイル書き戻さないといけません。この動作を**セーブ** (save) と呼びます。いいかえれ

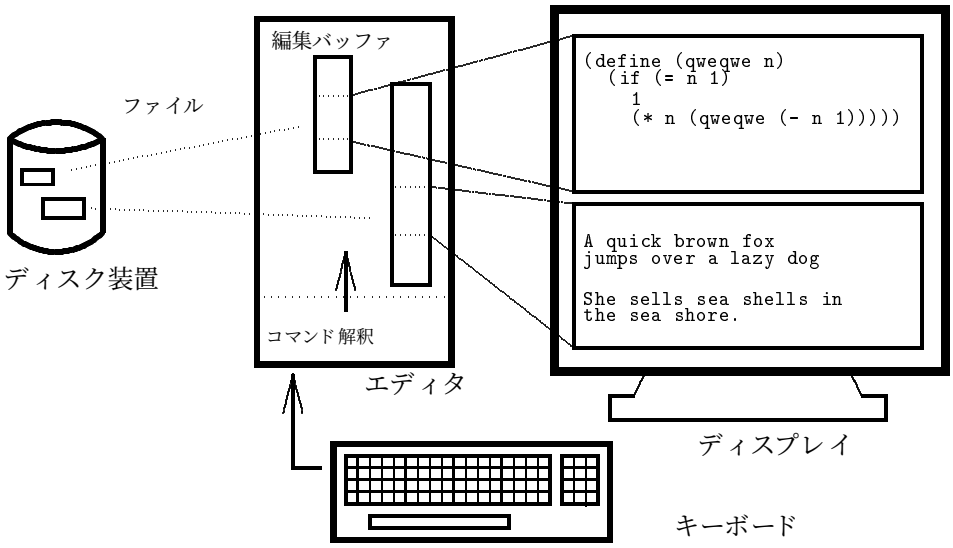


図 4.1: エディタの概念図

ば、いくら編集作業をしてもセーブしなければ、編集作業は意味を持ちません。

多くエディタでは、いくつものバッファを同時に使用することができます。この機能により、あるファイルの一部を別のファイルにコピーしたり、あるファイルの内容を参考にしながら別のファイルを変更する、といったことも可能になります。

バッファのうちのいくつかは画面に表示されます。画面は、ひとつあるいは複数の**ウインドウ** (window) より構成されます。それぞれのウインドウには、ひとつのバッファが対応付けられています。そしてウインドウに対応したバッファの内容が、そのウインドウに表示されます。(エディタの中には複数のウインドウを開くことができないものや、ひとつしか編集バッファを持てないものもあります。)

ウインドウごとに、編集作業の「注目場所」である**カーソル** (cursor) があります。文字の挿入や削除などの編集作業は、カーソルのある位置で適用されます。長いファイルの場合、ウインドウの範囲に収まらないことがあります。そのような場合、カーソルの周辺のバッファの内容が、ウインドウに収まる範囲で表示されます。ウインドウに表示されていない部分を見るには、カーソルを移動させることで見ることができます。

4.2 NGSCM の画面の構成

NGSCM の画面は、いくつかの部分より構成されています。

4.2.1 エコー領域

画面の一番下にある 1 行は**エコー領域**と呼ばれる部分です。エコー領域は、NGSCM からのメッセージを表示したり、利用者がファイル名などを入力するのに使われます。図 4.2

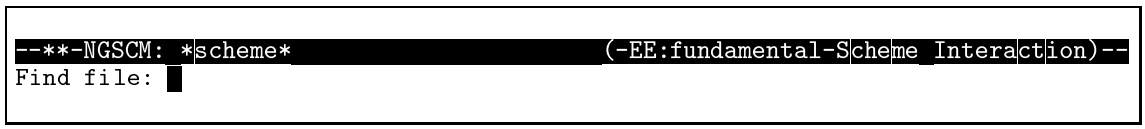


図 4.2: エコー領域 (ファイル名の入力をしているところ)

は、エコー領域でファイル名の入力をしているところです。なおこの図での白黒反転した行は、後で説明するモード行です。

4.2.2 モード行

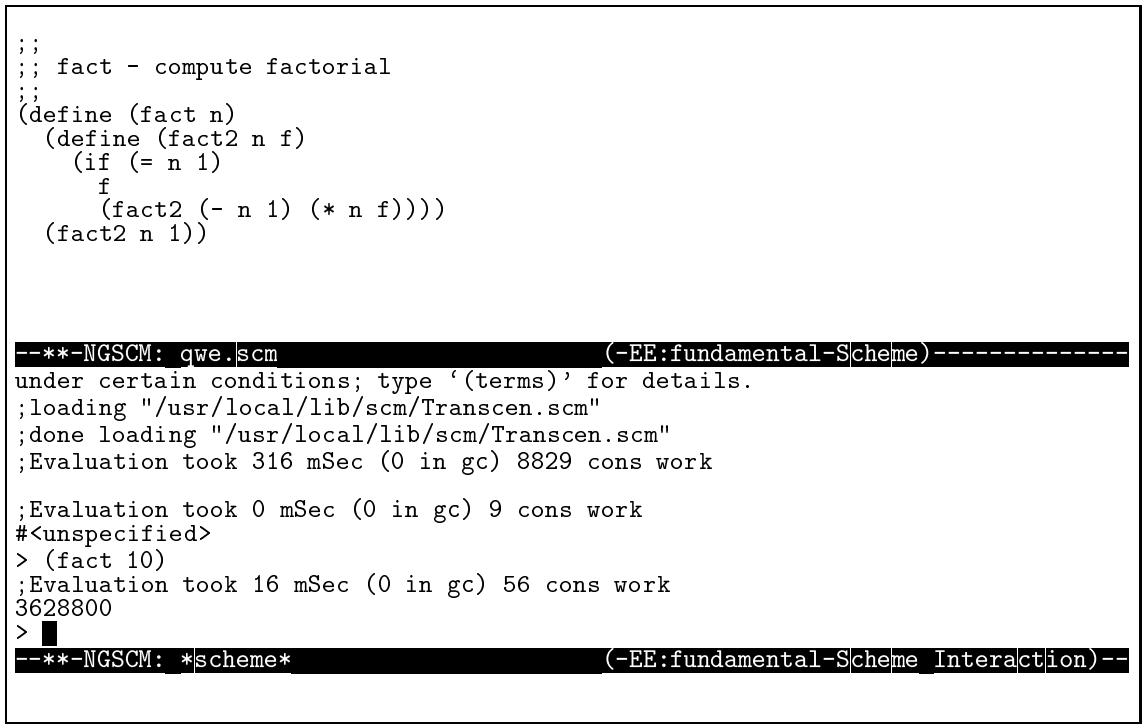


図 4.3: NGSCM の画面例

NGSCM では複数のウインドウを開いて、それぞれのウインドウで違ったバッファを表示できます。図 4.3での例では、2 つのウインドウが開かれています。上のウインドウでは

図 4.4: モード行

バッファ qwe.scm が、下のウインドウではバッファ *scheme* が表示されています。それぞれのウインドウの下にある白黒反転した行は、**モード行** (mode line) と呼ばれるものです。モード行には、そのウインドウが表示しているバッファの情報が書かれています。

モード行に表示されている情報には、次のものがあります。

- ファイルを変更したかどうか
  一番左の部分は、バッファに書換えを行なったかどうかを表しています。--**-NGSCM: となっているときは、そのバッファで編集しているファイルが書換えられてから、まだセーブがされていないことを表してます。-----NGSCM:となっているときは、まだ書き換えられていないことを表しています。

- バッファの名前
  NGSCM:の右には、バッファの名前が表示されています。図 4.3の例では、qwe.scm (上のウインドウ) と *scheme* (下のウインドウ) の、2 つのバッファがあります。

- 使用している漢字コード
  ファイル名の右にある (-EE: は、使用している漢字コードを表していますが、詳しいことは省略します。

- バッファのモード
  漢字コード表示のすぐ右にあるのが、バッファのモードの表示です。NGSCM や Mule では、モードごとに特有な編集コマンドを提供する機能があります。上のウインドウでは fundamental-Scheme となっています。これは、基本的な編集機能を用意した fundermental (基本) モードに加えて、 Scheme プログラムの編集に便利な機能を提供する Scheme モードが組み合わされていることを表しています。

4.3 NGSCM の起動と終了

NGSCM には、いくつかの起動方法があります。

4.3.1 ファイル名を指定した起動

次のようにして、 NGSCM を起動します。

```
% ngscm ファイル名 RET
```

ここで % は Unix のシェルのプロンプトです。

たとえば

```
% ngscm sum.scm RET
```

とすれば、 sum.scm というファイルを編集するために NGSCM が起動されます。指定したファイルがすでにある場合は、そのファイルを読み込みます。もし指定したファイルがない場合は、新しいファイルを作ることになります。

4.3.2 ファイル名を指定しない起動

次のようにすると、ファイルの読み込みはせずに NGSCM を起動できます。

```
% ngscm RET
```

NGSCM を起動した後でも、C-x C-f により、ファイルを読み込んで編集することができます。詳しくは、7.1.5 を参照して下さい。

4.3.3 終了の方法

NGSCM を終了するには、 C-x C-c を入力します。(C-x C-c は、CTL キーを押しながら x を押し、そして次に CTL キーを押しながら c を押すことを表しています。)

バッファを書換えたら、NGSCM はセーブするかどうかを尋ねてきます。

```
Save file /home/pooh/sum.scm? (y or n)
```

ここで y を押せば、ファイルがセーブされます。もし n を押すと、セーブせずに NGSCM を終了します。(なお /home/pooh/sum.scm は編集していたファイル名で、利用者や使用環境によって違う表示の場合があります。)

セーブをしないと編集結果は失われますので、十分に注意して下さい。(編集作業に大失敗をしたときは、セーブをせずに NGSCM を終了してください。セーブしていないので、ファイルは作業前の状態のままです。)

4.4 簡単な使い方

作ったプログラムを保存するには、ファイルにプログラムを書く必要があります。プログラムファイルに作成には、NGSCM の持つ編集機能を使います。ここでは、プログラムの作成と実行に最低限必要なものを、例を通して学びます。(初心者の人へ: この部分を学んだら、本章の残りと次の第8章を読み飛ばして結構です。高度な利用方法は、 Scheme プログラミングに慣れてきてからで十分です。)

任意に与えられた数 n に対して、1 から n までの総和を計算する Scheme プログラムを例として取り上げます。そしてそのプログラムをファイルに作成し、実行します。

1 から n までの総和は、次のようにして計算できます。

- $n = 1$ のとき
  1 が総和です。

- それ以外のとき
  $n-1$ までの総和に n を足したものが、総和です。

これを数学的に書くと、 次のようになります。

$$\mathrm{sum}(n) = \begin{cases} 1 & n = 1 \text{ のとき} \\ \mathrm{sum}(n-1) + n & \text{その他のとき} \end{cases}$$

以上の計算方法を Scheme のプログラムにすると、次のようになります。

```
(define (sum n)
  (if (= n 1)
    1
    (+ (sum (- n 1)) n)))
```

上で示したプログラムを、sum.scm という名前のファイルに作ることにします。ファイル名の最後の部分 .scm は**拡張子**(かくちょうし) といい、ファイルの内容が Scheme プログラムであることを表しています。.scm をつける必要は必ずしもありませんが、ファイル名を見ただけでファイルの内容が Scheme プログラムだと分かるので、.scm をつけるようにしましょう。

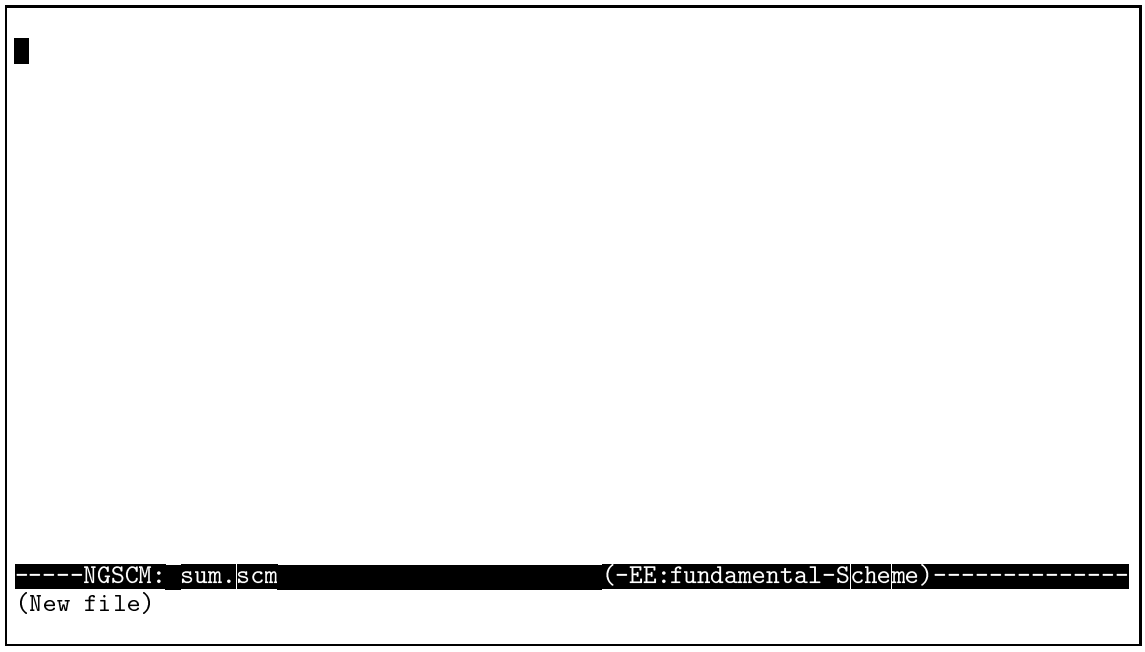


図 4.5: ngscm sum.scm 実行直後の画面

では、NGSCM を起動します。

% ngscm sum.scm RET

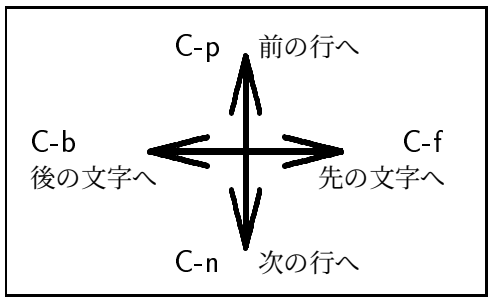


図 4.6: カーソルの移動

すると画面は、図 4.5のようになります。この画面のモード行には、`sum.scm` と表示されています。これは、ファイル `sum.scm` のバッファであることを表しています。エコー領域では `(New file)` と表示されています。ファイル `sum.scm` がないので、ファイルを新しく作ることを表しています。

プログラムの入力を始める前に、基本的な編集機能を説明しておきます。NGSCM では、基本的には押したキーの文字が入力されます。間違った入力をした場合の修正やカーソルの移動のために、いろいろな編集コマンド (編集のための動作) があります。

編集コマンドの多くは、**コントロールキー** (control key) を押しながらアルファベットキーなどを同時に押すことで実行できます。以下のものが基本的な編集コマンドです。

- C-b
  カーソルを 1 文字後に移動します (b は backward character の b)
- C-f
  カーソルを 1 文字先に移動します (f は forward character の f)
- C-p
  カーソルを前の行に移動します (p は previous line の p)
- C-n
  カーソルを次の行に移動します (n は next line の n)
- C-d
  カーソルの下の 1 文字を削除します (d は delete character の d)
- DEL
  カーソルの前の 1 文字を削除します
- RET
  改行します

カーソルの移動とキーの関係を、図 4.6に示します。実際に試してみて、使い方を習得してください。

以上で準備ができました。先ほどのプログラムの入力をして下さい。(図 4.7参照)

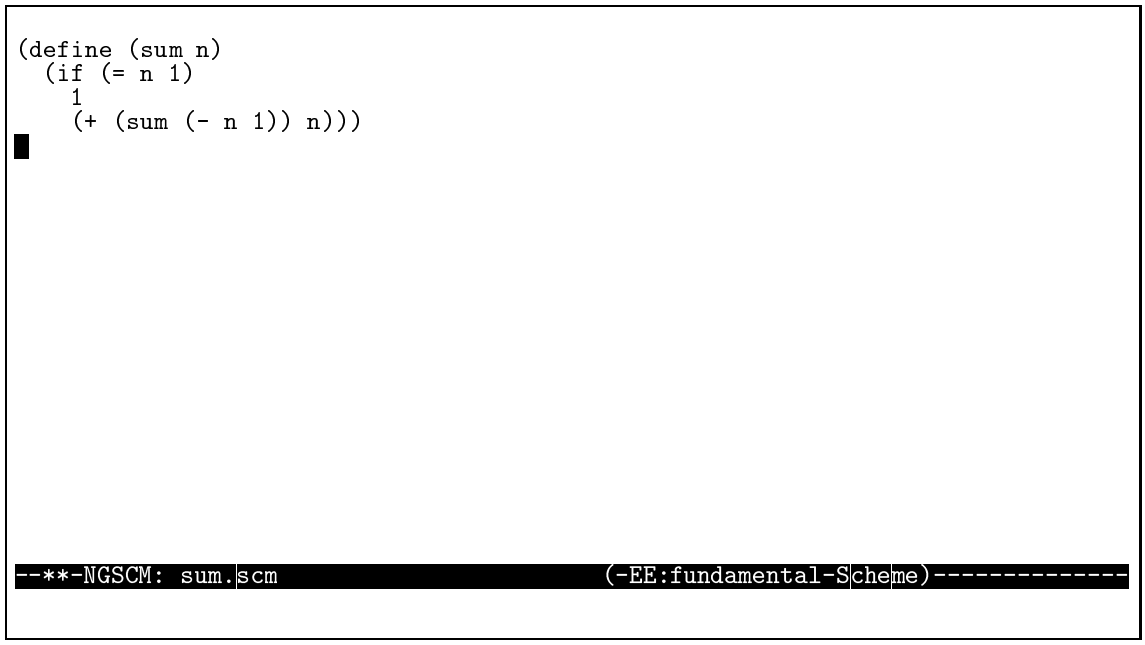


図 4.7: 手続き sum の入力

閉じ括弧) を入力すると、対応する開き括弧 (が一瞬点滅します。Scheme プログラムには括弧が多いですが、この機能のおかげで開き括弧と閉じ括弧の対応の勘違いを避けられます。

つぎに、いま書いたばかりのプログラムを実行させます。まず C-x 2 を入力してください。すると画面は、図 4.8のように 2 分割されます。図 4.8のように、カーソルを (define ...) の式の直後に移動して下さい。ここで C-c C-e を入力します。この C-c C-e は、カーソルの直前の式 (今の場合では (define (sum n)···) です) を Scheme に与えて評価 (実行) せよ、という編集コマンドです。これにより、Scheme にて手続き sum が定義されました。この C-c C-e を実行した直後の画面が図 4.9です。画面の上半分が作成中のプログラムを表示するウインドウ (表示部分のことをウインドウと呼びます) で、下半分が Scheme 処理系との対話に使われるウインドウです。

では、定義した手続き sum が正しく動作するか試してみましょう。まず、カーソルを他のウインドウに移動させる編集コマンド C-x o により、カーソルを Scheme のウインドウに移動させます。(図 4.10)

このウインドウで評価したい式を入力し最後に C-j を入力すれば、その式の評価ができます。では式 (sum 10) を評価してみましょう。(sum 10) を入力し、最後に C-j を忘れないで下さい。プログラムが正しければ、図 4.11のようになるでしょう。

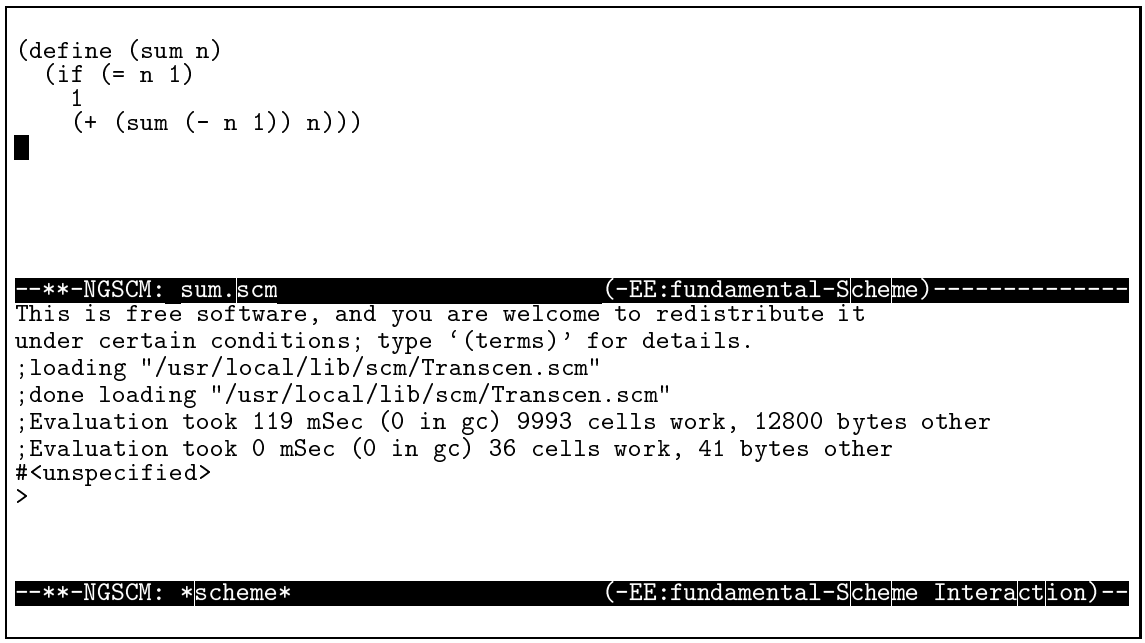


図 4.8: C-x 2 によるウインドウの 2 分割

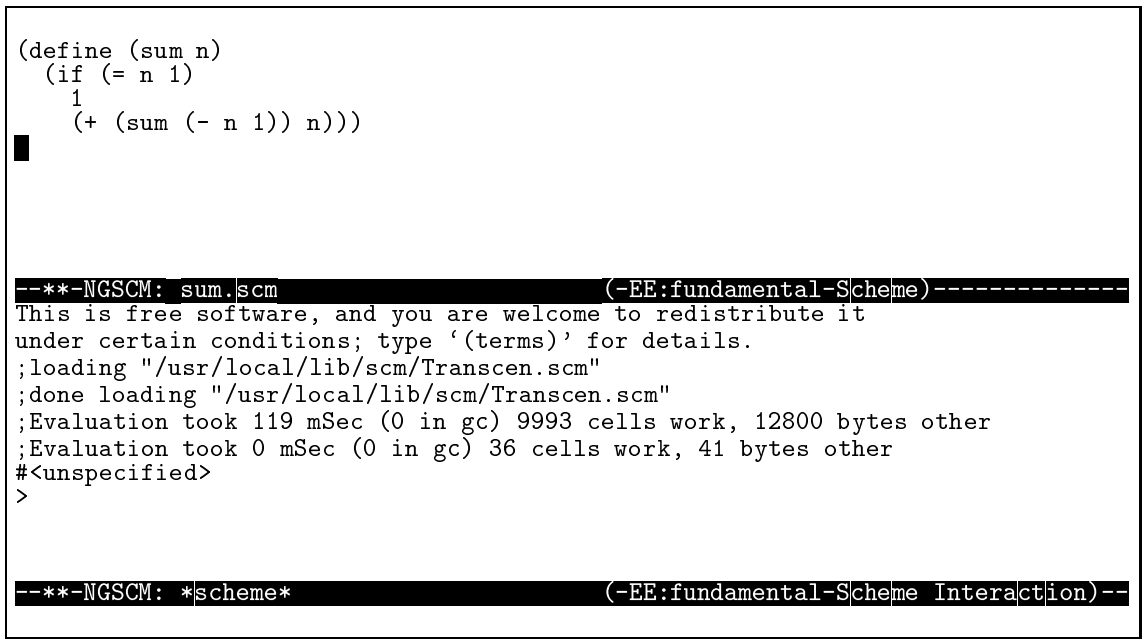


図 4.9: C-c C-e による式の評価

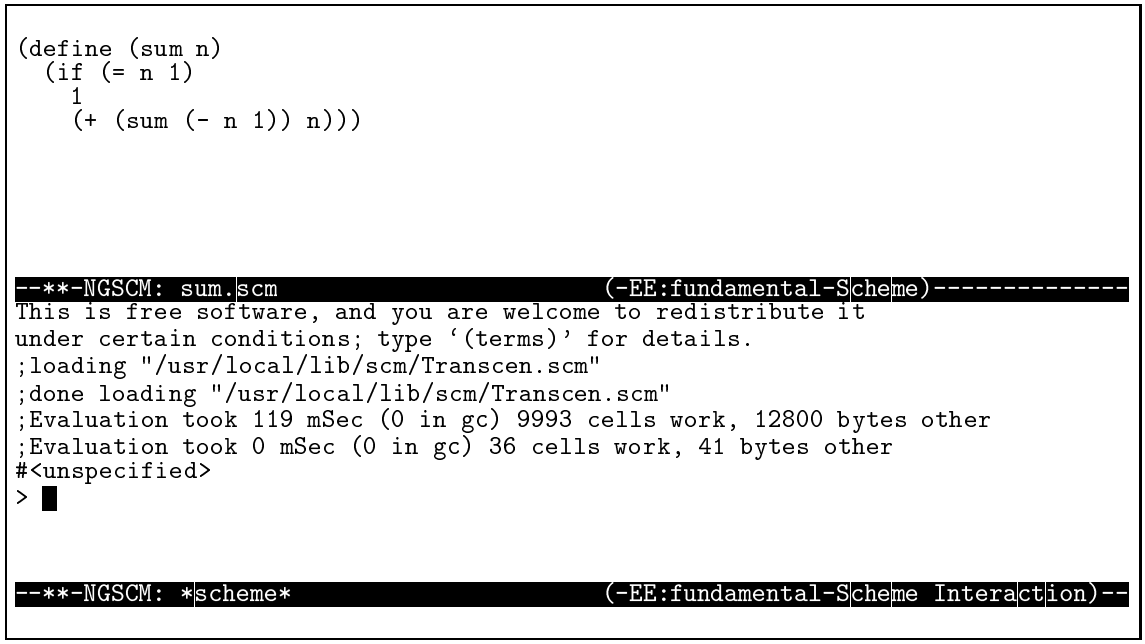


図 4.10: C-x o によりカーソルを Scheme ウインドウに移す

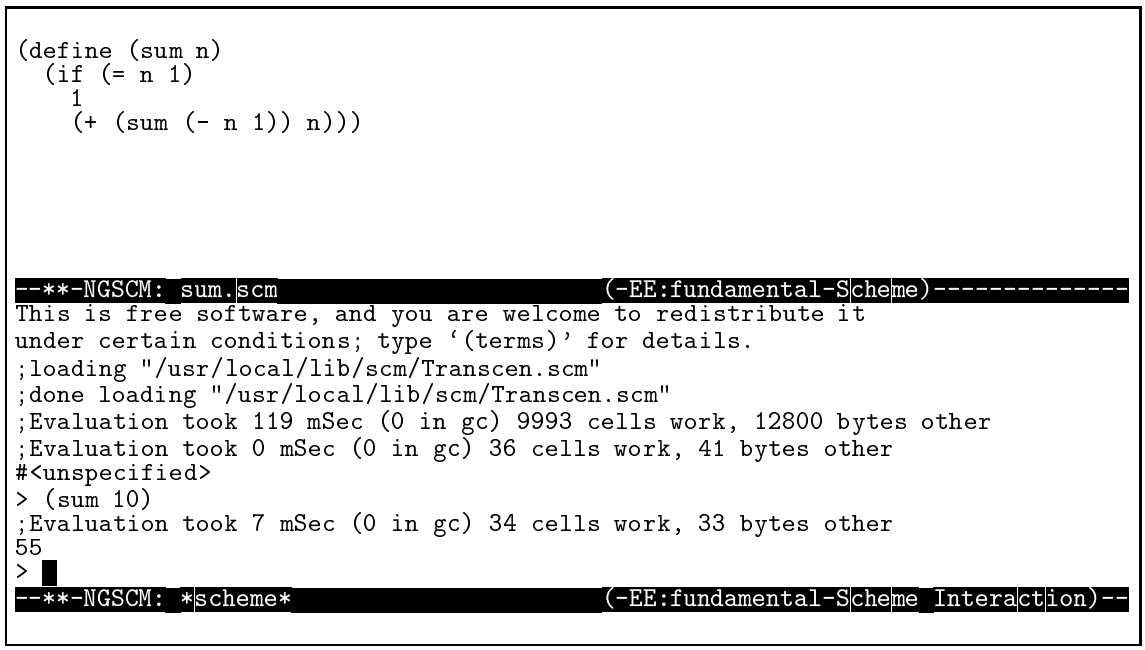


図 4.11: (sum 10) の評価

もしそのようにならなければ、プログラムに間違いがあります。プログラムの修正は、次のようにします:

1. C-x o により、カーソルをプログラムのウインドウに移動させます。
2. 間違った部分を修正します。
3. 修正した式の直後にカーソルを移動させ、C-c C-e により、変更された式 (手続き) を Scheme に与えて、定義し直します。
4. C-x o によりカーソルを Scheme ウインドウに移動させ、(sum 10) C-j を入力し、実行させます。

以上のことをプログラムが正しく動作するまで繰り返します。プログラムが正しく動くようだったら、10 以外にも色々な数を与えて計算させてみましょう。

以上でプログラムの作成ができ、そのプログラムを実行し、正しく動いていることが分かりました。NGSCM を終了するために、C-x C-c を入力します。このときに、編集したファイルをセーブするかどうかを尋ねられます:

```
Save file /home/pooh/sum.scm? (y or n)
```

(ファイル名の部分 /home/pooh/sum.scm は異なることがあります。) ここで y を入力してください。すると作成したプログラムがディスクに書き込まれ、保存されます。もし n と答えると、入力したプログラムは NGSCM の終了とともに消えてしまいます。

入力されたプログラムはファイルとして保存されているために、後日そのプログラムを再び読み込んで実行することができます。

補足: 計算させる式を入力するときに、式を 1 行に書く必要はありません。式が長いときは、適当なところでリターンキー RET を打って改行し、見やすくしてもかまいません。

たとえば、式

```
(* (+ 1 2) (* (+ 3 4) (+ 5 6)) (+ 7 (* 8 2)))
```

を 1 行で書くと、どのような構造になっているのか分かりにくいですね。これを

```
(* (+ 1 2)
   (* (+ 3 4) (+ 5 6))
   (+ 7 (* 8 2)))
```

と書けば、(+ 1 2) と (* (+ 3 4) (+ 5 6)) と (+ 7 (* 8 2)) をかけあわせたものだな、というのが一目瞭然です。数や手続き名の途中でなければ、改行したり空白文字をいくつ入れても、見た目が変わるだけでプログラムそのものは同じです。プログラムが複雑

になってくると、読みにくいプログラムは間違いの元となります。普段から読みやすいプログラムを書くように心がけましょう。

重要: プログラムの実行がいつまでも終らない場合は、プログラムに**無限ループ** (infinite loop) があって、プログラムのある部分が無限に繰り返し実行されている可能性があります。(画面の下で `Garbage collecting...` と `Garbage collecting...done` が交互に表示されていると、まず間違いなく無限ループにはまっています。) 無限ループとは、いつまでもプログラムの実行が終らないという種類のプログラムの間違いです。このようなときは C-g を入力することで、Scheme プログラムの実行が中断できます[1]。

以上をまとめると、次の通りです。

1. 作成するファイルを指定して、 NGSCM を起動します。
2. プログラムを入力します。
3. Ctlx2 により画面を 2 分割します。
4. C-c C-e によって手続きを Scheme に与えます。
5. C-x o でウインドウの移動をして、Scheme ウインドウで動作の確認をします。(無限ループに入ったときは、C-g でプログラムの実行を中断します。)
6. 間違いがあればプログラムウインドウに移って修正をして、4. からやり直します。
7. C-x C-c で、NGSCM を終了します。終了のとき、ファイルをセーブします。

4.4.1 Mule を使う場合

Unix などで動作する Mule エディタも、プログラムを書きながら、Scheme 処理系の実行ができます。使い方は、上で説明した方法とほとんど同じです。

1. ファイル名を指定して、 Mule を起動します。

```
% mule sum.scm RET
```

2. プログラムの入力をします。
3. C-x 2 で、ウインドウを 2 分割します。

[1] 別の Scheme 処理系では、C-c で中断するものもあります。

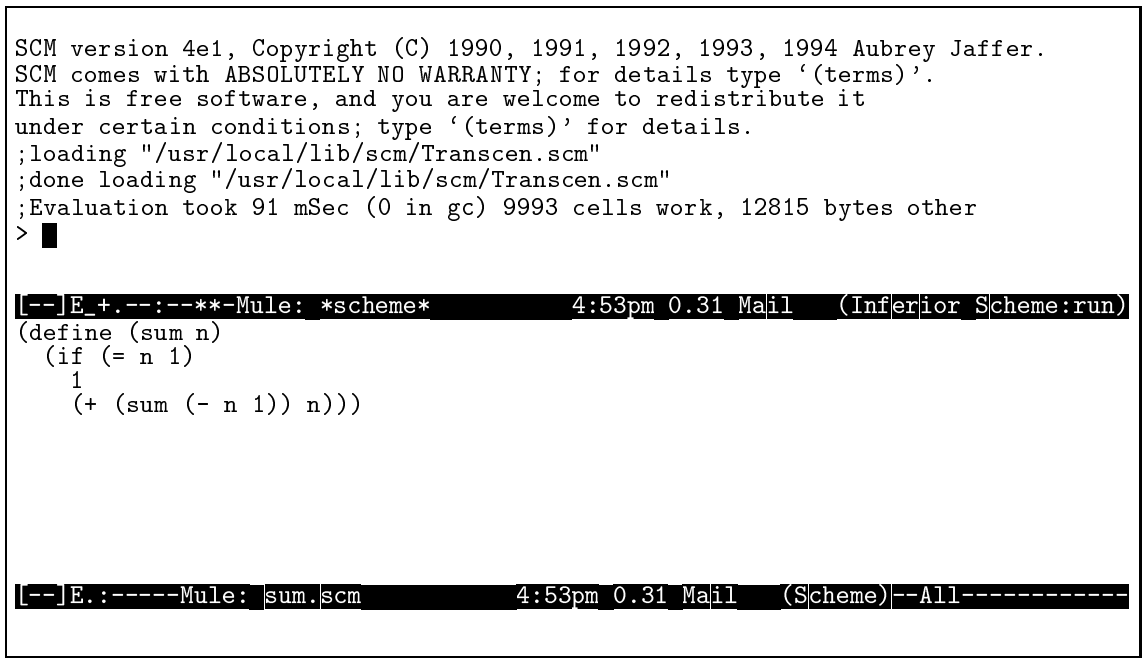


図 4.12: Mule での Scheme 処理系の実行

4. Scheme 処理系を始動させます。
   M-x run-scheme RET を入力し、Scheme 処理系を始動します。なお、M-x run-scheme の入力は、ESC キーを押してから x を押し、 run-scheme 入力し、最後に RET キーを押して下さい。(この動作の直後の様子が、図 4.12です。)

5. プログラムを実行します。式の入力の終りは RET キーで行ないます。

6. ウインドウ間のカーソルの移動は C-x o で、プログラムのバッファの式を Scheme 処理系に渡すには、C-c C-e を入力します。

4.4.2 その他の Scheme 処理系の場合

NGSCM 以外のテキストエディタと Scheme 処理系を使う場合は、次のような手順になります。ここでは例として、 テキストエディタとして vi を、Scheme 処理系として SCM を使った場合を示しますが、その他のシステムの場合でもほぼ同一です。

1. vi を起動し、プログラムをファイルに作成します。(vi の使い方は、NGSCM と大きく違っています。vi の使い方は、マニュアルを参照して下さい。)

   % vi sum.scm RET

2. ファイルをセーブし、エディタを終了します。

3. Scheme 処理系を起動します。

```
% scm
SCM version 4e1, Copyright (C) 1990, 1991, 1992, 1993, 1994 Aubrey Jaffer.
SCM comes with ABSOLUTELY NO WARRANTY; for details type `(terms)'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `(terms)' for details.
;loading "/usr/local/lib/scm/Transcen.scm"
;done loading "/usr/local/lib/scm/Transcen.scm"
;Evaluation took 333 mSec (0 in gc) 8829 cons work
>
```

4. load 手続きを使って、プログラムファイルをロードします。

```
> (load "sum.scm") RET
```

5. プログラムを実行します。無限ループに入ったときは、C-c でプログラムの実行を中断します。実行を中断するキーは処理系によって違うので、詳細はマニュアルを参照して下さい。(ですが、C-c または C-g で中断をするシステムがほとんどです。)

6. プログラムに誤りがあれば最初に戻って、プログラムの誤りを修正します。

さのみ、よし、あしきとは教ふべからず。餘りにいたく諫むれば、童は氣を失ひて、能ものぐさくなり立ちぬれば、やがて能は止まるなり。

世阿弥 「風姿花伝」
昭和三十八年 岩波書店刊

第 5 章
プログラミング実習

本章では、プログラミング言語 Scheme を使った基礎的なプログラミングの実習をします。それぞれの実習には、いくつかのプログラムが出てきます。プログラムを眺めるだけでなく、実際に入力し、実行してみて下さい。またプログラムをいろいろと改造してみて下さい。そのようなことをしているうちに、プログラムを作る能力が次第についてゆきます。

では具体的な学習内容について説明します。本章では、以下に示す実習テーマを通じて、プログラミングの基礎を学びます。

1. **有理数計算手続きの製作**
   — このテーマでは、有理数データとそれに対する演算手続きの作成を通して、抽象データ型の基礎について学びます。

2. **繰り返しの実行の練習**
   — 実用的なプログラムでは、多くのデータを取り扱います。そのようなプログラムでは、それぞれのデータに対して同じ処理を適用する、ということを繰り返します。この実習では、何らかの処理を繰り返し実行するといった、基礎的なプログラミング技法を練習します。

3. **住所録の製作 その 1: オンメモリ版**
   — このテーマでは、住所録プログラムを製作します。ここでは、抽象データ型の考えに基づくプログラミングの練習を行ないます。

それぞれのテーマに対して適当にファイル名を決め、(たとえば「住所録の製作」の場合には、`address.scm` などとして) 本文に出てきたプログラムをそのファイルに入力して下さい。そして 77ページ 4.4 で説明した方法を使っていろいろ実行させてみて、動作を確認しましょう。

テーマの終りには、2 通りの課題を準備しています。**練習問題** は、本文で学んだことに対しての質問とか、作成したプログラムの拡張をする問題です。ほとんどが基本的な問題

なので、できるだけ解いてみるようにしてみて下さい。*印が付いているものは少し難しい問題なので、解けなくても構いません。(* が多いほど難しくなっています。) **レポート**では、本文に関連している事項に関して、調査や考察をするものです。図書館などに行き、専門書を調べるなどの準備が必要かもしれません。

プログラムを作成するときは、適切な手続き名や変数名を付けるようにして下さい。その手続きの名前や変数の名前が不適切だと、プログラムを読むときや改良するときに勘違いの元となります。

以上のことを守って、分かりやすい (しかも正しい) プログラムを書く癖をつけるようにしましょう。また作成したプログラムがちゃんと動作していることを確認するために、適切な入力をいくつか選んで、実行してみて下さい。

5.1 有理数計算手続きの製作

本実習では、**有理数** (rational) を計算するためのプログラム群を作ります。基本的な機能を組み合わせてより複雑な機能をつくり出す方法を学びます。

有理数とは分子と分母よりなる数で、a/b のように表される数です。たとえば

$$3/10,\ -1/3,\ 101/9$$

などは有理数です。

有理数データを総称して、有理数型と呼ぶことにします。有理数型は Scheme の基本的なデータ (整数、対、記号など) を組み合わせて実現します。有理数データの演算のために、次の手続き群を作ります[1]。

- 分子と分母をそれぞれ整数で指定して、あたらしく有理数データを作る手続き `make-rat`
- 与えられたデータが、有理数かどうかを判定する手続き `is-rat?`
- 与えられた有理数データから、分子を取り出す手続き `rat-num`
- 与えられた有理数データから、分母を取り出す手続き `rat-den`
- 有理数同士の加算をする手続き `rat:+`
- その他、有理数に対する演算手続き

5.1.1 設計と実現

それぞれの手続きの実現について考えてゆきます。

式 (`make-rat` a b) を評価すれば、分子が a で分母が b である有理数が作られ、返されるものとします[2]。式 (`is-rat?` r) を評価すると、r が有理数ならば真 `#t` を、そうでなければ偽 `#f` が返されるものとします。以上のことを考えると、有理数データの中には、

- 有理数であることの印
- 分子
- 分母

[1] もちろん、ここに挙げているものが絶対的なものではありません。実習 14 では、違った手続き群を示しています。

[2] `rat` は、rational を表しています。

の 3 つが必要であることが分ります。

ここでは有理数データを、有理数であることの印 (記号 rat)、与えられた分子と分母を、リストとして表現する実現方法を採ります。これに従った手続き make-rat は、次のようになります。

```
(define (make-rat a b)
  (list 'rat a b))
```

上の手続きの中で、rat の前に ' が付いています。これは**引用符** (quote) と呼ばれるものでした。これがないと rat は変数として使われてしまい、記号 rat をリストの中に入れる、ということができません。

次に、与えられたデータが有理数データの判定をする手続きを考えます。有理数データどうかは、リストの第一要素が記号 rat であるかどうかで判別します。リストの第一要素を取り出す Scheme 手続きは car であることを思い出せば、手続き is-rat?は以下のようにすればいいでしょう。

```
(define (is-rat? r)
  (eq? (car r) 'rat))
```

手続き eq?は、与えられた 2 つの引数が同一のものかどうかを調べるものでした。これにより、リストの第一要素が印 rat と等しいか判定しています。

Scheme でこれら 2 つの手続きを定義してから、実行してみます。

```
> (define r1 (make-rat 1 3))      — 1/3 を作る
#<unspecified>
> (is-rat? r1)     — r1 は有理数データですか?
#t                          — そうです
```

こんどは、有理数データでないものを引数に与えて、is-rat? が正しく機能するか調べてみましょう[3]。

```
> (is-rat? 10)
ERROR: car: Wrong type in arg1 10     — エラーが生じた!
```

許されていないデータ型に対して car を行った、とエラーメッセージに出ています。is-rat? の定義をもう一度見てみれば、引数 r がリストかどうかを確認せず、いきなり car を行なっています。

[3]数学的な意味では $10 = 10/1$ なので、 10 は有理数であるといえます。ですが計算機上では、10 には「有理数型の印」がないために、有理数の数ではないと考えます。

上の例のように、数などに対して car を実行するとエラーになります。そのため、前もってデータがリストかどうかを調べるようにしないといけません。与えられたデータがリストかどうかを調べるには、手続き list? を使います。

手続き is-rat? を次のように改良します。これはどんな引数に対しても、エラーを起こさずに動作します。

```
(define (is-rat? r)
  (and (list? r)
       (eq? (car r) 'rat)))
```

上のプログラムでは、特殊形式 and が出てきています。これは、与えられた式を順に評価してゆき、評価結果が #f のものが現れたらそこで実行をやめて #f を返します。もしすべての引数の評価結果に #f のものがなければ、一番最後の引数の評価結果が and の値となります[4]。

以上で有理数を作る手続きと、データが有理数かどうかを判定する手続きができました。今度は有理数に対する演算手続きに移ります。有理数 r_1 と r_2 に対する演算は、次のように定められます。

$$[\text{加算}] \qquad r_1 + r_2 = \frac{r_1\text{の分子} \times r_2\text{の分母} \ + \ r_1\text{の分母} \times r_2\text{の分子}}{r_1\text{の分母} \times r_2\text{の分母}}$$

$$[\text{減算}] \qquad r_1 - r_2 = \frac{r_1\text{の分子} \times r_2\text{の分母} \ - \ r_1\text{の分母} \times r_2\text{の分子}}{r_1\text{の分母} \times r_2\text{の分母}}$$

$$[\text{乗算}] \qquad r_1 \times r_2 = \frac{r_1\text{の分子} \times r_2\text{の分子}}{r_1\text{の分母} \times r_2\text{の分母}}$$

$$[\text{除算}] \qquad r_1/r_2 = \frac{r_1\text{の分子} \times r_2\text{の分母}}{r_1\text{の分母} \times r_2\text{の分子}}$$

これらの手続きを作る前に、有理数データの分子部分、分母部分を取り出す手続き rat-num、rat-den を作ります[5]。リスト s の第 i 番目の要素をとり出す手続きとして、(list-ref s i) があるのでこれを使うことにします。(なお、リストの一番最初の要素は第 0 番目になります。) これらの手続きを作るとき、与えられたデータが有理数型かどうかのチェックを忘れてはいけません。与えられたデータが有理数型なら分子あるいは分母部分を抜き出して返すものとし、そうでなければエラーとします。エラーとするには、手続き error を使います。この手続き error は、引数に与えられた式を表示してプログラムの実行を強制中断します。(手続き error は Scheme 処理系に依存した手続きです。この手続きを使用するときは、使っている処理系のマニュアルを参照してください。)

rat-num, rat-den は、それぞれ次のようになります。

[4] ここで、左から「順に」評価をしてゆく、というのが大事な点です。引数すべてを評価してから、その結果を and に渡すと、引数に 10 を与えるとまた同じエラーになってしまします。

[5] 分子、分母のことを英語ではそれぞれnumerator, denominator といいます。

```
(define (rat-num r)
  (if (is-rat? r)
    (list-ref r 1)
    (error "rat-num - not rational number")))
(define (rat-den r)
  (if (is-rat? r)
    (list-ref r 2)
    (error "rat-den - not rational number")))
```

error の引数で rat-num - not rational number としているのは、どの手続きの中で何が原因でエラーが起きたかを分かるようにするためのものです。

ではこれらを試してみます。

```
> (define r (make-rat 1 3))
#<unspecified>
> (rat-num r)
1
> (rat-den r)
3
> (rat-den 10)
Error: rat-num - not rational number
>
```

これらを使って加算を行なう手続き rat:+ を作ります。もちろんこの手続きの中でも、引数が有理数であるかどうかをチェックしないといけません。有理数の加算の定義に従うと、次のような手続きになります。

```
(define (rat:+ r1 r2)
  (if (and (is-rat? r1) (is-rat? r2))
    (make-rat
      (+ (* (rat-num r1) (rat-den r2))
         (* (rat-den r1) (rat-num r2)))
      (* (rat-den r1) (rat-den r2)))
    (error "rat:+ - not rational number")))
```

このプログラムは、次のような構造をしています:

```
(define (rat:+ r1 r2)
  (if ⟨ r1 r2 ともに有理数データか?⟩
    (make-rat
```

〈 分子の計算 〉
〈 分母の計算 〉)
〈 エラーとする 〉))

まず最初に、2 つの引数 r1 と r2 がともに有理数データかどうか調べています。もしそうなら、 分子と分母を加算の定義にしたがってそれぞれ計算し、make-rat の引数に与えてあらたな有理数を作っています。もし 2 つの引数のいずれかが有理数データでなければ、error によってエラーとし、実行を中断します。

では、これを実行してみましょう。

```
> (define r (make-rat 1 3))    — 1/3
#<unspecified>
> (define s (make-rat 2 5))    — 2/5
#<unspecified>
> (define t (rat:+ r s))       — t = 1/3 + 2/5
#<unspecified>
> (rat-num t)       — 分子は 11
11
> (rat-den t)       — 分母は 15
15
```

ちゃんと $1/3 + 2/5 = 11/15$ が計算されていることが分かります。

有理数の減算を行なうプログラムも同様にして作れます:

```
(define (rat:- r1 r2)
  (if (and (is-rat? r1) (is-rat? r2))
    (make-rat
      (- (* (rat-num r1) (rat-den r2))
         (* (rat-den r1) (rat-num r2)))
      (* (rat-den r1) (rat-den r2)))
    (error "rat:- - not rational number")))
```

このプログラムが正しく動くかどうかは、各自でチェックしてみて下さい。

5.1.2 まとめ

本実習で作った有理数計算手続き群を振り返ってみます。まず、以下のような手続き群を作りました。

- 分子と分母から、有理数データを生成する手続き

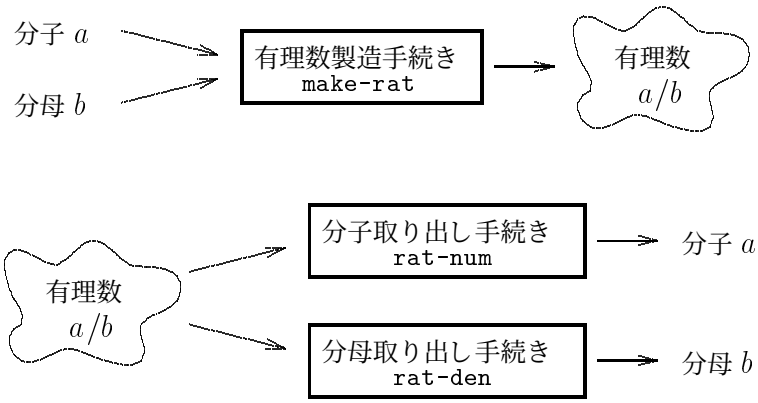


図 5.1: 有理数データとアクセス手続き

- 有理数データから分子と分母を取り出す、複合的なデータへのアクセスをする手続き
- 与えられたデータが有理数型かどうかを判定する手続き

次に、これらを組み合わせて、有理数型のデータに対する各種演算をする手続きを作りました。

本実習で採用した、印、分子、分母をリストにして保持する、という具体的な実現方法は、手続き `make-rat`, `rat-num`, `rat-den`, `is-rat?` によって完全に隠されています。そのため、具体的な実現方法を知らなくても、有理数に関する計算ができます。

別の見方をすれば、`make-rat`, `rat-den`, `rat:+` などを使っている限り、データの実現方法を変えてしまっても大丈夫です。データや手続きがどのような形で実現されているかが大切なのではなく、重要なのはその機能と性質で、データに対してどのような手続きが用意されていて、それらがどのような働きをするか、ということです。手続き群を改良する人の立場でいえば、手続きの呼び出し側から見た手続きの振舞いが以前のものと同じなら、どのように変更してもかまいません。

以上の考え方は**データ抽象** (data abstraction) と呼ばれます。データ抽象ではデータが実現されている方法を隠し、データへのアクセスするための手続きを提供します。そして、その手続きでしかデータへのアクセスを許しません。データを抽象化することにより、データの具体的な実現法には依存しなくても済みます。このようなデータ抽象によるデータ型を**抽象データ型** (abstract data type) と呼びます。また、データの詳細な内部構造を見えなくすることを、**情報隠蔽** (information hiding) といいます。

本実習で作成したプログラムでは、データとデータへアクセスするための手続きが一体となってはいません。言い替えれば、新たにつくり出されたデータ型を使ったプログラムを書くときに、使うデータ型とそれに対応した手続きをすべて把握しておき、適切な手続きを呼び出すようにしなくてはいけません。取り扱うデータ型の数が少ないときはあまり問題は生じませんが、データ型の数が膨大になってくると、それぞれのデータ型に対応し

た手続きの名前を記憶しておくことは難しく、間違ったプログラムを書いてしまう可能性が多くなります。

これを解決する方法として、「手続きにデータを渡す」のではなく、「データに指令を送る」方法があります。この方法なら、違うデータに対しても同一の指令を送ることができ、しかもその振舞いをデータごとに違ったものにすることができます。この方法は**メッセージ伝達** (message passing) と呼ばれますが、詳しいことは後の実習で学びます。

練習問題

1. 有理数同士の乗算を行なう手続き `rat:*` を作りなさい。

2. 関数 f を、$f(r) = r + r^2 + r^3$ とします。(ただし r は有理数とします。) この関数を計算する手続き `f` を、`rat:+` と `rat:*` を使って作りなさい。

3. 有理数同士の除算を行なう手続き `rat:/` を作りなさい。

4. 有理数の逆数を計算する手続き `rat:inv` を作りなさい。

5. 手続き `display` や `write` を使って有理数データを表示すると、`(rational-number 11 15)` のようになってしまいます。これだと美しいとはいえませんし、内部のデータ構造が表示されるために好ましくありません。そこで、`11/15` のように分数形式で表示する手続き `rat:print` を作りなさい。(ヒント: `display` あるいは `write` を使います。)

6. 引数として与えられた有理数が、約分可能かどうかを調べる手続き `reducible?` を作りなさい。もし約分可能であれば真 `#t` を返し、そうでなければ偽 `#f` を返すものとします。(ヒント: a と b との最大公約数を求めるには、手続き `(gcd` a b`)` を使います。)

7. 引数に与えられた有理数を約分した有理数を返す手続き `rat:reduce` を作りなさい。

8. もし分母が負ならば、分母を正にして分子の符号を反転するよう、手続き `rat:reduce` を改造しなさい。

9. 本実習で作った演算手続きでは、計算結果が約分されていません。各演算手続きが計算結果を約分するよう、それぞれの手続きを変更しなさい。

10. 2 つの有理数の加算を行なう過程を、順を追って表示しようと思います。与えられた 2 つの有理数を r_1 と r_2 とし、(1) 最初に与えられた 2 つの有理数を表示し、(2) 必要ならば通分を行なってその結果を表示し、(3) $r_1 + r_2$ の計算結果を表示します。さらに (4) 約分可能でならそのことを表示して、約分した結果を表示します。以上のこ

とを行なう手続き rat:show+ を作りなさい。例として、以下に $\frac{2}{3}+\frac{1}{4}$ の場合と $\frac{1}{8}+\frac{3}{8}$ の場合を示します。

```
> (rat:show+ (make-rat 2 3) (make-rat 1 4))
(2/3) + (1/4)
Tsuubun Shimasu
 = (4/4)*(2/3) + (3/3)*(1/4)
 = (8/12) + (3/12)
Tashi Masu
 = ((8+3)/12)
Kotae Desu
 = (11/12)

> (rat:show+ (make-rat 1 8) (make-rat 3 8))
(1/8) + (3/8)
Bunbo ga Onaji
Tashi Shimasu
 = ((1+3)/8)
Kotae Desu
 = (4/8)
Yakubun Shimasu
 = ((4*1)/(4*2))
Yakubun Shita Kotae Desu
 = (1/2)
```

11. 本実習で作った手続きはどれも、有理数型のデータ同士での演算しか取り扱っていません。引数に整数型のデータが与えられてもいいように、演算手続き群を改造しなさい。たとえば (rat:+ (make-rat 1 2) 2) は 5/2 となり、(rat:+ 1 2) は 3/1 となるようにしなさい。なお、データが整数かどうかを調べるには、手続き exact? を使いなさい。(exact? についての詳しいことは、6.1.3を参照して下さい。)

12. 上の問題に加え、実数型のデータも取り扱えるように rat:+ を改造しなさい。もし引数に実数が与えられたら有理数を実数に変換し、実数どうしの計算をし、実数で計算結果を返しなさい。たとえば (rat:+ (make-rat 1 2) 1.2) の結果は 1.7 となります。なお、データが実数かどうかを調べるには、手続き inexact? を使いなさい。(inexact? についての詳しいことは、6.1.3を参照して下さい。)

13. 本実習で作った手続きは、引数の型のチェックが厳密ではありません。たとえば make-rat に実数を引数に与えてもエラーになりません。それぞれの手続きにて、引数のチェックを厳密に行なうようにしなさい。

14. * 以下のような手続き群を実現しなさい。もちろん本文で使った手続き群の方が便利ですが、ここに挙げた手続き群でも同じように、有理数計算ができます。

    - 有理数データ 1/1 を作る手続き
    - 有理数データと整数が与えられるものとします。その有理数にその整数をかけあわせてできる有理数を返す手続き
    - 有理数データと整数が与えられるものとします。その有理数にその整数を割ってできる有理数を返す手続き
    - 与えられたデータが有理数データかどうかを判定する手続き
    - 与えられた有理数データの分子を取り出す続き
    - 与えられた有理数データの分母を取り出す続き
    - 各種の演算手続き

レポート

1. 巨大なプログラムの開発では、一人がすべてのプログラムを書くことはなく、ときには百人以上もの人がプログラミングに従事します。このような状況で、データ抽象はどのように有利であるか議論しなさい。

5.2 繰り返し実行の練習

本実習では、ある処理を繰り返し実行するための基礎技法を練習します。実用的なプログラムでは、「繰り返し実行」は必要不可欠なものです。たとえば、今月の誕生日の顧客をリストアップする場合を考えましょう。顧客リストに載っているそれぞれの顧客データをみて、その人の誕生日が今月かどうかを調べます。このことをデータすべてに対して、繰り返し実行することになります。

以下では、「繰り返し実行」をするいくつかのプログラムを示します。ある手続きの中で自分自身を呼び出すことを**再帰呼び出し** (recursion) といいます。再帰呼び出しを使うことで、繰り返し実行を実現することができます。

再帰呼び出しによるプログラム作成のコツは、次のようになります。

- 単純な仕事のときは、直接かたずけます。
- 難しい仕事は細かく分け、下請けに任せます。そして下請けの仕事の結果を組み合わせます。

再帰呼び出しでは「下請け」は自分自身ですが、仕事はより細かく分解されるので、いずれは直接に片付けられる単純な仕事になります。そのため、下請けに仕事を流すことが永遠に続くことはありません。

リスト中の数値データの個数を数える (その 1)

リストが引数で与えられ、その中にある数値データがいくつあるかを数える手続きを作ります。与えられるデータは、たとえば次のようなものです。

`(3 6 pooh piglet 9)` — この場合には `3` を返す
`(pooh)` — この場合には `0` を返す

リストの各要素には、さらにリストが現れることはないものとします。(たとえば `(1 (2 3) 4` のようなものは、データとして与えられないものとします。) この問題を解く手続きの構造は、次のようになるでしょう。この手続きの仮引数を `s` とします。

- 場合 1: もし `s` が空リストなら、`s` に入っている数値データの数は 0 です。
- 場合 2: もし `s` が空リストでないとき:
  1. 場合 2-1: もし `s` の `car` 部が数値データのとき:
     残りのリスト (`cdr s`) に入っている数値データの数に 1 加えたものが、`s` に入っている数値データの数です。
  2. 場合 2-2: もし `s` の `car` 部が数値データでないとき:
     残りのリスト (`cdr s`) に入っている数値データの数が、`s` に入っている数値データの数です。

これの考えに基づいて、手続き count-numbers-1a を作ります。

```
(define (count-numbers-1a s)
  (if (null? s)
    0                                    — 場合 1
    (if (number? (car s))
      (+ (count-numbers-1a (cdr s)) 1)   — 場合 2-1
      (count-numbers-1a (cdr s)))))      — 場合 2-2
```

この例では if を使っていますが、cond を使って書くこともできます。

```
(define (count-numbers-1b s)
  (cond
    ((null? s)
     0)                                  — 場合 1
    ((number? (car s))
     (+ (count-numbers-1b (cdr s)) 1))   — 場合 2-1
    (else
     (count-numbers-1b (cdr s)))))       — 場合 2-2
```

実際に実行してみましょう。

```
> (count-numbers-1a '(3 6 pooh piglet 9))
3
> (count-numbers-1a '(pooh piglet))
0
> (count-numbers-1a '())
0
> (count-numbers-1a '(3 6 9))
3
```

望み通りの動作をしていることが分かります。

上の実行例では、手続きに与える引数に ' がついています。これは**引用符** (quote) と呼ばれるものです。(クォートともいいます。) たとえば、(count-numbers1a '(3 6 pooh piglet 9)) の評価のときに引用符がなく (count-numbers1a (3 6 pooh piglet 9)) となっていたとしましょう。Scheme は (3 6 pooh piglet 9) を手続き呼び出しとしてこの式を評価しようとしてしまいます。

```
> (count-numbers1a (3 6 pooh piglet 9))
```

```
ERROR: Wrong type to apply:  (see errobj)
; in expression: (... 3 6 pooh piglet 9)
; in top level environment.
>
```

こうならないために、’ を置くことで、手続き呼び出しではなく、データそのものを引数として与えるようにしています。

リスト中の数値データの個数を数える (その 2)

先ほどの問題では、引数のトップレベルにはリストは現れないと仮定しましたが、こんどはリストが現れるのも許し、そのリストの中の数値も勘定する問題を考えます。たとえば次のようなデータも、引数として許します。

```
((3 alice 6) hatter (9 2))          — この場合には 4 を返す
((3 alice 6) hatter ((1 2) (3 4)))  — この場合には 6 を返す
((3 alice 6) ((1 (2))))             — この場合には 4 を返す
```

この問題を解く手続きの構造は、次のようになります。この手続きの仮引数を s とします。手続き count-numbers-1a とよく似ていますが、car 部に対して、さらに数値データの数を数えています。

- 場合 1: もし s が空リストなら、s に入っている数値の数は 0 です。
- 場合 2: もし s が空リストでないとき:
  1. 場合 2-1: もし (car s) がリストのとき:
     (car s) の数値データの数に、残りのリスト (cdr s) の数値データの数を加えたものが、s に入っている数値データの数です。
  2. 場合 2-2: もし (car s) が数値データのとき:
     残りのリスト (cdr s) に入っている数値データの数に 1 加えたものが、s に入っている数値データの数です。
  3. 場合 2-3: それ以外のとき:
     残りのリスト (cdr s) に入っている数値データの数が、s に入っている数値データの数です。

これに基づいて手続き count-number-2a を作ります。

```
(define (count-numbers-2a s)
  (if (null? s)
    0                                    — 場合 1
```

```
(cond
  ((list? (car s))
   (+ (count-numbers-2a (car s))
      (count-numbers-2a (cdr s))))    — 場合 2-1
  ((number? (car s))
   (+ (count-numbers-2a (cdr s)) 1))  — 場合 2-2
  (else
   (count-numbers-2a (cdr s))))))     — 場合 2-3
```

リストから、数値データだけを取り出したリストを作る

与えられたリストの中から数値データを取り出し、その数値データだけからなるリストを作る手続きを作ります。リストの要素に、さらにリストは含まれていないものとします。

```
(1 2 rabbit owl)    — この場合には (1 2) を返す
(pooh honey)        — この場合には空リスト () を返す
(4 5 6)             — この場合には (4 5 6) を返す
```

再帰呼び出しを使った場合での手続きの構造は、次のようになるでしょう。なお、仮引数を s とします。

- 場合 1: s が空リストのとき
  空リストを返します。
- 場合 2: s が空リストでないとき
  1. 場合 2-1: (car s) が数値データの場合
     (car s) と、(cdr s) から数値データを取り出したリストを cons したものが、望みのリストです。
  2. 場合 2-2: (car s) が数値データでない場合
     (cdr s) から数値データを取り出したリストが、望みのリストです。

この考えに沿って手続きを作ると、次のようになります。

```
(define (collect-numbers s)
  (if (null? s)
      '()                                          — 場合 1
      (if (number? (car s))
          (cons (car s) (collect-numbers (cdr s))) — 場合 2-1
          (collect-numbers (cdr s)))))             — 場合 2-2
```

数値データよりなるリストから、各要素を二乗したリストを作る

数値データだけから構成されるリストが与えられ、各要素を二乗したリストを作る手続きを作ります。リストの要素に、さらにリストは含まれていないものとします。

```
(0 1 2 3)          — この場合には (0 1 4 9) を返す
()                 — この場合には空リスト () を返す
```

手続きの構造は、次のようになります。なお、仮引数を s とします。

- 場合 1: s が空リストのとき
  空リストを返します。
- 場合 2: s が空リストでないとき
  (car s) の二乗と、(cdr s) の数値データを二乗したリストを cons したものが、望みのリストです。

この考えに沿って手続きを作ると、次のようになります。

```
(define (squared-numbers s)
  (if (null? s)
      '()                                  — 場合 1
      (cons (* (car s) (car s))            — 場合 2
            (squeared-numbers (cdr s)))))
```

練習問題

1. リストのトップレベルに現れる記号の数を求める手続きを作りなさい。次の例を参考にして下さい。

   ```
   (0 -1 2 -3)              — この場合には 0 を返す
   (pooh piglet 2001)       — この場合には 2 を返す
   (1192 538 (owl rabbit))  — この場合には 0 を返す
   ```

2. リストに含まれる記号の数を求める手続きを作りなさい。(要素がリストの場合は、再帰的に記号の数を求めなさい。) 次の例を参考にして下さい。

   ```
   (0 -1 2 -3)              — この場合には 0 を返す
   (pooh piglet 2001)       — この場合には 2 を返す
   (1192 538 (owl rabbit))  — この場合には 2 を返す
   ```

3. リストのトップレベルに現れる数値データのうち、0 以上のものがいくつあるかを求める手続きを作りなさい。次の例を参考にして下さい。

   ```
   (0 -1 2 -3)              — この場合には 2 を返す
   (pooh piglet 2001)       — この場合には 1 を返す
   (1192 538 (owl rabbit))  — この場合には 2 を返す
   ```

4. 数値データからなるリストが与えられ、その要素をすべてかけあわせた数を求める手続きを作りなさい。

5. 数値データからなるリストが与えられ、偶数である要素だけからなるリストを返す手続きを作りなさい。

6. 数値データと記号のみからなるリストが与えられるものとし、対応する要素が数値データのときは num を、記号のときは sym を要素とするリストを返す手続きを作りなさい。(次の例を参考にして下さい。)

   ```
   (1 pooh honey 2)  — この場合には（num sym sym num）を返す
   (owl rabbit)      — この場合には（sym sym）を返す
   ()                — この場合には（）を返す
   ```

7. 文字列データよりなるリストが与えられるとし、その文字列をすべてを順につないでひとつの文字列にする手続きを作りなさい。次の例を参考にして下さい。(ヒント: string-append)

   ```
   ("Winnie" "-the-" "Pooh")
         — この場合には"Winnie-the-Pooh"を返す
   ("Pooh")
         — この場合には"Pooh"を返す
   ```

5.3 住所録の製作 その 1: オンメモリ版

本実習では電話帳プログラムを作ります。住所録には色々な人の氏名、住所、電話番号、勤務先、生年月日などの情報が蓄えられています。ここで作る住所録プログラムは簡単化のために、名前と電話番号の 2 項目だけしか持たない住所録を作ります。それ以外の項目も取り扱えるようにするのは簡単なので、プログラムの拡張は練習問題とします。9.4 (住所録の製作 その 2: ファイル版) では、データをファイルに保持するようなプログラムを作ります。

コンピューターで住所録を実現すると、いろいろと便利なことができます。紙に書かれた住所録を使っていると、項目を目で探し、手でページをめくらないといけません。ですが、コンピューターにデータを記憶させておけば、検索はすべてコンピューターがしてくれますし、見つけるのもあっという間です。また、新しい住所データの追加やデータの変更も楽にできます。住所録データを他のプログラムからも検索できるようにすれば、プリンタを使って葉書の宛先を自動的に印刷するプログラムをつくることもできます。(紙の住所録なら、すべて手作業で葉書の宛先を書かないといけません。)

実生活の中で、ある人の電話番号を調べるにはどのようにしているか考えてみてください。たとえば A さんの電話番号を調べるには、住所録を使って次のようにするでしょう。

1. 住所録を探して来て、住所録を開きます。住所録には色々な人の名前と電話番号の組があります。
2. A さんの名前を探して、 順々に調べてゆきます。
3. A さんの名前が見つかれば、 電話番号の欄を見ます。
4. 見つからなければ、 あきらめます。
5. 最後に住所録を閉じ、住所録を元の場所にしまいます。

ここで作るプログラムは、この考え方に基づいて作ります。住所録をアクセス (読み書き) するに先だち、これから住所録の使用を宣言する手続きを呼び出します。これは、住所録にアクセスするための各種の下準備を、その手続きにさせるためのものです。これは実世界での、住所録をどこからか運んで住所録を開くことに対応していています。(手元にないと何もできませんし、閉じたままだと読むことができません!)

次に、探している項目を見つけないといけません。住所録は、氏名と電話番号でひとつのデータの単位と考えることができます。ある人の電話番号を見つけるには、まず最初のデータを読み、そのデータの氏名の部分と探している人の氏名を比べます。もしそのデータが探しているひとのものなら、そのデータの電話番号部分が望みの電話番号です。もし違っていれば、次のデータを読み、 さきほどと同じように氏名を比べます。この動作を繰り返します。(このことは、データが概念的に一列にならんでいることを意味しています。)

もしすべてのデータを調べても見つからなければ、その住所録にはその人は載っていないということになります。

最後に、住所録の使用の終了を宣言する手続きを呼び出します。これは開いていた住所録を閉じ、元にあった場所にしまうことに対応しています。

5.3.1 設計と実装

以上の考え方に基づいて、具体的なプログラムの設計に移ります。住所録データは、*ADDRESS-BOOK* という変数に保持することにします。変数 *ADDRESS-BOOK-OPENED* は、住所録が開かれているかどうかを表すのに使います。もしこの変数の値が真ならば、住所録が開かれていることを表します。以下では、「開く」、「閉じる」という言葉の代わりに、それぞれ「**オープン** (open) する」、「**クローズ** (close) する」という言葉を使うことにします[6]。

住所録をオープンした後はデータをひとつづつ順番に読んでゆきます。ですので、現時点でどこまでデータを読んだかを記憶しないといけません。このために、 変数 *ADDRESS-BOOK-STATE* を使います。

これらの変数の宣言は、次のようになります。

```
;;; 住所録データを保持するデータベース
(define *ADDRESS-BOOK* '())

;;; 住所録の状態を保持する変数群
(define *ADDRESS-BOOK-STATE*  #f)  ; 注目点
(define *ADDRESS-BOOK-OPENED* #f)  ; オープンされているかどうか
```

ここで ; に続くものは**注釈**あるいは**コメント**というものです。; から行の終りまでは、プログラムについての説明などを自由に書くことができます[7]。注釈はプログラムの実行にまったく影響を与えません。そのため、注釈はプログラムの実行に必要なものではありません。注釈は作成したプログラムに対する説明やメモなどを、プログラム中に残すためのものです。ですので、プログラムをつくるときは、プログラム説明や設計方針などを注釈として書き、理解しやすいプログラムになるよう心がけてください。

住所録のデータの構造を最初に設計しておきます。データの単位は、氏名が第一要素、電話番号が第二要素であるリストとします。このデータを、「個人データ」と呼ぶことにします。たとえば、(yumiko 01-23-4567) です。

住所録データは、個人データをリストにしたものとします。たとえば

```
((mayuko 01-23-4567) (misako 09876-7-654)
 (hikaru 56-7890) )
```

[6] 計算機科学の分野では、「オープン」「クローズ」の方が日常的に用いられているからです。

[7] 文字列内で ; が現れても、それは文字列の一部であって注釈になりません。

です。

データは上で説明した構造にしますが、将来のデータの構造の変更に備えて、上で説明した構造そのものに依存しない形でデータの内容を参照するようにします。というのも、個人データから氏名を取り出すのに car を使っていたとします。もし個人データの構造が変われば、個人データから氏名を取り出す部分を (洩れなく) 変更しないといけません。この変更は大変な仕事なので、「個人データから氏名を取り出す」手続きを定義しておきます。その手続きを呼び出せば、データの構造に依存せずに氏名を取り出すことができます。これ以外にも、データの具体的な構造を隠すいくつかの手続きを用意します。

氏名と電話番号から、個人データを作る手続き make-address-item を作ります[8]。

```
; 名前と電話番号から、個人データを作る
(define (make-address-item name phone)
  (list name phone))
```

個人データから氏名を取り出す手続きと、個人データから電話番号を取り出す手続きを作ります。

```
; 個人データから、名前を取り出す
(define (get-name item)
  (car item))
; 個人データから、電話番号を取り出す
(define (get-phone item)
  (cadr item))
```

住所録の最初の状態は、個人データがまったく何もない状態から始まり、順次個人データを加えてゆくことで、より大きな住所録とします。

次は住所録を真新しいものにする手続きです。

```
; 住所録を消去する
(define (make-new-address-book)
  (set! *ADDRESS-BOOK* '()))
```

前にも説明したように、住所録には個人データをリストの形で保持します。このため、個人データを住所録に追加するには、追加しようとしている個人データと住所録データとを cons し、その結果を新たな住所録データとします。

次の手続き add-to-address-book は、引数に与えられた氏名と電話番号を住所録に蓄えます。

[8] コメントに日本語を使っていますが、これは分かり易さのためです。Scheme 処理系によっては日本語が使えないものもありますので、プログラムを入力し実行する場合は注意して下さい。

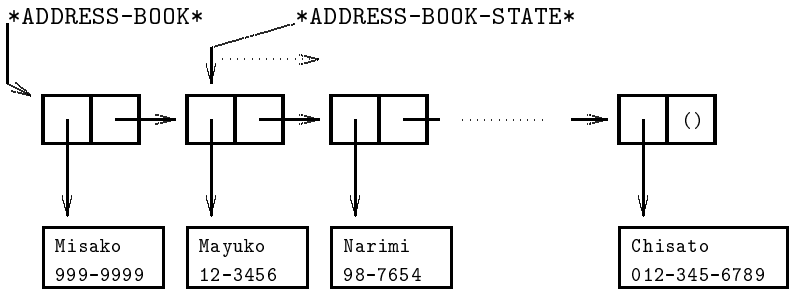


図 5.2: 住所録データと変数の関係

```
; 住所録に個人データを追加する
(define (add-to-address-book name phone)
  (set! *ADDRESS-BOOK*
        (cons (make-address-item name phone) *ADDRESS-BOOK*)))
```

この手続きの中で、手続き make-address-item を使って個人データをつくり出していることに注意してください。

以上により、住所録を作るのに必要な手続きが完成しました。以下の 2 つの手続きを定義した後で (address-book) を評価すると、Chisato, Mayuko, Narimi, Misako の 4 人分の電話番号が登録された住所録が作られます。

```
; 住所録を作り直す
(define (address-book)
  (make-new-address-book)
  (add-initial-address))

;;; 個人データをいくつか登録する
(define (add-initial-address)
  (add-to-address-book 'Chisato '012-345-6789)
  (add-to-address-book 'Narimi '98-7654)
  (add-to-address-book 'Mayuko '12-3456)
  (add-to-address-book 'Misako '999-9999))
```

次はこの住所録に対するデータの検索について考えます。最初に住所録のオープンとクローズのための手続きを作ります。オープンのときは、「注目点」を住所録の先頭の個人データにします。これとともに、住所録がオープンされていることを変数 *ADDRESS-BOOK-OPENED* に記録します。これは、オープンされてもいないのに住所録のデータを読み出す、といっ

たプログラムの間違いを検出するのに使います。住所録をクローズするときは、住所録はオープンされていないということを変数*ADDRESS-BOOK-OPENED* に記録します。

オープンとクローズのための手続きは、それぞれ以下のようになります。

```
; 住所録のオープン
(define (open-address-book)
  (set! *ADDRESS-BOOK-STATE* *ADDRESS-BOOK*)
  (set! *ADDRESS-BOOK-OPENED* #t))
; 住所録のクローズ
(define (close-address-book)
  (set! *ADDRESS-BOOK-OPENED* #f))
```

住所録データの様子を、図 5.2に図示しています。変数 *ADDRESS-BOOK* は、住所録データ全体を保持していることになります。変数 *ADDRESS-BOOK-STATE* は最初、住所録データの先頭を指します。対の cdr 部をたぐることで、次のデータを注目することになります。

では次に、個人データを読み出す手続きを作ります。前に述べたように、住所録は氏名と電話番号より構成される個人データが一列にならんだものです。ひとつの個人データを読んだら次へ、次を読んだらその次へ、というように読んでゆきます。

手続き read-address は、注目点の個人データを読んで次のデータに注目点を移動し、読んだ個人データを返します。この手続きは 2 つの手続き lookup-address と next-address を基にして作ることにします。以下にこれらの手続きを作ります。

手続き lookup-address は、住所録で注目点の個人データを読みます。まだ読まれていない個人データからなるリストが、変数*ADDRESS-BOOK-STATE*に保持されています。手続き lookup-address は、このリストの先頭の個人データを返します。

```
; 注目点の個人データを読む（注目点を変えない）
(define (lookup-address)
  (if *ADDRESS-BOOK-OPENED*
    (car *ADDRESS-BOOK-STATE*)
    (error:not-opened)))
```

ここで *ADDRESS-BOOK-OPENED* を調べて、ちゃんとオープンされているかを確かめていることに注意しましょう。というのも、住所録がオープンされていないと *ADDRESS-BOOK-STATE* にはどのような値が入っているか分かりません。そのため、最初の個人データから順々に読んでゆける保証はありません。(このことがアクセスに先だってオープンし、オープン手続きの中でいろいろな準備をすることの理由です。)

手続き next-address は、注目点を次の個人データにします。この手続きが実行されると、変数 *ADDRESS-BOOK-STATE* の次の要素を指すようにします。これも先ほどと同様に、住所録がオープンされていないとエラーにします。

```
; 注目点を次の個人データにする
(define (next-address)
  (if *ADDRESS-BOOK-OPENED*
    (set! *ADDRESS-BOOK-STATE* (cdr *ADDRESS-BOOK-STATE*))
    (error:not-opened)))
```

以上の 2 つの手続きを使って、手続き read-address を作ります。

```
; 注目点の個人データを読む（注目点は次に移る）
(define (read-address)
  (if *ADDRESS-BOOK-OPENED*
    (let ((value (lookup-address)))
      (next-address)
      value)
    (error:not-opened)))
```

上ではデータを順々に読むための手続きを作成しました。データを読み続けてゆけば、いつかはデータの終りにたどり着きます。このときの変数 *ADDRESS-BOOK-STATE* の値は空リストになります。ここでさらにデータを読もうとすると (car *ADDRESS-BOOK-STATE*) の実行がおこなわれ、空リストに対する car の適用によるエラーが起きます。これを避けるためには、住所録の終りかどうかを調べる必要があります。すなわち、住所録を読むときは、データの終りに到達したかを確かめないといけません。

手続き end-of-address-book? で、住所録の終りにたどり着いたかどうかを調べます。

```
; 注目点が住所録の終りかどうかを判定する
(define (end-of-address-book?)
  (if *ADDRESS-BOOK-OPENED*
    (null? *ADDRESS-BOOK-STATE*)
    (error:not-opened)))
```

以上の一連の手続きで用いたエラーのときに呼び出す手続きは、以下のようにします。

```
(define (error:opened)
  (error "Already opened"))
(define (error:not-opened)
  (error "Not opened"))
```

ここで手続き error は、エラーメッセージを出して実行を終了する手続きです。

以上により、住所録をアクセスするための基本的な手続き (オープン、クローズ、読み出しなど) を作りました。以上の手続き群をまとめたのが、以下のプログラムリストです。

```
;;; address.scm （住所録プログラム）

;;; 住所録データを保持するデータベース
(define *ADDRESS-BOOK* '())

;;; 住所録の状態を保持する変数群
(define *ADDRESS-BOOK-STATE*  #f)  ; 注目点
(define *ADDRESS-BOOK-OPENED* #f)  ; オープンされているかどうか

;;; 個人データ関連の手続き
; 名前と電話番号から、個人データを作る
(define (make-address-item name phone)
  (list name phone))
; 個人データから、名前を取り出す
(define (get-name item)
  (car item))
; 個人データから、電話番号を取り出す
(define (get-phone item)
  (cadr item))

;;; 住所録の初期化と個人データ登録関連の手続き
; 住所録を消去する
(define (make-new-address-book)
  (set! *ADDRESS-BOOK* '()))
; 住所録に個人データを追加する
(define (add-to-address-book name phone)
  (set! *ADDRESS-BOOK*
(cons (make-address-item name phone) *ADDRESS-BOOK*)))
; 住所録を作り直す
(define (address-book)
  (make-new-address-book)
  (add-initial-address))

;;; 個人データをいくつか登録する
(define (add-initial-address)
  (add-to-address-book 'Chisato '012-345-6789)
  (add-to-address-book 'Narimi '98-7654)
  (add-to-address-book 'Mayuko '12-3456)
  (add-to-address-book 'Misako '999-9999))

;;; 外部へ公開された、住所録アクセスのための手続き群
; 住所録のオープン
(define (open-address-book)
  (set! *ADDRESS-BOOK-STATE* *ADDRESS-BOOK*)
  (set! *ADDRESS-BOOK-OPENED* #t))
; 住所録のクローズ
(define (close-address-book)
  (set! *ADDRESS-BOOK-OPENED* #f))
```

```
; 注目点の個人データを読む（注目点は次に移る）
(define (read-address)
  (if *ADDRESS-BOOK-OPENED*
    (let ((value (lookup-address)))
      (next-address)
      value)
    (error:not-opened)))
; 注目点の個人データを読む（注目点を変えない）
(define (lookup-address)
  (if *ADDRESS-BOOK-OPENED*
    (car *ADDRESS-BOOK-STATE*)
    (error:not-opened)))
; 注目点を次の個人データにする
(define (next-address)
  (if *ADDRESS-BOOK-OPENED*
    (set! *ADDRESS-BOOK-STATE* (cdr *ADDRESS-BOOK-STATE*))
    (error:not-opened)))
; 注目点が住所録の終りかどうかを判定する
(define (end-of-address-book?)
  (if *ADDRESS-BOOK-OPENED*
    (null? *ADDRESS-BOOK-STATE*)
    (error:not-opened)))

;;; エラーメッセージ関連の手続き群
(define (error:opened)
  (error "Already opened"))
(define (error:not-opened)
  (error "Not opened"))
```

5.3.2 住所録のアプリケーションインターフェース

住所録に関する手続きをまとめると、以下の通りになります。

- (address-book)
  住所録を初期化します。
- (add-to-address-book ⟨氏名⟩ ⟨電話番号⟩)
  住所録に個人データを登録します。
- (open-address-book)
  住所録をオープンします。オープンにともない、注目点を住所録の先頭の個人データとします。
- (close-address-book)
  住所録をクローズします。

- `(lookup-address)`
  注目点の個人データを読み出します。注目点はそのままです。

- `(next-address)`
  注目点を次の個人データにします。

- `(read-address)`
  注目点の個人データを読み出し、注目点を次の個人データにします。

- `(end-of-address-book?)`
  注目点が住所録の終りかどうかを調べます。

住所録よりいろいろな情報を引き出す**アプリケーションプログラム** (application program) は、これらあらかじめ用意されている手続きを組み合わせるだけで作ることができます。これらは、どの手続きが用意され、それらがどのような機能を持っているかを示しており、アプリケーションプログラムとの間を橋渡しをするもの (インターフェース) となっています。

このように、何かの機能を利用しやすいように便利な手続き群 (あるいはその仕様) のことを、**アプリケーションインターフェース** (application interface) あるいは **API** と呼んでいます。

5.3.3 アプリケーションプログラムの作成

今度はこれらの手続きを使って、アプリケーションプログラムを作ります。ここでは例として、与えられた氏名からその人の電話番号を調べるプログラムを作ります。その手続きの名前は `phone` とし、実現例を以下に示します。

```
(define (phone name)
  (let ((phone-number #f))
    (open-address-book)
    (set! phone-number (search-phone name))
    (close-address-book)
    phone-number))
```

この中に出てくる手続き `serach-phone` は、引数に与えられた氏名からその人の電話番号を調べる手続きで、このすぐ後に作ります。この例のように、1 つの手続きですべての処理をしないようにしているのは、1 つの手続きを短く簡潔にして、プログラムを分かりやすくするためです。

手続き `search-phone` は、住所録を順々に読んでゆき、それぞれの個人データが探している人のものかを、氏名を比べることで調べます。もし探している人のもの個人データが

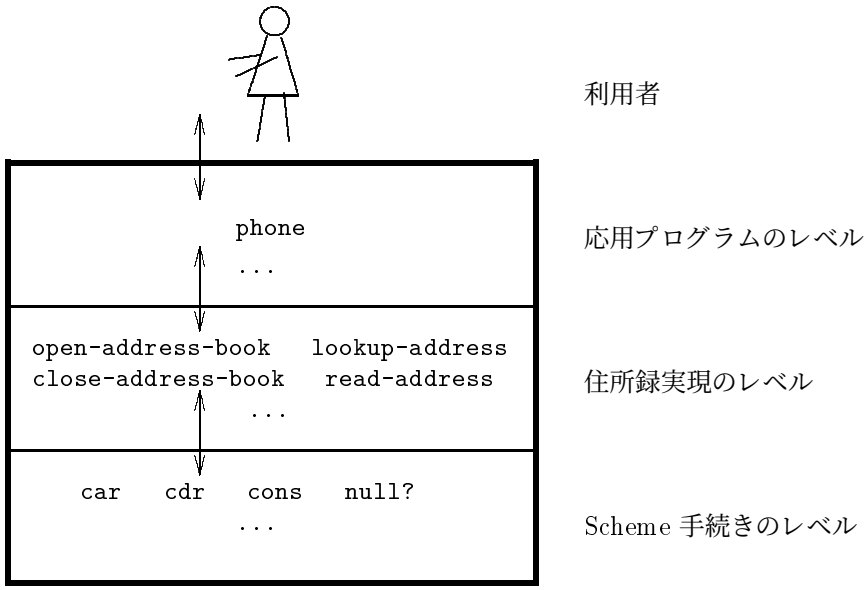


図 5.3: 3 つの抽象化のレベル

見つかれば、その個人データの電話番号部分を取り出し、手続きの値として返します。住所録から個人データを読む前には、住所録の終りがどうかをチェックすることを忘れてはいけません。手続き search-phone は、次のようになります。

```
(define (search-phone name)
  (if (end-of-address-book?)      — 住所録の終りか?
    "Not found"                   — 終りなら "Not Found"を返す
    (let ((item (read-address)))  — 住所録データをひとつ読む
      (if (equal? name (get-name item))
        (get-phone item)          — 名前が一致
        (search-phone name)))))   — 次を調べる
```

ではこれを実行してみます。以下に実行例を示します。

```
> (address-book)              — 住所録にデータを登録します
#<unspecified>
> (phone 'Mayuko)             — 万由子さんの電話番号は?
12-3456
> (phone 'Hiroko)             — 浩子さんの電話番号は?
"Not found"                   — 住所録には載っていない
```

5.3.4 まとめ

本実習で作ったプログラムを振り返ってみましょう。住所録という抽象的なデータを考え、それがどのような性質を持ち、どのような考え方でアクセスするかといった、データに関するモデルを最初に考えました。そしてそれに従って住所録のデータの構造を決め、それにアクセスするための手続き群を作りました。

住所録を参照したプログラムを作る人にとっては、データのモデルとアクセス方法のみが重要な関心事で、それが具体的にどのような内部構造になっているかは関係ありません。

ここで作成した住所録に載せてある個人データは、プログラムの中に固定していました。ですが実用的な住所録プログラムでは、プログラム中にデータを記述するのではなく、ディスク上にファイルとして記録しておきます。(こうすることで登録データに変更があっても、プログラムそのものは変えなくて済みます。) ここで作った住所録プログラムをファイルにデータを保持するようにするには、住所録のデータの表現方法とアクセス手続き (たとえば `read-addres`) を変更するだけで、応用プログラムは変更しなくて済みます。(9.4 (住所録の製作 その 2: ファイル版) では、同一の名前と機能を持ちながら、データををファイルに保持するような手続き群を作ります。)

以上の関係を図 5.3に示しています。Scheme は、基本的なデータとその操作手続きとして、`cons`, `car`, `null?`などが、Scheme 基本手続きのレベルで提供されています。これらを使って住所録を実現する手続き群が、住所録実現のレベルで作られています住所録の実現のレベルでは、手続き `read-address`, `lookup-address` などの手続きを、応用プログラムのレベルに対して提供しています。応用プログラムは、住所録実現のレベルによって用意されている手続きだけを使い、住所録を参照するプログラムの実現をしています。最後に利用者は、応用プログラムのレベルで提供されている手続きだけを使い、色々な人の住所を調べます。住所録実現のレベルや Scheme 基本手続きのレベルで提供される手続きを、利用者は意識する必要はありません。

以上のことで重要なことは、プログラムが階層的な構造をしていて、より基本的なレベルから一段抽象度の高いレベルを用意し、その実現には一段下のレベルで提供される概念だけを使えば十分、ということです。あるレベルでの実現方法を変更しても、レベル間でのデータのモデルとアクセス方法さえ守られていれば、他の部分には何も影響は及ぼしません。このため、プログラムの保守が容易になります。

練習問題

1. 住所録に載っている、すべての人の名前と電話番号を表示する手続き `show-all` を作りなさい。

2. 電話番号が引数で与えられるものとします。その電話番号を持つ人の氏名を調べる手続き `phone-to-name` を作りなさい。同じ電話番号を持つ人が複数いる場合が考えられますが、同じ電話番号を持つ人ぜんぶを表示しなさい。

3. 住所録に住所、年齢、 生年月日、性別の情報を保持できるよう、新たに欄を付け加えなさい。これにともない、 それぞれの欄を取り出す手続きも作りなさい。

4. 手続き name-to-address は、引数として与えられた氏名から、その人の住所を調べる手続きとします。この手続きを作りなさい。

5. 年齢がある範囲にある人の一覧を表示する手続き age-from-to を作りなさい。この手続きの呼び出しは、(age-from-to f t) とし、 f歳以上 t 歳以下の人を表示するようにしなさい。

6. 引数として与えられた月に生まれた人すべてを、リストにして返す手続き birthday-people を作りなさい。この手続きの返す値は、氏名、誕生日、住所のリストが該当者ひとりのデータとし、これをリストにしたものとしなさい。(この手続きの返す値を用いて、誕生日を祝うダイレクトメールを自動的に作り出す手続きを作れば、顧客サービス用のプログラムとなります。)

7. 今度の成人式に参加する女性を、住所録から調べ出す手続きを作りなさい。(あなたが呉服屋の店員なら、その手続きはとても役に立つでしょう。)

8. ** あなたが持っている (紙に書いた) 住所録のデータを、この実習で作った住所録プログラムに登録しなさい。友達に電話したり手紙を出すときは、住所録プログラムを使って電話番号や住所を調べるようにしなさい。しばらく使ってみた後に、次の質問に答えなさい。(1) 欲しいと思った機能はどのようなものですか? (2) どこをどうすれば、より使いやすくなると思いましたか? 実際に使ってみた感想と意見に従って、プログラムを改良しなさい。

レポート

1. 図 5.3に示したような抽象化の階層構造を持つものは、他にもいろいろあります。そのようなものをひとつ挙げなさい。それぞれの階層のレベル間には、どのような関係があるかを説明しなさい。

この比の稽古、やすき所を花に當てて、態を大事にすべし。働きをも確やかに、音曲をも、文字にさわさわと當たり、舞をも、手を定めて、大事にして稽古すべし。

世阿弥 「風姿花伝」
昭和三十八年 岩波書店刊

第 6 章

Scheme 入門 (中級編)

第 3章「Scheme 入門 (初級編)」では、Scheme の基本的な機能を勉強し、簡単なプログラムが作れるようになりました。こんどはもう少し高度な Scheme の機能を説明します。本章では (1) 他のデータ型 (文字型データ、ベクトル、数)、(2) 入出力機能、(3) 変数の内部定義、(4) 制御構造について学び、最後に良いプログラムの書き方について学びます。

6.1 さまざまなデータ型 (その 2)

6.1.1 文字

文字 (character) は文字通り、A や 6 などの「文字」を表すデータ型です。Scheme プログラム中で文字データは、たとえば #\A (大文字の A) とか、 #\b (小文字の b) のように、#\⟨ 文字 ⟩ または #\⟨ 文字の名前 ⟩といった形式で書かれます。

#\⟨ 文字 ⟩の形式での、文字データの例を見てみましょう。

```
> #\A
#\A#<unspecified>
> #\b
#\b#<unspecified>
> (display #\A)
A#<unspecified>
> (display #\b)
b#<unspecified>
```

この例で分かるように、文字データは自己評価的です。

次は #\⟨ 文字の名前 ⟩ の例を見てみましょう。なお ⟨ 文字の名前 ⟩ による文字の表し方では、空白文字や改行文字などの**制御文字** (control character) を表すのに使います[1]。

たとえば #\space は、空白文字を表します。

```
> (display #\space)
  #<unspecified>
> (display #\tab)
            #<unspecified>
> (display #\del)
^?#<unspecified>
> (display #\bel)
^G#<unspecified>
```

ASCII 文字集合での制御文字の一覧を、Appendix Aに掲げたので参考にして下さい。

#\⟨ 文字 ⟩の形式で文字を指定する場合には大文字と小文字の区別はありますが、#\⟨ 文字の名前 ⟩ の場合には大文字/小文字は関係ありません。

```
> (equal? #\A #\A)
#t
> (equal? #\A #\a)
#f
> (equal? #\tab #\tab)
#t
> (equal? #\tab #\TAB)
#t
> (equal? #\tab #\TaB)
#t
```

それぞれの文字には、その文字を表す整数が割り当てられています。ASCII 文字集合の場合では、文字 A には (10 進数で) 65 が割り当てられています[2]。文字に割り当てられた整数のことを、その文字の**文字コード** (character code) といいます。文字コードに関連した手続きには、以下のものがあります。

◇ (char->integer ⟨ 文字 ⟩)
— ⟨ 文字 ⟩ の文字コードを返します。

[1]制御文字とは、A や z のように文字そのものを表すのではなく、カーソルを次の行に移動させたり、画面の表示を消去するといった、カーソルや画面などを制御するためのものです。制御文字は普通の文字のように入力できないので、名前や文字コード (Appendix A参照) によって文字を指定します。

[2]文字とそれにどの整数を割り当てるかを表にしたものが、**文字集合** (character set) です。ASCII 文字集合や EBCDIC 文字集合は代表的な文字集合です。

◇ (integer->char ⟨ 文字コード ⟩)
— ⟨ 文字コード ⟩ の文字を返します。

次に、文字と文字との比較について説明します。文字と文字の間には順序が付けられていて、この順序に従って大小関係が決められています。たとえば、A は B よりも小さい、となっています。文字と文字との比較のための手続きには、次のものが用意されています。なお、名前の最後に -ci の付いた手続きは、大文字と小文字の区別をせずに比較をします[3]。

◇ (char=? ⟨ 文字$_1$⟩ ⟨ 文字$_2$⟩)
— ⟨ 文字$_1$⟩ = ⟨ 文字$_2$⟩ か?

◇ (char<? ⟨ 文字$_1$⟩ ⟨ 文字$_2$⟩)
— ⟨ 文字$_1$⟩ < ⟨ 文字$_2$⟩ か?

◇ (char>? ⟨ 文字$_1$⟩ ⟨ 文字$_2$⟩)
— ⟨ 文字$_1$⟩ > ⟨ 文字$_2$⟩ か?

◇ (char<=? ⟨ 文字$_1$⟩ ⟨ 文字$_2$⟩)
— ⟨ 文字$_1$⟩ ≤ ⟨ 文字$_2$⟩ か?

◇ (char>=? ⟨ 文字$_1$⟩ ⟨ 文字$_2$⟩)
— ⟨ 文字$_1$⟩ ≥ ⟨ 文字$_2$⟩ か?

◇ (char-ci=? ⟨ 文字$_1$⟩ ⟨ 文字$_2$⟩)
— ⟨ 文字$_1$⟩ = ⟨ 文字$_2$⟩ か?

◇ (char-ci<? ⟨ 文字$_1$⟩ ⟨ 文字$_2$⟩)
— ⟨ 文字$_1$⟩ < ⟨ 文字$_2$⟩ か?

◇ (char-ci>? ⟨ 文字$_1$⟩ ⟨ 文字$_2$⟩)
— ⟨ 文字$_1$⟩ > ⟨ 文字$_2$⟩ か?

◇ (char-ci<=? ⟨ 文字$_1$⟩ ⟨ 文字$_2$⟩)
— ⟨ 文字$_1$⟩ ≤ ⟨ 文字$_2$⟩ か?

◇ (char-ci>=? ⟨ 文字$_1$⟩ ⟨ 文字$_2$⟩)
— ⟨ 文字$_1$⟩ ≥ ⟨ 文字$_2$⟩ か?

では、これらの実行例を見てみましょう。

[3] -ci は、case insensitive の略で、大文字/小文字の違いは気にしない、という意味です。

```
> (char=? #\A #\A)
#t
> (char=? #\A #\a)
#f
> (char-ci=? #\A #\a)
#t
> (char<? #\G #\H)
#t
> (char<? #\G #\G)
#f
> (char>? #\5 #\8)
#t
```

与えられたデータが文字型かどうかを調べるには、手続き char?を使います。

◇ (char? 〈データ〉)
— 〈データ〉の値が文字データなら真 #t を、そうでなければ偽 #f を返します。

```
> (char? #\A)
#t
> (char? "A")
#f
```

文字は、数字とかアルファベット文字とかのいくつかのグループに分かれています。文字がどのグループに属するかを調べる手続きも用意されています。

◇ (char-alphabetic? 〈文字〉)
— 〈文字〉はアルファベット文字か?

◇ (char-numeric? 〈文字〉)
— 〈文字〉は数字か? 数字とは、0,1,...,9 をいいます。

◇ (char-whitespace? 〈文字〉)
— 〈文字〉は空白文字か? 空白文字とは、(ASCII 文字集合の場合) スペース (space)、タブ (tab)、改行 (line feed)、改頁 (form feed)、復改 (carriage return) のことをいいます。

◇ (char-upper-case? 〈文字〉)
— 〈文字〉は大文字か?

◇ (char-lower-case? 〈文字〉)
— 〈文字〉は小文字か?

実行例を以下に示します。

```
> (char-alphabetic? #\b)
#t
> (char-alphabetic? #\4)
#f
> (char-number? #\z)
#f
> (char-number? #\0)
#t
> (char-whitespace? #\space)
#t
> (char-whitespace? #\tab)
#t
> (char-upper-case? #\A)
#t
> (char-upper-case? #\b)
#f
> (char-lower-case? #\A)
#f
> (char-lower-case? #\z)
#t
```

大文字から小文字に、あるいはその逆を行なうには、次の手続きを使います。

◇ (char-upcase ⟨ 文字 ⟩)
— ⟨ 文字 ⟩ を大文字に変換します[4]。

◇ (char-downcase ⟨ 文字 ⟩) ⟨ 文字 ⟩ を小文字に変換します[5]。

文字を表示するには、手続き display と write を使います。 この他、write-char を使って文字を表示することもできます。

```
> (write-char #\9)
9#<unspecified>
> (write-char #\newline)

#<unspecified>
```

[4,5] これらの手続きにおいて、⟨ 文字 ⟩ は必ずしもアルファベット文字である必要はありません。⟨ 文字 ⟩ がアルファベット文字のときに大文字または小文字に変換し、それ以外のときは ⟨ 文字 ⟩ をそのまま返します。

文字と文字列との橋渡しをする手続きとして、以下の 2 つが用意されています。

◇ (list->string ⟨ 文字のリスト ⟩)
— ⟨ 文字のリスト ⟩ を、文字列にします。

◇ (string->list ⟨ 文字列 ⟩)
— ⟨ 文字列 ⟩ を、文字のリストにします。

実行例を見てみましょう。

```
> (list->string (list #\p #\o #\o #\h))
"pooh"
> (list->string '())
""
> (string->list "owl")
(#\o #\w #\l)
```

6.1.2 ベクトル

いくつかのデータをまとめるのにリストを使えばよいことは、以前に学びました。ですが、 たとえばリストの 3 番目の要素を取り出すには、cdr を 2 回行なった後 car をしないといけません。これくらいなら実行時間は気になりませんが、100 番目の要素を取り出すとなると、多くの回数 cdr をしないといけません。そのため、要素のデータを取り出すのに非常に時間がかかってしまいます。

リストは、先頭から順々に要素を調べてゆく場合には大変便利ですが、上のような場合には実行時間がかかってしまいます。**ベクトル** (vector) と呼ばれるデータ型を使うことで、速くデータを取り出すことができます。ベクトルでは要素の番号を指定することで、読み出しや書き込みの対象とする要素を指定します。そのためリストと違い、先頭から順々に car や cdr で要素を「たぐって」ゆかなくて済みます。ベクトルは、**配列**(はいれつ)とも呼ばれます。何番目のデータでも素早く取り出せる反面、ベクトルデータを作ったときにその大きさが決まってしまい、後から大きくすることができません。(リストの場合は cons を使って、リストを長くしてゆくことができました。)

図 6.1に、主記憶でのリストとベクトルの様子を模式化したものを示します。この図では要素数が 4 で、A, B, C, D を要素とするリストとベクトルを示しています。この図から分かるように、ベクトルでは要素が順番にならんでいるので、最初の要素の記憶番地から i 番目の要素の記憶番地は簡単に計算できます。しかしリストの場合にはそうはできません。

手続き vector と make-vector で、ベクトルデータを作ります。なお、ベクトルデータは #(...) と表示されます。

◇ (vector ⟨ データ$_1$⟩ ⟨ データ$_2$⟩ ···)
— 最初の要素が ⟨ データ$_1$⟩、次の要素が ⟨ データ$_2$⟩ ··· であるベクトルを作ります。

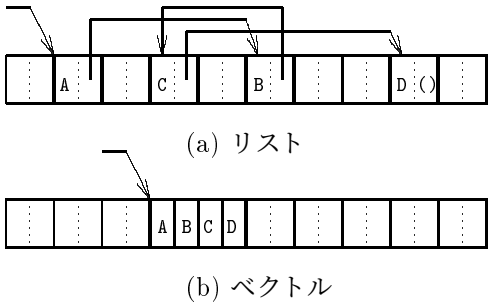


図 6.1: 主記憶上でのリストとベクトル

◇ (make-vector n)
— 大きさ (要素数) が n のベクトルを作ります。(それぞれの要素の初期値がどうなるかは、処理系に依存しています。)

△ (make-vector n ⟨ 初期値 ⟩)
— 大きさ (要素数) が n で、それぞれの要素の値が ⟨ 初期値 ⟩であるベクトルを作ります。

なお処理系によっては make-vector が用意されていないことがあるので注意して下さい。

```
> (vector 'pooh 'piglet 'tiger)
#(pooh piglet tiger)
> (vector 'pooh 'piglet '(kanga roo) 'tiger)
#(pooh piglet (kanga roo) tiger)
> (make-vector 3 'honey)          — 初期値を指定した場合
#(honey honey honey)
> (make-vector 3)                 — 初期値を指定しない場合
#(#<unspecified> #<unspecified> #<unspecified>)
```

ベクトルの要素の値の参照や変更、要素数を調べるには、次の手続きを使います。

◇ (vector-ref ⟨ ベクトル ⟩ i)
— ⟨ ベクトル ⟩ の i 番目の要素の値を返します。

◇ (vector-set! ⟨ ベクトル ⟩ i ⟨ データ ⟩)
— ⟨ ベクトル ⟩ の i 番目の要素の値を、⟨ データ ⟩ に書き換えます。

◇ (vector-length ⟨ ベクトル ⟩)
— ⟨ ベクトル ⟩ の大きさ (要素数) を返します。(なお、#() は長さ 0 のベクトルです。)

△ (vector-fill! 〈ベクトル〉〈データ〉)
— 〈ベクトル〉のすべての要素の値を〈データ〉に書き換えます。

ベクトルの要素を指定する番号は **添字**(そえじ)(index) といいます。

注意: ベクトルの最初の要素の添字は、0 と定められています。すなわち、〈ベクトル〉の最初の要素を参照するには、(vector-ref 〈ベクトル〉 0) として下さい。

実行例を見てみましょう。

```
> (define v (vector 'pooh 'piglet 'tiger))
#<unspecified>
> v
#(pooh piglet tiger)
> (vector-ref v 0)
pooh
> (vector-ref v 1)
piglet
> (vector-ref v 2)
tiger
> (vector-set! v 1 'rabbit)
#<unspecified>
> v
#(pooh rabbit tiger)
> (vector-length '#(alice hair hatter))
3
> (vector-length '#(alice))
1
> (vector-length '#())
0
```

データがベクトルかどうかを調べるには、手続き vector? を使います。

◇ (vector? 〈データ〉)
— 〈データ〉の値がベクトルデータなら真 #t を、そうでなければ偽 #f を返します。

```
> (vector? '#(alice cheshire-cat caterpiller))
#t
> (vector? '(alice cheshire-cat caterpiller))
#f
```

注意: プログラムの中に'#(...) という形で書いてあるベクトルは、書き換えてはいけません。プログラム中のベクトルは、その値が固定されているべきもの (定数) なので、書き換えると意図に反した動作となることがあります。 ベクトルは数とは違って自己評価的ではないために、定数ベクトルをプログラム中で使用するときには、次のように引用符を付けないといけません。

```
> (define a '#(kanga roo))
#<unspecified>
> a
#(kanga roo)
```

ベクトルデータとリストデータとは、次の手続きを使うことにより、相互に変換することができます。

◇ (vector->list ⟨ ベクトル ⟩ ⟨ リスト ⟩
— ⟨ ベクトル ⟩ を ⟨ リスト ⟩ に変換します。

◇ (list->vector ⟨ リスト ⟩ ⟨ ベクトル ⟩
— ⟨ リスト ⟩ を ⟨ ベクトル ⟩ に変換します。

```
> (vector->list '#(mouse lory duck dodo))
(mouse lory duck dodo)
> (list->vector '(mouse lory duck dodo))
#(mouse lory duck dodo)
> (vector-ref (list->vector '(mouse lory duck dodo) 2))
duck
```

6.1.3 数 (もう少し詳しく)

3.6.2 では、基本的なデータ型である整数と実数と基本的な演算について学びました。ここでは、もう少し**数** (number) について詳しく勉強します。

標準 Scheme では、**整数** (integer)、**有理数** (rational) (**分数**とも呼ばれます)、**実数** (real)、**複素数** (complex) が、数の部分型として用意されています。

数学的な立場で考えると、数 26 は整数ですが、$\frac{26}{1}$ とも考えることができるので、26 を有理数ともみなせます。また 26.0 と考えれば 26 は実数でもあり、$26.0 + 0.0i$ と考えれば複素数ともみなせます。一方、1.5 は実数で、$1.5 + 0.0i$ と考えれば複素数とみなせますが、この数は整数でも有理数でもありません。これらの関係を図示すれば、図 6.2のような部分型の関係を形成することになります。これと同様な概念が Scheme にも用意されています。

データ型を判定する手続きとして、次のものが用意されています。

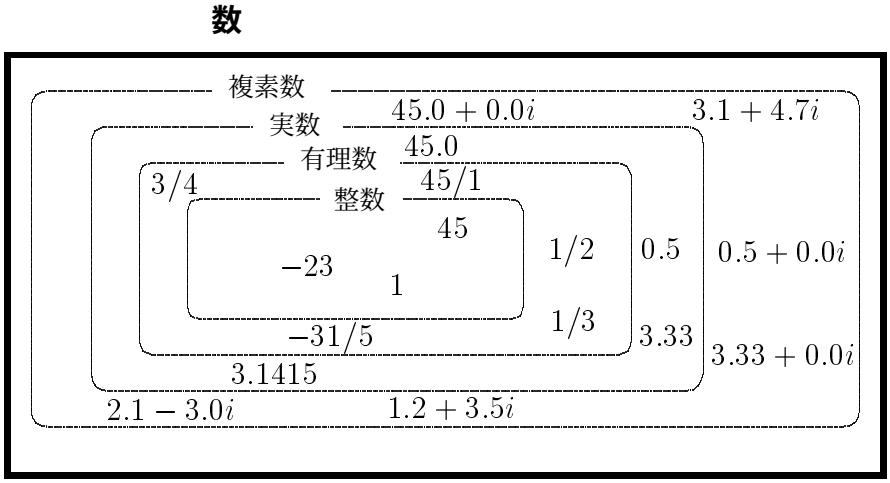


図 6.2: 数の部分型の関係

◇ (number? 〈データ〉)
— 〈データ〉が数なら真 #t を、そうでなければ偽 #f を返します。

◇ (integer? 〈データ〉)
— 〈データ〉が整数なら真 #t を、そうでなければ偽 #f を返します。

◇ (rational? 〈データ〉)
— 〈データ〉が有理数なら真 #t を、そうでなければ偽 #f を返します。

◇ (real? 〈データ〉)
— 〈データ〉が実数なら真 #t を、そうでなければ偽 #f を返します。

◇ (complex? 〈データ〉)
— 〈データ〉が複素数なら真 #t を、そうでなければ偽 #f を返します。

これらを試してみましょう。

```
> (integer? 26)
#t
> (real? 26)
#t
> (complex? 26)
#t
> (integer? 1.5)
#f
> (real? 1.5)
#t
```

```
> (complex? 1.5)
#t
```

有理数や複素数に関しては、それら独特の手続きがあります。いずれも Scheme の必須な手続きではないので、これらの手続き (特に有理数関連) を持っていない Scheme 処理系もいくつかあります。これらを使用する前に、処理系のマニュアルを参照して下さい。

△ (rationalize x y)
— x を有理数にした値を返します。ただし返される有理数は x から y よりも大きく離れていない有理数で、もっとも単純なものです[6]。

△ (numerator n)
— n の分子部分を返します。 返される値は n が既約分数で表したときの分子です。

△ (denominator n)
— n の分母部分を返します。 返される値は n が既約分数で表したときの分母で、常に正の値です。n が 0 のときの分母は 1 と決められています。

△ (make-rectangular x y)
— 複素数 $x + yi$ (i は虚数単位) を返します。

△ (make-polar r t)
— 複素数 $r \cdot e^{it}$ (i は虚数単位) を返します。

△ (real-part z)
— 複素数 z の実数部を返します。

△ (imag-part z)
— 複素数 z の虚数部を返します。

△ (magnitude z)
— 複素数 z の絶対値を返します。

△ (angle z)
— 複素数 z の偏角を返します。

次はプログラムの中に数を書く方法を説明します。これまででは数の書き方は 10 とか 3.14 といった方法だけでしたが、この以外にも色々な数の書き方が用意されています。たとえば、これまで数は 10 進数で書いていましたが、16 進数でも書くこともできます[7]。16 進数で数を書くには、 まず #x を書き、その後に数を 16 進数で書きます。(x は hexadecimal の x を表しています。)

[6] 2 つの既約分数 $r_1 = n_1/d_1$ と $r_2 = n_2/d_2$ に対し $|n_1| \leq |n_2|$ かつ $|d_1| \leq |d_2|$ であれば、r_1 は r_2 よりも単純である、といいます。たとえば、 3/5 は 4/7 よりも単純です。$0 = 0/1$ はすべての有理数より単純ですが、2/7 と 2/5 ように、単純さの比較ができない有理数の組もあることに注意して下さい。

[7] 10 進数は私たちが普段使っている数の表記方法で、ひとつの桁に 0 から 9 までの 10 個の数字を使い、数を書き表しています。16 進数とは、ひとつの桁に 0 から 9 と、a から f までの 16 個の文字を使い、数を書き表す方法です。(「数」と「数字/文字」の違いに気をつけて下さい。) たとえば、16 進数で a を 10 進数で書けば 10 で、16 進数で 13 を 10 進数で書けば 19 ($= 1 \cdot 16 + 3 \cdot 1$) です。

```
> #x10          — 16 進数での 10 (10 進数では 16)
16              — 10 進数で表示されている
> #x1f
31
```

このほか、2 進数 (#b)、8 進数 (#o)、10 進数 (#d) で数を書くことができます。

```
> #b1011        — 2 進数での 1011 は
16                — 10 進数では 10 です
> #o71          — 8 進数での 71 は
57                — 10 進数では 57 です
> #d26          — 10 進数での 26 は
26                — 10 進数で 26 です
> 26            — なにも指定しなければ 10 進数と解釈されます
26
> (+ #b10 #o10 #d10 #x10)     — これは (+ 2 8 10 16) に等しい
36
```

0 がたくさん付いている非常に大きな数や 0 に近い小さな数を書く場合、0 の数を間違って入力してしまう可能性がありますし、読む側も 0 の数を間違ってしまう可能性があります。Scheme ではこのような場合のために、実数を表すのに便利な記法が用意されています。

たとえば 54.0e3 は $54.0 \times 10^3 = 54000.0$ を、167.5e-5 は $167.5 \times 10^{-5} = 0.001675$ をそれぞれ表しています。一般に MeEは $M \times 10^E$ を表す書き方で、**浮動小数点表示**(floating-point representation) といいます[8]。 (ただし E は整数でないといけません。) M は**仮数部**(mantissa)、E は**指数部**(exponent) とそれぞれ呼ばれています。

```
> (+ 32.0e2 1.1e3)
4.3e3
> (* 5.0e+10 2)
100.0e9
> (* 1.0e64 31e10)
3.1e75
```

プログラム中で数の厳密性を陽に指定することもできます。厳密数であることを指定する数の書き方は、数の前に #e を書きます。いっぽう非厳密数であることを指定するには、数の前に #i を書きます。e は exact (厳密型) の、i は inexact (非厳密型) の頭文字となっています。

[8] e 以外にも、s,f,d,l を使うこともできます。これらは数の精度を指定するもので、それぞれ short, single, double, long の精度を表しています。e は、処理系の標準の精度を使うことを表しています。

```
#f
> #e3
3
> #i3
3.0
> (inexact? #e3)
#f
> (inexact? #i3)
#t
```

数を文字列へ変換したり、文字列を数に変換する手続きもあります。

◇ (number->string ⟨数⟩)
— ⟨数⟩を文字列に変換します。

◇ (string->number ⟨文字列⟩)
— ⟨文字列⟩を数に変換します。

```
> (string->number "300")
300
> (string->number "#x1f")
31
> (string->number "31.4e-1")
3.14
> (number->string 300)
"300"
> (number->string 6.626e-34)
"662.6e-36"
> (number->string (+ 1 2))
"3"
```

手続き string->number に、数の文字列での表現として正しくないものが渡されたときは#f が返されます。

```
> (string->number "pooh")
#f
> (string->number "zero")
#f
```

6.1.4 文字列 (もう少し詳しく)

3.6.3 では、文字列の基本的な概念と基礎的な使い方を学びました。ここでは文字列についてもう少し詳しく説明し、文字列に対する他の手続きを紹介します。

3.6.3 で学んだのは、"Pooh loves honey." のようにあらかじめプログラム中に書いておき、それを表示させたり、他の文字列とつなげたりするだけでした。

まず最初に新しい文字列をつくり出す手続きを紹介します。

◇ (make-string k)
— 長さが k の文字列を新しく作り、その文字列を返します。ただし、どのような文字より構成される文字列になるかは、処理系によって違います。

◇ (make-string k ⟨ 文字 ⟩)
— ⟨ 文字 ⟩ が k 個連なった (すなわち、長さが k の) 文字列を新しく作り、その文字列を返します。

◇ (string ⟨ 文字$_1$⟩ ⟨ 文字$_2$⟩ $\cdots$)
— 最初の文字が ⟨ 文字$_1$⟩、二番目の文字が ⟨ 文字$_2$⟩、$\cdots$ である文字列を新しく作り、その文字列を返します。

上の説明で「新しく作る」というのは、その文字列を保持するための記憶領域を新たに確保する、という意味です。このことを利用して、プログラムの実行中に文字列用の記憶領域を増やしてゆくことができます。

次に文字列に含まれている文字を調べたり、文字列の内容 (の一部) を書き換える方法を説明します。文字列 s は $c_0c_1...c_{n-1}$ のように、文字 c_i がならんだものと考えます。Scheme では、一番最初の文字を 0 番目の文字とし、以降 1 番目、2 番目、$\cdots$ としています。文字列の中身の参照や変更のために、以下の手続きが用意されています。

◇ (string-ref ⟨ 文字列 ⟩ k)
— ⟨ 文字列 ⟩ の k 番目の文字を返します。文字型のデータが返されます。

◇ (string-set! ⟨ 文字列 ⟩ k ⟨ 文字 ⟩)
— ⟨ 文字列 ⟩ の k 番目を、⟨ 文字 ⟩ に書き換えます。

これらの手続きに引数として与える k は、$0 \leq k <$ (string-length ⟨ 文字列 ⟩) でないといけません。

```
> (define s (make-string 8 #\x))
#<unspecified>
> s
"xxxxxxxx"
```

```
> (string-set! s 0 #\y)
#<unspecified>
> s
"yxxxxxxx"
> (string-ref s 0)
#\y
> (string-ref s 1)
#\x
```

その他、次の手続きも用意されています。

△ (string-copy ⟨ 文字列 ⟩)
— ⟨ 文字列 ⟩ と同じ長さの文字列を新しく作り、その文字列に ⟨ 文字列 ⟩ をコピーして返します。つまり、⟨ 文字列 ⟩ と同じ内容を持つ文字列を新しく作ります。

△ (string-fill! ⟨ 文字列 ⟩ ⟨ 文字 ⟩)
— ⟨ 文字列 ⟩ のすべての文字を ⟨ 文字 ⟩ に書き換えます。

注意: プログラムの中に書いてある文字列を書き換えてはいけません。次のような間接的な書き換えてもいけません。

```
(let ((name "pooh"))
  (string-set! name 1 #\w))
```

プログラム中の文字列は、その値が固定されているべきもの (定数) なので、書き換えると意図に反した動作となることがあります。 ある文字列を必要に応じて書き換えるときは、string-copy などの手続きでその文字列のコピーを作り、それを書き換えるようにして下さい。新たに作り出した文字列に対しては、自由に書換えをすることができます。

文字列と記号の橋渡しをする手続きとして、以下のものが用意されています。

◇ (symbol->string ⟨ 記号 ⟩)
— ⟨ 記号 ⟩ を文字列にします[9]。

◇ (string->symbol ⟨ 文字列 ⟩)
— ⟨ 文字列 ⟩ を記号にします。

[9] もし ⟨ 記号 ⟩ がプログラム中にかかれているものや手続き read によって返されたものとき、この手続きの返す文字列が大文字か小文字であるかは処理系に依存します。もし ⟨ 記号 ⟩ が手続き string-symbol によって返された記号のときは、この手続きの返す文字列の大文字/小文字は手続き string-symbol に与えられた文字列と一致します。この手続きで返された文字列を、手続き string-set! などの書換えをする手続きで書き換えることは許されていません。

6.2 入出力

これまででは、display を使ってデータを画面に表示させていました。この他にもデータ表示のための手続きが用意されています。データを画面に表示するだけでなく、ファイルに書き込んだり、ファイルやキーボードからデータを読み込むこともできます。ここでは Scheme でのデータの入出力の方法を詳しく説明します。

6.2.1 データの表示と改行

データの表示のための手続きには、次のものが用意されています。

◇ (display ⟨ データ ⟩)
— ⟨ データ ⟩ を表示します。もし ⟨ データ ⟩ が文字列のときは、二重引用符 (") で囲まずに表示します。

◇ (write ⟨ データ ⟩)
— ⟨ データ ⟩ を表示します。もし ⟨ データ ⟩ が文字列のときは、二重引用符 (") で囲って表示します。

◇ (write-char ⟨ 文字 ⟩)
— ⟨ 文字 ⟩ を表示します。

これらの手続きは、標準の出力先にデータを表示するものです。標準の出力先は、とくに何も指定しなければ画面となっていますが、他に切替えることもできます。これについては、後で詳しい説明をします。

これらの手続きの実行例は次の通りです。

```
> (write 34)
34#<unspecified>
> (write '(bear "Pooh"))
(bear "Pooh")#<unspecified>
> (write '(bear "Pooh"))
(bear Pooh)#<unspecified>
```

これらの例では、データを表示した直後に#<unspecified>が表示されています。これは、write や display の返す値が#<unspecified> であるため、式の評価による表示 (34 など) のあと、式の評価結果が表示され、上記のような表示となっているのです。(write や display の返す値は、 Scheme 処理系に依存しています。)

write や display は、改行をしないことに注意してください。

```
>(begin (display "Winnie-") (display "the-") (display "Pooh"))
Winnie-the-Pooh#<unspecified>
```

改行するには次の手続きを使います。

◇ (newline)
— 改行をします。カーソルは次の行の左端に移動します。

```
>(begin (display "Pooh") (newline) (display "Piglet"))
Pooh
Piglet#<unspecified>
```

(newline) の代わりに、write-char を使って (write-char #newline) としてもまったく同じです。

6.2.2 入出力とファイル

以上では画面にデータを表示させる方法でした。コンピューターに実務的な仕事をさせるには、ディスク上のファイルに書かれている内容を読みとって計算をしたり、その結果を別のファイルに書き込むことが必要になります。このように、データの書き込みや読み出しのことを、**入出力**(にゅうしゅつりょく)(input/output) といいます。

ファイルはディスクに書き込まれ、それぞれのファイルには名前 (ファイル名) が付けられます。 ファイルを読み書きするときは、ファイル名で対象のファイルを指定します。入出力はディスク上のファイルに対してだけでなく、キーボード、画面、プリンタなどの機器に対しても行なうことができます。これは、ディスク上のファイルと同じ手順で機器に対する読み書きができる仕組みを、オペレーティングシステムが用意しているからです。このため、利用者は入出力の対象がどの装置になっているかを気にすることなく、統一的な方法で入出力をするプログラムを書くことができます[10]。

それぞれのファイルは、文字が順にならんだもの、とみなされています。たとえば、あるファイルの中身が

```
Pooh
Owl
```

だったとしましょう。画面に表示すると二次元的な構造をしているように見えますが、それぞれの行の右端に改行文字があり、次の行の左端に続いてます。 上の例では P o o h #\newline O w l という具合に、文字がならんでいます。

[10]ただし、ファイルに対するいくつかの操作が制限されることがあります。ディスクファイルなら何度も同じデータを読んだり、内容を書き換えたりできます。しかし、キーボードからの入力を何度も同じデータを読むことはできませんし、プリンタに印刷した内容を書き換えることはできません。

6.2.3 ファイルのオープンとポート

Scheme では入出力先の名前を直接指定して入出力をするのではなく、**ポート** (port) と呼ばれる Scheme データで指定します。入出力先はポートを生成するときに指定します。すなわち入出力先はポートに結びつけられていて、ポートによって入出力先を間接的に指定します。ポートには**入力ポート** (input port) と**出力ポート** (output port) があり、それぞれデータの入力、出力に使われます。

ファイルへの入出力は、以下の流れで行ないます。

1. ファイルを開き、 そのファイルに対するポート p を得ます。
2. ポート p から、データを読み出します。(あるいは、p にデータを書きます。)
3. ポート p を閉じて、入出力を終了します。

ファイルのオープンには次の手続きを使います。

◇ `(open-input-file` 〈 ファイル 〉`)`
— 〈 ファイル 〉を読み込みのためにオープンし、入力ポートを返します。〈 ファイル 〉は文字列でないといけません。

◇ `(open-output-file` 〈 ファイル 〉`)`
— 〈 ファイル 〉を書き込みのためにオープンし、出力ポートを返します。〈 ファイル 〉は文字列でないといけません。

ポートへデータを書き込むには次の手続きを使います。(ファイルの読み込みについては次節で説明します。)

◇ `(write` 〈 データ 〉〈 ポート 〉`)`

◇ `(display` 〈 データ 〉〈 ポート 〉`)`

◇ `(write-char` 〈 データ 〉〈 ポート 〉`)`
— 〈 データ 〉を 〈 ポート 〉に書き込みます。

〈 ポート 〉は省略可能で、もし省略されていれば**現在出力ポート** (current output port) に書き込まれます。ふつうは画面が現在出力ポートでの出力先になっています。このために、ポートを指定せずにこれらの手続きを呼び出せば、画面に表示がされるわけです。

ポートをクローズするには、次の手続きを使います。

◇ `(close-input-port` 〈 入力ポート 〉`)`
— 〈 入力ポート 〉をクローズします。

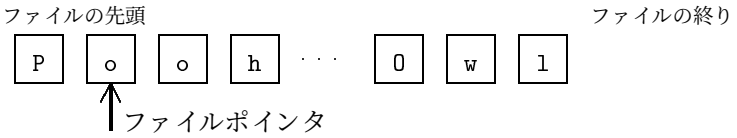


図 6.3: ファイルポインタの概念図

◇ (close-output-port 〈出力ポート〉)
— 〈出力ポート〉をクローズします。

データがポートかどうかを調べる手続きもあります。

◇ (input-port? 〈データ〉)
— 〈データ〉が入力ポートなら真 #f を、そうでなければ偽 #f を返します。

◇ (output-port? 〈データ〉)
— 〈データ〉が出力ポートなら真 #f を、そうでなければ偽 #f を返します。

多くの場合、キーボードからデータを読みとり、画面へ出力します。このため、入出力をする手続きでポートの指定が省略されていれば、標準のポートに対して入出力をするようになっています。標準の入力には**現在入力ポート** (current input port) が、標準の出力には**現在出力ポート** (current output port) が使われます。普通では、現在入力ポートはキーボードで、現在出力ポートは画面になっています[11]。

現在入力ポートと現在出力ポートは、次の手続きで得ることができます。

◇ (current-input-port)
— 現在入力ポートを返します。

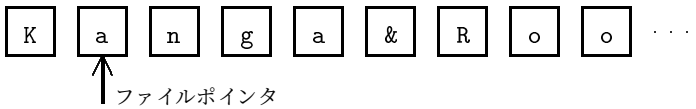
◇ (current-output-port)
— 現在出力ポートを返します。

6.2.4 読み込み

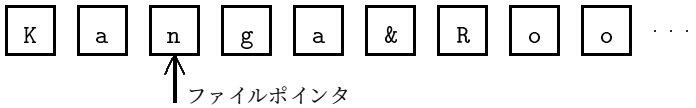
ファイルの読み込みをするときは、「いまどこを読もうとしているか」が重要となります。「ファイルのどこを読もうとしているか」を表すものを、「ファイルポインタ」といいます。ファイルをオープンした直後では、ファイルポインタは最初の文字を指しています。ファイルポインタは、データを読み込むにつれて進められます。図 6.3に、ファイルポインタの概念図を示します。この図では、内容が Pooh · · · Owl であるファイルの 2 番目の文字にファイルポインタがあります。

ファイルからのデータを読み込みは、以下の 2 通りの方法が用意されています。

[11]ほとんどのオペレーティングシステムでは、ファイルへの入出力と同じ方法で、キーボードや画面に対する入出力ができるようになっています。



(a) 読み込み前のファイルポインタ



(b) 1 文字読み込んだ後のファイルポインタ

図 6.4: 文字単位での読み込み

- 「文字」を単位とした読み込み
- Scheme の「式」を単位とした読み込み

文字を単位とした読み込みでは、ファイルポインタの指す文字を 1 文字ずつ、順番に読み出してゆきます。図 6.4では、文字を読む前と読んだ後のファイルポインタを図示しています。(この場合での読み込まれた文字は a です。)

式を単位とした読み込みでは、おおまかに言って次のようにしてデータが読み込まれます。

1. 式として関係ない空白文字を読み飛ばす。
2. 空白文字などを読み飛ばした後、最初に現れた文字によって次のような動作をします。
   - コメントの始まりの文字 ; のとき。
     行の終り (改行文字) まで読み飛ばします。
   - 数字やアルファベット文字のとき。
     次の空白文字が現れるまで、数または記号として読み込みます。
   - 二重引用婦 " のとき。
     対応する二重引用婦 " に出会うまで、文字列として読み込みます。
   - 開き括弧 (のとき。
     対応する閉じ括弧) に出会うまで読み込みます。

図 6.5に、式を単位とした読み込みを図示しています。(何も書かれていない四角は、空白文字を表しています。)

読み込みのための手続きとして、以下のものがあります。

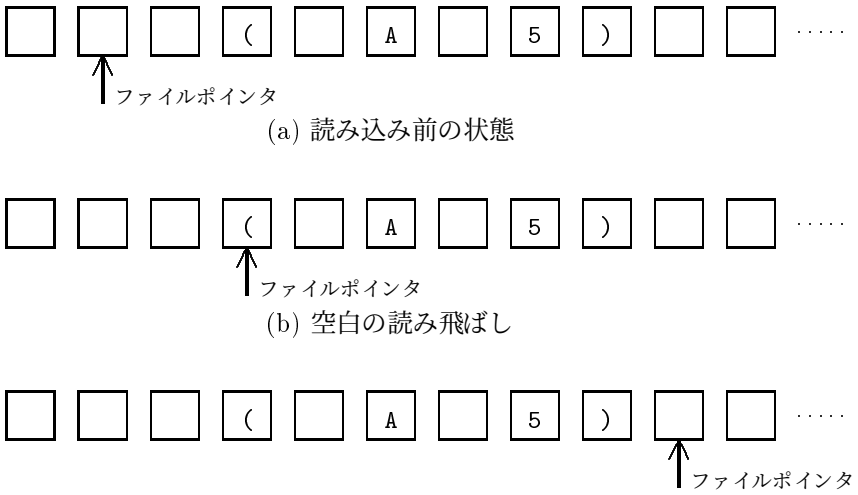


図 6.5: 式単位での読み込み

◇ (read ⟨ ポート ⟩)
— ⟨ ポート ⟩ から (Scheme データとしての) 式を 1 つ読み込み、その式を返します。もしポートが与えられなければ、現在入力ポートから読み込みます。

◇ (read-char ⟨ ポート ⟩)
— ⟨ ポート ⟩ から 1 文字読み込んで、文字型のデータを返します。もしポートが与えられなければ、現在入力ポートから読み込みます。

ファイルは無限に長くはなく、終りがあります。ファイルの終りにたどり着いたときは、ファイルの終りを表す特別なデータが返されます。この特別なデータを**ファイル終端子** (end-of-file object) といいます。ファイルの終りにたどり着いたことは、read や read-char の返したデータがファイル終端子かどうかを調べることで分かります。

◇ (eof-object? ⟨ データ ⟩)
— ⟨ データ ⟩ がファイル終端子なら真 #t を、そうでなければ偽 #f を返します。

文字単位での読み込みに関連して、次のものも用意されています。

◇ (peek-char ⟨ ポート ⟩)
— ファイルポインタを進めずに、⟨ ポート ⟩ から 1 文字読み込み、文字型のデータを返します。もしポートが与えられなければ、現在入力ポートから読み込みます。

△ (char-ready? ⟨ ポート ⟩)
— ⟨ ポート ⟩ で少なくとも 1 文字が読み込み可能となっているかを調べます。もしポートが与えられなければ、現在入力ポートを調べます。読み込み可能な文字があれば真 #t を、そうでなければ偽 #f を返します。

手続き read-char でキーボードから読む場合を考えます。もしキーが押されていないければ、read-char はキーが押されるまで待ち、そして押されたキーの文字を返します。これではビデオゲームのように、リアルタイムな動作をするプログラムは作れません。そのようなプログラムでは、まず手続き char-ready?を使ってキーが押されているかどうかを調べます。もしキー入力があればそれを読むようにすることで、プログラムの実行が入力待ちで中断しないようにします。

6.2.5 その他の手続き

以上で説明した入出力の方法では、ファイルのオープンとクローズをする手続きを使わないといけませんでした。これを省略できる手続きも用意されています[12]。

△ (call-with-input-file ⟨ ファイル ⟩ ⟨ 手続き ⟩)
— ⟨ ファイル ⟩ を読み込みオープンをしてポートを作り、⟨ 手続き ⟩ にそのポートを引数として呼び出します。⟨ 手続き ⟩ の評価が終ればポートは自動的にクローズされ、⟨ 手続き ⟩ の評価結果を返します。なお⟨ ファイル ⟩ は文字列で、⟨ 手続き ⟩ は 1 引数の手続きでないといけません。

△ (call-with-output-file ⟨ ファイル ⟩ ⟨ 手続き ⟩)
— ⟨ ファイル ⟩ を書き込みオープンをしてポートを作り、⟨ 手続き ⟩ にそのポートを引数として呼び出します。⟨ 手続き ⟩ の評価が終ればポートは自動的にクローズされ、⟨ 手続き ⟩ の評価結果を返します。なお ⟨ ファイル ⟩ は文字列で、⟨ 手続き ⟩ は 1 引数の手続きでないといけません。

◇ (with-input-from-file ⟨ ファイル ⟩ ⟨ 手続き ⟩)
— ⟨ ファイル ⟩ を読み込みオープンをしてポートを作り、そのポートを現在入力ポートにして ⟨ 手続き ⟩ を呼び出します。⟨ 手続き ⟩ の評価が終ればポートは自動的にクローズされ、⟨ 手続き ⟩ の評価結果がこれらの手続きの評価値として返され、現在入力ポートはもとのポートに戻ります。なお ⟨ ファイル ⟩ は文字列で、⟨ 手続き ⟩ は無引数の手続きでないといけません。

◇ (with-output-to-file ⟨ ファイル ⟩ ⟨ 手続き ⟩)
— ⟨ ファイル ⟩ を書き込みオープンをしてポートを作り、そのポートを現在出力ポートにして ⟨ 手続き ⟩ を呼び出します。⟨ 手続き ⟩ の評価が終ればポートは自動的にクローズされ、⟨ 手続き ⟩ の評価結果がこれらの手続きの評価値として返され、現在出力ポートはもとのポートに戻ります。なお ⟨ ファイル ⟩ は文字列で、⟨ 手続き ⟩ は無引数の手続きでないといけません。

[12] call-with-input-file と call-with-output-file は Scheme に必須な手続きとは定められていません。そのため、処理系によってはこれらが用意されていない場合があります。

標準 Scheme で定められている手続きだけでは、細かなファイル操作はできません。多くの Scheme 処理では実用的な Scheme プログラムを作るために、ファイル操作に関する独自の手続きをいろいろと用意しています。たとえば次のような手続きがあります。詳しくは使っている処理系のマニュアルを参照して下さい。なおこれらの手続きに引数として与えるファイルの指定は、すべて文字列でないといけません。

√ (file-exists? ⟨ファイル⟩)
— ⟨ファイル⟩が存在するかどうかを調べます。もしあれば真 #t が、なければ偽 #f が返されます。

√ (force-output ⟨ポート⟩) または (flush-output-port ⟨ポート⟩)
— バッファリングされている出力文字を、強制的に出力します。(バッファリングとは、入出力命令が実行されるたび入出力をするのではなく、ある程度の入出力データが溜るまで実際の入出力を延期することをいいます。) ⟨ポート⟩が省略されたときは、現在出力ポートを指定したことになります。

√ (delete-file ⟨ファイル⟩)
— ⟨ファイル⟩を削除します。

√ (rename-file ⟨元ファイル⟩ ⟨新ファイル⟩)
— ファイルの名前を変更します。⟨元ファイル⟩を⟨新ファイル⟩という名前にします。

6.3 内部定義

次の手続きを見てください:

```
(define (foo n)
  (define (bar i)
    (* i i))
  (list n (bar n)))
```

手続き foo の内部で、 別の手続き bar が定義されています。ある手続きの内部での手続き (あるいは変数) の定義を、**内部定義** (internal definition) といいます。いっぽう内部定義ではない定義を**トップレベル定義** (top level definition) と呼びます。

内部定義の利点は、ある手続きの中だけで使える手続きや変数を定義できる点にあります。手続き foo がすでに定義してあるものとして、次の例を見てください。

```
> (define (bar) (+ i 1))
#<unspecified>
> (foo 3)
9
> (bar 3)
4
```

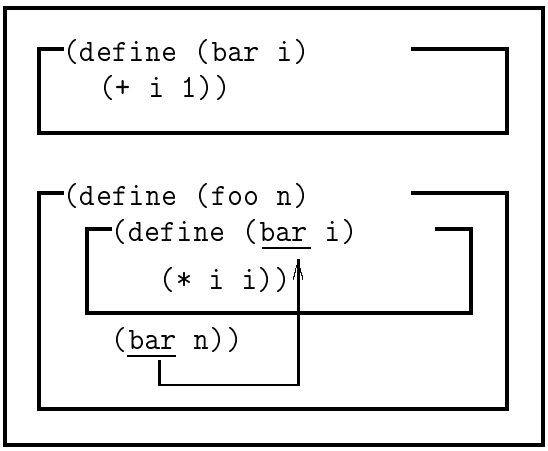


図 6.6: 内部定義 1

この例では、トップレベルで手続き bar を定義しています。手続き foo を呼び出すと、その中に現れる bar は図 6.6のように、内部で定義されている二乗を計算する手続きが使われます。手続き foo はその本体である式 (bar n) を評価し、その結果が foo の実行結果として返されます。

この例では、トップレベルでの bar の定義は、foo の実行に影響を与えていません。また逆に foo の実行は、bar のトップレベル定義にも影響を与えていません。

もし内部定義が使えないとすれば、別の手続きが同じ名前の変数を使っていて、その変数の値を書き換えてしまうかもしれません。そのことが原因でプログラムが予期しない動作をする可能性があります。このため、使おうとしている変数が他の部分で使われていないかどうか、注意深く調べる必要があります。しかしそれは大きなプログラムを作るときには大変な作業となってしまいます。内部定義を使うと他の部分で使われているかどうかをまったく気にせず、自由に変数を使うことができます。

今度は少し複雑な例を見てみましょう。次の一連の定義をした後に、式 (foo 2) を評価するとどうなるでしょうか?

```
(define (foo n)
  (define (bar n)
    (* n n))
  (list (bar n) (qwe n)))
(define (qwe n)
  (bar n))
(define (bar n)
  (+ n 1))
```

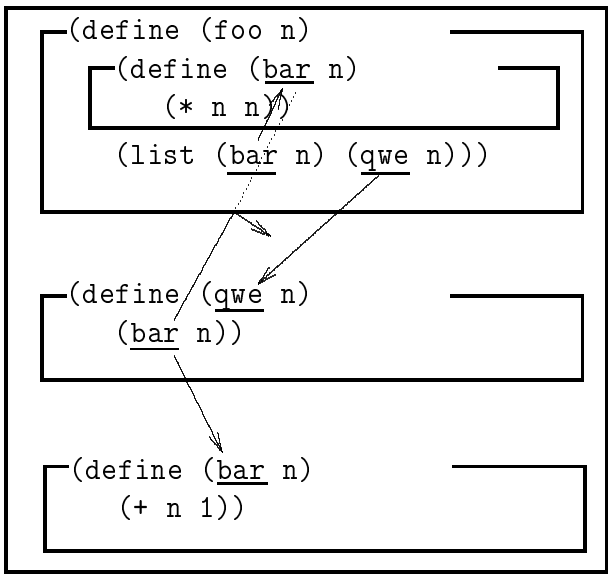


図 6.7: 内部定義 (2)

答えは (4 3) です。手続き foo の本体では、(list (bar n) (qwe n)) の評価結果を使っています。上で説明した理由で、(bar n) の評価結果は 4 になります。では (qwe n) の評価を考えましょう。手続き qwe の本体を見ると、(bar n) の評価結果を返すようになっています。ここで使われる bar は foo の内部で定義されたものではなく、トップレベルで定義されている方 (引数に 1 加える手続き) が使われます。ですので (bar n) の評価結果は 3 になります。

大切なことは、ある手続きの内部で定義されたものは外から見えなくなっている、ということです。上の例では、foo の内部定義の bar は foo の外からは見えません。図 6.7のように、プログラムの構造を箱が入れ子になっていると考えればよく分かります。同じところか外側にあるものは見えても箱の内側に入っているものは見えないという、マジックミラーで囲まれた部屋のようになっています。

変数の参照のとき文面的に一番近いものが参照される規則を**文面的有効域則** (lexical scoping) と呼んでいます。変数とその値を組を、**束縛** (binding) といい、変数名から値を決めるための表みたいなものです。参照される変数が、プログラムの字面によって (プログラム作成時に) 静的に決まることにより、**静的束縛** (static binding) とも呼ばれています。逆に、実行の (時間的な) 順序によって参照される変数が定まる規則を、**動的有効域則** (dynamic scoping) あるいは **動的束縛** (dynamic binding) と呼びます[13]。

[13]古い Lisp (Lisp 1.5 や MacLisp など) では動的有効域則が採用されていましたが、Scheme や Lisp の国際標準仕様である Common Lisp では静的有効域則が採用されています。

以上で説明した通り、手続きの中でしか使えない変数 (あるいは手続き) を、内部定義を使うことで作ることができます。大切なことは、手続きの中で変数 v を定義すると、その手続きの中だけで参照できる v の値を保持する領域があたらしく用意される、ということです。

次の手続き f を見てください。これは、「f を呼び出しときの引数 n に対し、 n^2を返す手続き」を作り出す手続きです。

```
(define (f n)
  (define value (* n n))
  (lambda ()
     value))
```

ここで、手続き f の本体は、変数 value を返す引数を持たないラムダ式です。

手続き f を呼び出すと、ラムダ式が実行され、手続きデータが返されます。これによって作られた手続きは、変数 value の値を返すだけのものです。

では、これを実行してみましょう。

```
> (define g2 (f 2))
#<unspecified>
> (g2)
4
> (define g6 (f 6))
#<unspecified>
> (g6)
36
> (g2)
4
```

この例では、f が呼ばれるごとに value のための記憶領域があらたに用意され、この記憶領域に引数の二乗が保持されます。手続き f は、lambda によって作られる手続きデータを返すのですが、その中で使われている value は、先ほどあらたに用意された記憶領域を使うものです。変数 value の値を保持する場所は、f を呼び出すたびに違った場所が使われます。このために、手続き g2 と g6 の呼び出しではそれぞれ違った値が返され、上の例で見たような結果になるわけです。

ラムダ式を評価して得られるデータは手続きデータですが、それにはラムダ式の定義 (引数のならびと本体) と、そのラムダ式が評価されたときの束縛の情報が含まれています。

この束縛の情報は**環境** (environment) と呼ばれ、変数を参照するとき使う束縛を表しています。手続きデータに引数が与えられてその本体が評価されるときは、手続きデータのもつ環境で本体が評価されます。

上の例では、g2 (に束縛されている手続きデータ) の環境では value の値は 4 で、g6 (に束縛されている手続きデータ) の環境では 36 となっていますので、字面上ではおなじ手続き (lambda () value) を評価しても、返される値が違うことになります。

6.4 制御構造 (その 2)

ここでは知っておくと便利な制御構造のいくつかを勉強します。

6.4.1 case による場合分け

場合分けをして評価する式を選ぶ方法として、 if と cond による方法を前に学びました。もうひとつの方法の case をここで紹介します。まずは例を見てください。

```
> (case 'kennedy
    ((kennedy roosevelt lincoln) 'president)
    ((tanaka nakasone murayama)  'prime-minister))
president
> (case 'murayama
    ((kennedy roosevelt lincoln) 'president)
    ((tanaka nakasone murayama)  'prime-minister))
prime-minister
```

このように case は cond と違って、判定式は書きません。一般に case は以下の形式をしています。

◇ (case ⟨ キー ⟩
⟨ 節$_1$⟩
⟨ 節$_2$⟩
⋮
⟨ 節$_n$⟩)

ただし ⟨ 節 ⟩ は

((⟨ データ ⟩ ···) ⟨ 式$_1$⟩ ⟨ 式$_2$⟩ ···)

の形をしていて、それぞれの ⟨ データ ⟩ は互いに違うものでないといけません。

最後の節を **else 節** (else clause) にすることもできます。else 節とは、

(else ⟨ 式$_1$⟩ ⟨ 式$_2$⟩ ···)

の形をした節のことです。⟨ キー ⟩ がどの ⟨ データ ⟩ にも一致しないとき、else 節が選ばれます。

case の動作は次の通りです。まず最初に、⟨ キー ⟩ が評価されます。その値と等しい ⟨ データ ⟩ を持つ ⟨ 節 ⟩ を見つけます。(キーの評価値と⟨ データ ⟩ の比較には eqv? を使います。) 次に見つけた ⟨ 節 ⟩ の ⟨ 式 ⟩ を左から順に評価してゆき、最後の ⟨ 式 ⟩ の評価値を case の値として終了します。

もしキーの評価値と等しい ⟨ データ ⟩ を持つ ⟨ 節 ⟩ がないとき、case が返す値は処理系に依存します。もし ⟨else 節 ⟩ があれば⟨else 節 ⟩ の式が左から評価され、最後の式が case の評価値となります。

⟨else 節 ⟩ を使った例として、次の手続き number->name を見てみましょう。

```
(define (num->name n)
  (case n
     ((1)  'one)
     ((2)  'two)
     ((3)  'three)
     (else 'many)))
```

これを実行させてみると、次のようになります。

```
> (num->name 1)
one
> (num->name 2)
two
> (num->name 3)
three
> (num->name 4)
many
> (num->name 5)
many
```

6.4.2 名前付き let による繰り返し

繰り返しを実行のための構文、名前付き let を紹介します。名前付き let は標準 Scheme の仕様では必須な構文でないため、使えない Scheme 処理系もあります。使えるかどうかは、使用している処理系の説明書を参照してください。(名前付き let がなくても次に紹介する letrec で同じことができます。)

名前付き let は let とほぼ同一で、局所変数を持つ環境を作ります。let に名前を付け、その名前を手続きとして呼び出すことで名前付き let の本体の先頭に再び実行を移すことができるところが let と違う点です。

構文は

△ (let ⟨ 名前 ⟩ ((⟨ 変数$_1$⟩ ⟨ 式$_1$⟩)
(⟨ 変数$_2$⟩ ⟨ 式$_2$⟩)
⋮
(⟨ 変数$_n$⟩ ⟨ 式$_n$⟩))
⟨ 本体 ⟩)

となっています。普通の let と違う点は ⟨ 名前 ⟩ の部分です。

では例を見てみましょう。

```
(let loop ((l '(a b c)) (n 0))
  (if (null? l)
      n
      (loop (cdr l) (+ n 1))))
```

この例では loop という名前が付けられていて、局所変数として l と n があります。名前付き let の本体は、 次のような形になっています。

- もし l が空リストなら、 n を返す。
- そうでないときは、(cdr l) と (+ n 1) の評価結果を引数にして、 loop を呼び出す。

これがどのような動作をするかを説明します。まず、 局所変数 l と n が用意されて、それらの初期値はそれぞれ (a b c) と 0 になります。そして名前付き let の本体の式 (if ...) の評価がされます。名前付き let の本体の評価は、名前付きでない普通の let と同じですが、(loop ⟨arg$_1$⟩ ⟨arg$_2$⟩) の評価は次のようになります。まず ⟨arg$_1$⟩ と ⟨arg$_2$⟩ を評価し、その結果結果を名前付き let の局所変数の新しい値にします。そして名前付き let 文の本体を再び評価します。これにより、繰り返し実行が実現されます。

上の例では、局所変数 l と n の最初の値はそれぞれ (a b c) と 0 です。l は空リストでないので、(loop (cdr l) (+ n 1)) が評価されます。これにより局所変数 l と n の値はそれぞれ (b c) と 1 となり、再び名前付き let の本体が評価されます。最終的には l は空リストになり、n を名前付き let の値として返して実行が終了し、評価値 3 を得ます。

別の例を見てみましょう。

```
(define (sum n)
  (let loop ((i n) (total 0))
    (display (list i total))
    (newline)
    (if (= i 0)
```

```
        total
        (loop (- i 1) (+ total i)))))
```

この例は 名前付き let を使い、引数に与えられた整数 n に対して 1 から n までの総和を求める手続きです。(局所変数の変化を表示するよう、display, newline を入れています。) i は n から 1 まで順々に値が変わり、total は n から i までの和を保持します。i が 0 になれば繰り返しは終了で、求める総和は total に保持されているので、この値を返しています。

この手続きに引数 6 を与えて評価してみると、次のようになります。

```
> (sum 5)
(5 0)
(4 5)
(3 9)
(2 12)
(1 14)
(0 15)
```

6.4.3 letrec による繰り返し

letrec を使うことで、繰り返し実行や相互再帰を実現できます。その構文は、

◇ (letrec
((⟨ 変数$_1$ ⟩ ⟨ 式$_1$ ⟩)
(⟨ 変数$_2$ ⟩ ⟨ 式$_2$ ⟩)
⋮
(⟨ 変数$_n$ ⟩ ⟨ 式$_n$ ⟩))
⟨ 本体 ⟩)

となっています。⟨ 式 ⟩ は λ 式でないといけません。

まずは letrec の簡単な使い方である、名前付き let と同様な使い方を例を使って説明します。

```
(define (sum n)
  (letrec
    ((loop
      (lambda (i total)
        (display (list i total))
        (newline)
        (if (= i 0)
```

```
        total
        (loop (- i 1) (+ total i))))))
  (loop n 0)))
```

これは名前付き let で出てきた例を letrec を使って書き直したものです。

この例では、 lambda で作られる手続きを値として持つ局所変数 loop を作っています。そしてこの局所変数のある環境の中で、letrec の本体 (loop n 0) を評価しています。これにより手続き呼び出しが実行され、(lambda ···) の中身が評価されます。(lambda ···) の中で、(loop ···) という部分があります。これを評価すると再び loop の先頭から実行が開始され、繰り返しが行なわれることになります。局所変数の更新は名前付き let で説明したときと同じです。

これまでは単純な繰り返しだけでしたが、複数の手続きの間で繰り返しをすることもできます。次の例は引数が偶数かどうかを判定する手続きで、2 つの手続きの間で相互に呼び出し合う形の繰り返し実行の例です。

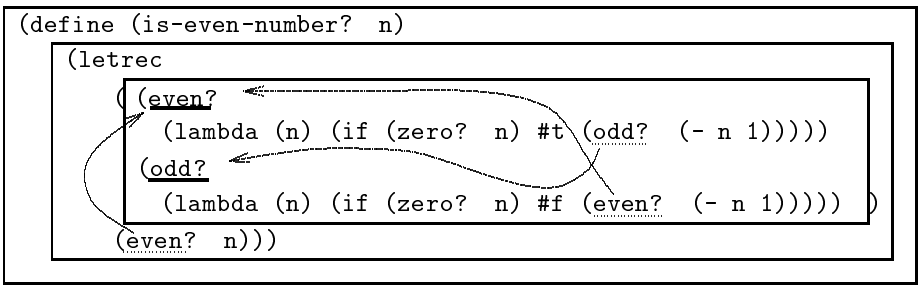
```
(define (is-even-number? n)
  (letrec
    ((even?
       (lambda (n) (if (zero? n) #t (odd? (- n 1)))))
     (odd?
       (lambda (n) (if (zero? n) #f (even? (- n 1))))))
    (even? n)))
```

これを実行してみましょう。

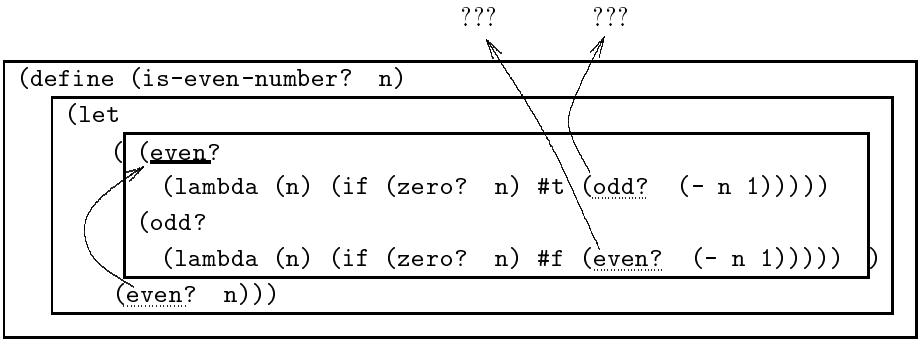
```
> (is-even-number? 4)
#t
> (is-even-number? 5)
#f
```

手続き is-even-number? では、局所手続き even? と odd? のそれぞれの中で、他の局所手続きを呼び出しています。

このように、局所手続き同士が互いに呼び出し合うことを**相互再帰** (mutual recursion) と呼びます。letrec ではその局所変数を持つ環境を作り、その環境の中で初期値が評価されます。いっぽう let では、局所変数の初期値は let の置かれた環境で評価されます。このため、もし図 6.8(b) のように let を使うと、even? の中で使われている odd? は let での局所変数ではなく、値を持っていないことになります。いっぽう letrec ではそのようなことはありません。letrec はこのような目的のために用意されています。



(a) letrec



(b) let

図 6.8: letrec と let

6.5 美しいプログラムを — 字下げと変数名

ここでは、美しいプログラムの書き方について考えます。プログラムが「美しい」とは、結果の出力の体裁や計算機画面の表示の美しさではなく、プログラムそのものの美しさのことをいいます。美しいプログラムは、プログラムの構造が一目で分かり、どのような働きをするプログラムかが容易に理解できます。さらに単に字面だけでなく、プログラムを構成するいくつかの部分がどのような機能を持ち、そしてどのように組み合わされて全体を構成するか、といった巨視的な観点からの美しさもあります。ここでは最初の一歩として、プログラムの構造が一目で分かるプログラムの書き方について学びます。

6.5.1 字下げ

階乗の計算を例にして考えてみましょう。自然数 n の**階乗**(factorial) $n!$ は、1 から n までの数を掛け合わせたもので、$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$ です。これの数学的な定義は次のようになります。

$$n! = \begin{cases} 1 & n = 1 \text{ のとき} \\ n \cdot (n-1)! & n \geq 2 \text{ のとき} \end{cases}$$

```
; procedure "factorial"
;   ---  computes factorial of n : n! = n (n-1) (n-2) ... 1
;   argument: n  -  integer (must be positive)
(define (factorial n)
  (if (= n 1)
    1
    (* n (factorial (- n 1)))))
```

(a) 美しいプログラム

```
;; purocedure f
(define (f x) (if (= x 1) 1
  (* x (f (- x 1)))))
```

(b) 見にくいプログラム

図 6.9: いろいなプログラムの書き方

階乗を計算する 2 つのプログラムを図 6.5.1(b) と (c) に示します。これらは同じ計算をしますが、図 6.5.1(a) の方が読みやすくなっています。引数の意味や手続きが何を計算するかが、コメントとして明確に記述されています。いっぽう (b) の方は何を計算する手続きなのかが記述されておらず、さらにプログラム自体も変な位置で改行してあるため、プログラムの構造が一目で理解できません。さらに手続きの名前も単に `f` としているだけで、手続き名から何をする手続きかを連想することが難しくなっています。

プログラムの構造が一目で分かるよう、行のはじめの部分に適切な数の空白文字をいれることを**字下げ** (indent) といいます。図 6.5.1(a) では適切な位置で改行をし、ちょうどいい数の空白文字が入れられています。

たとえ適切な位置で改行しても空白文字の数が適切でないと、プログラムの構造を簡単には読みとれません。こんどはフィボナッチ数列

$$F_i = \begin{cases} 1 & n = 1 \text{ のとき} \\ 1 & n = 2 \text{ のとき} \\ F_{i-1} + F_{i-2} & n \geq 3 \text{ のとき} \end{cases}$$

を例にします。図 6.5.1(a),(b) は、フィボナッチ数列の n 番目の数を計算するプログラムで、それぞれ字下げだけを変えたものです。

図 6.5.1(b) では変な字下げがされているために、プログラムがフィボナッチ数列の定義に合っているかどうかを確かめるのは大変です。

プログラムの字面が美しかろうが醜かろうが、プログラムの動作の正しさそのものには関係はありません。ですがプログラムを作成する段階で正しい字下げをし、適切な変数名を選ぶことで、プログラムに誤りが入る可能性はずっと減ります。別な見方をすれば、プ

```
; compute n-th Fibonacci
(define (fibonacci n)
  (if (= n 1)
    1
    (if (= n 2)
      1
      (+ (fibonacci (- n 1)) (fibonacci (- n 2))))))
```

(a) 良い字下げ

```
; compute n-th Fibonacci
(define (fibonacci n)
  (if (= n 1)
    1
  (if (= n 2)
      1
      (+ (fibonacci (- n 2))
  (fibonacci (- n 1))))))
```

(b) 良くない字下げ

図 6.10: 字下げ

ログラムの間違い探しに時間を費やすこることを避けるためにも、字下げと変数名には注意を払うべきです[14]。

以下では Scheme での字下げの方法のいくつかを示します。字下げのルールは人の好みによって違いますので、どれが良いとは一概にはいえません。ここでは筆者が使っている字下げの方法を紹介します。大原則のひとつは、ひとつの行が画面の右端を越えないようにする、ということです。画面の幅はたいてい 80 文字で、これを越えた長さの行は画面の次の行に折り返されて、画面の左端から表示される場合が多いです。もしそのように表示されると、プログラムの構造を読みとるのが困難になります。そのためにも、1 行 を 80 文字以下にすることを奨めます。

- `define` 構文
  手続きの本体を 2 つの空白を入れて字下げします。

  `(define (`⟨ 手続き名 ⟩ ⟨ 引数のならび ⟩`)`
  ⟨ 手続きの本体 ⟩`)`

[14]プログラムの字下げを自動的にしてくれるソフトウエアがあります。ですが、いい加減な字下げでプログラムを書き、一番最後に字下げプログラムを使って字下げして完成品にすることは、上記の趣旨に反することなのでするべきではありません。

手続きでないときは、次のようにします。

```
(define ⟨変数名⟩ ⟨式⟩)
```

もし ⟨式⟩ が長くて画面の右端を越えるときは、次のようにします。

```
(define ⟨変数名⟩
  ⟨式⟩)
```

- if 構文
  if の中の式を、2 つの空白を入れて字下げします。

```
(if ⟨式⟩
  ⟨式1⟩)
```

あるいは

```
(if ⟨式⟩
  ⟨式1⟩)
  ⟨式2⟩)
```

- cond 構文
  cond 構文は、 節がならぶように書きます。

```
(cond
  (⟨テスト1⟩ ⟨式⟩ ···)
      ⋮
  (⟨テストn⟩ ⟨式⟩ ···))
```

もし ⟨テスト⟩ が長いときは、次のようにします。

```
(cond
  (⟨テスト1⟩
   ⟨式⟩ ···)
      ⋮
  (⟨テストn⟩
   ⟨式⟩ ···))
```

- let 構文
  局所変数と本体とで字下げを変えて、 区別しやすくします。

```
(let ((⟨変数1⟩ ⟨式1⟩)
        ⋮
      (⟨変数n⟩ ⟨式n⟩))
  ⟨式1⟩
    ⋮
  ⟨式n⟩)
```

もし局所変数の初期値を与える式が長いときは、次のようにします。

```
(let ((⟨変数1⟩
        ⟨式1⟩)
        ⋮
      (⟨変数n⟩
        ⟨式n⟩))
  ⟨式1⟩
    ⋮
  ⟨式n⟩)
```

もし局所変数のならびが特に短いときは、一行で書くこともあります。

```
(let ((⟨変数1⟩ ⟨式1⟩) ··· (⟨変数n⟩ ⟨式n⟩))
  ⟨式1⟩
    ⋮
  ⟨式n⟩)
```

- 一般の式
  一般の式は、手続き名とそれに対する引数のならびから成ります。引数のならびが短いときは、一行で書きます。

```
(⟨手続き⟩ ⟨式⟩ ··· ⟨式⟩)
```

もし一行に書き切れないときは途中で改行をしますが、引数は ⟨ 手続き ⟩ よりも右に寄せます。

```
(⟨手続き⟩ ⟨式⟩ ··· ⟨式⟩
          ⟨式⟩ ··· ⟨式⟩
              ⋮
          ⟨式⟩ ··· ⟨式⟩)
```

または

(⟨ 手続き ⟩
⟨ 式 ⟩ ··· ⟨ 式 ⟩
⟨ 式 ⟩ ··· ⟨ 式 ⟩
⋮
⟨ 式 ⟩ ··· ⟨ 式 ⟩)

とします。もし ⟨ 式 ⟩ が長いときは、一行にひとつの式を書きます。

6.5.2 手続きと変数の名前

変数と手続きの名前について考えます。プログラムを書くときには色々な変数を使いますが、何のために使われているかを表した変数名にすることで、プログラムが読みやすくなります。

図 6.5.1 には、価格のリストと税率から消費税の総和を計算する 3 つのプログラムが示されています。(a),(b),(c) いずれも、その手続きが何をするものかがコメントとして記述してあります。図 6.5.1(a) では、仮引数の名前が `b`, `c` なために、引数として何を与えて良いのかがまったく分かりません。そのために、プログラムの中身を解読しないと使えません。(a) を改良したのが (b) です。このプログラムでは、どの引数が何の役割をしているかをコメントとして記述してあるために、その手続きを利用することができます。

(a) も (b) も、そのプログラムがすることと関連性のない `S` を手続きの名前にしています。この名前だとすぐにでも忘れてしまいそうです。また別のプログラムの中で `S` が現れても、そのプログラムを読むひとはそれが何をする手続きかさっぱり分からないでしょう。

プログラム (c) では、手続きの名前と引数の名前ともに、どのような役割を持つものかを名前にしています。また手続きの最初の部分には、どのようにして使うかがコメントとして詳しく書かれているために、この部分を見るだけで使い方がすぐ分かります。

こんどは手続きを作成している時点のことを考えてみましょう。ここで挙げた例はいずれも紙面の都合で短いものだけですが、実際のプログラムはこれよりも長く複雑です。そのような場合に、変数名が `b` や `c` だったらどうでしょうか? どの変数が何の役割をしているかをすべて記憶するのは大変です。変数の数が増えてくるとそれらを完全に憶えることはまず不可能で、必ずといっていいくらい勘違いをしてしまいます。

コメントに書いておけばいいと思われるかも知れませんが、そのようなことに努力するよりも、適切な変数名を選び、プログラムの構造を良くするべきです。図 6.5.1(c) のプログラムでは、手続きの名前や変数名は少々長めですが、こうすることでプログラム自体が読みやすくなるとともに、ケアレスミスによってプログラムに**バグ** (プログラムの間違い) を入れてしまうことが少なくなります。

以上のように、字下げや適切な変数名を選ぶことは、プログラムの読みやすさ、書きやすさの点から大変重要です。美しいプログラムを追求するということは、見た目のきれい

```
;; SYOUHIZEI
(define (S b c)
  (if (null? b)
    0
    (+ (* c (car b)) (S (cdr b) c))))
```

(a) 悪い例 1

```
;; procedure "S"
;;  - computes SYOUHIZEI
;;  argument: b - list of price
;;            c - tax rate
(define (S b c)
  (if (null? b)
    0
    (+ (* c (car b)) (S (cdr b) c))))
```

(b) 悪い例 2

```
;; procedure "SyouhiZei"
;;  - computes sum of SYOUHIZEI of each item
;;  argument: list-of-price  - list of price (e.g. )
;;            tax-rate       - tax rate (e.g. be 0.03)
;;  Example:
;;    (SyouhiZei '(1000 200 800) 0.03)
;;    ---> 60     = 0.03*1000+0.03*200+0.03*800
;;
(define (SyouhiZei list-of-price tax-rate)
  (if (null? list-of-price)
    0
    (+ (* tax-rate (car list-of-price))
       (SyouhiZei (cdr list-of-price) tax-rate))))
```

(c) 良い例

図 6.11: 変数の名前の選び方

さだけでなく、プログラムの内容そのものに十分な配慮を加え、質の高いプログラムを書こうとする行為にほかなりません。

初心者にとっては、最初から美しいプログラムを書くことは難しいことです。プログラムが動き出したあとは放り投げるのではなく、より美しいプログラムにするために、何度も書き換えてみてください。徐々にいいプログラムを書く技能が身についてゆくことと思います。更に詳しく勉強したい人は、以下の文献を参考にして下さい。

- B. W. Kernighan, P. J. Plauger (木村 泉 訳), “プログラム書法”, 第二版, 共立出版, 1982.
- B. W. Kernighan, P. J. Plauger (木村 泉 訳), “ソフトウエア作法”, 共立出版, 1981.
- F. P. Brooks Jr. (山内 正彌 訳), “ソフトウエア開発の神話”, 企画センター, 1977.
- D. E. Knuth (有澤 誠 訳), “文芸的プログラミング”, アスキー出版局, 1994.
- 筧 捷彦 他, “プログラミング·セミナー”, 共立出版, 1985.

ああ、死んだ作家は仕合せだ。生き長らえてゐる愚作者は、おのれの作品をひとりでも多くのひとに愛されようと、汗を流して見當はづれの註釋ばかりつけてゐる。そして、まづまづ註釋だらけのうるさい駄作をつくるのだ。

太宰治 「道化の華」
太宰治全集第一巻収録 昭和四十二年 筑摩書房刊

第 7 章

NGSCM の編集コマンド

この章では NGSCM の編集コマンドについて学びます。 NGSCM では複数のファイルを同時に編集したり、画面を複数に分割して別のファイルを参照しながら編集する機能が用意されています。これらの機能はそれぞれ、マルチバッファ機能、マルチウインドウ機能と呼ばれています。NGSCM にはこの他いろいろな編集機能がありますが、すべてを説明することは誌面の都合上しません。

7.1 NGSCM の編集コマンド

7.1.1 キー

NGSCM では、 基本的には押されたキーが入力されます。たとえば a を押すと、文字 a がバッファに挿入されます。

ですが 文字の入力だけではなく、カーソルの移動やタイプミスした文字の消去なども、ファイルを編集する上で必要です。これらの編集動作の実行に、それぞれの編集コマンド専用のキーを準備する方法が考えられます。ですが編集コマンドの数が多いと特別なキーの数が多くなります。するとこの方法では利用できるキーボードの種類が限定されてしまいます。

このようなことを回避するために、NGSCM では数個のある特別なキーを押しながら、あるいは押した後に、編集コマンドを普通のキーを使って指定する方法を使っています。たとえばカーソルを左の文字に移動するという編集コマンドを実行するには、コントロールキーを押しながら b キーを押します[1]。この他、エスケープキーを押した後にさらにキーを入力する方法もあります[2]。たとえば、カーソルの後ろの 1 つの単語を大文字にするには、

[1]多くのキーボードにおいてコントロールキー (controle key) は、キーボード左端のシフトキー (shift key) とタブキー (tab key) の間にあります。キーには、 'CTL'、 'CTRL'、 あるいは 'Control' と書いてあります。

[2]多くのキーボードでは、エスケープキー (escape key) は、 キーボードの左端上のタブキーの上にあります。普通は「エスケープキー」と書いてあります (キーボード上には 'ESC' と書いてあることが多い) が、

```
This is an Eample.
```

(a) ここで x を押すと...

```
This is an Example.
```

(b) 文字が挿入される

図 7.1: 文字の挿入

エスケープキーを押した後に u を押します。

本書では、編集コマンドを指定する方法を記述するのに、次の記法を使います。

- C-k
  コントロールキーを押しながらキー k を押すことを表します。
- M-k
  エスケープキーを押した後に、キー k を押すことを表します。
- C-x k
  C-x を押した後に、キー k を押すことを表します。
- C-c k
  C-c を押した後に、キー k を押すことを表します。
- M-x *command-name*
  エスケープキーを押した後に x を押し、さらに編集コマンドの名前 *command-name* を入力して RET キーを押すことを表します。(これは編集コマンドを名前で指定する実行方法です。)

7.1.2 入力とカーソルの移動

文字の入力はカーソルのある文字の直前に挿入されます。

たとえば図 7.1のように、Eample の a にカーソルがあるときにキー x を押すと Example となり、カーソルは a に移ります[3]。

カーソルの移動をおこなう編集コマンドとして、つぎのものがあります。

Emacs エディタが開発/使用された歴史的な背景により、Emacs 系のエディタの世界では、**メタキー** (meta key) と呼ばれています。

[3]Emacs ではカーソル位置のことをポイント **ポイント** (point) と呼んでいますが、本書では単にカーソルと呼びます。

- C-b
  カーソルを 1 文字後に移動します。

- C-f
  カーソルを 1 文字先に移動します。

- C-n
  カーソルを次の行に移動します。

- C-p
  カーソルを前の行に移動します。

- RET
  改行します。

- M-b
  カーソルを 1 語前に移動します。

- M-f
  カーソルを 1 語後に移動します。

- C-a
  カーソルを行の最初に移動します。

- C-e
  カーソルを行の最後に移動します。

- C-v
  カーソルを 1 画面分先に移動します。

- M-v
  カーソルを 1 画面分前に移動します。

- M-<
  編集バッファの最初にカーソルを移動します。

- M->
  編集バッファの最後にカーソルを移動します。

- C-l
  画面の中心にカーソルが来るように、 画面を再表示します。

7.1.3 マークとリージョン

長いファイルを編集しているときは、カーソルを別の場所に移してそこの内容を読み、また元の場所に戻るということがよくあります。元の場所に戻るときに目で見ながら元の場所に戻ってゆくのは大変です。このことを助けるために、テキストの好きな場所に**マーク** (mark) をつけることができます。カーソルがバッファ内のどの位置にあっても、すぐにマークの位置にカーソルを戻すことができます。

マークに関連した編集コマンドには、つぎのものがあります。

- C-SPC または C-@ または M-x set-mark-command
  現在のカーソル位置にマークを設定します。

- C-x C-x
  現在のカーソル位置にマークをつけ、マークの付いていた場所にカーソルを移動させます。

C-x C-x は、マークのついた位置と現在のカーソル位置を交換します。そのため C-x C-x を続けて実行することで、バッファ内の 2 つの場所を交互に見ることができます。

バッファの中のテキストの内、ある範囲をすべて削除するとか、ある範囲内の英単語をすべて大文字にしたい、ということがあります。そのような範囲を指定するのに、マークと現在のカーソル位置の間を使います。この範囲のことを**リージョン** (region) と呼びます。

テキストのある部分をリージョンとして指定するには、まず指定したいテキストの始めの位置にカーソルを移動してマークをつけます。つぎに終りの位置にカーソルを移動させると、望みの部分がリージョンとして指定されたことになります。リージョンに対して適用される編集コマンドは後に説明します。なおリージョンの指定には、マークが先に来てもカーソル位置が先に来ても関係はありません。単にカーソル位置とマークの間のテキストがリージョンとなります。

7.1.4 消去、削除、ヤンク

バッファ内の不要なテキストは、**消去** (delete) または**削除** (kill) によって消します。削除されたテキストは**削除バッファ**(kill buffer) と呼ばれる特別の領域に入れられます。削除バッファは最後に削除されたテキストを保持し、このテキストをあとからバッファに挿入することもできます。テキストの移動やコピーはこの機能を使います。なお、移動は同じバッファの中でけでなく、他のバッファへもできます。いっぽう消去されたテキストは削除バッファに入りません。これが「削除」と「消去」の違いですので、注意してください。

消去と削除をする編集コマンドには、次のものがあります。

- C-d
  カーソルの下の 1 文字を消去します。

```
"But what happens when you come to the beginning again?"
Alice ventured to ask.
"Exactly so," said the Hatter: "as the things get used up."
"Suppose we change the subject," the march Hare
interrupted yawning.
```

(a) 元のテキスト

```
"Exactly so," said the Hatter: "as the things get used up."
"Suppose we change the subject," the march Hare
interrupted yawning.
```

(b) 4 回続けて C-k を実行した後。
("But what ... to ask. の 2 行が削除されています。)

```
"Exactly so," said the Hatter: "as the things get used up."
"But what happens when you come to the beginning again?"
Alice ventured to ask.
"Suppose we change the subject," the march Hare
interrupted yawning.
```

(c) "Suppose... で始まる行の先頭で C-y を実行。
テキストの移動が行なわれました。

図 7.2: 削除とヤンク
(L.Carroll “Alice’s Adventures in Wonderland” より)

- DEL
  カーソルの前の 1 文字を消去します。

- C-k
  行の最後まで削除します。

- M-d
  カーソルの後ろの１語を削除します。

- C-w
  リージョンのテキストをバッファより削除し、削除バッファに入れます。

- M-w
  リージョンのテキストを削除バッファにコピーします。

削除されたテキストを挿入する機能が、**ヤンク機能** (yank) です。

- C-y
  最後に削除されたテキストを現在のカーソル位置に挿入します。

C-k および M-d が連続して実行されたときは、一連の動作で削除されたテキストがヤンクで挿入されるテキストとなります。たとえば 4 回続けて (そのあいだは他の編集コマンドは実行せずに) C-k をおこなった後に C-y を実行すると、4 回の C-k で削除されたテキストすべてがカーソル位置に挿入されます。図 7.2を参考にしてください。

テキストの移動は C-w でもできます。この機能は紙の書類の一部を切って別の場所に糊で貼り付けることに似ていることから、**カットアンドペースト** (cut and paste) とも呼ばれます。

まず移動したい領域の先頭にカーソルを移動して、C-SPC でマークを付けます。つぎにその部分の終りの部分にカーソルを移動します。これで移動させたい領域をリージョンとして指定できました。ここで C-w でリージョンを削除します。削除バッファには削除したテキストが入ります。テキストを移動させたい場所にカーソルを移動させ、C-y を実行します。これでテキストの移動ができます。削除してから C-y を実行するまでの間は、他の削除コマンドは実行しないで下さい。もし他の削除コマンドを実行すると、あらたに削除されたテキストが削除バッファに入れられ、最初に削除したテキストは失われてしまいます。

リージョンのテキストの移動でなく、テキストのコピーもできます。この方法は先ほどと同様ですが、コピーの場合は C-w の代わりに M-w を使います。この編集コマンドはリージョンのテキストを削除バッファにコピーするだけなので、リージョンのテキストは削除されません。

7.1.5 ファイルの読み込みと書き込み

ファイルを編集するのに起動時にファイル名を指定する方法がありますが、起動後に編集するファイルを指定して編集をはじめることもできます。さらに複数のファイルを同時に編集することもできます。

ファイルに関連した編集コマンドには、つぎのものが用意されています。

- C-x C-f
  新しくバッファを作って、 そのバッファにファイルを読み込みます。

- C-x C-r
  新しくバッファを作ってファイルを読み込みますが、バッファ内容の変更はできないようにします。

  この編集コマンドは、ファイルを見るだけの時に使えます。というのも、バッファの変更を禁止にすることで、ファイルを間違って書き換えてしまうことを防止できるからです。(この編集コマンドで読み込んだファイルを変更するには、C-x C-q により、バッファ内容の変更禁止を解く必要があります。)

- C-x i
  カーソル位置に別のファイルを挿入します。

- C-x C-s
  バッファの内容をセーブします。

- C-x C-w
  バッファの内容を、指定したファイルに書き込みます。

以上の編集コマンドはいずれも、ファイル名を指定する必要があります。これらのコマンドを実行すると、エコー領域でファイル名の入力を求められます。たとえば C-x C-f では、

```
Find file: 
```

と表示され、ファイル名の入力はここで行ないます。

重要: ファイルの読み込みなどの編集コマンドを中止するには、C-g を入力します。これにより、 編集コマンドを実行する前の状態に戻ることができます。

注意: ファイルの読み込みのとき、同じファイルが既にバッファにとり込まれていれば、そのバッファが画面に表示されます。またファイル名は同一でも置かれているディレクト

リ[4]が違う場合は、バッファの名前を fact.scm<2> とか fact.scm<3> のように、 ファイル名の後ろに番号をつけて区別します。

注意:: バッファを変更した後にセーブしていなければ、NGSCM を終了するときにそのバッファをセーブするかどうか尋ねられます。もしここで n と答えると、 編集作業の結果はすべて失われます。

7.1.6 入力補完機能

ファイル名を指定するときには、**補完** (completion) 機能が使えます。補完とは名前の最初の部分を与えるだけで対象の名前が特定できれば、自動的に入力を補ってくれる機能です。この機能のおかげで名前をすべて手で入力しなくて済みます。もし複数の候補がある場合は、候補をすべて表示してくれます。この機能のおかげで、ファイル名の綴りを完全に覚えておく必要はありません。この補完機能はファイル名のほか、バッファ名の指定や M-x によるコマンド名を指定した編集コマンドの実行のときにも使えます。

たとえば sum.scm というファイルを編集する場合を考えます。C-x C-f に対して sum.scm を入力すれば、 sum.scm が読み込まれます。もし s で始まるファイルが sum.scm だけだとしましょう。C-x C-f に対して s を入力し、TAB を入力します。

```
Find file: s TAB
```

すると qwe.scm がエコー領域に現れ、入力が補完されます。

```
Find file: sum.scm
```

もし s で始まるファイルが sum.scm の他に sum1.scm がある場合を考えます。この場合も先ほどと同じように s を入力して TAB を入力します。今度は sum まで補完されるます。これは s で始まるファイルのいずれもが、ファイル名のはじめに sum を共通に持っているので、sum まで補完されたのです。このとき図 7.3のように、*Completion* バッファが作られ、ファイル名が sum で始まるファイルすべてを表示します。

さらに 1 を入力して (この時点での入力は、sum1 です) TAB を入力します。

```
Find file: sum1 TAB
```

sum1 で始まるファイルは sum1.scm ひとつだけなので、sum1.scm と補完されます。

```
Find file: sum1.scm
```

[4]ファイルの数が増えてくると、いろいろな種類のファイルがあって雑然としてきます。オフィスでのキャビネットごと、あるいは棚ごとに区分けをして、書類を保持するのと似た感じで、ファイルをまとめる方法が必要となってきます。ディレクトリとは、このような目的のためのものです。詳しくは、オペレーティングシステムの本を参照してください。

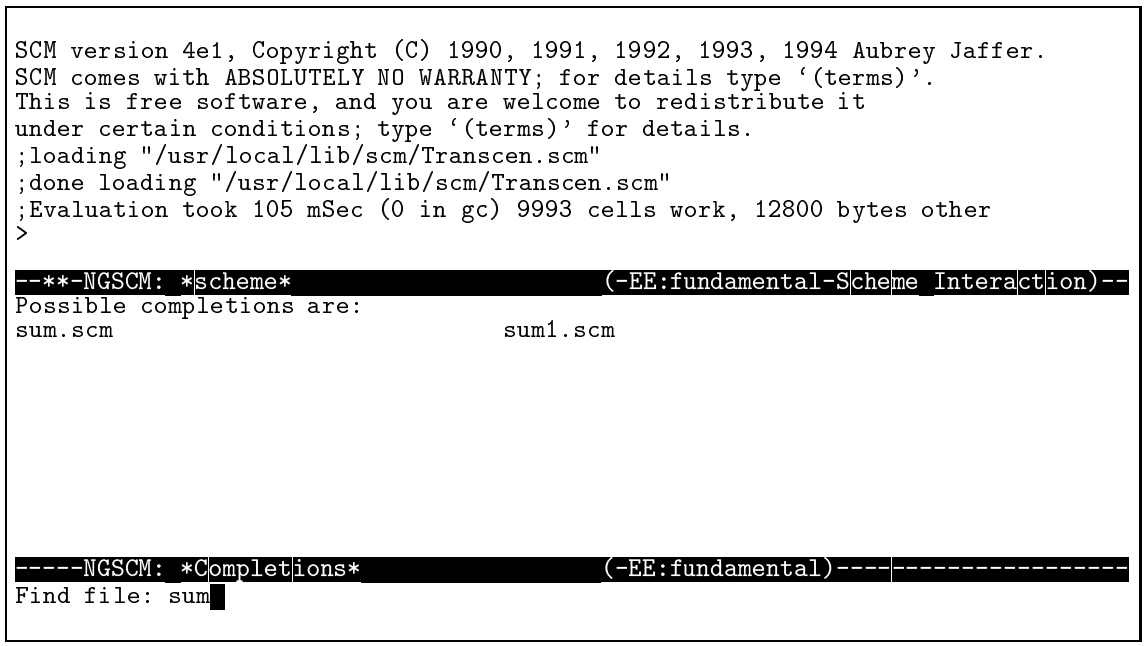


図 7.3: 補完候補の一覧

ここで RET を入力すれば、 ファイル名の指定は完了です。
もし s で始まるファイルがひとつもなければ、

```
Find file: s TAB
```

と入力しても、

```
Find file: s [No match]
```

と表示され、候補がひとつもないことが分かります。

7.1.7 探索と置換

探索 (search) とは、ある文字列がファイルのどこにあるかを探し、見つかった場所にカーソルを移動させる編集コマンドです。**置換** (replace) とは、ある文字列を他の文字列に置き換える編集コマンドです。単なる文字列だけでなく、正規表現 (regular expression) による探索をする編集コマンドもありますが、省略します。

- C-s
  順方向インクリメンタル探索
- C-r
  逆方向インクリメンタル探索

- M-x search-forward
  順方向探索

- M-x search-backword
  逆方向探索

- M-%
  対話的置換

普通の探索では探索する文字列を完全に与えてから探索を始めますが、インクリメンタル探索は探索文字列を順時与えてゆきます。例として `define` という文字列を、インクリメンタル探索で見つける場合を考えます。C-s でインクリメンタル探索を始めます。このときエコー領域に

```
I-search:
```

と表示されます。探索する文字列 `define` の最初の文字 d を入力します。すると文字 d のある部分にカーソルが移動します。ここでのカーソルの移動先が目的の場所なら ESC キーを押し、探索を終了します。もしそうでないなら、 次の文字である e を入力します。この入力により、文字列 `de` のある部分にカーソルが移動します。このときのエコー領域は

```
I-search: de
```

となっています。

このように、目的の部分が見つかるまで、d, e, f, i, n, e を順に入力してゆきます。もし探索文字列を順次与えているときに入力を間違えば、BS または DEL で探索文字列の最後の文字を取り消すことができます。途中で `define` が見つかれば、 上でも述べたように ESC キーを押し、探索を終了します。

次の部分を探すには、 もう一度 C-s を押します。すると、次に見つかった部分にカーソルが移動します。いったん探索を終ってから、もう一度前回と同じものを探索をするには、C-s を 2 回押します。

C-s は現在のカーソル位置からファイル終りに向かってインクリメンタル探索をする編集コマンドです。一方 C-r は、現在のカーソル位置からファイル始めに向かって (C-s とは逆方向に) インクリメンタル探索をする編集コマンドです。

M-x search-formard, M-x search-backword も探索をする編集コマンドですが、探索文字列を一括して与えて探索する編集コマンドです。これを実行するには、エコー領域で探索文字列を一度に与えないといけません。

```
Search:
```

ここで探索文字列を入力し、 最後に RET を押します。

```
Search: define RET
```

すると探索が行なわれ、見つかればその場所にカーソルが移動します。もし見つからなければエラーとなり、 エコー領域に

```
Search failed: "define"
```

と表示されます。

重要: 探索や置換を途中で中断するには、C-g を入力します。

置換とは、バッファ内のある文字列を他の文字列に置き換える編集コマンドです。文字列の置換をするには M-% を使います。例として、バッファ中の Eeyore という語を Rabbit に置き換える方法を示します。

M-%を押すと、 エコー領域は

```
Query replace: 
```

となります。ここで置き換えようとする文字列 Eeyore を入力し、 RET を押します。するとエコー領域は

```
Query replace: Eeyore with: 
```

となります。ここで置き換える文字列 Rabbit を入力して RET を押します。これにより置換が始まります。

置き換えるべき候補の文字列にカーソルが移動し、候補を置き換えるかどうかが尋ねられます。

```
Query replaceing Eeyore with Rabbit:
```

ここでの入力は以下の通りです。

- SPC
  その候補を置き換え、さらに次の候補を探します。
- DEL
  その候補は置き換えず、さらに次の候補を探します。
- !
  それ以降、置き換えをするかを尋ねずに、候補をすべて置換えします。
- ESC
  置換命令を終了します。

Mule エディタの場合は以下の通りです。y で置き換えをします。n で置き換えをしません。! は NGSCM の場合と同じで、q で置換命令を終了します。

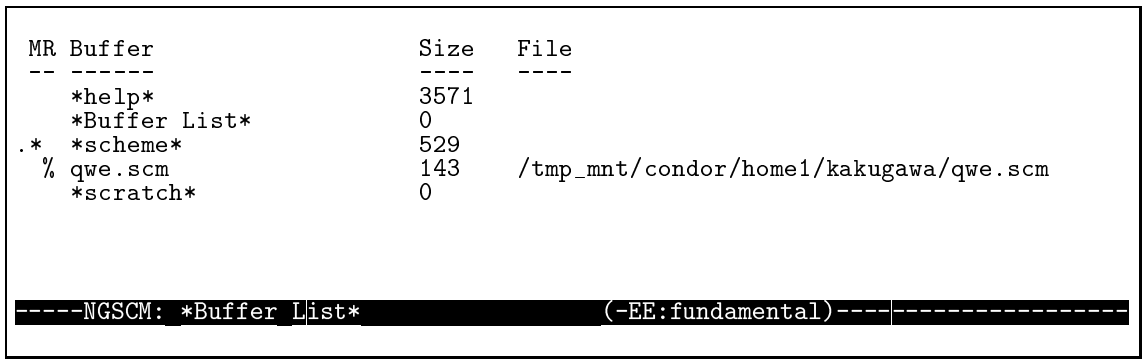


図 7.4: バッファの一覧

7.1.8 マルチバッファ

前にも説明したように、ファイルはいったんバッファに入れられて編集されます。複数のバッファを使って、複数のファイルを同時に編集してゆくことができます。

バッファの一覧を見るには、次の編集コマンドを使います。

- C-x C-b
  新たに `*Buffer List*` という名前のバッファを作り、現在のバッファの一覧を表示します。

図 7.4に例を示します。この図での各行は、左から次の意味を持ちます。. のついているバッファが現在選択されている (カーソルのある) バッファです。* の付いているものは変更が加えられたことを示しています。% の付いているバッファは書き込み禁止のバッファです。(C-x C-r でファイルを読み込んだときなどに、このようなバッファが作られます。) それらのつぎの欄はバッファ名で、その右がそのバッファの大きさを表しています。一番右の欄はバッファに対応したファイル名を表しています。

バッファの選択と作成には、次の編集コマンドがあります。

- C-x b
  他のバッファを選択します。

C-x b を実行すると、 エコー領域で

`Switch to buffer: (default` *buffer*`)` ▉

と、切替えるバッファ名が尋ねられます。括弧で囲まれているバッファ名は、バッファ名を指定せずに RET だけを入力したときに切替えられるバッファ名です。もし選択したいバッファが *buffer* なら、RET を押すだけでそのバッファに切替わります。

それ以外のバッファを選択する場合は、バッファ名を入力して最後に RET を押します。するとそのバッファに切替り、表示されます。もし指定したバッファがない場合は、その名前のバッファが新しく作成されます。

たとえば

```
Switch to buffer: (default *scratch*) f.scm RET
```

とすれば、f.scm バッファが選択されます。

バッファ名の指定には補完機能が使えるので、バッファ名を完全に覚えておいたり、バッファ名をすべて入力しなくて済みます。

このほかバッファに対する編集コマンドには、次のものがあります。

- C-x k
  バッファを削除します。

- C-x C-q
  選択されているバッファの変更禁止/変更可能を、交互に切替えます。

C-x k を実行すると、 エコー領域で

```
Kill buffer: (default buffer) ▉
```

と表示されます。もし削除したいバッファが *buffer* なら、単に RET を入力することで *buffer* が削除されます。それ以外のバッファを削除するのなら、そのバッファ名を入力して最後に RET を押します。(この編集コマンドでも、 バッファ名の指定に補間機能が使えます。)

たとえば

```
Kill buffer: (default *scheme*) f.scm RET
```

とすれば、f.scm が削除されます。もしバッファに変更 (書き込み) が加えられた後にセーブがされていなければ、本当にバッファを削除して良いかどうかをエコー領域で尋ねられます:

```
Buffer modifed; kill anyway? (yes or no) ▉
```

これはファイルに修正を加えたけれども、うっかりセーブを忘れてバッファを削除してしまうのを防止するためのものです。というのも、セーブしなければバッファの書き換えはファイルに反映されないからです。ですがバッファが削除されても、ディスク上のファイル自体は残ります。もちろんそのファイルの内容は、編集作業を始める前のものです。もし編集操作を誤ってどうしようもなくなったときは、セーブせずにバッファを削除することで、編集前のファイルのままにできます。

C-x C-q を実行すると、 バッファのモード行の左の部分--**-NGSCM が --%%-NGSCM になります。もう一度 C-x C-q を実行すると元通りになります。表示が --%%-NGSCM になっているときはバッファは変更禁止になっていて、文字の削除や追加などはできません。

選択するバッファを切替えることで、 複数のファイルを同時に編集できます。ヤンク機能と組み合わせれば、あるファイルの一部を他のファイルに移動させることができ、編集作業が大変楽になります。

7.1.9 マルチウインドウ

画面をいくつかに分割し、複数のバッファを同時に表示する機能を**マルチウインドウ** (multiple windows) といいます。たとえば図 4.3では、画面を 2 分割して 2 つのバッファ qwe.scm と *scheme* を表示しています。このような表示窓を**ウインドウ** (window) と呼んでいます。ウインドウは 2 つだけでなく、いくつも作ることができます。

カーソルのあるウインドウが選択されたウインドウです。また、選択されたウインドウに対応しているバッファが、選択されているバッファとなります。編集作業は選択されたバッファに対して適用されます。

ウインドウにはひとつのバッファが対応していて、対応するバッファがそのウインドウに表示されます。(表示されていないバッファには、どのウインドウも対応していないことになります。) ひとつのバッファを複数のウインドウに対応付けることもできます。この場合では、ひとつのバッファが複数のウインドウで表示されることになります。各ウインドウはそれぞれのカーソル位置を記憶しています。このため同一のバッファでもウインドウが違えば、違う部分を表示することが可能となります。

ウインドウの作成と消去に関する編集コマンドには、次のものがあります。

- C-x 2
  選択しているウインドウを 2 つに分割します。

- C-x 0
  選択しているウインドウを消去します。

- C-x 1
  選択しているウインドウだけを残し、他のウインドウを消去します。

図 7.5は、ひとつのウインドウだけの画面を示しています。ここで C-x 2 を実行すると、図 7.6のようにウインドウが 2 分割されます。この図から分かるように、分割されたウインドウはそれぞれ同じバッファに対応しています。ここでファイルを選択したり、他のバッファの選択をすることで、他のファイルや他のバッファが表示されます。こうすることで、別のファイルを参照しながらプログラムの編集をすることができます。たとえば図 7.6 の状態でファイル fact.scm を選択した場合を、図 7.7に示します。

その他のウインドウ関連の編集コマンドには、次のものがあります。

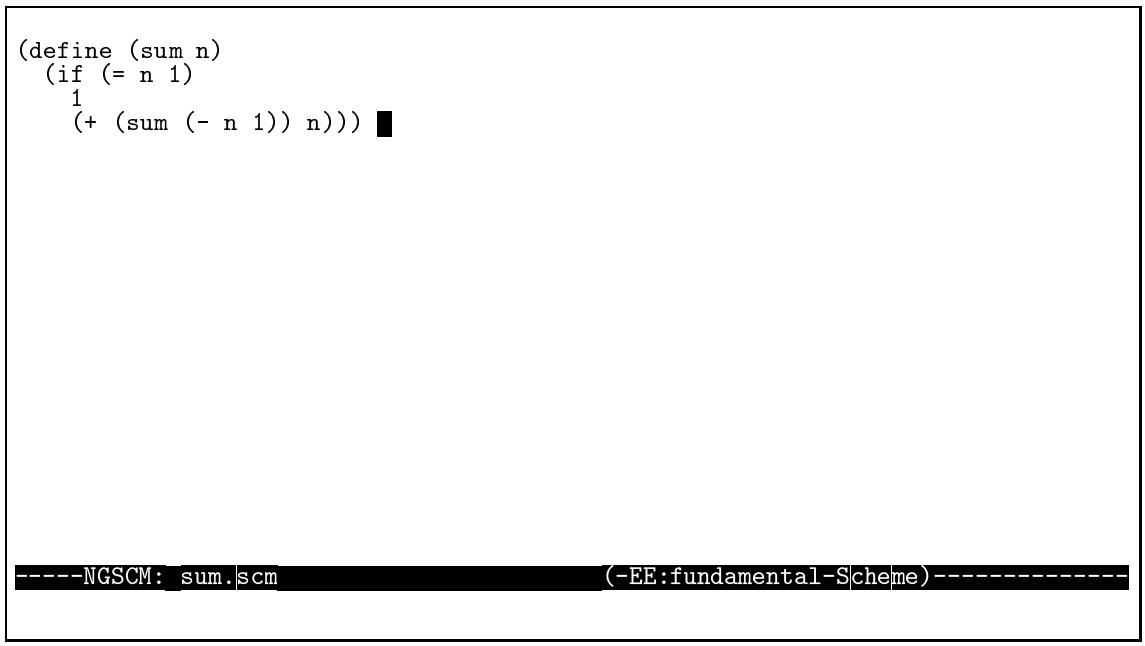


図 7.5: ウインドウが 1 つの状態

- C-x o
  他のウインドウを選択します。

- C-x ^
  選択されたウインドウを大きくします。

- M-x shrink-window
  選択されたウインドウを小さくします。

他のウインドウに表示されているバッファを編集するには、カーソルをそのウインドウに移動しなくてはいけません。C-x o により、カーソルを他のウインドウに移動させます。

C-x^は、 カーソルのあるウインドウを 1 行分大きくします。これに伴い、他のウインドウは小さくなります。M-x shrink-window は、 カーソルのあるウインドウを 1 行分小さくします。

たとえば図 7.7のように同じ大きさのウインドウを 2 つ作り、あるファイルを参照しながら別のファイルの編集をする場合を考えます。画面が 25 行程度しかない計算機 (あるいは端末機) を使っている場合では特に、画面が狭くて不便です。このようなときは、 編集作業をしているウインドウを大きくすると快適です。ウインドウを大きくするには、まず大きくしたいウインドウにカーソルを移動します。そしてそのウインドウで C-x^を入力します。図 7.8では、 5 回 C-x^を実行した例です。

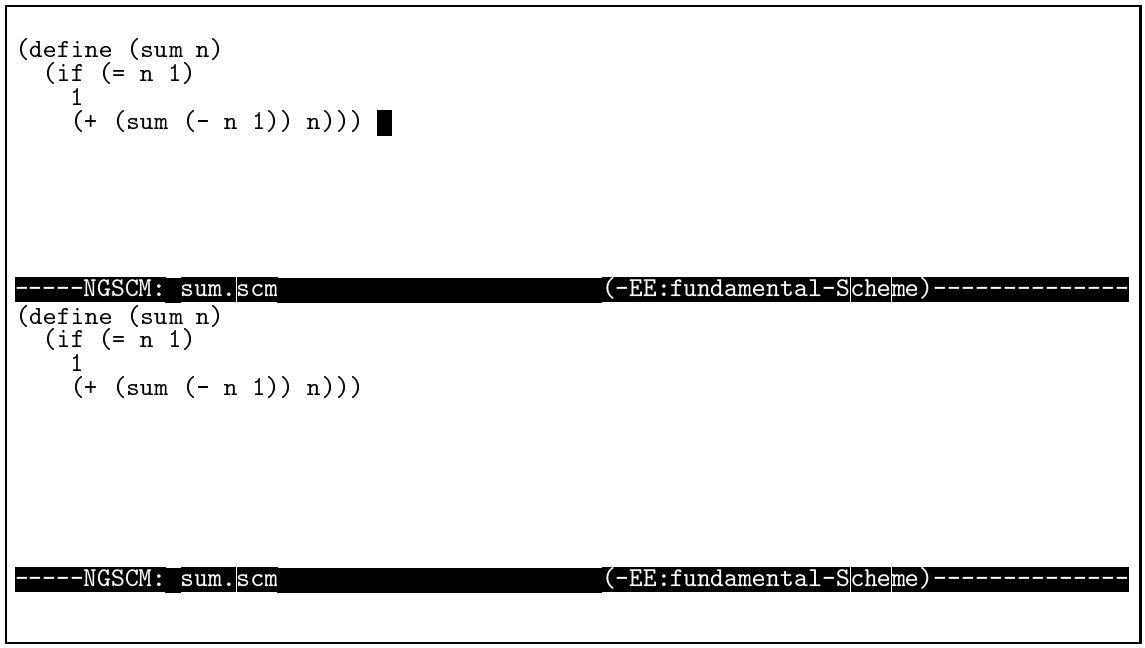


図 7.6: ウインドウを 2 つに分割した状態

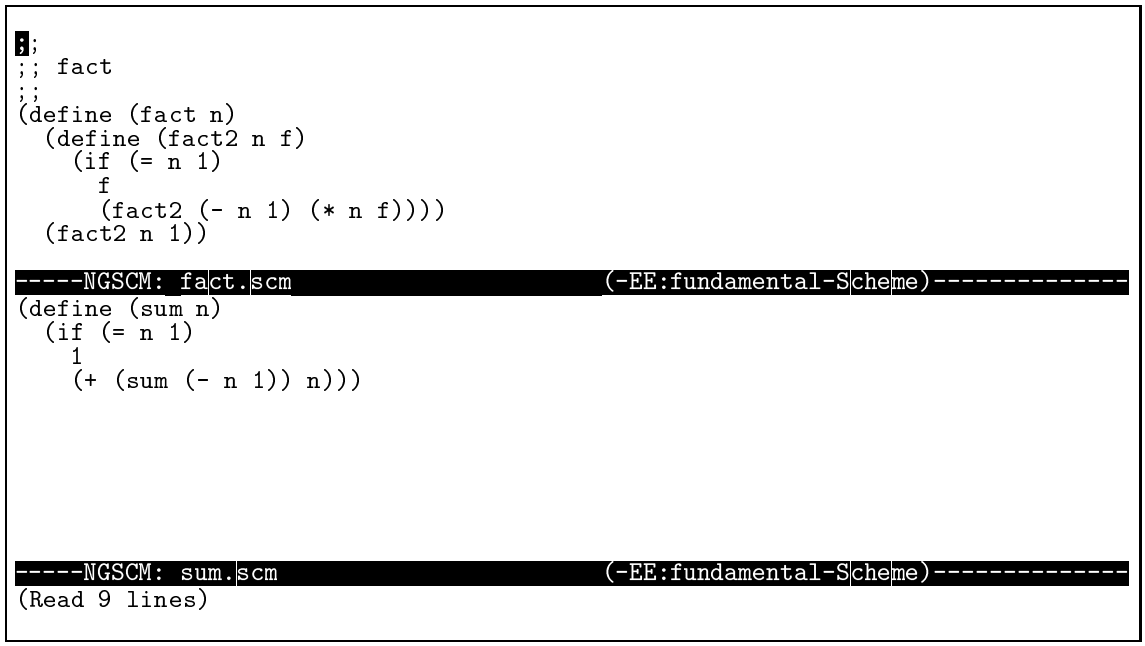


図 7.7: さらに, ファイル `fact.scm` を選択

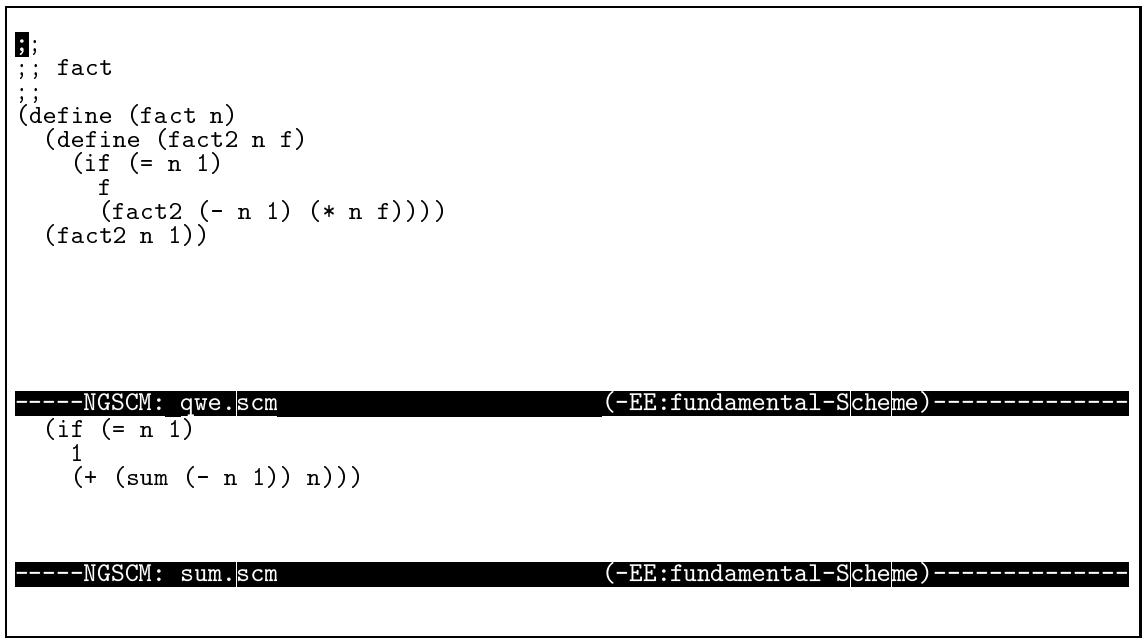


図 7.8: ウインドウの大きさを変更した画面

プログラムを作成する人、あるいは多くの文書を作る人にとっては、エディタは人間とコンピューターとの接点である重要なソフトウエアです。エディタについてより詳しく知りたい人へは、 以下の文献の一読をお奨めします。

- N. Meyrowitz, A. van Dam (石井 博 訳), “対話型編集システム: 第 I 部”, bit 別冊 コンピューターサイエンス, pp.75–104, 共立出版, 1983. (N. Meyrowitz, A. van Dam, “Interactive Editing Systems: Part I”, *ACM Computing Surveys*, Vol. 14 No. 3, pp.321–352, September 1982.)
- N. Meyrowitz, A. van Dam (瀬川 清 訳), “対話型編集システム: 第 II 部”, bit 別冊 コンピューターサイエンス, p.105–166, 共立出版, 1983. (N. Meyrowitz, A. van Dam, “Interactive Editing Systems: Part II”, *ACM Computing Surveys*, Vol. 14 No. 3, pp.353–415, September 1982.)
- 永田 守男, 黒川 利明 (編), “エディタ”, 情報処理, 第 25 巻 第 8 号, p.758–874, 情報処理学会, 1984 年 8 月.
- Richard M. Stallman (竹内 郁夫, 天海 良治 訳), “GNU Emacs マニュアル”, 共立出版, 1988 年 2 月.
- Craig A. Finseth (岩谷 宏 訳), “The Craft of Text Editing – 手作りのテキストエディタ –”, 株式会社ビレッジセンター出版局, 1994 年 4 月.

練習問題

1. 3 つの数 a, b, x が引数で、$ax + b$ を計算する手続き fx

```
(define (fx a b x)
  (+ (* a x) b))
```

をファイルに作成し、実行しなさい。なおプログラムファイルは、 fx.scm という名前にしなさい。実行例は、以下の通りです。

```
> (fx 2 3 4)
;Evaluation took 16 mSec (0 in gc) 5 cons work
11
> (fx 2 3 8)
;Evaluation took 0 mSec (0 in gc) 5 cons work
19
```

2. 次の手続き power を、power.scm という名前のファイルに作り、実行しなさい。この手続きには 2 つの引数 a, b があり、a^b を計算する手続きです。

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

実行例は、以下の通りです。

```
> (power 2 3)
;Evaluation took 0 mSec (0 in gc) 18 cons work
8
> (power 2 6)
;Evaluation took 0 mSec (0 in gc) 28 cons work
64
```

3. 次の手続き ave を、ave.scm という名前のファイルに作り、実行しなさい。この手続きは、引数で与えられた 3 つの数の平均を計算します。

```
(define (ave a1 a2 a3)
  (/ (+ a1 a2 a3) 3))
```

実行例は以下の通りです。

```
> (ave 2 3 4)
;Evaluation took 0 mSec (0 in gc) 8 cons work
3
> (power 4 5 12)
;Evaluation took 0 mSec (0 in gc) 8 cons work
7
```

一回で書き上がることはまずない。下書きをして、文法的に変ちくりんな文章にだけはならないようにする。ところがやっとの思いで書いても、先方がこれを読んでどう感じるかが、こちらにはわからないのが、手紙の困ったところである。

群ようこ 「二重三重の楽しみ方」
『三島由紀夫レーター教室』解説 平成三年 筑摩書房刊

第 8 章

Scheme プログラムの作成と実行

本章では NGSCM の編集機能を使った Scheme プログラムの編集方法と、Scheme プログラムを実行する方法を学びます。4.4 で最低限必要な操作法を学びましたので、初心者の人はこの章は飛ばして構いません。

NGSCM には Scheme プログラムの編集や、Scheme プログラムの実行を支援するいくつかの編集コマンドが準備してあります。ここで学ぶ NGSCM の持つ高度な機能を学べば、より効率のいい Scheme プログラムの編集と実行方法を身に付けることができます。NGSCM の機能は Mule に似せてあるので、Mule を使うときもここでの説明の多くがあてはまります。

8.1 Scheme モードと Scheme Interaction モード

Scheme プログラムの編集をより便利にするために、Scheme モードが用意されています。

モード (mode) とは、 バッファの性質みたいなものです。どのキーが押されたらどの編集動作を実行するかという、キーと編集動作の対応をモードごとに定めることができる仕組みがあります。そのためバッファのモードが違うと、押されたキーと編集動作の対応がまったく違うこともあります。第 7章で学んだ編集コマンドは、実は基本的なモードである fundermental モードでの編集コマンドです[1]。ですがモードごとにキーと編集動作の対応がまるっきり違うのは混乱の元ですので、基本的なキーの使い方はできるだけ同じように設計されています。

Scheme モードでは、Scheme プログラムの作成と実行のに便利な編集コマンドが、通常の編集コマンド体系に追加されています。このために、第 7章で学んだ編集コマンドはすべて使えます。

NGSCM では、普通のファイルは fundamental モードという標準的な編集モードで編集されます。ファイル名が `.scm` で終るファイルを読み込むときは、バッファのモードを

[1] GNU Emacs や Mule では、 Fundermental モードです。

Scheme モードにします。4.4ページで、ファイル名の最後は .scm にしましょう、と書いたことのもうひとつの理由がこれです。

もし .s で終る Scheme プログラムファイルを読み込んで Scheme モードにして編集したいときは、バッファのモードを Scheme モードに切替えます。Scheme モードに切替えるには、M-x scheme-mode を実行します。すると、バッファのモード行が

```
-----NGSCM: sum.s                          (-EE:fundamental)----------------
```

だったのが

```
-----NGSCM: sum.s                          (-EE:fundamental-Scheme)--------------
```

となります。これは fundermental モードの機能に、Scheme モードの機能が加わったことを表しています。

では Scheme Interaction モードについて説明します。Scheme プログラムを実行するために Scheme 処理系とのやりとりをするバッファのモードが、Scheme Interaction モードです。Scheme Interaction モードのモード行は

```
--**-NGSCM: *scheme*                        (-EE:fundamental-Scheme Interaction)--
```

と表示されています。*scheme* バッファが Scheme 処理系とのやりとりをするバッファで、Scheme 処理系の実行結果はすべてこのバッファに挿入されます[2]。

以下に Scheme プログラムファイル編集、ならびに Scheme プログラム実行のための編集コマンドを示します。

Scheme 用のモード

- M-x scheme-mode
  バッファを Scheme モードにします。拡張子が .scm のファイルを読み込むと、自動的に Scheme モードになります。

式の評価

- C-j または, C-c C-e
  カーソルの前にある式を評価し、その結果を Scheme バッファ (*scheme*) で表示します。(C-j は Scheme Interaction モードでのみでしか使えません。)

[2] さきほど学んだように、バッファのモードを編集コマンドによって明示的に Scheme モードにすることはできますが、Scheme Interaction モードにする編集コマンドは用意されていません。というのも、 Scheme Interaction モードになり得るのは *scheme* バッファだけだからです。

- M-ESC
  エコー領域から式をひとつ読み込んで評価し、その結果を Scheme バッファで表示します。

- CtlcC-r
  リージョン内のすべての式を順に評価し、Scheme バッファで表示します。

リストと式

- C-M-f
  式をひとつ越えて、先へカーソルを移動します。

- C-M-b
  式をひとつ越えて、逆方向へカーソルを移動します。

- C-M-k
  カーソルの右にある式をひとつ削除します。

プログラムの字下げ

- TAB
  カーソルのある行を字下げします。

- LFD
  RET に続いて TAB を押すのと同じ動作を行ないます。

Defun

Scheme プログラムが書かれたファイルは、複数の式がならんだ形になっています。Scheme での式の中にさらに別の式を書くことができますが、一番外の式、すなわち、どの式の一部にもなっていない式は**トップレベル** (top level) の式と呼ばれます。バッファ内で括弧で括られたトップレベルの式で、行の左端から始まっているものは defun と呼ばれます[3]。defun を単位とする編集コマンドとして以下のものが用意されています。

- C-M-a
  いまカーソルがある式の defun またはカーソルの前の defun の先頭にカーソルを移動します。

- C-M-e
  いまカーソルがある式の defun またはカーソルより後ろの defun の最後にカーソルを移動します。

[3]プログラミング言語 Lisp では、関数を定義するのに `defun` というキーワードが使われていたことと、プログラムファイル中のトップレベル式のほとんどが (`defun` ...) で始まることに由来して、 この名前が付きました。

- C-M-h
  いまカーソルのある defun またはカーソルより後ろの defun にリージョンを設定します。

トップレベルでの手続き定義の中にカーソルがあれば、C-M-a を実行することでその手続き定義の先頭部 (`define` ... の開き括弧の部分にカーソルが移動します。トップレベルでの式の外にカーソルがあるときに C-M-a を実行すれば、カーソルの前の式を前にひとつ飛び越したところにカーソルが移ります。このように、 defun に関する編集コマンドを使うことで、手続き定義式を単位としてカーソルの移動ができます。

8.1.1 履歴機能

`*scheme*` バッファでは、式の入力とそれに対する評価結果のバッファへの挿入が繰り返されます。プログラムのテストをするとき、同じ式や似た式を何度も入力する場合がしばしばあります。短い式ならいいのですが、少し長い式を何度も入力するのは大変です。

NGSCM には以前に入力したものを呼び出す編集コマンドが用意されています。この機能は**履歴機能** (history) と呼ばれ、入力の煩わしさを解消できます。

- M-p
  1 つ前の入力をカーソル位置に挿入します。

- M-n
  1 つ後の入力をカーソル位置に挿入します。

たとえば NGSCM を起動した後に次の一連の式を入力し、 評価したとします。

```
(list 1 2 3 4 5)
(+ 1 2 3 4 5)
(* 1 2 3 4 5)
(vector 1 2 3 4 5)
```

一連の式の評価が終った後の画面が図 8.1です。

ここで M-p を入力すると、ひとつ前の入力 (すなわち `(vector 1 2 3 4 5)`) がカーソル位置に現れます。さらにもう一度 M-p を入力すると、もうひとつ前の入力 (この場合は `(* 1 2 3 4 5)`) がカーソル位置に現れます。M-p を繰り返して入力すれば、どんどん前の入力がカーソル位置に現れてきます。もし M-p を入力し過ぎて目的の入力を過ぎてしまったときは、M-n を入力することで図 8.2のように、その後に入力された式がカーソル位置に現れます。

この履歴機能の上手な使い方を説明します。入力しようとしている式が以前に入力したものとまったく同じときは、履歴機能で得られた過去の入力をカーソル位置に呼び出し、

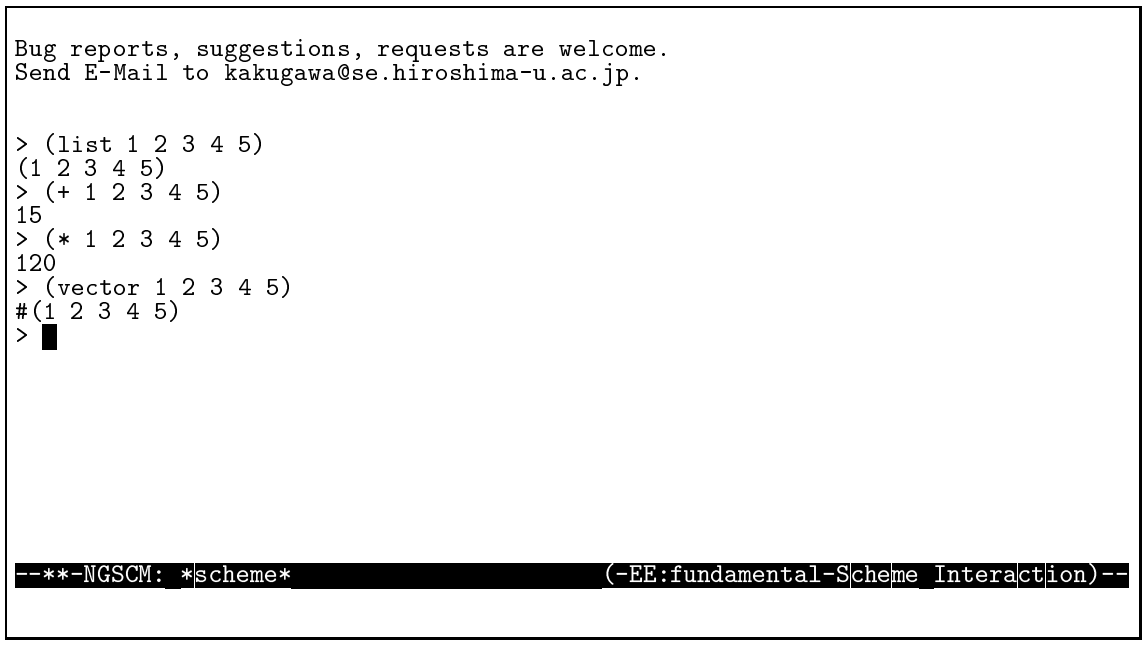
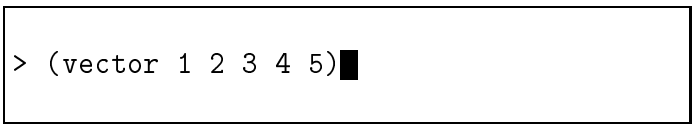


図 8.1: 一連の式の入力後

C-j でその式を評価します。もし同一なものがなくても、以前に入力した式の中に似たものがあればそれを呼び出します。そしてその式を編集コマンドを使って自分の入力したい式に書き換え、C-j で評価します。この方法を使うことで、キーを叩く回数を大幅に減少させることができます。また、一度正確な入力がされれば、それを呼び出す限りタイプミスをする可能性が大幅に減ります。ですので、ぜひとも使い方を覚えるようにしてください。

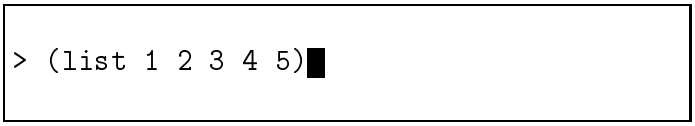
注意: 履歴機能で得られる式は、NGSCM が利用者の入力すべてを記憶しているのではありません。`*scheme*` バッファの中で `>` で始まる行を、(実際に書かれている内容には関係なく) 利用者からの入力だと判断します。そしてその行の内容を、カーソル位置に挿入するようにしています。そのために、入力が 2 行以上にまたがっている場合は、最初の 1 行しかカーソル位置に得られないことになります。また `*scheme*` バッファの一部を削除してしまった場合には、削除された部分にあった入力を履歴機能で呼び出すことはできません。

この履歴機能で得られる過去の入力は画面上に見えている部分だけでなく、バッファの先頭の方ある直接見えていない過去の入力に対しても有効です。もし M-p によって、一番過去の (すなわち `*scheme*` バッファ内で一番先頭の) 入力よりも前の入力を得ようとすると、今度は一番最後に入力した式が得られます。上の例で 5 回続けて M-p を入力すると、直前の入力である (`vector 1 2 3 4 5`) がカーソル位置に現れます。M-n についても同様で、もし一番最新の入力の次の入力を得ようとすると、一番古い入力がカーソル位置に得られることになります。



(1) 1 度目の M-C-p の入力

(2) 更に M-C-p を入力



(3) 更にもう 2 回 M-C-p を入力

(4) M-C-n を入力

図 8.2: 履歴機能

以上はカーソル位置に過去の入力を得る方法でしたが、過去の入力がされたバッファ位置にカーソルを移動させる編集コマンドも用意されています。

- C-c C-p
  1 つ前の入力がされた場所にカーソルを移動します。
- C-c C-n
  1 つ後の入力がされた場所にカーソルを移動します。

8.1.2 NGSCM での Scheme システム

前にも述べたように、NGSCM は Scheme 処理系とテキストエディタより構成されています。このために、プログラムの編集と実行が 1 つのシステムでできるようになっています。ここでは、どのような仕組みでこのことが行なわれているかを簡単に説明します。

注意: なおこの部分は初心者には難しいので、興味がなければ読み飛ばしても構いませんし、理解できなくても気にすることはありません。

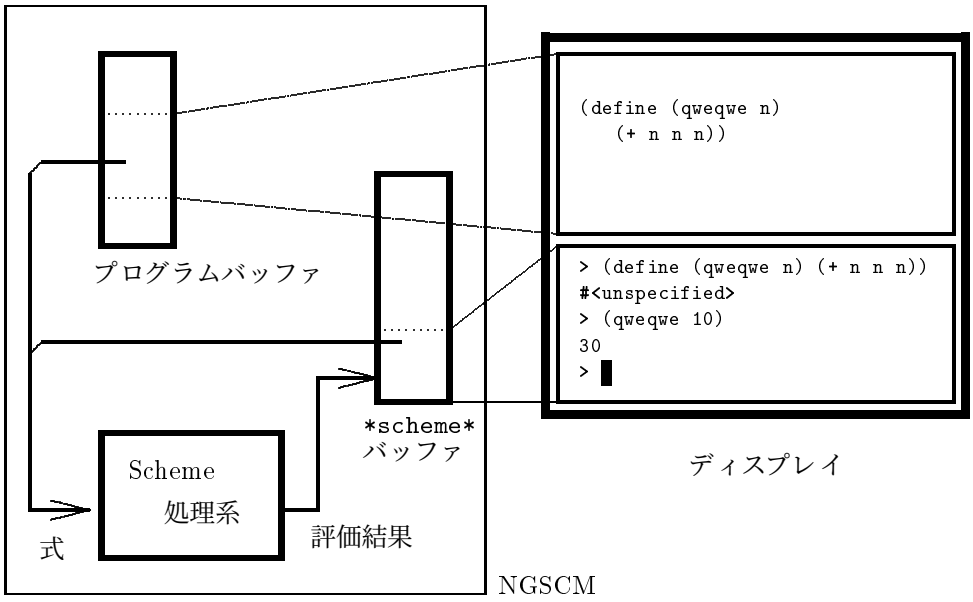


図 8.3: NGSCM での Scheme 処理系

NGSCM での仕組みは、 Mule や GNU Emacs で Scheme を実行する仕組みと違っています。Mule では、エディタと Scheme 処理系は別々のプログラムで、Scheme 処理系を別個に起動します。実行中のプログラム同士で通信をする機構 (プロセス間通信と呼ばれています) を使い、評価すべき式を Scheme 処理系に送り、その評価結果を受けとってバッファに挿入する、という方法をとっています。

C-c C-e または C-j を実行すると、カーソルの前の S-式を評価され、その評価結果が *scheme* に表示されます。これは図 8.3に示す機構によって行なわれます。

まず NGSCM の起動時に、Scheme 処理系用のバッファ *scheme* が作られます。C-c C-e や C-j が押されると、これらの動作をする編集コマンドが呼び出されます。これらの編集コマンドはまず最初に、カーソルの直前の式のコピーを取ります。次に Scheme 処理系を呼び出し、Scheme 処理系は式のコピーを見て評価 (実行) します。そしてその結果は、*scheme* バッファに (文字として) 挿入されます。

C-c C-e の動作説明では,「評価結果を*scheme* に表示する」という表現をしていましたが、評価結果を文字にして *scheme* バッファに挿入しているだけです。*scheme* バッファもバッファですので、そこで各種の編集命令を実行することができます。

このバッファは Scheme 処理系とのやりとりに使われますから、Scheme プログラムの実行に便利な編集コマンドがあれば大変便利です。このため、*scheme* バッファを Scheme

Interaction モードとし、履歴機能などの編集コマンドが使えるようにしてあるわけです。

傳藏等はその翌年の夏ころまで官費で生活してゐたが、次第に土地の言葉風儀にも通じて來て、簡単な用事や使い走りなら半端ながら用を辨じるやうになつた。それでいつまでも役所の世話になつていては氣の毒だというところから、何か仕事か勞働を仰せつけてもらひたい役所の長官ドクトル・ジョーヂに申し出た。

井伏鱒二 「ジョン万次郎漂流記」
『ジョン万次郎漂流記』収録 昭和二十二年 文學界社刊

第 9 章
プログラミング実習 2

本章では第 5章の実習 1 よりも複雑なプログラムを作ります。実習 1 のときと同じように、出てきたプログラムを実際に入力し、実行してみて下さい。本章では以下に示す実習テーマを通じて、プログラミングの練習をします。

1. **自動販売機のシミュレート**
   — メッセージ伝達法によるプログラミング技法の基礎を学びます。Scheme には 局所変数を持った手続きを生成する機能がありますが、これを利用して局所データを持つオブジェクトを作成する技法の基礎を学びます。

2. **ファイル処理**
   — ディスク上にファイルとして記録してあるデータを読みとって処理を加え、その結果を別のファイルに記録することが実用的なプログラムでは必要となります。この実習ではファイルを使ったデータの入出力の基礎技法を学びます。

3. **整列**
   — 整列とは、データを順にならびかえることをいいます。入試で成績の上位 30 名を選ぶとか、利用金額の多い順にならべた顧客リストを作るとかいった場合に必要です。この実習では 2 通りの整列方法を学びます。

4. **住所録の製作 その 2: ファイル版**
   — このテーマでは、再び住所録プログラムを製作します。今度のプログラムでは、ファイルに住所データを保持しておき、ファイルを読みながら検索をするようにします。ファイルを利用することで、より実用的なプログラムとなります。

本実習の内容は、前の実習と比べていくぶん高度になっています。本書に出てきたプログラムをじっくりと読んで理解するようにして下さい。ここで学ぶことは実用性のあるプログラムを作るのに十分役に立ちますので、いろいろな便利なプログラムを自分で設計開発してみてください。

9.1 自動販売機のシミュレート

本実習では局所状態を持ったデータ (オブジェクト) による、プログラミング技法の基礎を学びます。 オブジェクトとは、いくつかのデータとそれを操作する基本的な手続き群を封じ込めたものです。これまでに紹介したプログラムでは、(一時的な作業用の局所変数を除いては) データはトップレベルの変数に保持されていました。このため、他の手続きが同一の名前の変数を別の目的で使っていて、その変数の値が他の手続きによって書き換えられてしまう可能性がありました。

ここで紹介する技法では、オブジェクトの中にいくつかの固有のデータと、それを操作する手続き群を封じ込めています。データの値を変更するには、オブジェクトにメッセージを送ります。**メッセージ** (message) とはオブジェクトに対する指令で、何らかの動作を要求するものです。メッセージを受けとったオブジェクトは、メッセージが正しい (許される) ときだけメッセージに対応した手続きを実行し、必要に応じて内部データを変更します。

データはオブジェクトの中に封じ込められているために、データを変更するにはオブジェクトにメッセージを送るしか方法がなく、他の手続きが勝手にデータを書き換えることはできません。このために、プログラムの誤りによるデータの破壊を防ぐことができます。

9.1.1 固有の局所データを持つオブジェクト

まずは局所データを持つオブジェクトを作る方法から学んでゆきましょう。次の例は「財布」オブジェクトを作る手続きです。

```
(define (make-wallet money)
  (lambda (amount)
    (if (< money amount)
      "Not enough money"
      (begin
        (set! money (- money amount))
        money))))
```

手続き make-wallet は、ひとつの引数 money を持ちます。式 (make-wallet 100) を評価すると、(lambda (amount) ···) の評価結果が (make-wallet 100) の値として返されます。

返される値は「手続き」と呼ばれる型のデータです。(そのため、手続き make-wallet は、「手続きを作る」手続きです。) 作られたデータは、手続きと同じように呼び出すことができます。この手続きは (lambda (amount) ···) となっているように、1 つの引数を持っています。では、試してみましょう。

```
> (define w (make-wallet 100))
#<unspecified>
> (w 10)
90
> (w 10)
80
> (w 10)
70
```

変数 money は w だけが参照できる変数です。このように参照できるものが一部に限られていることから、変数 money は w の局所変数と呼ばれます。(詳しくは 6.3ページの説明を読み返して下さい。) 局所変数は同一名のトップレベルの変数にはまったく作用しません。

```
> (define money 10000)
#<unspecified>
> (define w (make-wallet 100))
#<unspecified>
> (w 10)
90
> (w 10)
80
> money
10000
```

手続き w は、もし財布に十分なお金があれば、変数 money から w の引数に与えられた amount を引き、その結果を money に代入しています。そして財布に残った金額 money を実行結果として返しています。上の実行例では 2 度続けて (w 10) を評価していますが、(set! money (- money amount)) によって財布の残金を代入しているので、(w 10) の値は呼び出すごとに違っています。変数 money が財布の状態 (残金) を表す変数で、時々刻々移り変わる財布の状態を保持しています。(図 9.1参照を参照して下さい。)

w の値となっている手続きの定義を見ると

```
(lambda (amount)
  (if (< money amount)
    "Not enough money"
    (begin
      (set! money (- money amount))
      money)))
```

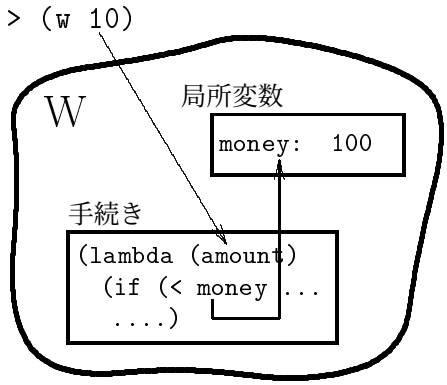


図 9.1: 局所変数を持つ手続きオブジェクト

となっていて、変数 money は一見トップレベルで定義された変数のように見えます。ですが money はトップレベルで定義された変数ではなく、手続き make-wallet の引数として用いられた変数です。手続きの引数として使われる変数は局所的な変数で、その手続きの内部でしか使えません。式 (lambda (amonut) ···) は手続き make-wallet の内側に置かれているので、w が set! で代入をする変数 money は手続き make-wallet の引数です。このことより、代入をする変数は make-wallet の内側だけでしか参照できない局所変数で、トップレベル変数ではないことが分かります[1]。

これまでの説明で想像つくように、変数 money は make-wallet が呼ばれるたびに新しいものが用意されます。すなわち最初に評価された (make-wallet 100) での 変数 money と、次に評価された (make-wallet 100) での変数 money では、 名前は同じでも実体が違います。この手法を使うことで、「同じ種(しゅ)の、異なった個体」をいくつも作ることが可能となります。

```
> (define w1 (make-wallet 100))
#<unspecified>
> (define w2 (make-wallet 100))
#<unspecified>
> (w1 20)
 80
> (w2 10)
90
```

[1] (make-wallet 100) が評価されたときに、新たに money の値が 100 である環境が作られ、その環境のもとで (lambda (amount) ···) が評価されます。λ式が評価されるとその評価値である手続きデータには、評価された時の環境と式の定義 ((lambda (amount) ···)) が含まれています。そしてその手続きデータが評価されるときには、make-walletが評価されたときの環境に、手続きデータに対する引数に対する環境を加えて拡大された環境のもとで、ラムダ式の本体が評価されます。このような理由のため、単なるλ式を局所変数を持ったオブジェクトとすることができる訳です。

```
> (w1 40)
40
```

以上で手続きデータに固有のデータを保持させるための、基本技法を学びました。

9.1.2 自動販売機オブジェクト

次は上で学んだ技法を使って、缶ジュースの自動販売機 (vending machine) の真似をするプログラムを作ります。自動販売機には数種類の缶ジュースが、それぞれの種類ごとにたくさん入れられていて、売れるたびに残りのジュースは 1 つづつ減ってゆき、売上のお金がたまって行きます。本実習で作るプログラムは、簡単化のために取り扱うジュースの種類を 1 つに限定します。(多くの種類を取り扱うことのできるプログラムは、練習問題で取り上げます。)

ジュースの在庫数は販売機の状態ですので、これを局所データとして保持します。ジュースを買うにはお金を入れ、選んだジュースのボタンを押さないといけません。ジュースが売り切れているかどうかを調べる機能も必要です。以上のことより、本実習では次のような機能を持つ自動販売機プログラムを作ります。

- 販売機を動かし始めるとき (販売機オブジェクトを作るとき)、最初に入れるジュースの総数を指定します。
- 販売機のボタンが押されたときに (販売機オブジェクトに「ボタン」メッセージが送られるとき)、ジュースがあれば売ります。もし売り切れなら、そのことを知らせます。
- ジュースの残りを表示します。(販売機オブジェクトに「残り」メッセージを送ると、在庫数を返します。)

ここでは自動販売機をオブジェクトとみなしていますが、財布の例の場合と違ってジュースを買うとか、在庫を調べるなど、ひとつのオブジェクトがいろいろな動作をします。このために、どの動作をするかを指定する必要があり、動作の指定は引数で与えるようにします。

作ろうとしているプログラムの大まかな構造は、次のようにすれば良いでしょう:

```
(define (new-vend stock)
  (lambda message
    (cond
      (⟨ button メッセージである ⟩
        ⟨ ジュースを売る ⟩ )
      (⟨stock メッセージである ⟩
        ⟨ 在庫数を返す⟩)
      (⟨ それ以外なら, エラー ⟩ ))))
```

手続き new-vend は、最初に販売機に入れるジュースの数 stock を引数に持っています。この変数は局所的な変数で、販売機の持つジュースの数を憶えるのに使います。

これを基にして自動販売機オブジェクトを生成する手続きを作ると、次のようになります。

```
(define (new-vend stock)
  (lambda message
    (cond
      ((eq? (car message) 'button)
        (if (= stock 0)
          "Sold out"
          (begin
            (set! stock (- stock 1))
             'juice)))
      ((eq? (car message) 'stock)
        stock)
      (else
        (error "Unknown message" message)))))
```

(lambda messages ···) で分かるよう、販売機オブジェクトは引数 message を持ちます。(引数のならびが (message) ではなく message であり、括弧で囲まれていない点に注意して下さい。) 仮引数がひとつで括弧に囲まれていない場合は任意個の実引数を与えることができ、それらはリストになって仮引数 message に渡されるようになっています。

もしメッセージ (つまり message の第 1 要素) が button なら、ジュースを販売する動作をします。もしジュースがなければ (つまり (= stock 0) のとき)、"Sold out" を返し、売り切れを知らせます。まだジュースがあるときは在庫の数を 1 つ減らし、販売した商品を渡します。販売機オブジェクトへのメッセージが stock のときは、ジュースの残り数 (stock の値) を返します。もしそれ以外のメッセージが渡されたらエラーにします。

では 2 つのジュースを入れた販売機を作り、実際に試してみましょう。

```
> (define v (new-vend 2))     — 2 つのジュースを持つ販売機を作る
#<unspecified>
> (v 'button)          — ジュースを 1 つ買う
juice
> (v 'stock)           — 残りは?
1
> (v 'button)          — もう 1 つジュースを買う
juice
> (v 'button)          — 更にジュースを買う
"Sold out"           — 売り切れです
```

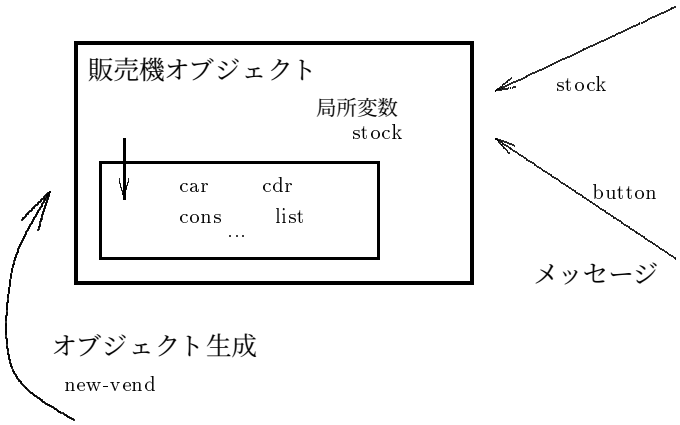


図 9.2: 販売機オブジェクトとメッセージ

9.1.3 まとめ

「ジュースを買う」ということをプログラムで表現するのに、(v 'button) という書き方をしていることに注意して下さい。これは、「販売機」に「ジュースの販売」を依頼する、という書き方になっています。以前でのプログラミング手法では、「販売機データ」を引数にして「ジュースを買う」手続きを呼び出す (たとえば (buy-juice v)) という書き方をしていました。(したがって、データと手続きとその実行に関する考え方が大きく違っています。)

ジュースを売る方法 (在庫の管理方法や売り上げ金の管理など) は、自動販売機自身が熟知していることです。一般の利用者がジュースを買うのに、自動販売機の内部構造に依存したジュースの取り出し方法や、売り切れチェックをする方法を知らないといけないとしたら、大変に使いにくいことでしょう。(buy-juice v) という書き方は、まさにそのような方法です。また別の種類の自動販売機でジュースを買うには、(buy2-juice v2) というように、別の手続きを使わないといけないかもしれません。色々な種類の販売機を使うときには、それぞれどの手続きを使うのかを完全に記憶しておかないといけません。

本実習で示した方法では、どのような方法でジュースが買えるかは知らなくても、「ジュースを売って下さい」というメッセージを自動販売機に送るだけでジュースが買えます。別な見方をすれば、「ジュースを売って下さい」というメッセージでジュースが買えるようになってさえいれば、内部構造が他の販売機とまったく違っていても、そのことはまったく気にせずにジュースを買うことができます。(図 9.2を参照して下さい。)

このように本実習で学んだプログラムの記述法は、非常に自然なものであることが分かると思います。またオブジェクトの内部にデータが埋め込まれて、外からはメッセージを介してでなければアクセスできないようになっています。これによってプログラミングミスによるデータの破壊を防ぐことができ、プログラムの信頼性を高めることができます。

練習問題

1. 以下の ⟨???⟩ を埋めて、手続き accumulate を完成しなさい。

```
(define (accumulate)
  (define ⟨???⟩ '())
  (define new-obj
    (lambda (d)
      (⟨???⟩ s (cons d s))
       s))
  new-obj)
```

この手続きは、引数に与えられたものをどんどんリストに蓄えてゆくオブジェクトを生成します。実行例は次の通りです。

```
> (define a (accumulate))
#<unspecified>
> (a 'pen)
(pen)
> (a 'a)
(a pen)
> (a 'is)
(is a pen)
> (a 'this)
(this is a pen)
```

2. 手続き make-wallet をもっと簡潔にしようとして、次の手続きを作りました。

```
(define (make-wallet-2 money amount)
  (if (< money amount)
    "Not enough money"
    (begin
      (set! money (- money amount))
      money)))
```

この手続きは、make-wallet と同じ動作をしません。どのような動作をするか推測し、推測が当たっているかどうか確かめなさい。また、どうして make-wallet と同じ動作しないのか、その理由を説明しなさい。

3. メッセージに対する処理が長くなる時は、処理を手続きとしてまとめた方がすっきりとしたプログラムになります。次のように、 new-vend を書き換えて new-vend2 にしました。⟨???⟩ を埋めなさい。

```
(define (new-vend2 stock)
  (define (button-msg)
    ⟨???⟩)
  (define (stock-msg)
    ⟨???⟩)
  (lambda message
    (cond
      ((eq? (car message) 'button) ⟨???⟩)
      ((eq? (car message) 'stock) (stock-msg))
      (else (error "Unknown message" message)))))
```

4. 手続き new-vend では、1 つづつしかジュースを買うことができません。新たにまとめ買い用のボタンを付け、5 個のまとめ買いができるようにプログラムを改造しなさい。

5. 手続き new-vend での販売機は、ジュースが売り切れたらそれまでです。(販売機の使い捨てです!) そこで、vend にジュースを補充する機能を付け加えなさい。なお、補充は supply と言うメッセージと、補充するジュースの数をメッセージとして販売機に渡すようにしなさい。たとえば (v 'supply 10) とすれば、販売機 v に 10 個のジュースが補充されるようにしなさい。

6. 販売機に売上金を蓄えるようにし、さらにその売上金を取り出すことができるようにしなさい。誰でも売上金を取り出せると困るので、「鍵」として合い言葉を決め、お金を取り出す時には合い言葉を言わないといけないようにしなさい。もちろん、間違った合い言葉を言う人に売上金を渡してはいけません。

   もし合い言葉がどの販売機でも同じなら、ある販売機の合い言葉がばれてしまうと、他の販売機ぜんぶが盗難にあってしまいます。このようなことが起きないよう、自動販売機オブジェクトを作るとき、引数にそのオブジェクトの合い言葉を指定することで、販売機ごとに違った合い言葉を決めることができるようにしなさい。

7. * 本実習で作った自動販売機は、1 種類のジュースしか取り扱いませんでした。そこで、複数のジュースを販売する自動販売機を作成しなさい。販売機はオブジェクトを作成するときに、ジュース毎に名前、個数、単価を指定できるようにしなさい。また、販売しているジュースの名前の一覧を調べる機能も付けなさい。

レポート

1. メッセージ伝達によるプログラミングの利点を、例を述べて説明しなさい。また、メッ

セージ伝達によるプログラミングが可能な言語をいくつか選び、それぞれについて言語の概略、開発の目的、特徴、そして欠点を調べなさい。

9.2 ファイル処理

本実習の目的は、ファイルの読み書き (入出力) をするプログラムの作成です。これまでの入出力は、画面に対して表示をするだけでした。実用的なプログラムでは、膨大な入力データをファイルに記録しておき、プログラムは入力データをファイルから読んで処理をし、そしてその結果を他のファイルに書き込んだりプリンタに印刷したりします。最近の多くのオペレーティングシステムでは、プリンタや画面などの入出力装置に対する入出力も、ディスク上のファイルと同じ方法でできるようになっています[2]。

ディスクには多くのファイルを作成し保持することができます。そのためファイルを読み書きするときは、たくさんのファイルの内のどれにするのかを指定しないといけません。ファイルの指定をするには、名前でファイルを指定します。ファイルの名前は**ファイル名** (file name) と呼んでいます。ファイルをアクセス (読み書き) する手順は、6.2で説明した通りです。

9.2.1 ファイルからの読み込み

では手始めとして、式がいくつか書かれているファイルがあるものとします。そのファイルに書かれている式すべてを表示するプログラムを作成します。これを行なう手続きを show-s-exp とし、引数としてファイル名を渡すものとします。6.2 で示したファイルへのアクセス方法の手順に従うと、いま作ろうとしている手続きの骨格は以下のようになるでしょう。

```
(define (show-s-exp filename)
  ⟨filename をオープンし、port をそのポートとします⟩
  ⟨port から式をひとつ読み込み、 exp に入れます⟩
  ⟨exp がファイル終端子でない間、次を繰り返し実行します⟩
        ⟨exp を表示します⟩
        ⟨port から新たに式を exp に読み込みます⟩
  ⟨port をクローズし、実行を終ります⟩ )
```

ではこの骨格を元に、プログラムを作成します。ファイル filename から読み込むので、オープンをするのに手続き open-input-file を使います。

```
(define (show-s-exp filename)
  (let ((port (open-input-file filename)))
    (letrec
      ((loop
```

[2]たとえば MS-DOS では、PRN というファイル名にデータを書き込むことで、そのデータがプリンタに送られて印刷されます。このようにファイルと装置に対する入出力を区別しなくて済むようにするのが、近年のオペレーティングシステムの主流です。

```
        (lambda (exp)
          (if (eof-object? exp)
            'done      — ファイルの終端なら終了
            (begin     — そうでなければ表示し、繰り返します
              (write exp)
              (newline)
              (loop (read port)))))))     — 繰り返します
      (loop (read port)))       — 繰り返しのはじまり
    (close-input-port port)))
```

このプログラム例では、letrec を使って繰り返しをしています。まず letrec の中で変数 exp を持つ手続き loop を作り、(loop (read port)) によって繰り返しを開始します。これにより、まず最初のデータをファイルから読みとります。loop の本体では、引数 exp がファイル終端子かどうかを調べ、もしそうなら letrec の実行結果として done を返します。(この値そのものは特に意味はありません。) exp がファイル終端子でなければそれを表示し、改行します。そして (loop (read port)) にて別の式を読み込み、繰り返し loop を実行します。letrec が終了すると (close-input-port port) により、ポートをクローズして手続き show-s-exp の実行が終ります。

ではこれを実行してみましょう。入力データファイルとして、次の内容のファイル testdata.scm を使います。(プログラムを実行する前に、エディタを使ってこのデータファイルを作っておいて下さい。)

```
(a b c)
123 111
a b c d

"STRING!!"

; a comment line
end?
```

式 (show-s-exp "testdata.scm") を評価してみます。

```
> (show-s-exp "testdata.scm")
(a b c)
123
111
a
b
```

```
c
d
"STRING!!"
end?
#<unspecified>
```

式を単位として読み込みをしているので、入力ファイルに 1 行に複数の式が書かれていても別々のものとして読み込まれ、空行があってもそれは無視され、コメント部分は読み飛ばされていることに注意して下さい。

上の例では 1 つの式を読み込んでは表示するだけの動作でしたが、今度はもう少し複雑なことをする手続きを作ってみましょう。データファイルにいくつかの数が書かれていてそれらの合計を求める、という簡単なデータ処理をする手続き sum-file を作ります。手続きの骨格は先ほどの場合と似ています。

```
(define (sum-file filename)
  ⟨filename をオープンし、port をそのポートとします⟩
  ⟨変数 total の値を 0 にします⟩
  ⟨port から exp に式を読み込みます⟩
  ⟨exp がファイル終端子ない間、以下を繰り返します:⟩
        ⟨total を exp だけ増やします⟩
        ⟨port から exp に式を読み込みます⟩
  ⟨port をクローズします⟩
  ⟨total を結果として返します⟩ )
```

では手続き sum-file を作ります。

```
(define (sum-file filename)
  (let ((port (open-input-file filename))
        (total 0))
    (letrec
      ((loop
         (lambda (exp)
           (if (eof-object? exp)
             'done
             (begin
               (set! total (+ total exp))
               (loop (read port)))))))
      (loop (read port)))
    (close-input-port port)
```

```
    total))     — total を手続きの結果とする
```

total が最後に置かれているのは、sum-file の評価結果を total の値とするためのものです[3]。

入力データを記録したファイルとして、以下の内容が書かれたファイル numdata.scm を使うことにします:

```
89 78 12 45 60 92
100 88
```

では sum-file を実行してみましょう。

```
> (num-file "numdata.scm")
564
```

確かに合計が計算されていることが分かります。

9.2.2 ファイルへの書き込み

これまではファイルからの読み込みだけでしたが、今度はファイルへの書き込みについて学びます。引数に与えられたリストの各要素を、1 行にひとつの Scheme データをファイル out.scm に書く手続き out-list を示します。

```
(define (out-list s)
  (let ((port (open-output-file "out.scm")))
    (do ((rest s (cdr rest)))
        ((null? rest))
      (write (car rest) port)
      (newline port))
    (close-output-port port)))
```

ファイルを書き込みオープンするために、open-output-file を使っています。またファイルへのデータの書き込みに write と newline を使っていますが、ポートを引数に与えることで書き込み先を指定し、ファイルへの書き込みを実現しています。

この手続きを実行してみましょう。

```
> (out-list '(a b 200 (foo bar baz) ((0 1 2) (3 4 5))))
#<unspecified>
```

実行後にファイル out.scm を見てみると、

[3] let の評価結果は、本体の最後の式の評価結果なので、本体の最後の式として total を置いています。

```
a
b
200
(foo bar baz)
((0 1 2) (3 4 5))
```

となっています。

9.2.3 文字単位での入出力

以上での入出力は式を単位にしていました。しかしこれだけでは、

- ファイル中の行数を数える
- ファイル中の小文字を全て大文字に変える
- ファイル中にある 'A' から 'Z' までのそれぞれの文字の出現頻度を数える

などの処理ができません。これらを可能にするには、文字単位での入出力が必要となります。以降では、文字単位での入出力をするプログラムを作ります。文字の読み込みをする手続は `read-char` で、文字の書き込みをする手続は `write-char` なので、これらを使います。

ファイルの内容が何行あるかを数える手続き `count-lines` を作ります。行の終りには**改行文字** (return character) があるので、この数を数えることで行数が分かります。なお改行文字は、`#\newline` または `#\nl` という形でプログラム中に記述されます。プログラムの構造は以下のようになります。

```
(define (count-lines filename)
  ⟨filename をオープンし、 port をそのポートとします⟩
  ⟨ 変数 lines の値を 0 にします⟩
  ⟨port から ch に 1 文字読み込みます⟩
  ⟨ch がファイル終端子でない間、以下を繰り返します⟩
        ⟨ もし ch が改行文字なら、lines を 1 つ増やします⟩
        ⟨port から ch に 1 文字読み込みます⟩
  ⟨port をクローズします⟩
  ⟨lines を手続きの値として返します⟩ )
```

これを元にすると、以下のプログラムができます。

```
(define (count-lines filename)
  (let ((port (open-input-file filename))
```

```
        (lines 0))
    (letrec
        ((loop
          (lambda (ch)
           (cond
             ((eof-object? ch)
              (close-input-port port)
              lines)
             ((equal? ch #\newline)
              (set! lines (+ lines 1))
              (loop (read-char port)))
             (else
              (loop (read-char port)))))))
      (loop (read-char port)))))
```

ではこれを実行してみましょう。

```
> (count-lines "testdata.scm")
8
> (count-lines "numdata.scm")
2
```

次は文字単位での書き込みの例として、ファイル中の小文字を大文字に変更する手続き to-upper を作ります。この手続きは、入力ファイルと出力ファイルのファイル名が引数として与えられ、入力ファイルに現れる小文字をすべて大文字にし、それ以外の文字はそのままにして出力ファイルに書き出します。このプログラムの構造は、以下のようになります。

```
(define (to-upper infile outfile)
  ⟨infile を入力オープンし、inport をそのポートとします⟩
  ⟨outfile を出力オープンし、outport をそのポートとします⟩
  ⟨inport から ch に 1 文字読み込みます⟩
  ⟨ch がファイル終端子でない間、以下を繰り返します⟩
     ⟨もし ch が小文字なら、ch を大文字にし outport に書きます⟩
     ⟨そうでなければ、ch を outport に書きます⟩
     ⟨inport から ch に 1 文字読み込みます⟩
  ⟨inport と outport をクローズします⟩ )
```

この骨格に従って実現した手続きは、以下の通りです[4]。

[4]手続き char-upcase は、小文字でない文字が引数に与えられたときは、引数をそのまま返します。そのために、わざわざ手続き char-lower-case? を使って、小文字であるかどうかで場合分けをしなくて構いま

```
(define (to-upper infile outfile)
  (let ((inport  (open-input-file infile))
        (outport (open-output-file outfile)))
    (letrec
        ((loop
          (lambda (ch)
            (cond
             ((eof-object? ch)
              (close-input-port inport)
              (close-output-port outport))
             ((char-lower-case? ch)
              (write-char (char-upcase ch) outport)
              (loop (read-char inport)))
             (else
              (write-char ch outport)
              (loop (read-char inport)))))))
      (loop (read-char inport)))))
```

動作を確かめるための入力ファイルとして、次の内容を持つファイル testfile を使います:

```
This is the TEST file.
```

では手続き to-upper を実行してみましょう。

```
> (to-upper "testfile" "out")
done
```

ファイル out を見ると、testfile で小文字だったものが大文字となっているのが分かります:

```
THIS IS THE TEST FILE.
```

練習問題

1. ファイル中に現れる式の数を数える手続き count-s-exp を作りなさい。

2. ファイル中の式の内、記号の数を数える手続き count-symbol を作りなさい。

せん。

3. 引数に与えられた式と同じものが、ファイルの中にいくつあるかを数える手続き count-occur を作りなさい。この手続きは (count-occur 式 ファイル名) の形式で呼び出すものとします。なおファイルから読み込んだ式と引数に与えられた式を比較するのに、手続き equal? を使いなさい。

4. ファイルの文字数を数える手続き count-char を作りなさい。(空白文字や改行文字などの制御文字も、文字として数えます。)

5. ファイルの単語数を数える手続き count-word を作りなさい。単語と単語の区切りは、空白文字 (#\space) や桁飛ばし文字 (#\tab)、改行文字 (#\newline) によるものとします。単語の間に区切りの文字が続いて現れるときも、正しくカウントできるようにしなさい。

6. ファイル内での、'A' から 'Z' までの各アルファベット文字の出現頻度を数える手続き count-freq を作りなさい。なお対象とするファイル名は、引数に与えられるものとします。

7. 入力ファイルとして Scheme のプログラムファイルが与えられるものとします。このファイルの中で、トップレベルで定義されている手続きおよび変数の名前をリストにして返す手続き get-define を作りなさい。

8. * 引数として与えられる入力ファイルに英文が入っているものとします。これを単語と単語の間を区切り、出力ファイルの 1 行にひとつの単語が現れるようにしなさい。なお、この手続きは (a-word/line 入力ファイル 出力ファイル) として使うものとします。

9. * 入力ファイルとして Scheme のプログラムファイルが与えられるものとします。このファイルに現れるすべての記号をリストにして返す手続き symbols-in-file を作りなさい。ただし、実行結果として返されるリストに同じ記号が重複して現れないようにしなさい。

10. * 名前と科目 (国語、数学、体育) のテストの採点結果が、1 人につき 1 つリストになって入力ファイルに入っているものとします。このファイルを読み込んで、科目毎の平均点および最高点を求めるプログラムを作りなさい。さらに、各科目毎に最高点を取った人 (複数いるかも知れません) の名前も出力しなさい。処理結果はファイルに書き出すようにしなさい。このプログラムは (test-score 入力ファイル 出力ファイル) の形で呼び出すものとします。

    個人のデータは (名前 国語の成績 数学の成績 体育の成績) という形式で表されていて、入力ファイルには各個人のデータがならんでいます。たとえば、入力データファイルは次のようなものです。

```
("Tanaka Misako"     98  68  84)
("Nishida Hikaru"    98 100  76)
("Takahashi Yumiko"  50  72  82)
("Takata Mayuko"     97  67  78)
("Yoshida Mayuko"    68  80  79)
("Yoshida Makiko"    86  99  77)
("Amuro Namie"       69  89  98)
("Uchida Yuki"       71  89  70)
```

11. * 上の問題でのデータファイルに対して、科目ごとにテストの成績順で整列し、それをファイルに出力する手続きを作りなさい。さらに総合成績で整列した結果をファイルに出力する手続きも作りなさい。

レポート

1. オペレーティングシステムの教科書を読んで、ファイルシステムの持つ役割と機能を調べなさい。

2. 実習 10 で使った入力データファイルは、多分採点済みのテストの回答用紙を見ながら、手作業でファイルに入力したものでしょう。手による入力ではタイプミスや勘違いによって、間違った入力をしてしまうことがあります。入力が正しいかをチェックするには何度も見直しをする必要がありますが、データ量が多いと大変です。

   そこで明らかなミスを自動的に検出することを考えます。テストの採点結果に限って考えて、どのような入力の誤りが生じそうかを考察しなさい。またそれらの誤りのうち、どのようなものなら自動的な検出できますか? そしてその誤りを検出するプログラムを作り、故意に誤りを入れた入力ファイルに対して実行してみなさい。

9.3 整列

本実習では、ばらばらにならんだデータを大きい順 (あるいは小さい順) にならびかえる、**整列** (sorting) について学びます。たとえば入学試験では、試験結果の良い順で受験者をならべ (整列し)、定員が 100 名なら上から順に 100 名を選び出すことで、合格者を簡単に決めることができます。別の例として、ある人の住所を住所録から調べたいときには、データベースの先頭から順々に調べてゆけば確かに見つけることができます。もし名前の順で住所録データベースの項目がならべてあれば、辞書で英単語を調べるときのように関係ないところは飛ばすことができます。('system' の意味を辞書で調べる場合、'a' から 'r' で始まる単語は読み飛ばすことができます。) データ数が少ない場合はそのありがたみはあまり感じませんが、データ数が数千、数万となると、データの検索時間はまるきり違ってきます。

これらの例で分かるように、整列法を学ぶことは実用的なソフトウエアを作るために必要なことです。ここでは整列時間は長くても単純な方法の**バブル法** (bubble sort) と、少々複雑ながら整列時間が短い**クイック法** (quick sort) の 2 つの整列法を紹介し、整列プログラムを作ります。

9.3.1 バブル法

トランプのカードを大きい順にならべるときは、たぶん次のようにするでしょう:

1. 手持ちのカードの一番の大きなカードを見つけて抜き出して、机に置きます。
2. 再び一番の大きなカードをみつけて抜き出して、先ほどのカードの上に置きます。
3. 同じことを、カードがなくなるまで繰り返します。

これを少しだけコンピュータープログラムに近い形で記述すると、次のようになります。なお整列すべきデータは、横にならんで置かれていると考えます。

1. 一番左から右端までのデータ中の最大のものを見つけ、それと一番左のデータを入れ換えます。
2. 左から 2 番目から右端までのデータの中の最大のものを見つけ、それと左から 2 番目のデータを入れ換えます。
3. 同様なことを、右端にたどり着くまで繰り返します。

上の動作が終った状態での、左から i 番目のデータを a_i とおきます。するとどの i に対しても、$a_i \geq a_{i+1}$ が成立します。というのも、もし $a_i < a_{i+1}$ であるような i があったとすると、i 番目の繰り返しのときに最大のデータを選ばなかったことになり、上で説明した整

列の方法に反しています[5]。どの i に対しても $a_i \geq a_{i+1}$ なので、ただしく整列できることが分かります。

なお、このようにある問題を解くための手順 (あるいは解法) のことを、**アルゴリズム** (algorithm) といいます[6]。ひとつの問題 (たとえば整列問題) に対していくつものアルゴリズムが考えられますし、ひとつのアルゴリズムをいく通りものプログラムによって実現することができます。ですので、「問題」、「アルゴリズム」、「プログラムによる実現」は違うものですので、混同しないようにしましょう。

上で説明した方法による整列アルゴリズムは、**選択法** (selection sort) (または **単純選択法** (straight selection sort)) と呼ばれています。ここではこれに少し変更を加えた整列アルゴリズムの**バブル法** (bubble sort) を説明します。バブル法は、次の手順による整列アルゴリズムです。

- 最初、全体での最大値を左端に置くために、次のことをします。左から 2 番目から右端までデータを順々に調べ、左端よりも大きなものが現れたらデータを入れ換えます。これを右端まで続けます。

- 次に、左から 2 番目から右端までの最大値を左から 2 番目に置くために、次のことをします。左から 3 番目から右端までデータを順々に調べ、左から 2 番目のデータより大きなものが現れたら、データを入れ換えます。これを右端まで続けます。

- 以降、同様なことを右端にたどり着くまで繰り返します。

バブル法による整列アルゴリズムを、Scheme 手続きとして実現する `bubble-sort!` を作ります[7]。なお入力となるデータの列は、ベクトルで与えられるものとします。

手続きのおおまかな構造は以下のようになります。

```
(define (bubble-sort! numbers)
  ⟨各 i (0 から⟨numebrs の要素数-2⟩) に対し、以下を実行します⟩
    ⟨各 j (i+1 から⟨numebrs の要素数-1⟩) に対し、以下を実行します⟩
      ⟨もし左から j 番目の要素が、左から i 番目の要素よりも
       大きい場合は、それらを交換します⟩
  ⟨整列した結果を返します⟩ )
```

これを Scheme 手続きとして実現したものを以下に示します。

[5] この手法による証明は、**背理法**と呼ばれています。

[6] アルゴリズムは、**算法**とも呼ばれることがあります。

[7] 手続き `set!` や `vector-set!` は、その名前の最後に ! がついています。Scheme では (新しいデータを作って返すのではなく) データの値を直接書き換える手続きの名前の最後に ! をつける慣習があります。ここで作る整列手続きは入力データを書き換えるので、この慣習に従って名前を付けています。

```
(define (bubble-sort! numbers)
  (do ((i 0 (+ i 1))
       (swap #f))        — i と j の交換のとき、一時的に使用
      ((= i (- (vector-length numbers) 1)) numbers)
    (do ((j (+ i 1) (+ j 1)))
        ((= j (vector-length numbers)))
      (if (< (vector-ref numbers i) (vector-ref numbers j))
        (begin      — 左から i 番目の要素と、左から j 番目の要素を交換
          (set! swap (vector-ref numbers j))
          (vector-set! numbers j (vector-ref numbers i))
          (vector-set! numbers i swap))))))
```

この手続きの中で、左から i 番目の要素と j 番目の要素を交換するのに、一時的な変数として swap が使われています。ではこれを実際に動かしてみましょう。

```
> (bubble-sort! (vector 2 3 43 2 1 5 0))
#(43 4 3 2 2 1 0)
> (bubble-sort! (vector 1 2))
#(2 1)
> (bubble-sort! (vector 100))
#(100)
```

正しく整列されています。他にもいろいろ入力を与えて、試してみて下さい。

9.3.2 クイック法

今度は**クイック法** (quick sort) と呼ばれる整列アルゴリズムを紹介します。バブル法に比べて複雑ですが、より高速に整列をするアルゴリズムです。

クイック法の基本的なアイディアは次の通りです。

1. まず整列すべきデータの中から、基準値をひとつ選びます。
2. 基準値以上のグループと基準値以下のグループの、2 つのグループに入力のデータを分割します。
3. 基準値以上のグループを整列をします。
4. 基準値以下のグループを整列をします。
5. 各グループの整列結果をつなげて、整列結果とします。

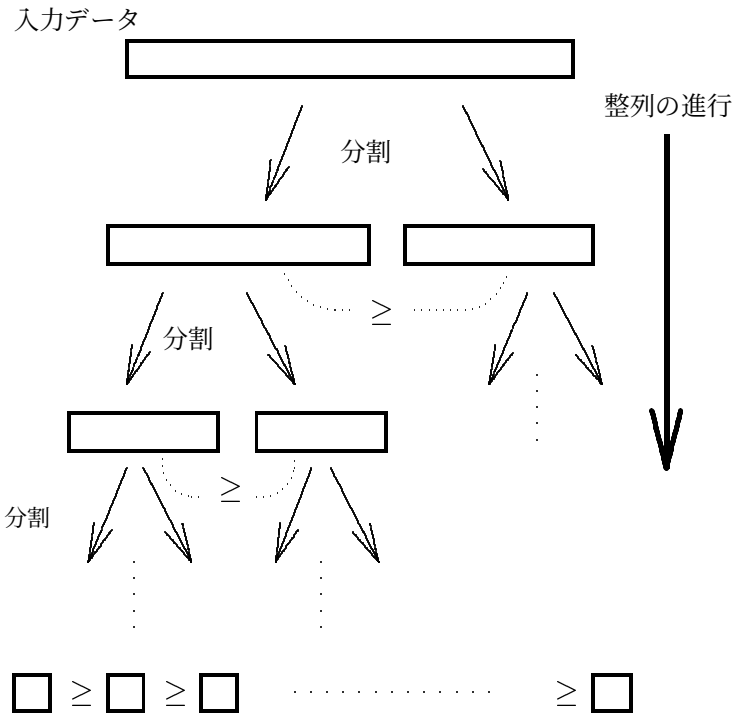


図 9.3: クイック法の概略図

これを図示すると、図 9.3 のようになります。この手続きの特徴は、まず最初に大まかにデータを 2 つのグループに分けるところにあります。こうすることで、より小さくなったデータのグループの整列を考えればよいことになります。最終的にはグループの大きさが 2 (あるいは 1) となり、このように小さなデータ数の入力の整列は、簡単に行なえます。(データ数が 2 なら、2 つを比べて入れ換えをするだけです。)

クイック法による整列アルゴリズムの構造を考えると、

- 与えられた問題をより小さくて簡単な問題 (**部分問題**といいます) に分けるステップの**分割段階**と、
- 部分問題の解をまとめるステップである**統合段階**

の 2 つに分かれていると考えることができます。これはクイック法だけでなく他のアルゴリズムにも応用可能な手法で、**分割統治法** (devide and conquer) と呼ばれています。

入力データを分割する手続き `partition!` がすでにあるものとして (後でこの手続きを作ります)、整列の手続きを作ります。手続き `partition!` は、(`partition!` $v\ x\ y$) として呼び出されるものとし、ベクトル v の添字 x から添字 y までの要素について分割し、2 つのグループの境界である要素の添字を返します。もし (`partition!` $v\ x\ y$) の評価結果が z とすれば、次のことが実行終了時に成り立ちます。

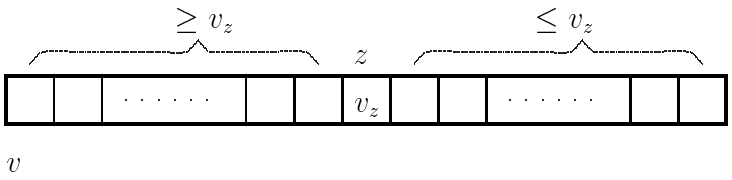


図 9.4: partition! による分割

- v の z 番目の要素は partition! が内部で選んだある基準値で、
- v の添字 x から添字 $z-1$ までには基準値以上のものが、
- v の添字 $z+1$ から添字 y までには基準値以下のものが入れられます。

厳密にいうと、次の通りです:

- すべての i $(x \leq i \leq z-1)$ に対して (vector-ref v i) $\geq$ (vector-ref v z) が成り立ち、さらに
- すべての i $(z+1 \leq i \leq y)$ に対して (vector-ref v z) $\geq$ (vector-ref v i) が成り立ちます。

これを図 9.4に示します。

クイック法による整列は、partition! を使えば簡単に作れます。

```
(define (quick-sort! vec)
  ; 内部手続き sub-sort!
  ;   ベクトル v の第 x 要素から第 y 要素までを整列
  (define (sub-sort! x y)
    (if (< x y)
      (let ((z (partition! vec x y)))    — 分割をする
        (sub-sort! x (- z 1))            — 左のグループを整列
        (sub-sort! (+ z 1) y))))         — 右のグループを整列
  ; quick-sort! の本体:
  ; sub-sort!を呼び、第 0 要素から第ベクトルの最後の要素までを整列
  (sub-sort! 0 (- (vector-length vec) 1))
  vec)
```

上の quick-sort! では、手続き sub-sort! が内部定義されています。手続き sub-sort! では最初に分割をし、そして分割した 2 つのグループそれぞれをさらに整列しています。分割してさらに整列された後は、すでに 2 つのグループは 1 つのベクトルの中で連続して置かれているので、統合段階は暗黙のうちになされています。

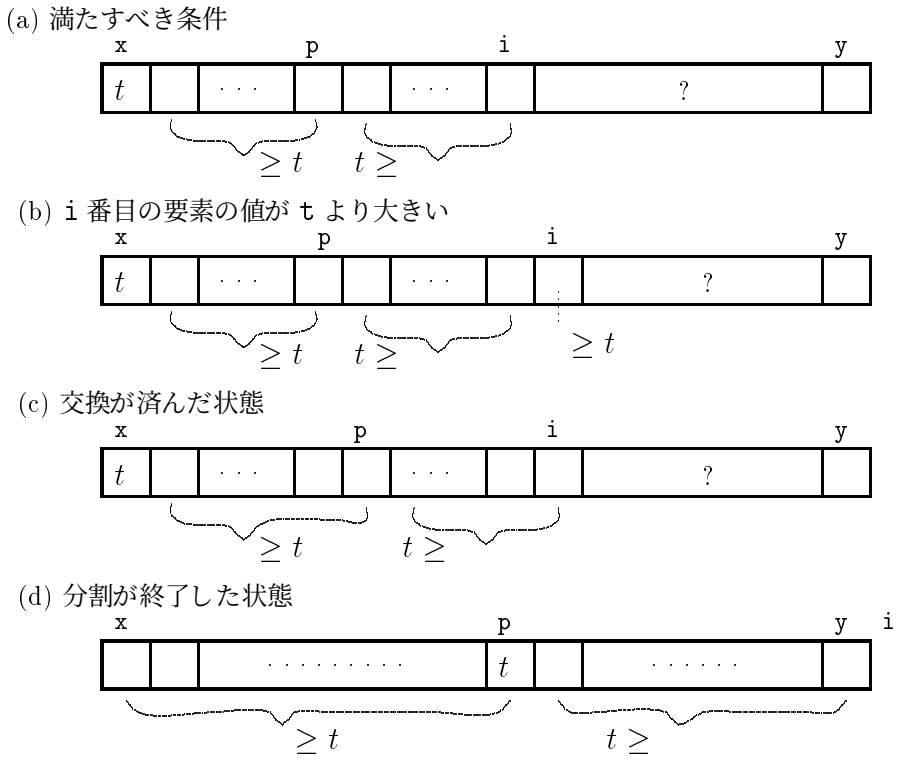


図 9.5: 分割

手続き quick-sort! の本体部分は (sub-sort! 0 (- (vector-length vec) 1)) となっていて、引数に与えられたベクトルの左端から右端までを sub-sort! を使って整列するようにしているだけです。手続き sub-sort! には、入力データが置かれているベクトル vec を引数として渡していないのにも関わらず中で vec が使われていることに注意して下さい。手続き sub-sort! の中での vec は、手続き quick-sort! の引数である vec を参照することになります。

注意しないといけない点は、どちらのグループにも partition! が返した添字の要素は除外されていることです。これにより分割後の 2 つのグループの要素数の和は、もとのものより 1 小さくなっています。このために、ソートすべき要素数は再帰呼び出しのたびに確実に小さくなってゆくので、再帰呼び出しが永久に続くことはありません。

では、データを二分割する手続き partition! を実現します。分割の大まかな考え方は以下の通りです。

1. x 番目の要素を基準値 t とします。

2. p の値を x にします。

3. i≤y である間、以下を繰り返します:
   i の値を 1 つ増やし、もし i 番目の要素の値が t より大きいと p+1 番目の要素と i 番

目の要素を交換し、p の値を 1 増やします。

4. x 番目の要素と p 番目の要素を交換します。

5. 評価結果として、p の値を返します。

この手順では、x 番目の要素から p 番目の要素は基準値 t 以下で、p 番目の要素から i 番目の要素は基準値 t 以上という性質が成り立つようになっています。そしてその範囲を徐々に広げていっています。最後に x 番目の要素と p 番目の要素を交換するのは、図 9.4で示されている、partition!に対する条件を満たすようにするためのものです。このように、繰り返しの本体が実行されても常に成立する性質を、**ループ不変量** (loop invariant) と呼びます。このループ不変量は、プログラムが正しいかどうかを調べるのに役立ちます。以上の一連の動きを、図 9.5に示しています。

分割方法を Scheme 手続きに直したものを以下に示します。上では説明しなかった細かな点にも十分注意して下さい。

```
(define (partition! vec x y)
  (let ((t (vector-ref vec x)))    — tが基準値
    ;
    (define (swap i j)   — i 番目の要素と j 番目の要素を交換する
      (let ((temp (vector-ref vec i)))
        (vector-set! vec i (vector-ref vec j))
        (vector-set! vec j temp)))
    ;
    (define (loop i p)
      (if (< y i)
        (begin
          (swap p x)
          p)
        (if (> (vector-ref vec i) t)
          (begin
            (swap (+ p 1) i)
            (loop (+ i 1) (+ p 1)))
          (loop (+ i 1) p))))
    ;
    (loop (+ x 1) x)))
```

ではこの partition! を試しに実行させてみましょう。

```
> (define qweqwe (vector 7 4 9 10 3 4 12 3 33 1 0 2))
#<unspecified>
> qweqwe
#(7 4 9 10 3 4 12 3 33 1 0 2)
> (partition! qweqwe 0 11)
4
> qweqwe
#(33 9 10 12 7 4 4 3 3 1 0 2)
```

この実行例では、partition! の評価結果は 4 なので、基準値はベクトルの 4 番目の要素である 7 です。ベクトルの 0 番目から 3 番目までの要素 33, 9, 10, 12 は基準値以上で、添字が $4+1$ 番目から 11 番目までの要素 4, 4, 3, 3, 1, 0, 2 は基準値以下であることが分かります。

分割をする手続きが完成したので、手続き quick-sort! を動作させることができます。

```
> (quick-sort! (vector 7 4 9 10 3 4 12 3 33 1 0 2))
#(33 12 10 9 7 4 4 3 3 2 1 0)
> (quick-sort! (vector 8 4 6 1 10 3 44 11 1 1 1))
#(44 11 10 8 6 4 3 1 1 1 1)
```

正しく整列できていることが分かります。

9.3.3 整列アルゴリズムの実行性能

バブル法よりもクイック法の方がより高速に整列するといいました。ではどれくらいの違いがあるのか、速さを比べてみましょう。実用的なビジネスソフトウエアでは、数万個あるいはそれ以上のデータを整列する必要があるので、整列にかかる時間は大変重要な関心事です。遅い整列アルゴリズムでは 1 週間かかるけれど、もし速い整列アルゴリズムでは 3 時間もあれば十分というのなら、速い整列アルゴリズムを使うべきでしょう。

データ数が少ないときはどちらの整列アルゴリズムでもあっという間に終了するので、データ数を多くして、整列の実行時間を測定します。ですが、データ数の大きな入力を手作業で作るのは大変なので、まずは入力データを自動的に作り出す手続きを作ります。

以下の手続き make-randam-vector は、引数に与えられた整数 n の大きさをもつ、要素が疑似乱数であるようなベクトルを作り出す手続きです[8]。疑似乱数の生成は、第 2 引数に渡された手続きを呼び出すことで得るものとします。手続き make-random は、疑似乱数を生成するオブジェクトを作る手続きです。

[8] **乱数** (random number) とは、規則性のない数の列です。ここで紹介する方法で得られる乱数は、真の乱数ではありません。というのも、プログラムによって次の値を何にするかが決まっているからです。ですが、得られた値を見てみると、一見乱雑な数が出てきているように見えるため、**疑似乱数** (pseudo random number) と呼ばれています。

```
; ランダムな値を持つ大きさ n のベクトルを作る
(define (make-random-vector n random-obj)
  (do ((rv (make-vector n))
       (i 0 (+ i 1)))
      ((= i n) rv)
    (vector-set! rv i
      (inexact->exact (floor (* (random-obj) 1000))))))

; 乱数生成オブジェクトを返す
(define (make-random)
  (define seed 1)
  (define (obj)
    (let* ((hi (truncate (/ seed 127773.0)))
           (lo (- seed (* 127773.0 hi)))
           (test (- (* 16807.0 lo) (* 2836.0 hi))))
      (if (> test 0.0)
        (set! seed test)
        (set! seed (+ test 2147483647.0)))
      (/ seed 2147483647.0)))
  obj)
```

疑似乱数を生成する方法は、“乱数生成系で良質なものはほとんどない”, (西村恕彦訳, bit, Vol.25 No.4, 共立出版, 1993 年 4 月) で紹介された方法を使っています。疑似乱数生成オブジェクトは 0 以上 1 未満の値を返すので、手続き `make-random-vector` は疑似乱数生成オブジェクトの返す値を使って、0 から 999 までの整数の疑似乱数を作っています。

ではこれらを試してみましょう。

```
> (define rand (make-random)
#<unspecified>
> (make-random-vector 10 rand)
#(0 131 755 458 532 218 47 678 679 934)
> (make-random-vector 10 rand)
#(383 519 830 34 53 529 671 7 383 66)
```

次にこれを使って、2 つの整列プログラムの実行にかかる時間を測定します。測定を楽にするために、以下の手続き `test-sort` を使います。この手続きは、第 1 引数の整列手続きに、データの大きさが第 2 引数 `size` であるデータに対して実行手続きを第 3 引数 `times` 回の実行し、実行 1 回にかかった平均時間をミリ秒単位で返します。

```
(define *rand* (make-random))
(define (test-sort times sorting-method size)
  (define (get-time)
    (/ (get-internal-run-time) internal-time-units-per-second))
  (do ((start (get-time))
       (i 1 (+ i 1)))
      ((> i times)
       (inexact->exact (* 1000 (/ (- (get-time) start) times))))
    (sorting-method (make-random-vector size *rand*))))
```

手続き get-internal-run-time はプログラム実行の経過時間を得る手続きで、変数 internal-time-units-
だけ値が増えれば 1 秒経過したことを表し、これらはいずれも Scheme 処理系 SCM で特有の手続きや変数です。他の Scheme 処理系でも同様な手続きが用意されている可能性がありますので、使用している処理系のマニュアルを参照して、処理系に合わせてプログラムを修正してください。もし時刻データを返す手続きがなければ自動的な時間測定は不可能で、手作業で測定することになります。

これを実行すると、以下のようになりました。

```
> (test-sort 10 bubble-sort! 10)
500
> (test-sort 10 quick-sort! 10)
317
```

それぞれ大きさ 10 のデータに対する整列を 10 回行ない、平均値をとったものが示されています。バブル法による手続きでは 500 ミリ秒、クイック法による手続きでは 317 ミリ秒かかっています。(当然これらの値は、使用するコンピューターによって違ってきます。)

これを色々な大きさのデータに対して行なった結果を、図 9.6に示します。実験は、Sun マイクロシステムズ社 SPARC Station ELC (主記憶 24MByte, SunOS 4.1.3) 上での SCM バージョン 4c4 で行ないました。この図を見て明らかですが、バブル法と比べてクイック法は圧倒的に速いことが分かります。

では、どうしてこのような違いが出てくるのかを、数学的な手法を使って調べてみましょう。(もしこれ以降の内容が難しければ、とりあえずは理解できなくても結構です。) 入力データの要素数を n とおき、要素数 n と手続きの実行時間がどのような関係にあるかを調べます。バブル法は、0 から $n-2$ までのそれぞれの i に対し、jを $i+1$ から $n-1$ まで変えながら第 i 要素と第 j要素を比較し、必要ならば要素の交換をしています。このため、

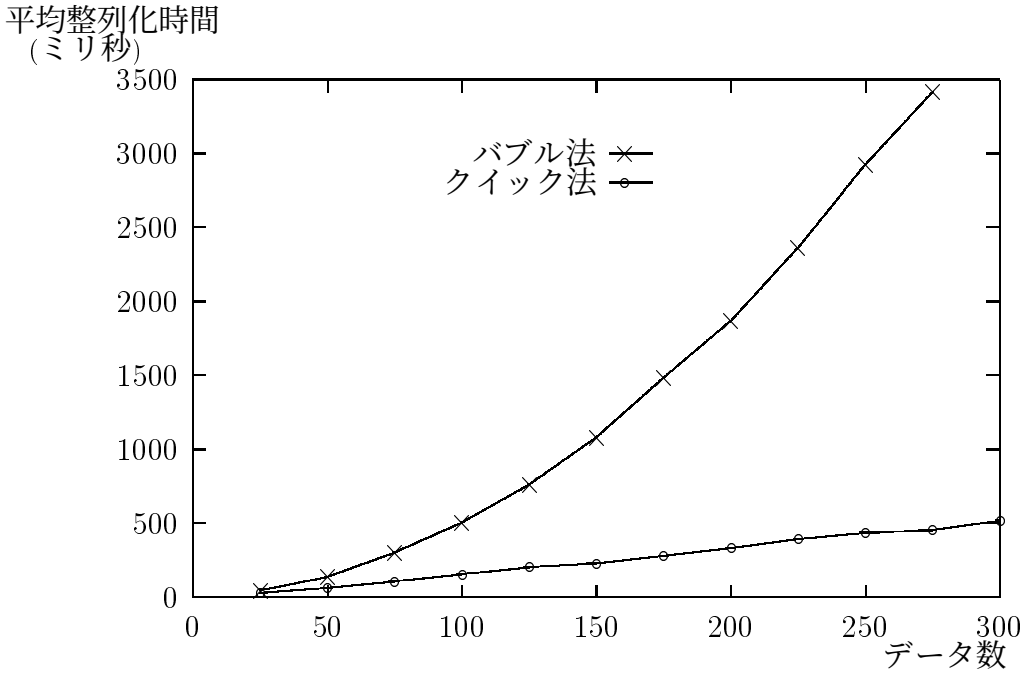


図 9.6: 整列に要する時間 (10 回の平均値)

比較を行なう回数を勘定すれば、

$$\begin{aligned}\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 &= \sum_{i=0}^{n-2}(n-i-1)\\ &= n(n-1)/2\end{aligned}$$

となり、要素数の 2 乗の半分程度の回数の比較をしていることが分かります。逆の見方をすれば、要素の交換回数は最大でも要素数の 2 乗の半分程度です。これらのことから、バブル法は要素数の 2 乗に比例した時間がかかる整列法であることが分かります。

クイック法の整列時間の見積もりは少し複雑です。まず分割段階にかかる時間ですが、これはデータ数 n に比例した時間しかかかりません。というのも分割段階では、どの要素も高々2 回しか他の要素と比較されないからです。よって分割段階では n に比例した時間がかかります。2 つの分割されたグループは、大きさがまちまちなのが普通ですが、ちょうど半分に分割されると仮定して考えます。すると要素数が n のときのクイック法での整列時間を $T(n)$ とおくと、次の漸化式が成立します。

$$\begin{aligned}T(n) &= \langle\ \text{分割段階での時間}\ \rangle\\ &\quad +\langle\ \text{左のグループの整列時間}\ \rangle + \langle\ \text{右のグループの整列時間}\ \rangle\\ &= \langle\ \text{分割段階での時間}\ \rangle + 2\langle\ \text{要素数}\ \ n/2\ \ \text{での整列時間}\ \rangle\\ &= \langle n\ \text{に比例した時間}\ \rangle + 2T(n/2)\end{aligned}$$

複雑なので $T(n)$ の求め方は省略しますが、これを解くと $T(n)$ は $n\log_2 n$ に比例したも

のとなります[9]。このことより、クイック法は $n \log_2 n$ に比例した時間で整列します。もし2つのグループが大きさ1と $n-1$ の2つに分割されるような入力のときは、$n \log_2 n$ では整列はできず、n^2 に比例した時間になってしまいます。

練習問題

1. バブル法のプログラムを改造し、データを小さい順にならべるようにしなさい。

2. クイック法のプログラムを改造し、データを小さい順にならべるようにしなさい。

3. 本実習で出てきた整列は、数の整列でした。文字列の整列をする手続き `string-sort!` を作りなさい。

4. 数を絶対値の大きい順に整列する手続き `absolute-sort!` を作りなさい。

5. $(x\ \ y)$ という形の、2つの数がリストになったものを整列する手続き `num2-sort!` を作りなさい。ただし、 $x_1 < x_2$ または $(x_1 = x_2$ かつ $y_1 < y_2)$ であるときに $(x_1\ \ y_1) < (x_2\ \ y_2)$ であると大小関係を定めるものとします。たとえば、`(7 4)<(11 0)` であり、また `(1 2)<(1 3)` です。

6. 上の3つの問題では、いろいろな形のデータの整列手続きを作りましたが、データ型ごとに整列手続きを作成するのは面倒です。そこで、整列手続きに大小の比較をする手続きを引数として渡すことを考えます。すると、整列手続きの実現で、引数に与えられるデータの型と比較方法を分離でき、大小比較の手続きさえ作れば整列手続きを利用できます。この考えに基づいた整列手続きを作りなさい。またその整列手続きの利用例として、上の問題5を解いてみなさい。

7. * 次は大小比較手続きを引数として渡すのではなく、データそのものが大小の比較方法を知っているようにしなさい。すなわちデータにメッセージとして比較指令と比較対象を送り、そのメッセージに対する返答として大小関係が返されるようにしておき、それに従って整列するようにしなさい。

8. 要素数 n の入力データに対して、バブル法を使った整列手続きでの整列時間を An^2、クイック法を使った整列手続きでの整列時間を $Bn \log_2 n$ とおきます。ここで A と B はある定数です。図9.6から、定数 A と B を求めなさい。求めた A と B から、要素数が1万のときの整列時間を予想しなさい。さらに、要素数が10万のとき、100万のときはどうですか?

9. 読者の使用している計算機で、バブル法とクイック法の整列時間を測定し、要素数との関係をグラフにして示しなさい。

[9] $\log_2 n$ は2を底とする n の対数です。$2^d = n$ のとき、$d = \log_2 n$ と表せます。

10. 選択法による整列をする Scheme 手続き `selection-sort!` を作りなさい。

11. バブル法はクイック法と比べて単純な整列法であるぶん、要素数の少ない場合はより早く整列が終る可能性があります。読者の使っている計算機では、要素数がいくらまでならバブル法がクイック法よりも高速であるか調べてみなさい。

12. * データを大きい順にならべるクイック法の手続きを考えます。入力データ数を n とし、入力データが小さい順にならんでいるときに、整列時間と n の関係を求めなさい。

13. 数がいくつか書き込まれているファイルを読んで整列し、結果を別のファイルに書き出す手続き `file-sort!` を作りなさい。ただしこの手続きは (`file-sort!` *infile outfile*) として呼び出されるものとします。入力となるファイル *infile* には、たとえば

    ```
    195 22 89
    123
    345
    ```

    といった形式で数が書かれているものとします。出力ファイルへは、1 行に数がひとつ書かれているものとします。

    ```
    345
    195
    123
    89
    22
    ```

14. * 9.2の実習 10を再び考えます。各科目および総合成績それぞれについて、スコア順に整列し、順位、スコア、氏名をファイルに書き出す手続きを作りなさい。

レポート

1. 本実習で紹介した整列法以外にも、挿入法、ヒープ法、シェル法などがあります。これらはどのような整列法であるかを調べ、特徴、欠点を述べなさい。またこれらの整列法に基づく整列手続きを作りなさい。

2. 比較による整列では、$n \log n$ に比例した時間よりも早い整列はできません。なぜそうなのかを調べ、説明しなさい。

参考文献

- 石畑 清, “アルゴリズムとデータ構造”, 岩波書店, 1989 年.
- S. K. Park, K. W. Miller, (西村 恕彦 訳), “乱数生成系で良質なものはほとんどない”, bit, Vol.25 No.4, pp. 19–27, 共立出版, 1993 年 4 月, および Vol.25 No.7, pp.12–20, 1993 年 5 月.

9.4 住所録の製作 その 2: ファイル版

本実習では、もうひとつの住所録プログラムを作ります。ここで作成する手続き群は、5.3 で作成したものと同一の機能を持ちながら、ディスク上のファイルに記録してあるデータを参照するようにします。5.3 での住所録データは、プログラムの中に書いてありました。そのために、新しいデータを追加するたびに、プログラムファイルを書き換えないといけませんでした。ファイルを利用することでデータとプログラムを分離し、データのファイルだけを書き換えるようにできます。

9.4.1 データファイル

まずさいしょに、データファイルの書式を決めます。こ k どえは単純に（〈名前〉〈電話番号〉）のように、S 式の形で書くことにします。こうすると、手続き read を使うことで、一度に名前と電話番号をリストの形で読み出せるからです。

以下に、いくつかのデータを含むデータファイル ADDRESS.DAT の例を示します。(これは 5.3 で出てきたものと同一のデータです。)

```
;; DATA FILE FOR ADDRESS BOOK
;    NAME         PHONE NO.
   (Chisato     012-345-6789 )
   (Narimi      98-7654      )
   (Mayuko      12-3456      )
   (Misako      999-9999     )
; end
```

このファイルには、; によるコメントがあります。手続き read は、Scheme プログラムの読み込みのとき同じように、コメント部分は読み飛ばすのを思い出して下さい。5.3に出てきたデータには引用符が使われていましたが、このデータファイルには引用符がないことに注意して下さい。5.3 のデータはプログラム中に書かれていたために、そのままでは式とみなされます。式として評価されてしまわないようにするために、引用符を使う必要がありました。ところが、データファイルを読む時はそのようなことはないので、引用符はいりません。

9.4.2 設計

5.3で作ったオンメモリ版のアプリケーションインターフェースでは、すべてのデータをメモリ上に保持していました。この方法ではメモリ量を越えたデータを取り扱えません。これから作るファイル版のアプリケーションインターフェースは、データの一部だけをメ

モリに保持し、巨大なデータファイルも扱えるようにします。このような性質は実用的なプログラムには必要不可欠で、事実、多くのプログラムがそのようになっています。

以下にアプリケーションインターフェースをもう一度示します。

- (address-book)
  住所録を初期化します。

- (add-to-address-book 〈氏名〉〈電話番号〉)
  住所録に個人データを登録します。

- (open-address-book)
  住所録をオープンします。オープンにともない、注目点を住所録の先頭の個人データとします。

- (close-address-book)
  住所録をクローズします。

- (lookup-address)
  注目点の個人データを読み出します。注目点はそのままです。

- (next-address)
  注目点を次の個人データにします。

- (read-address)
  注目点の個人データを読み出し、注目点を次の個人データにします。

- (end-of-address-book?)
  注目点が住所録の終りかどうかを調べます。

このことから分かるように、住所録は最初から順々に読んでゆくようになっています。そのため、注目している場所のデータだけを保持すれば、それで十分そうなことが予想できると思います。そのため、住所録を読み進めるごとに、ファイルからデータを読み込むようにします。

簡単化のために、データファイルは ADDRESS.DAT というファイルに固定します。住所録のオープンの時に住所録のデータファイルをオープンし、そのポートを保持しておきます。データを読む時は、そのポートからデータを読むようにします。アプリケーションインターフェースに、現在の注目場所のデータを読むという手続きがあるので、そのデータはあらかじめ読み込んでおき、変数に保持しておく、というようにしておきます。

9.4.3 実現

以上の考えに従い、手続き群を作ります。まず最初は、使用する変数を決めます。

変数 *ADDRESS-BOOK-FILE* は、データファイルのファイル名を保持します。このように変数は使わず、ファイルをオープンする場所で直接ファイル名を書く方法もあります。しかしそれでは、あとからデータファイル名を変更する場合、あちこちを変更しなくてはいけなくなる可能性があります。一箇所に書いておけばそこを変更するだけで済むので、プログラムの保守が容易になります。

```
;;;---------------------------------------------------------------
;;; 住所録のデータファイル名
(define *ADDRESS-BOOK-FILE*  "ADDRESS.DAT")
```

住所録の状態を保持する、3 つの変数を用意します。これらはアプリケーションインターフェース内部のもので、アプリケーションプログラムはこれら変数を参照すべきではありません。次のような変数を用意します。

- *ADDRESS-BOOK-PORT* — データファイルのポートを保持します。
- *ADDRESS-BOOK-DATA* — 注目点のデータを保持します。
- *ADDRESS-BOOK-OPENED* — 住所録がオープンされているときに真 #t を、そうでないときは偽 #f を保持します。オープンされてもいないのにデータを読もうとする場合を検出するのに使います。

```
;;;---------------------------------------------------------------
;;; 住所録の状態を保持する変数群（非公開）
(define *ADDRESS-BOOK-PORT*   #f)  ; 住所録ファイルに対するポート
(define *ADDRESS-BOOK-DATA*   #f)  ; 注目点データ
(define *ADDRESS-BOOK-OPENED* #f)  ; オープンされているかどうか
```

まず最初に、データファイルから個人データを読み込むための手続き read-data-item を作ります。これは単にデータファイルのポートから、手続き read を使って S 式をひとつ読み込むだけです。読み込んだデータは、変数 *ADDRESS-BOOK-DATA* に保持します。

```
;;; データファイルから個人データ１つを読む
(define (read-data-item)
  (set! *ADDRESS-BOOK-DATA* (read *ADDRESS-BOOK-PORT*)))
```

住所録のオープン手続き open-address-book は、次のようになります。まず最初に *ADDRESS-BOOK-OPENED* の値を#f にし、住所録がクローズされているものとします。次

にデータファイルをオープンし、そのポートを変数 *ADDRESS-BOOK-PORT* に保持します。もしデータファイルがなければエラーになり、実行が中断されます。もしオープンに成功すれば、変数 *ADDRESS-BOOK-OPENED* の値を #t にして、住所録がオープンされたことを記録します。そして最後に、一番最初の個人データを読んで、変数 *ADDRESS-BOOK-DATA* に保持します。

```
; 住所録のオープン
(define (open-address-book)
  (set! *ADDRESS-BOOK-OPENED* #f)
  (set! *ADDRESS-BOOK-PORT* (open-input-file *ADDRESS-BOOK-FILE*))
  (set! *ADDRESS-BOOK-OPENED* #t)
  (set! *ADDRESS-BOOK-DATA* (read-data-item))
  #t)
```

クローズは簡単で、以下の通りです。住所録がオープンされていなければ、エラーにします。

```
; 住所録のクローズ
(define (close-address-book)
  (if *ADDRESS-BOOK-OPENED*
    (close-input-port *ADDRESS-BOOK-PORT*)
    (error:not-opened))
  (set! *ADDRESS-BOOK-OPENED* #f)
  #t)
```

個人データを読む手続き read-address では、変数 *ADDRESS-BOOK-DATA* に保持されている注目点のデータを返します。しかし、注目点の移動をしなければならず、それに伴って変数 *ADDRESS-BOOK-DATA* の値も更新しなければいけません。そのため、いったん変数 *ADDRESS-BOOK-DATA* の値を一時的な変数 value に保持し、次の個人データを読み込んで変数 *ADDRESS-BOOK-DATA* の値も更新し、最後に変数 value の値を返すようにします。(住所録がオープンされていなければ、エラーにします。)

```
; 注目点の個人データを読む（注目点は次に移る）
(define (read-address)
  (if (not *ADDRESS-BOOK-OPENED*)
    (error:not-opened)
    (begin
      (let ((data *ADDRESS-BOOK-DATA*))
        (set! *ADDRESS-BOOK-DATA* (read-data-item))
        data))))
```

注目点の個人データを返す手続き lookup-address は、単に変数 *ADDRESS-BOOK-DATA* の値を返すだけです。(住所録がオープンされていなければ、エラーにします。)

```
; 注目点の個人データを読む（注目点を変えない）
(define (lookup-address)
  (if *ADDRESS-BOOK-OPENED*
    *ADDRESS-BOOK-DATA*
    (error:not-opened)))
```

手続き next-address は、次の個人データに注目点を移す手続きです。これは手続き read-address を呼び出すことで、注目点が次の個人データに移り、しかも変数 *ADDRESS-BOOK-DATA* の値がその個人データとなります。このことを利用して、単に手続き read-address を呼び出すことで十分となります。

```
; 注目点を次の個人データにする
(define (next-address)
  (read-address))
```

注目点が住所録の終りかどうかを判定する手続き end-of-address-book? は次のようになります。変数 *ADDRESS-BOOK-DATA* は手続き read の返した値なので、もし注目点の個人データが住所録の最後のデータなら、その値はファイル終端子となっています。このことより、変数 *ADDRESS-BOOK-DATA*の値がファイル終端子かどうかを調べることで、住所録の最後かどうかを判定します。

```
; 注目点が住所録の終りかどうかを判定する
(define (end-of-address-book?)
  (if *ADDRESS-BOOK-OPENED*
    (eof-object? *ADDRESS-BOOK-DATA*)
    (error:not-opened)))
```

その他の手続き (error:opened, error:not-opened, make-address-item, get-name, get-phone) は、5.3 で作ったものと同じものを使います。

9.4.4 プログラムリスト

以上の手続き群をまとめると、以下のプログラムリストとなります。

```
;;;
;;; address.scm （住所録プログラム FILE VERSION）
;;;
```

```
;;;-----------------------------------------------------------------
;;; 住所録のデータファイル名
(define *ADDRESS-BOOK-FILE*  "ADDRESS.DAT")


;;;-----------------------------------------------------------------
;;; 住所録の状態を保持する変数群（非公開）
(define *ADDRESS-BOOK-PORT*   #f)  ; 住所録ファイルに対するポート
(define *ADDRESS-BOOK-DATA*   #f)  ; 注目点データ
(define *ADDRESS-BOOK-OPENED* #f)  ; オープンされているかどうか


;;;-----------------------------------------------------------------
;;; 住所録アクセスのための手続き群（公開）

; 住所録のオープン
(define (open-address-book)
  (set! *ADDRESS-BOOK-OPENED* #f)
  (set! *ADDRESS-BOOK-PORT* (open-input-file *ADDRESS-BOOK-FILE*))
  (set! *ADDRESS-BOOK-OPENED* #t)
  (set! *ADDRESS-BOOK-DATA* (read-data-item))
  #t)

; 住所録のクローズ
(define (close-address-book)
  (if *ADDRESS-BOOK-OPENED*
    (close-input-port *ADDRESS-BOOK-PORT*)
    (error:not-opened))
  (set! *ADDRESS-BOOK-OPENED* #f)
  #t)

; 注目点の個人データを読む（注目点は次に移る）
(define (read-address)
  (if (not *ADDRESS-BOOK-OPENED*)
    (error:not-opened)
    (begin
      (let ((data *ADDRESS-BOOK-DATA*))
(set! *ADDRESS-BOOK-DATA* (read-data-item))
data))))

; 注目点の個人データを読む（注目点を変えない）
(define (lookup-address)
  (if *ADDRESS-BOOK-OPENED*
    *ADDRESS-BOOK-DATA*
    (error:not-opened)))

; 注目点を次の個人データにする
(define (next-address)
```

```
  (read-address))

; 注目点が住所録の終りかどうかを判定する
(define (end-of-address-book?)
  (if *ADDRESS-BOOK-OPENED*
    (eof-object? *ADDRESS-BOOK-DATA*)
    (error:not-opened)))


;;;-----------------------------------------------------------------
;;; その他の手続き（非公開）

; エラーメッセージ関連の手続き群
(define (error:opened)
  (error "Already opened"))
(define (error:not-opened)
  (error "Not opened"))

;;; 個人データ関連の手続き
; 名前と電話番号から、個人データを作る
(define (make-address-item name phone)
  (list name phone))
; 個人データから、名前を取り出す
(define (get-name item)
  (car item))
; 個人データから、電話番号を取り出す
(define (get-phone item)
  (cadr item))

;;; データファイルから個人データ１つを読む
(define (read-data-item)
  (set! *ADDRESS-BOOK-DATA* (read *ADDRESS-BOOK-PORT*)))

;end
```

9.4.5 実行

本実習で作ったファイル版の住所録アプリケーションインターフェースは、5.3 で作ったものと同じ手続き名、同じ引数、同じ意味合いを持っています。そのため、5.3で作ったアプリケーションプログラム phone が一切の変更なしに、そのまま使えます。

```
  (define (phone name)
    (let ((phone-number #f))
      (open-address-book)
      (set! phone-number (search-phone name))
```

```
    (close-address-book)
    phone-number))
(define (search-phone name)
  (if (end-of-address-book?)      — 住所録の終りか?
    "Not found"                   — 終りなら "Not Found"を返す
    (let ((item (read-address)))  — 住所録データをひとつ読む
      (if (equal? name (get-name item))
        (get-phone item)          — 名前が一致
        (search-phone name)))))   — 次を調べる
```

これを試しに実行してみましょう。

```
> (phone 'Chisato)      — 千里さんの電話番号は?
012-345-6789
> (phone 'Mayuko)       — 万由子さんの電話番号は?
12-3456
> (phone 'Ayako)        — 亜矢子さんの電話番号は?
"Not found"             — 住所録には載っていない
> (phone 'Misako)       — 美佐子さんの電話番号は?
999-9999
```

9.4.6 まとめ

本実習では、アプリケーションインターフェースの仕様を保ったまま、ファイルに記録されている住所データを参照する手続き群を作りました。アプリケーションインターフェースの仕様を厳密に守ったことにより、アプリケーションプログラムに何も影響を与えずに、手続き群をそっくり入れ換えることができました。

このように、モジュール間のインターフェースを正確に定義しそれを厳密に守ることで、ソフトウエアを完全に独立した部分に分割することが可能となります。この手法を使うことで、モジュールごとに互いに影響を及ぼすことなく、独立して保守が可能となります。モジュールの中だけのことを考えれば済むので、モジュールを順次入れ換えて、システムを改良してゆくこともできます。開発のときには、ひとつの小規模なモジュールだけに注目すれば済むため、大きなプログラムも上手にモジュール分割をすることで開発が容易になります。

練習問題

1. 本実習で作った住所録でのそれぞれのデータの項目 (氏名と電話番号) は、Scheme の記号として取り扱っていました。これを文字列とするよう、プログラムを変更しな

さい。

2. データ項目の氏名を部分を変更し、姓と名を分けて別の項目としなさい。

3. 実習 5.3 の練習問題を解きなさい。アプリケーションプログラム作成の問題については、本実習でのアプリケーションインターフェースでも正しく動くことを確かめなさい。

4. ** 本実習では一種類のデータファイルしか使いませんでした。住所録ファイルのファイル名を指定できるように、機能の拡張を考えます。手続き `open-address-book` は無引数で呼び出す仕様となっています。これまでに書いたアプリケーションプログラムがそのまま使えるようにするには、アプリケーションインターフェースの仕様をどうすればよいかを設計しなさい。

5. * 上の問題で考え出した仕様に従い、アプリケーションインターフェースのプログラムを変更しなさい。

6. ** 本実習では、住所録ファイルのデータの書式をひとつ決めて、その書式でデータを記録していました。プログラムの機能の拡張に伴って、ファイルのデータ書式の変更は当然考えられます。いろいろな書式のデータをひとつのプログラムで取り扱うには、データファイルの中身がどの書式で書かれているかを、プログラムが分からないといけません。

   そこで、データファイルの始めの部分に書式を表す記号 (たとえば `FORMAT-1`) を書くことを考えます。まず最初に書式記号を読み、それが `FORMAT-1` かどうかを確認するよう、プログラムを変更しなさい。もし `FORMAT-1` でなければ取り扱えない書式なので、エラーとしなさい。

7. *** (上の問題の続き) もうひとつ別のデータの書式を決め、その書式に従ったデータファイルを作りなさい。そしてその書式のデータファイルを、もとの書式のデータファイルとまったく同じに取り扱えるよう、アプリケーションインターフェースを改造しなさい。データファイルの書式が何であるかをアプリケーションプログラムが知らなくても、住所録データにアクセスできるようにしなくてはいけません。(ファイルの始めにある書式記号を読み、それに従ってファイルアクセスに使う手続きを切替える必要があります。)

8. ** 何種類もの書式のデータファイルを取り扱うようになってくると、ファイルアクセスに使う手続きを選ぶのが大変です。そこで、「住所録オブジェクト」を設計し、データファイルへのアクセスや各項目へのアクセスを、メッセージによってオブジェクトに依頼するようにしなさい。

9. 住所録のデータファイルは人が手で入力するものなので、入力ミスはどうしても避けられません。そこで、簡単な入力ミスを自動に見つけ出す、データ検証をするプログラムを作りなさい。このプログラムは住所録を読んでおかしいな項目を見つけ、それを画面に表示しなさい。

この比の稽古には、ただ、指を差して人に笑わるるとも、それをば顧みず、内にては、聲の届かん調子にて、宵·暁の聲を使ひ、心中には、願力を起こして、一期の堺ここなりと、生涯にかけて能を捨てぬより外は、稽古あるべからず。ここにて捨つれば、そのまま能は止まるべし。

世阿弥 「風姿花伝」
昭和三十八年 岩波書店刊

第 10 章

Scheme 入門 (上級編)

10.1 制御構造 (その 3)

10.1.1 do による繰り返し実行

再帰の他にも、繰り返し実行をするための構文 do が用意されています[1]。次の手続き f は引数に与えられた整数 n に対し、$\sum_{i=1}^{n} i^2$ を計算します。

```
(define (f n)
  (do ((i 0 (+ i 1))
       (the-sum 0))
      ((> i n) the-sum)
    (set! the-sum (+ the-sum (* i i)))))
```

この例では 2 つの局所変数 i と the-sum を用意し、それらの初期値をともに 0 としています。繰り返しは (> i n) が真となると終了します。逆をいえば、(<= i n) が真である間、do の本体 (set! the-sum (+ the-sum (* i i))) が実行されます。本体の実行後に、局所変数 i は (+ i 1) の評価結果が代入されます。(the-sum には更新値がないために、何も行なわれません。) 以上が繰り返し実行され、ついには (> i n) が真となって繰り返しは終了します。そして the-sum の値が、do の評価結果として返されます。

特殊形式 do の構文は次の通りです。

△ (do ((⟨変数$_1$⟩ ⟨初期値$_1$⟩ ⟨次の値$_1$⟩)
  ...
  (⟨変数$_n$⟩ ⟨初期値$_n$⟩ ⟨次の値$_n$⟩))
  (⟨終了条件⟩ ⟨終了時の式$_1$⟩ ··· ⟨終了時の式$_n$⟩)
  ⟨式$_1$⟩ ··· ⟨式$_n$⟩)

[1] do は Scheme に必須な機能とは定められていません。そのため、処理系によっては使えない可能性もあります。

⟨ 変数$_1$⟩···⟨ 変数$_n$⟩は局所変数で、do の中だけで使えます。⟨ 変数$_i$⟩ の初期値は ⟨ 初期値$_i$⟩ で与えられます。次に ⟨ 終了条件 ⟩ が評価され、その結果が #f 以外なら ⟨ 式$_1$⟩ ···が順次評価されます。⟨ 式$_n$⟩ の評価が終了すると各 ⟨ 次の値$_1$⟩ が評価され、対応した ⟨ 変数$_1$⟩ にその値が代入されます。そして ⟨ 終了条件 ⟩ が評価して、その結果が #f 以外なら、先ほどと同様に ⟨ 式$_1$⟩ ···が順次評価されます。このようして、繰り返しが実行されます。

もし ⟨ 終了条件 ⟩ の評価結果が #f なら繰り返しは終了し、⟨ 終了時の式$_1$⟩ ···⟨ 終了時の式$_n$⟩ を順次評価して、⟨ 終了時の式$_n$⟩ の評価結果が do の値となります。

なお ⟨ 次の値$_i$⟩, ⟨ 終了時の式$_i$⟩, ⟨ 式 ⟩は省略することもできます。上で示した手続き f は、変数 the-sum に対する ⟨ 次の値 ⟩ が省略されています。

もうひとつ例を示します。次の手続き sum-of-list は、引数として与えられた数のリストに対し、要素の総和を計算します。

```
(define (sum-of-list s)
  (do ((the-sum 0)
       (rest s (cdr rest)))
      ((null? rest) the-sum)
      (set! the-sum (+ the-sum (car rest)))) )
```

以下に実行例を示します。

```
> (sum-of-list '(1 2 3 4))
10
> (sum-of-list '(2 3))
5
> (sum-of-list '())
0
```

10.1.2 for-each による繰り返し実行

一連の式を繰り返して実行する方法として再帰法と do を学びましたが、for-each で繰り返し実行をすることもできます。

for-each はリストで与えた各要素を引数にして、与えられた手続きを呼び出します。

```
> (define (proc s) (display s) (newline))
#<unspecified>
> (for-each proc '(a b c))
a
b
c
```

この例では、リスト (a b c) の各要素を手続き proc の引数として (左から順に) 呼び出します。すなわち

```
(begin
  (proc 'a)
  (proc 'b)
  (proc 'c))
```

と同じ動作をします。for-each の構文は、

◇ (for-each ⟨ 手続き ⟩
⟨ リスト$_1$⟩
⟨ リスト$_2$⟩
⋮
⟨ リスト$_n$⟩)

となっています。ここで ⟨ 手続き ⟩ は n 個の引数を取ることのできる手続きで[2]、⟨ リスト ⟩ はリストです。

最初に ⟨ リスト$_1$⟩ ⟨ リスト$_2$⟩ ... ⟨ リスト$_n$⟩それぞれの第 1 要素を引数として、手続き ⟨ 手続き ⟩ を呼び出します。次に、⟨ リスト$_1$⟩ ⟨ リスト$_2$⟩ ... ⟨ リスト$_n$⟩それぞれの第 2 要素を引数として、手続き ⟨ 手続き ⟩ を呼び出します。以上のことが繰り返されます。

別の例を見てみましょう。

```
(let ((v 0))
  (for-each (lambda (n) (set! v (+ v n)))
            '(1 3 5 7 9))
  v)
```

この例では、まず最初に局所変数 v を用意しています。その環境の中で、リスト (1 3 5 7 9) の各要素を順番に、手続き (lambda (n) (set! v (+ v n)) の引数にして呼び出すことで、リストの要素を v に加えています。そして最後に結果として v の値を返しています。この式の評価値は 25 となります。

このように for-each は、⟨ 手続き ⟩ が返す値よりもむしろ**副作用** (side effect) を目的として呼び出します。副作用とは、手続きを呼び出した結果、その手続きの内部にはない変数の値が書き換えられることをいいます。上の例では、変数 v は手続きの内部にはなく、手続きの外部にある変数です。そして set! で変数 v の値が変更され、for-each の終了後にその結果を利用していることが分かります。

⟨ リスト ⟩ が複数ある場合の例を見てみましょう。

[2] n 引数の手続きもしくは、+ のような任意個の引数を持つ手続きのことです。

```
(let ((v 0))
  (for-each (lambda (a b) (set! v (+ v (* a b))))
            '(10 20 30) '(1 2 3))
  v)
```

この例では $10 \cdot 1 + 20 \cdot 2 + 30 \cdot 3$ の計算をしていて、評価値は 140 となります。

10.1.3 map による繰り返し

map はリストの要素それぞれを手続きの引数にして呼び出し、その結果をリストにします。

```
> (map * '(1 2 3) '(10 20 30))
(10 40 90)
```

これは (list (* 1 10) (* 2 20) (* 3 30)) と同じ動作です。

map の構文は、

◇ (map ⟨ 手続き ⟩
⟨ リスト$_1$⟩
⟨ リスト$_2$⟩
⋮
⟨ リスト$_n$⟩)

となっています。

map は次のように実行されます。まず最初はそれぞれの ⟨ リスト ⟩ の第一要素を、⟨ 手続き ⟩の引数として呼び出します。次にそれぞれの ⟨ リスト ⟩ の第二要素を、⟨ 手続き ⟩ の引数として呼び出します。これを ⟨ リスト ⟩ の長さだけ繰り返します。map の返す値は ⟨ リスト ⟩ の第一要素をに対する⟨ 手続き ⟩ の返した値、⟨ リスト ⟩ の第二要素をに対する ⟨ 手続き ⟩ の返した値、···をリストにしたものです。

以下にいくつかの例を示します。

```
> (map car '((1 2 3) (a b c) (x y z)))
(1 a x)
```

これは (list (car '(1 2 3)) (car '(a b c)) (car '(x y z))) と同じ動作です。

```
> (map (lambda (a b c) (+ a b c))
       '(1 2 3) '(10 20 30) '(100 200 300))
(111 222 333)
> (let ((v 0))
```

```
    (map (lambda (n) (set! v (+ v n)))
         '(1 2 3 4))
    v)
10
```

10.1.4 aplly による手続き呼び出し

これまで学んだ手続きの呼び出しは、(⟨ 手続き ⟩ ⟨ 引数 ⟩ ···) の形をしていました。この方法では、たとえば

```
(define (calc op a b)
  (op a b))
```

といったように、⟨ 手続き ⟩ の部分 op を引数として渡し、手続き呼び出しをすることができます。

ですがこの方法で呼び出すことができる手続きは、引数の数が 2 でないといけません。引数の数が 3 の手続きや、引数の数が 4 の手続きなどを、統一的な方法で呼び出すことはできません。たとえば、

```
(define (calc op2 a b)
  (op a b))
(define (calc op3 a b c)
  (op3 a b c))
(define (calc op4 a b c d)
  (op4 a b c d))
```

のように、呼び出す手続きの持つ引数の数に応じて、「呼び出し手続き」を準備するのもいいですが、少々めんどうです。

手続き apply は、手続きの引数に関係なく、同一の形式で手続きに引数を与えて呼び出します。

◇ (apply ⟨ 手続き ⟩ ⟨ 引数 ⟩)
— ⟨ 手続き ⟩ は手続きで、⟨ 引数 ⟩ はリストです。⟨ 引数 ⟩ の第一要素、第二要素、··· それぞれを、⟨ 手続き ⟩の第一引数、第二引数、··· として ⟨ 手続き ⟩ を呼び出します。⟨ 手続き ⟩ の評価結果が apply の評価結果となります。

◇ (apply ⟨ 手続き ⟩ ⟨ 引数$_1$⟩ ···⟨ 引数リスト ⟩)
— (apply ⟨ 手続き ⟩(append (list ⟨ 引数$_1$⟩ ···) ⟨ 引数リスト ⟩)) と等価です。(この形式が使えるかどうかは処理系に依存します。)

例を見てみましょう。

```
> (apply cons (list 'kanga 'roo))
(kanga . roo)
> (apply number? (list 2001))
#t
> (apply number? (list 2001 2010))
                                    —  引数の数が違うとエラー
ERROR: apply: Wrong number of args
> (apply + (list 1 2 3 4 5))    —  +は任意の数の引数をとる
15
> (apply (lambda (f1 f2 x) (f1 (f2 x))) (list - sqrt 2))
-1.4142135623731
```

10.2 リスト操作関数

31ページ 3.4 では、リストについての基本的な概念といくつかの手続きを学びました。ここでは、より高度なリスト操作を学びます。

10.2.1 A リスト

A リスト (alist) とは、car 部が対であるようなリストのことです。alist とは、association list (連想リスト) の略です。 なぜ連想リストというかといえば、連想のキーとその連想値からなるリストとして使われるからです。

次の例をみてみましょう。

```
((yama kawa) (tsu- ka-) (uteba hibiku))
```

これは A リストの例で、3 つのキー yama, tsu-, uteba があります。それぞれの連想値は kawa, ka-, hibiku となっています。

あるキーから連想値を検索するための手続きとして、次の手続きが用意されています。

◇ (assq ⟨ 連想リスト ⟩ ⟨ 検索キー ⟩)
— ⟨ 連想リスト ⟩ の先頭から順に、⟨ 連想リスト ⟩ の中の連想キーと⟨ 検索キー ⟩ が等しいものを探します。もし一致するものがあればそのキーと連想値の組を返します。もし見つからなければ、偽 #f が返されます。⟨ 検索キー ⟩ と連想キーとの比較には、eq? を使います。

◇ (assv ⟨ 連想リスト ⟩ ⟨ 検索キー ⟩)
— assoq と同じですが、⟨ 検索キー ⟩ と連想キーとの比較に eqv? を使う点が違います。

◇ (assoc ⟨ 連想リスト ⟩ ⟨ 検索キー ⟩)
— assoq と同じですが、⟨ 検索キー ⟩ と連想キーとの比較に equal? を使う点が違います。

実行例を見てみましょう。

```
> (define alist1 '((yama kawa) (tsu- ka-) (uteba hibiku)))
#<unspecified>
> (assq 'yama alist1)
(yama kawa)
> (assq 'uteba alist1)
(uteba hibiku)
> (assq 'jiji alist1)
#f
> (define alist2 '((1 one) (2 two)))
#<unspecified>
> (cadr (assv 1 alist2))     —    連想値を取り出す
one
> (cadr (assv 2 alist2))
two
> (define alist3 '(((A) 1) ((B) 2) ((C) 3)))
#<unspecified>
> (assoc '(A) alist3)
((A) 1)
> (assoc 'B alist3)
#f
```

連想キーを項目、連想値を表の中の項目の値とみなせば、A リストを使って一種の「表」を実現することができます。しかも、A リストに変更をすることで、その表を実行時にどんどん変更してゆくことができます。cond や case でも表を実現することはできますが、実行時にどんどん表を変更してゆくことはできません。

10.2.2 リストのメンバーを調べる

あるリストの中に、あるものがメンバーとして含まれているかどうかを調べる、ということがよくあります。こういう目的のための手続きとして、次のものが用意されています。

◇ (memq ⟨ 検索キー ⟩ ⟨ リスト ⟩)
— ⟨ 検索キー ⟩ が⟨ リスト ⟩ のトップレベルに現れるかどうかを調べます。もし見つかれば、見つかった部分から残りのリストを返します。見つからなければ、偽 #f が

返されます。⟨ 検索キー ⟩ と ⟨ リスト ⟩ のトップレベルの要素との比較には、eq? を使います。

◇ (memv ⟨ 検索キー ⟩ ⟨ リスト ⟩)
— memq と同じですが、⟨ 検索キー ⟩ と ⟨ リスト ⟩ のトップレベルの要素との比較に eqv? を使う点が違います。

◇ (member ⟨ 検索キー ⟩ ⟨ リスト ⟩)
— memq と同じですが、⟨ 検索キー ⟩ と ⟨ リスト ⟩ のトップレベルの要素との比較に equal? を使う点が違います。

例を見てみましょう。

```
> (define friends '(pooh piglet owl rabbit tiger))
#<unspecified>
> (memq 'owl friends)
(owl rabbit tiger)
> (memq 'tiger friends)
(tiger)
> (memq 'alice friends)
#f
> (member (list 'pooh) '((pooh) (alice)))
((pooh) (alice))
> (member (list 'alice) '((pooh) (alice)))
((alice))
> (memv 11 '(1 2 3 5 7 11 13 17))
(11 13 17)
> (memv 9 '(1 2 3 5 7 11 13 17))
#f
```

10.2.3 対の書き換え

これまでに学んだ方法でリスト構造を変えるには、cons を使うしかありません。cons を使ってリストを伸ばすことはできますが、すでにあるリストを「変更」することはできません。

次の例を見てみましょう。まず変数 alist の値を ((p . 3.14) (e . 2.718)) とします。ここで、ドット対と小数点の違いに注意して下さい。(説明を簡単にするために、A リストの連想キーと連想値の組は⟨ 連想キー ⟩ . ⟨ 連想値 ⟩ としています。10.2.1 での説明と少し違うので注意して下さい。)

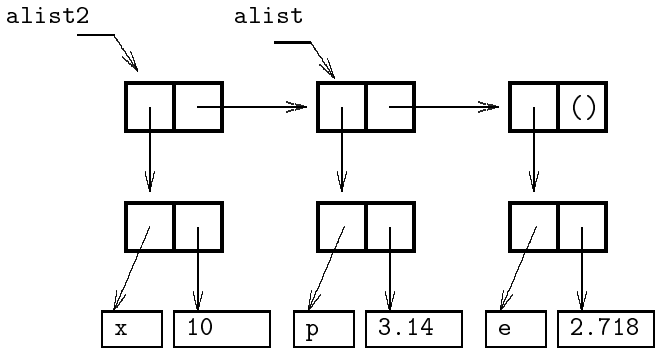


図 10.1: A リスト (x . 10) (p . 3.14) (e . 2.1718))

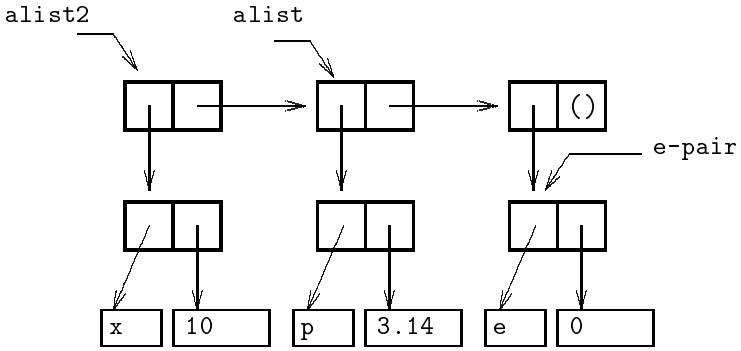


図 10.2: (set-cdr! e-pair 0) 後の A リスト

```
> (define alist (list (cons 'p 3.14) (cons 'e 2.718)))
#<unspecified>
> alist
((p . 3.14) (e . 2.718))
```

この A リストに、連想キーと連想値の組をあらたに付け加えるのは簡単です。

```
> (define alist2 (cons (cons 'x 10) alist))
#<unspecified>
> alist2
((x . 10) (p . 3.14) (e . 2.718))
> alist
((p . 3.14) (e . 2.718))
> (assoc x alist2)
(x . 10)
> (assoc x alist)
```

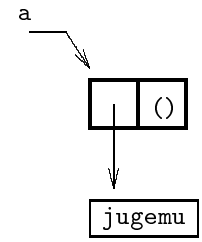


図 10.3: リスト（Jugemu）

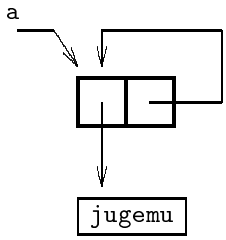


図 10.4: ループのある構造

```
(e . 2.718)
```

この様子を図 10.1に示します。

では、e の連想値を 0 にするにはどうすればいいでしょうか？連想キーが e 以外のものからなる A リストを作り、その A リストに（e . 0）を加えるという方法がまず考えられます。A リストがとても長い場合は、作業にとても時間がかかるという欠点があります。図 10.1での、（e . 2.718）を作っている対の cdr 部を書き換えることができれば、そのような問題は生じません。

このような目的のために、対の car 部や cdr 部を書き換える手続きが用意されています。

◇ (set-car! 〈対〉〈データ〉)
— 〈対〉の car 部を、〈データ〉に書き換えます。

◇ (set-cdr! 〈対〉〈データ〉)
— 〈対〉の cdr 部を、〈データ〉に書き換えます。

上の例をもう一度考えてみましょう。図 10.1のように変数 e-pair は、A リストの（e 2.718）を指しているものとします。

```
> (define e-pair (caddr alist2))
#<unspecified>
> e-pair
```

```
(e . 2.718)
```

ここで (set-cdr! e-pair 0) とすれば、図 10.2のように e の連想値は 0 に変更されます。

```
> (set-cdr e-pair 0)
#<unspecified>
> alist2
((x . 10) (p . 3.14) (e . 0))
> alist
((p . 3.14) (e . 0))
```

こんどは e-pair の car 部を書き換えてみましょう。

```
> (set-car e-pair 'zero)
#<unspecified>
> alist2
((x . 10) (p . 3.14) (zero . 0))
> alist
((p . 3.14) (zero . 0))
```

手続き set-car!, set-cdr! は、対の car 部、cdr 部それぞれを書き換えてしまうので、手続き cons では作れない構造をしたデータも作ることができます。まず変数 a の値を (jugemu) とします。

```
> (define a (cons 'jugemu '()))
#<unspecified>
> a
(jugemu)
```

この様子を図 10.3に示します。次に a の指す対の cdr 部を、a の指す対に書き換えます。

```
> (set-cdr! a a)
#<unspecified>
```

すると変数 a の値となっている対は、図 10.4 に示す構造になります。この対に対して何度 cdr を適用しても、永遠にリストの終りにたどり着きません。また画面に表示させようとしても、多くの処理系では

```
> a
jugemu jugemu jugemu jugemu jugemu jugemu jugemu jugemu
jugemu jugemu jugemu jugemu jugemu jugemu jugemu jugemu
jugemu jugemu jugemu jugemu jugemu jugemu jugemu jugemu
  ...
```

と永遠に表示が終りません[3]。(処理系によっては、ループになっていることを判定して、特別な形式で表示するものもあります。)

このように set-car!, set-cdr! は、使い方を間違えると大変面倒ですので、十分に注意して使いましょう。

10.3 準引用

引用符 ’ を使えば、その直後に書いた式は評価されないことをこれまでに学びました。引用符 ’ に似ていますが、一部だけを評価する引用符モドキが用意されているので紹介します。完全な引用とは違ってそれに準じたものであることから、**準引用** (quasiquotations) と呼ばれます。

- ‘⟨データ⟩(または (quasiquote ⟨データ⟩))
  — ⟨データ⟩ を評価せずそのままが値となりますが、⟨データ⟩ の中に , または ,@ (あるいはこれらの非省略形式) があれば、それらに従います。, や ,@ がないときは ’ と同じです。‘ は**バッククオート** (back quote) と読みます。

- ,⟨データ$_i$⟩(または (unquote ⟨データ$_i$⟩))
  — ‘⟨データ⟩ の中で ,⟨データ$_i$⟩ が現れると、‘⟨データ$_i$⟩ を評価したものと置き換えます。(, は ‘ の中でのみ使われます。)

- ,@⟨データ$_i$⟩(または (unquote-splicing ⟨データ$_i$⟩))
  — ‘⟨データ⟩ の中で ,@⟨データ$_i$⟩ が現れると、まず ⟨データ$_i$⟩ を評価します。(その結果はリストでないといけません。) そして評価結果のトップレベルの要素を ,@⟨データ$_i$⟩に置き換えます。これはちょうどリストの括弧と取り払った置き換えになります。(,@ は ‘ の中でのみ使われます。)

例を見てみましょう。

```
> 'pooh                          —  ただのクオート
pooh
> `(pooh piglet rabbit)          —  クオートと同じ使い方
(pooh piglet rabbit)
> (define x 'honey)
#<unspecified>
> (list 'pooh 'loves x)          —  クオートだけいいのですが、、、
(pooh loves honey)
```

[3] このようになったら、強制的に実行を中断しないといけません。多くの処理系では、C-g または C-c で中断できます。(Mule 上で Scheme を使っている場合は、C-c を 2 回押す必要があります。)

```
> `(pooh loves ,x)          —   バッククオートを使うと簡単です
(pooh loves honey)
> (define y '(piglet rabbit owl))
#<unspecified>
> `(,y are friends of pooh)
((piglet rabbit owl) are friends of pooh)
> `(,@y are friends of pooh)
(piglet rabbit owl are friends of pooh)
```

, や ,@ は、リストのトップレベルにある必要はありません。リストに限らず、ベクトルに対しても使えます。

```
> (define z 2)
#<unspecified>
> `((1 2 3) (,(* 1 z) ,(* 2 z) ,(* 3 z)))
((1 2 3) (2 4 6))
> `#(1 2 3 ,(* 1 z) ,(* 2 z) ,(* 3 z))
#(1 2 3 2 4 6)
```

` を入れ子にして使うこともできます。` が入れ子になるごとに、入れ子の深さが 1 つ深くなります。そのなかで , が現れると、入れ子の深さが 1 つ浅くなります。入れ子の深さが 0 の , のデータが、置き換えられます。

```
> `(2001 `(2010 x))
(2001 (quasiquote (2010 x)))
                        —   (2001 `(2010 x)) の非省略形
> `(2001 `(2010 ,x))
(2001 (quasiquote (2010 (unquote x))))
                        —   (2001 `(2010 ,x)) の非省略形
> `(2001 `(2010 ,,x))
(2001 (quasiquote (2010 (unquote honey))))
                        —   (2001 `(2010 ,honey)) の非省略形
```

上の例での一番最後の実行例では x の入れ子の深さは 0 なので、,x の部分は x の値 honey に置き換えられています。

10.4 継続

継続 (continuation) とは、プログラム実行の途中の点のことです。プログラム実行がどこに続いているかに関する情報、変数の状態などが継続を構成しています。Scheme には基本的なデータ型として継続が用意されています。継続を使うことでプログラムの実行を

いろいろな形で制御でき、非常に奥が深いものです。複雑な使い方は省略し、ここでは簡単な使い方を学びます。

たとえば式 ((+ 1 (+ 2 3)) の評価を考えてみましょう。部分式 (+ 2 3) の評価がすんだ後は、その結果と 3 を加えます。このように「X が済んだ後に Y をする」というときは、X が済んだ後の実行は Y に続く、ということにほかなりません。

手続き call-with-current-continuation は、継続を引数として手続きを呼び出します。

◇ (call-with-current-continuation ⟨ 手続き ⟩)
— ⟨ 手続き ⟩ は 1 引数の手続きです。この式が評価されると、継続を引数として ⟨ 手続き ⟩ を呼び出します。

⟨ 手続き ⟩ の引数として与えられる継続 C は、手続き呼び出しと同じように (C ⟨ 引数 ⟩) として、何度でも継続呼び出しの「続き」に実行を移すことができます。もし ⟨ 手続き ⟩ の中で (C ⟨ 引数 ⟩) が評価されれば、(call-with-current-continuation ⟨ 手続き ⟩) の値は ⟨ 引数 ⟩の評価結果になります。もし (C ⟨ 引数 ⟩) が評価されなければ、呼び出した ⟨ 手続き ⟩ の評価値を (call-with-current-continuation ⟨ 手続き ⟩) の評価値となります。

手続き procedure? は、引数に継続が与えられたときも真 #t を返します。

10.4.1 非局所的な脱出

継続を手続きと同じように「呼び出し」て、継続呼び出しの「続き」に戻ることができます。このことを利用して、入れ子になって呼び出された手続きから、呼び出し元に一気に戻ることができます。

例を見てみましょう。

```
(define a 3)
(define (f cont)
  (if (= a 0)
    (cont #f)
    (set! a (- a 1)))
  (display "a=")
  (display a)
  (newline)
  a)
```

手続き f をみると、変数 a の値が 0 なら継続の続きに戻るようになっています。もし a の値が 0 以外なら値を 1 つ減らし、a の値を表示、改行して f の実行を終っています。f は a の値を返していることにも注意して下さい。

これらの定義した後、(call-with-current-continuation f) の評価を考えてみましょう。これは手続き f を継続を引数として呼び出しています。最初 a の値は 3 なので、a の値が 1 つ減らされ、a の値が表示されます。そして f の評価結果である a の値が、(call-with-current-continuation f) の値となります。これを試してみましょう。

```
> (call-with-current-continuation f)
a=2   — display による表示
2     — (call-with-current-continuation f) が返した値
> (call-with-current-continuation f)
a=1
1
> (call-with-current-continuation f)
a=0
0
```

さらにやってみましょう。今度は a の値が 0 なので、手続き f の中の式 (cont #f) が評価されます。

```
> (call-with-current-continuation f)
#f    — (call-with-current-continuation f) が返した値
```

ここで注意してほしいのは、a=という行がないことです。これは (cont #f) の評価によって、継続呼び出しの「続き」に制御が移ったためです。(cont #f) は手続き呼び出しの形をしていますが、手続き呼び出しと違い、呼び出し元には戻ってきません。

もう少し複雑な例を見てみましょう。こんどは f の中で呼び出された別の手続きの中で、「続き」に制御を移す例です。

```
(define b 3)
(define (g cont)
  (if (= b 0)
    (h cont)
    (set! b (- b 1)))
  (display "b=")
  (display b)
  (newline)
  b)
(define (h cont)
  (cont #f))
```

これも前の例と同じ動作をします。

```
> (call-with-current-continuation g)
b=2   — display による表示
2     — (call-with-current-continuation g) が返した値
> (call-with-current-continuation g)
b=1
1
> (call-with-current-continuation g)
b=0
0
> (call-with-current-continuation g)
#f
```

このように、さらに呼び出された手続きでも、途中で式の評価を終了して呼び出し元に戻る、ということができます。このことは**非局所的な脱出** (non-local exit) とも呼ばれます。

継続を使わずに手続きの実行を終るには、その手続きの最後に制御がたどり着かないといけませんでした。継続を使うことで手続きの途中でも、さらには何重にも手続き呼び出しが入れ子になったときでも、呼び出し元に実行を戻すことができます。上で示した継続の使い方は、C 言語にある関数の setjmp, longjmp と同じ使い方です。

10.4.2 実行の中断と再開

継続は「脱出」だけでなく、「再突入」することもできます[4]。継続を保存して実行を中断し、別の仕事をした後に中断したときの継続を呼び出し、実行を再開することで「再突入」ができます。(中断されたときの局所変数も、ちゃんと保存されています。)

次の例は、実行している手続きの途中で中断をし、後からその実行に戻るための手続き群です。

```
(define *cont-top*  #f)
(define *cont-proc* #f)

(define (start p)
  (define (start2 c)
    (set! *cont-top* c)    — start の呼び出し元の継続を保存
    (p))
  (call-with-current-continuation start2))
```

[4] C 言語での setjmp, longjmp 関数ではスタックの浅くなる場合にしか longjmp を使えません。いっぽう Scheme の継続を使うと、スタックの深くなる場合でも実行の継続ができます。これは継続にはスタックの内容も保持しているからです。(「再突入」ができない、制限された継続しか用意していない Scheme 処理系もあるので注意して下さい。)

```
(define (break)
  (define (break2 c)
    (set! *cont-proc* c)   — 中断する手続きの継続を保存
    (*cont-top* "BREAK"))
  (call-with-current-continuation break2))
(define (resume)
  (*cont-proc* #t))   — 中断点に実行を移す (引数は何でもよい)
```

手続き start は、引数に与えられた手続きを呼び出します。この手続きを $proc$ とします。$proc$ の実行の途中で手続き break が呼ばれると $proc$ の実行は一時中断され、start の呼び出し元に実行が戻ります。その後で手続き resume を実行すると、$proc$ の実行が中断された場所から再び実行が再開されます。
$proc$ の例として、次のものを試してみましょう。

```
(define (winnie)
  (display "POOH") (newline)
  (break)
  (display "BEAR") (newline))
```

では、実行してみましょう。

```
> (start winnie)
POOH
"BREAK"
> 
```

手続き winnie が中断され、プロンプトが出ています。ここで別のことをやった後、winnie の実行を再開します。

```
> (list "Alice" "Hatter")   — 別のこと 1
("Alice" "Hatter")
> (+ 2001 9)                — 別のこと 2
2010
> (resume)          — 実行の再開
BEAR
#<unspecified>
```

別の $proc$ の例として、次のものを試してみましょう。

```
(define (one-day)
  (display "BREAKFAST") (newline)
```

```
  (break)
  (display "LUNCH") (newline)
  (break)
  (display "TEA") (newline)
  (break)
  (display "DINNER") (newline))
```

実行すると、次のようになります。

```
> (start one-day)
BREAKFAST
"BREAK"
> (resume)
LUNCH
"BREAK"
> (resume)
TEA
"BREAK"
> (resume)
DINNER
#<unspecified>
```

このように、継続を使うことで実行の中断と再開ができます。いくつかの手続きを交互に切替えて実行することにより、いくつものを手続きが同時に実行されているように見せかける、並行処理を簡単に実現できます。

「わしのやる彫金は、ほかの彫金と違つて、片切彫といふのでな。一たい彫金といふものは、金で金を截る術で、なまやさしい藝ではないな。精神の要るもので、毎日どぜうでも食わにや全く續くことではない」

岡本かの子 「家霊」
『鮨』収録 昭和十六年 改造社刊

第 11 章
プログラミング実習 3

本章では以下に示す実習テーマを通じて、より高度なプログラミングを学びます。

1. **円周率の計算**
   — ちょっとした息抜きとして、円周率 $\pi = 3.141592\cdots$ を計算するプログラムを作ります。ここで作るプログラムは、主記憶と計算時間の許す限り任意の精度で計算します。そして実際に小数点以下 1000 桁の計算をします。

2. **プログラミング言語処理系の基礎 — 式の計算**
   — 単純な式を解釈し、計算するプログラムを作ります。これは簡単な言語処理系で、実習「Scheme 処理系の作成」の基礎となる部分です。この実習では、プログラミング言語処理系の基礎を学びます。

11.1 円周率の計算

11.1.1 円周率の基礎概念と単純な計算法

図 11.1.1のように円の直径を d とすれば、その円周 (円の周囲の長さ) L は、

$$L = \pi d$$

という関係があります。この式は、円周は直径 d に比例し、その比例定数が π であることを表しています。この比例定数 π のことを、**円周率**といいます。本実習では、円周率を超高精度で計算するプログラムを作ります。いきなり超高精度で計算するプログラムを作るのは大変なので、簡単なプログラムから段階的に複雑なものにしてゆきます。

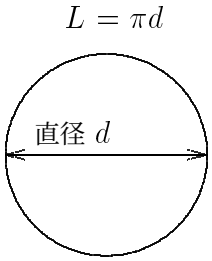


図 11.1: 円

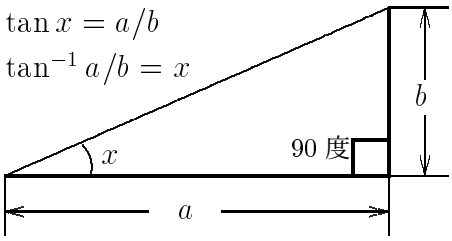


図 11.2: tan と $\tan^{-1}$

円周率を求める方法ですが、最初は逆正接関数 $\tan^{-1}$ を使った方法を使います。逆正接関数 $\tan^{-1}$ とは、図 11.1.1 のようにして定義される関数です。($\tan^{-1}$ はアークタンジェントと読みます。)

角度を表すのに普段では「度」を使いますが、数学では「度」の代わりに「ラジアン」を使います。360 度は 2πラジアンと等しいという関係にあります。以降では角度を表すのにラジアンを使います。

では図 11.1.1において、$x = \pi/4 (= 45$ 度$)$ のときを考えてみましょう。このとき $a = b$ なので、$\tan(\pi/4) = a/b = 1$ という式が成り立ちます。このことより $\tan^{-1}(1) = \pi/4$ となります。以上より

$$\pi = 4 \tan^{-1}(1)$$

となります。

$\tan^{-1}$ を計算する Scheme 手続き `atan` があるので、試してみましょう。

```
> (* 4 (atan 1))
3.1415926535898
```

本当の π は $3.141592653589793\cdots$ なので、小数点以下 14 桁目が四捨五入された結果が得られています。

11.1.2 円周率の計算法

実は $\tan^{-1}$ による計算方法は時間がかかる方法なので、別の計算公式を使って π の計算をするのが普通です。以下によく使われる円周率の公式を示します。(これらの公式の導き方は本書の内容を越えるので、興味のある人は円周率に関する本を調べて下さい。)

- マチンの公式

$$\pi/4 = 4\tan^{-1}(\frac{1}{5}) - \tan^{-1}(\frac{1}{239})$$

- オイラーの公式

$$\pi = 20\tan^{-1}(\frac{1}{7}) - 8\tan^{-1}(\frac{3}{79})$$

- ガウスの公式

$$\pi/4 = 12\tan^{-1}(\frac{1}{18}) + 8\tan^{-1}(\frac{1}{57}) - 5\tan^{-1}(\frac{1}{239})$$

これらで本当に計算できるか、試してみましょう。

```
> (* 4 (- (* 4 (atan (/ 1 5))) (atan (/ 1 239))))
3.1415926535898
> (+ (* 20 (atan (/ 1 7))) (* 8 (atan (/ 3 79))))
3.1415926535898
> (* 4 (+ (* 12 (atan (/ 1 18))) (* 8 (atan (/ 1 57)))
          (- (* 5 (atan (/ 1 239))))))
3.1415926535898
```

いずれも円周率が計算できていることが分かります。

円周率を計算する前に、逆正接関数 $\tan^{-1}$ の計算方法を説明します。Scheme データとしての実数に対する逆正接関数を計算する手続き `atan` はありますが、計算精度に限りがあります。そのため任意の桁数で計算をするには、高精度で計算する手続きを自分で作らないといけません。逆正接関数の計算方法として、次の 2 つが知られています[1]。(このように項を順次加えて関数の値を求めるものを**級数**といいます。)

[1] これらの計算方法の導き方は本書の内容を越えるので、説明はしません。興味のある人は数学の本を調べて下さい。

- オイラー級数

$$\begin{aligned}\tan^{-1} x &= \frac{1}{1+x^2}\left\{1+\sum_{i=1}^{\infty}\left\{\frac{\prod_{m=1}^{i}(2m)}{\prod_{m=1}^{i}(2m+1)}\cdot\left(\frac{x^2}{1+x^2}\right)^i\right\}\right\} \\ &= \frac{1}{1+x^2}\left\{1 + \frac{2}{3}\cdot\frac{x^2}{1+x^2} + \frac{2\cdot 4}{3\cdot 5}\cdot\left(\frac{x^2}{1+x^2}\right)^2 \right. \\ &\quad \left. + \frac{2\cdot 4\cdot 6}{3\cdot 5\cdot 7}\cdot\left(\frac{x^2}{1+x^2}\right)^3 + \cdots\right\}\end{aligned}$$

- クレゴリー級数

$$\begin{aligned}\tan^{-1} x &= \sum_{i=1}^{\infty}\frac{(-1)^{i-1}}{2i-1}x^{2i-1} \\ &= x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \cdots\end{aligned}$$

以下にオイラー級数による計算方法を Scheme で実現した手続き arctan を示します。

```
;;; ARCTAN  -  オイラー級数による atan(x)
(define (arctan x)

  ;; LOOP - 項の和を繰り返し加えてゆく
  ;  第 k 項と第 k 項までの和の比率が e 以下になるまで繰り返す
  ;  sum  - 第 k-1 項までの和
  ;  Ak   - 第 k 項の値
  ;  k    - 繰り返しの回数
  ;  e    - 精度
  (define (loop sum Ak k e)
    (if (<= (/ Ak sum) e)
      (/ (* sum x) (+ 1 (* x x)))
      (loop
        (+ sum Ak)
        (* (/ (* 2 (+ k 1)) (+ (* 2 (+ k 1)) 1))
           (* Ak (/ (* x x) (+ 1 (* x x)))))
        (+ k 1)
        e)))

  ;; ARCTAN の本体
  (loop 1 (* (/ 2 3) (/ (* x x) (+ 1 (* x x)))) 1 1e-14))
```

オイラーの計算方法の式を見ると、和の計算部分で kの値を 1 から無限大 (∞) まで変えないといけません。これをそのまま実現すると計算に無限時間かかってしまいます。ですが実

数変数にはある精度しかないので、ある程度の項まで加え合わせるだけで十分です。内部手続き loop は、$\sum$の中の第 k項と第 $k-1$ 項までの和の比率が e= 1.0×10^{-14} $=$ $\overbrace{0.000\cdots0}^{0\text{が}14\text{個}}1$ になるまで和を計算し続けるようにしています。上の手続きでは $\sum$ の中の第 $k+1$ 項を計算するのに、第 k項を利用しています。第 k項を A_k とすると $A_{k+1}=\frac{2(k+1)}{2(k+1)+1}A_k\frac{x^2}{1+x^2}$ という関係を利用して、各 kごとに A_kを最初から計算しないようにしています。

ではこの手続きが正しく動くか試してみましょう。

```
> (atan 0.1)        — Scheme で用意されている手続き
99.668652491162e-3
> (arctan 0.1)      — オイラーによる方法
99.668652491162e-3
> (arctan 10)
1.4711276743023
> (atan 10)
1.4711276743037
```

一番上の桁から 14 桁まで正しいことが分かります[2]。

試しにマチンの公式とオイラー級数を使って、円周率を求めてみましょう。

```
> (- (* 16 (arctan (/ 1 5))) (* 4 (arctan (/ 1 239))))
3.1415926535898
>(+ (* 20 (arctan (/ 1 7))) (* 8 (arctan (/ 3 79))))
3.1415926535898
```

次にマチンの公式を使った別の実現法を示します。(後で作る高精度の円周率を計算するプログラムは、この手続きを基にします。) このプログラムでは $x=1/y$ とおき、$\frac{x^2}{1+x^2}=\frac{1}{y^2+1}$ の関係を使い、$y=5,239$ の場合それぞれに対して y^2+1 をあらかじめ求めておいて、プログラム中に直接書いています。また $\tan^{-1}\frac{1}{5}$ を求めた後で係数の 16 をかけるのではなく、最初からかけて計算しています。$\tan^{-1}\frac{1}{239}$についても同様です。

この手続きのあらすじは次の通りです。変数 P に円周率を求めます。最初に $\tan^{-1}\frac{1}{5}$ を計算しますが、オイラー級数の各項を順々に P に入れます。その後 $\tan^{-1}\frac{1}{239}$ を計算しますが、各項を P から引いてゆき、最終的な p の値が円周率の計算結果となります。

```
;; マチンの公式による円周率の計算
;; PI = 16 atan(1/5) - 4 atan(1/239)
;;   NOTE:     26 = 5*5 + 1
;;            956 = 4*239
;;          57122 = 239*239 + 1
```

[2]なおこの正しい桁の数は e の値を変えることで制御できますが、精度を高くしようとして e の値を変えても、Scheme での計算の精度の限界を越ることはできません。

```
(define (simple-pi)
  ;; 繰り返しの回数
  (define *max-loops*  10)
  ;; 16 atan(1/5) を計算する
  (define (loop-16atan1_5 P Ak k)
    (if (> k *max-loops*)
      (loop-4atan1_239       ; 16atan(1/5) が終ると、その結果に
        (- P (/ 956 57122))  ; -4atan(1/239) を加える
        (* (/ 956 57122)
           (/ 2 3)(/ 1 57122))
        1)
      (loop-16atan1_5
        (+ P Ak)
        (* Ak (/ (* 2 (+ k 1)) (+ (* 2 (+ k 1)) 1)) (/ 1 26))
        (+ k 1))))
  ;; - 4 atan(1/239) を計算する
  (define (loop-4atan1_239 P Ak k)
    (if (> k *max-loops*)
      P
      (loop-4atan1_239
        (- P Ak)
        (* Ak (/ (* 2 (+ k 1)) (+ (* 2 (+ k 1)) 1)) (/ 1 57122))
        (+ k 1))))
  ;; PI の本体
  (loop-16atan1_5
    (/ (* 16 5) 26)
    (* 16 (/ 5 26) (/ 2 3) (/ 1 26))
    1))
```

実行結果は次の通りです。

```
> (simple-pi)
3.1415926523898
```

11.1.3 円周率の高精度計算

以上では Scheme にあらかじめ用意されている精度の限られた方法を使っての計算でしたが、任意の桁数での円周率の計算を次に考えます。桁数の大きな数をひとつの変数に蓄えることは不可能なので、いくつかの桁ごとに分けて保持するようにします。たとえばある数が $d_0.d_1d_2d_3d_4\cdots d_{n-1}$ (ただし d_i は 0 から 9 までの数字) であるとき、ベクトル大きさ n のベクトルの第 i 要素に d_iを保持するようにすれば、ベクトルの大きさを変えることで好きなだけ桁数を用意することができます。(説明を簡単にするために整数部分に 1 桁しかありませんが、それ以外のときでも同じです。) 本実習では高精度の数データを「高精度数」と呼ぶことにします。

それぞれの高精度数データをオブジェクトとします。演算をするときはオブジェクトに演算依頼メッセージを送るようにします。高精度数を蓄えるオブジェクトのことを、レジ

スタと呼ぶことにします。実現を簡単にするために、以下の形の手続きとメッセージを使うものとします。

- (make-register ⟨ 桁数 ⟩)
  小数点以下の桁数が、引数 ⟨ 桁数 ⟩ で指定されただけの精度を持つレジスタを作ります。

- (obj ':=0!)
  レジスタの値を 0.0 にします。

- (obj ':=1!)
  レジスタの値を 1.0 にします。

- (obj '+! obj_2)
  レジスタの値に obj_2の値を加えます。

- (obj '-! obj_2)
  レジスタの値に obj_2の値を引きます。

- (obj '*i! i)
  レジスタの値に整数 i をかけます。

- (obj '/i! i)
  レジスタの値を整数 i で割ります。

- (obj 'SHOW)
  レジスタの値を表示します。

これらのメッセージを組み合わせて逆正接関数の計算をし、そして円周率の計算をします。ここで少し注意しないといけないのは、演算をした結果を新しく作った別のレジスタオブジェクトに入れて返すというのではなく、メッセージを受けとったオブジェクトがそれ自身の値を変更する、という点です。高精度数では多くの桁数を使うためにたくさんの記憶領域を使います。そのためすこしでも余計な記憶領域を使わないように、注意深くプログラムを作らないといけません。

レジスタオブジェクトを生成する手続きの骨格は次のようになります。

```
;;; (make-register 桁数)
;;;     レジスタオブジェクトを生成する。
;;;     小数点以下の精度は、引数で指定した桁数である。
;;;     生成されたとき、レジスタの初期値は 0.0 である。
;;;
;;; オブジェクトが解釈するメッセージ:
;;;   (OBJ ':=0!)     - レジスタの値を 0.0 にする
;;;   (OBJ ':=1!)     - レジスタの値を 1.0 にする
```

```
;;;   (OBJ '+! OBJ2) - レジスタの値に obj2 の値を加える
;;;   (OBJ '-! OBJ2) - レジスタの値に obj2 の値を引く
;;;   (OBJ '*i! i)   - レジスタの値に整数 i をかける
;;;   (OBJ '/i! i)   - レジスタの値を整数 i で割る
;;;   (OBJ 'SHOW)   - レジスタの値を表示する
;;;
(define (make-register digits)

  ;; 内部データ
  ; 数の保持
  (define *num*            #f)  ; 数を保持するベクトル
  (define *size*            0)  ; *num* の大きさ
  (define *decimal-point*   3)  ; 小数点の位置
  (define *base*          100)  ; 1 要素では 0 から 99 までを保持
  (define *uwidth*          2)  ; 1 要素が保持する 10 進数での桁数

  ; 表示関連
  (define *digits/block* 10)     ; 1 ブロックに表示する桁数
  (define *digits/line*  50)     ; 1 行に表示する桁数

  ⟨手続き zero!⟩    — レジスタの値を 0.0 にする
  ⟨手続き one!⟩     — レジスタの値を 1.0 にする
  ⟨手続き add!⟩     — レジスタの値にレジスタ obj2 を加える
  ⟨手続き sub!⟩     — レジスタの値からレジスタ obj2 を引く
  ⟨手続き imult!⟩   — レジスタの値に整数 I をかける
  ⟨手続き idiv!⟩    — レジスタの値を整数 I で割る
  ⟨手続き show⟩     — レジスタの値を表示する

  ;; DISPATCH --- メッセージの解析をする手続き
  (define (dispatch . msg)
    (case (car msg)
      ((:=0!)   (zero!))
      ((:=1!)   (one!))
      ((+!)     (add!   (cadr msg)))
      ((-!)     (sub!   (cadr msg)))
      ((*i!)    (imult! (cadr msg)))
      ((/i!)    (idiv!  (cadr msg)))
      ((SHOW)   (show))
      ((NUM)   *num*)  ; 内部のみで使用するメッセージ
      (else
        (error "Unknown Message" msg))))

  ;; MAKE-REGISTER の本体
  ;   ベクトルを確保する
  (set! *size* (+ (quotient (+ digits *uwidth* -1) *uwidth*)
                  *decimal-point*))
  (set! *num*  (make-vector *size* 0))
  ;   レジスタの値を 0.0 に初期化する
  (zero!)
  ; メッセージ解析手続きを値として返す
  dispatch)
```

それぞれの内部手続きは徐々に説明してゆきます。

ベクトルのひとつの要素に 1 桁しか入れないのは記憶領域がもったいないので、この実現ではひとつの要素に 2 桁入れるようにしています。*decimal-point*は小数点の位置に

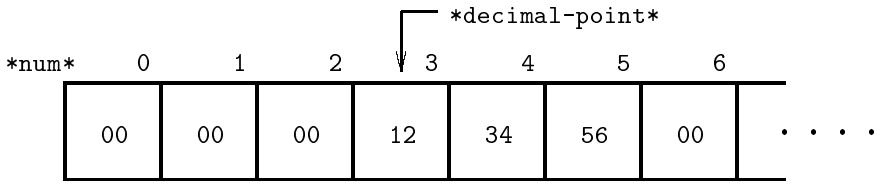


図 11.3: ベクトルによる高精度数の実現 (値が 0.123456 の場合)

当たるベクトルの添字を表す変数です。*num*の第 0 要素から第 (*decimal-point* − 1) 要素までが整数部分で、第*decimal-point*要素からが小数部分になります。配列の添字が大きいものほど位の小さい部分を保持します。たとえば 0.123456 は、図 11.1.3のように、*num*の第*decimal-point*要素に 12 を、第 (*decimal-point*+1) 要素に 34 を、第 (*decimal-point*+2) 要素に 56 を、それぞれ保持することになります。

make-register の本体部では確保すべきベクトルの要素数を計算し、内部変数*size*に代入しています。そしてその大きさのベクトルを生成し、内部変数*num*に代入しています。そしてレジスタの初期値を 0.0 にし、メッセージを解析する内部手続きを値として返しています。

高精度数どうしの演算をする手続きは自分で作らないといけません。高精度数の演算は筆算での計算と同じようにします。2 つの高精度数 $D = d_0.d_1d_2d_3d_4\cdots d_{n-1}$ と $E = e_0.e_1e_2e_3e_4\cdots e_{n-1}$ の足し算 $F = D + E = f_0.f_1f_2\cdots f_{n-1}$ の方法について説明します。

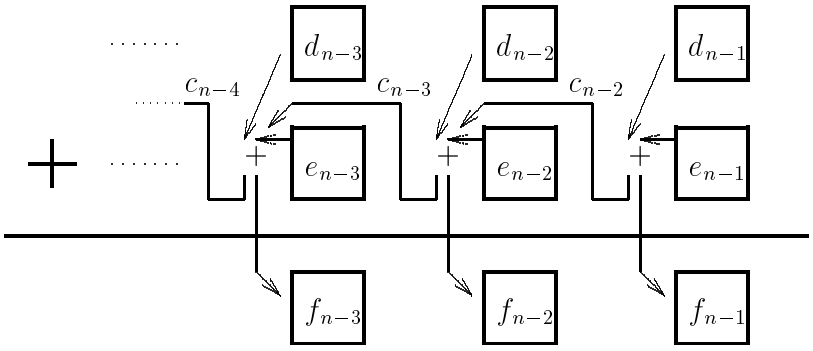


図 11.4: 高精度数どうしの加算

最初に $d_{n-1} + e_{n-1}$ を計算します。もしこの値が 10 を越えなければ、$f_{n-1} = d_{n-1} + e_{n-1}$ とします。もし 10 を越えるときは $d_{n-1} + e_{n-1}$ の 1 の桁を f_{n-1} とし、繰り上がりを c_{n-2} とします。(もし繰り上がりがなければ、$c_{n-2} = 0$ となります。)

次に f_{n-2} を求めます。f_{n-2} および繰り上がり c_{n-3} はそれぞれ、$d_{n-2} + e_{n-2} + c_{n-2}$ の 1 の桁、10 の桁となります。この様子を図 11.1.3に示します。これの考えを実現したものが、次の内部手続きです。

```
;; OBJ := OBJ+OBJ2   ---  レジスタの値にレジスタ OBJ2 を加える
(define (add! obj2)
  (let ((num2 (obj2 'num)))
    (define (add-loop j carry)
      (if (>= j 0)
        (let ((mm (+ carry
                     (vector-ref *num* j)
                     (vector-ref num2 j))))
          (vector-set! *num* j (modulo mm *base*))
          (add-loop (- j 1) (quotient mm *base*)))))
    ;
    (add-loop (- *size* 1) 0)))
```

この手続きを実現するにあたり、引数に与えられたオブジェクトにメッセージ num を送り、高精度数を保持しているベクトルを得ていることに注意しましょう。このメッセージはレジスタオブジェクトが内部で使うだけで、一般の利用には使いません。

引き算のときは繰り上がりの代わりに上の桁からの借りが起きるので、これに注意して引き算をします。

```
; OBJ := OBJ-OBJ2   ---  レジスタの値からレジスタ OBJ2 を引く
(define (sub! obj2)
  (let ((num2 (obj2 'num)))
    (define (sub-loop j borrow)
      (if (>= j 0)
        (let ((mm (- (vector-ref *num* j)
                     (vector-ref num2 j) borrow)))
          (if (>= mm 0)
            (begin
              (vector-set! *num* j mm)
              (sub-loop (- j 1) 0))
            (begin
              (vector-set! *num* j (+ *base* mm))
              (sub-loop (- j 1) 1))))))
    ;
    (sub-loop (- *size* 1) 0)))
```

こんどは高精度数 $D = d_0.d_1d_2\cdots d_{n-1}$ と整数 I のかけ算 $E = D \cdot I = e_0.e_1e_2\cdots e_{n-1}$ の方法を考えます。$d_i \cdot I$ は高精度数を使うまでもない桁数なので、次のようにして計算できます。

1. $x_{n-1} = d_{n-1} \cdot I$ とおく。$e_{n-1} = x_{n-1} \bmod 10$, $c_{n-2} = x_{n-1} \div 10$
2. $x_{n-2} = d_{n-2} \cdot I + c_{n-2}$ とおく。$e_{n-2} = x_{n-2} \bmod 10$, $c_{n-3} = x_{n-2} \div 10$
3. $x_{n-3} = d_{n-3} \cdot I + c_{n-3}$ とおく。$e_{n-3} = x_{n-3} \bmod 10$, $c_{n-4} = x_{n-3} \div 10$
4. 以下同様

これは各桁に I をかけ、繰り上がりがあればそれを次の桁に持ってゆく、というものです。

```
;; OBJ := OBJ*I  ---  レジスタの値に整数 I をかける
(define (imult! i)
  (define (mult-loop j carry)
    (if (>= j 0)
      (let ((mm (+ carry (* i (vector-ref *num* j)))))
        (vector-set! *num* j (modulo mm *base*))
        (mult-loop (- j 1) (quotient mm *base*)))))
  ;
  (mult-loop (- *size* 1) 0))
```

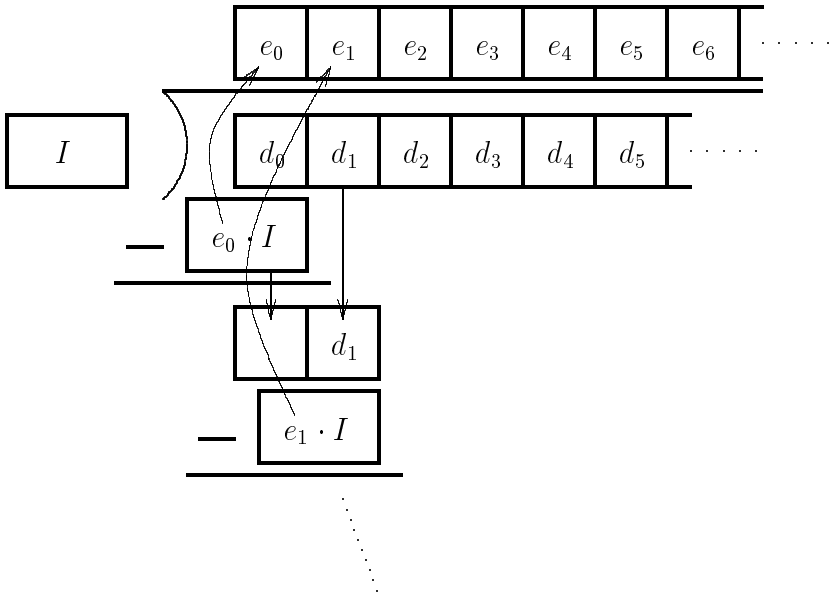


図 11.5: 高精度数を整数で割る

次は高精度数 $D = d_0.d_1d_2 \cdots d_{n-1}$ と整数 I の割算 $E = D/I = e_0.e_1e_2 \cdots e_{n-1}$ の方法について説明します。これも筆算と同じように計算します。上で説明した 3 つの演算はどれも一番下の桁から計算しましたが、割算のときは上の桁から計算してゆきます。図 11.1.3 のように、まず割られる数 D の上の 1 桁 d_0 を取り、I で割ります。その値が商のひとつの桁 e_0 になります。割った余りは下の桁に加えられます。これを最後の桁まで順におこないます。

```
;; OBJ := OBJ/I   ---  レジスタの値を整数 I で割る
(define (idiv! i)
  (define (div-loop j m)
    (if (< j *size*)
      (let* ((mm (+ (* m *base*) (vector-ref *num* j))))
        (vector-set! *num* j (quotient mm i))
        (div-loop (+ j 1) (modulo mm i)))))
  ;
  (div-loop 0 0))
```

次はレジスタオブジェクトの値の表示です。読みやすくするために 10 桁ごとに空白文字を入れ、50 桁表示したら改行するようにしています。少々トリッキーなプログラムですが、十分に吟味してみて下さい。

```
;; レジスタの値を表示する
(define (show)
  ; 整数部分を表示する
  (define (show-integer-part)
    (loop 0 1 '() *decimal-point* ""))
  ; 小数部分を表示する (一定桁ごとに空白文字を入れる)
  (define (show-decimal-part)
    (loop *decimal-point* 1 '() *size* " "))
  ; 数の表示をする
  (define (loop index digits strs max-index space-str)
    (define (mk-nstr n)
      (let* ((str (string-append
                   (make-string *uwidth* #\0) (number->string n))))
        (string->list (substring str
                        (- (string-length str) *uwidth*)
                        (string-length str)))))
    (if (null? strs)
      (if (< index max-index)
        (loop (+ index 1) digits
          (mk-nstr (vector-ref *num* index)) max-index space-str))
      (begin
        (display (car strs))
        (if (= (modulo digits *digits/block*) 0)
          (display space-str))
        (if (= (modulo digits *digits/line*) 0)
          (newline))
        (loop index (+ digits 1) (cdr strs) max-index space-str))))
  ; SHOW の本体
  (show-integer-part)
  (display ".")
  (newline)
  (show-decimal-part)
  (newline))
```

では以上のプログラムを試してみましょう。

```
> (define a (make-register 50))  — 小数以下 50 桁を持つレジスタ
#<unspecified>
> (a ':=1!)
#<unspecified>
> (a 'show)
000001.
0000000000 0000000000 0000000000 0000000000 0000000000
#<unspecified>
> (a '/i! 3)
#<unspecified>
> (a 'show)
```

```
000000.
3333333333 3333333333 3333333333 3333333333 3333333333
#<unspecified>
> (define b (make-register 50))
#<unspecified>
> (b ':=1!)
#<unspecified>
> (b '/i! 9)          — b = 0.11111...
#<unspecified>
> (b '+! b)           — 0.33333...+0.11111...
#<unspecified>
> (b 'show)
000000.
4444444444 4444444444 4444444444 4444444444 4444444444
#<unspecified>
```

手続き simple-pi を基にして、高精度で円周率を求めるプログラムを作ります。注意しないといけないのが、演算の精度に関することです。もし小数点以下 1000 桁計算したいとします。ですが、第 1001 桁からの繰り上がりや繰り下がりも正しく計算できないと、第 1000 桁は正しい値にはなりません。そのため、レジスタの小数点以下の桁数にちょうど 1000 桁を用意のではなく、少し多めに用意して計算しないといけません。

また別の注意点として、$\tan^{-1}$ の計算をするのに第何項まで求めればいいのか、ということがあげられます。これは求めようとする精度よりも、項が小さくなるまで繰り返せばよいことになります。レジスタが小数点以下 D 桁計算する場合には、第 k項を A_k とし、$A_k < 10^{-D}$ を満たす k の内で最小のものを K とすれば、$\tan^{-1}$ の計算の繰り返しを K回すればよいことになります。

小数点以下 D を求めるのに必要なレジスタの桁数と、必要な繰り返しの回数を求める式について考えてみましょう。オイラー級数の$\sum$の部分の第 k項は、

$$A_k = \frac{\prod_{m=1}^{k}(2m)}{\prod_{m=1}^{k}(2m+1)} \cdot (\frac{x^2}{1+x^2})^k$$

となっています。これより $A_k \leq (\frac{x^2}{1+x^2})^k$ を得ます。少々大きめの見積りですが、次の式を満たす k の内で最小のものを K とします。

$$(\frac{x^2}{1+x^2})^k < 10^{-D}$$

両辺を底を 10 とする対数をとり式を変形すると、次の式を得ます。

$$k > \frac{D}{\log_{10} \frac{x^2}{1+x^2}}$$

よって

$$K = \left\lfloor \frac{D}{\log_{10} \frac{x^2}{1+x^2}} \right\rfloor$$

となります[3]。$x = 1/y$とおけば、

$$K = \left\lfloor \frac{D}{\log_{10} \frac{1}{y^2+1}} \right\rfloor$$

となります。(マチンの公式では y の値として 5 と 239 を使います。) これを手続きとして実現したのが次の loops です。

```
;; ループ回数の計算
(define (loops d y)
  (inexact->exact (+ (floor (/ d (log10 (+ (* y y) 1)))) 1)))
;; 底を 10 とする対数関数
(define (log10 x)
  (/ (log x) (log 10)))
```

次に、必要な精度を得るのに必要なレジスタの桁数について考えます。たとえば小数点以下 1000 桁まで求めるとしましょう。この場合 $\tan^{-1}\frac{1}{5}$ と $\tan^{-1}\frac{1}{239}$ をオイラー級数で計算するのに必要な繰り返し回数はそれぞれ、$n_1 = 707$ と $n_2 = 211$ となります。$\tan^{-1}\frac{1}{5}$ の計算の場合、最悪の場合は一番下の桁での誤差が 707 回蓄積されます。そのため下から 3 桁は違ったものになる可能性がありますし、下から 3 桁目が違うことで下から 4 桁目への繰り上がりが生じ、下から 4 桁目が違った値になる可能性もあります。よって 4 桁は余計に計算しておく必要があります。このことを任意の桁数に対して求めるようにした手続き register-size を以下に示します。

```
;; 必要な精度を得るために必要な桁数の計算
(define (register-size D y)
  (+ D
     (inexact->exact (ceiling (+ (log10 (loops D y)) 1)))))
```

ではいよいよ超高精度で円周率を求めるプログラム pi を示します。これは上で作成した simple-pi を基にして、レジスタオブジェクトを使って計算するようにしたものです。

```
;;;
;;; PI  ---     高精度の円周率の計算
;;;
(define (pi D)

  ; MEMO:
  ;       26 =  5*5 + 1
  ;      956 = 4*239
  ;    57122 = 239*239 + 1
```

[3] $\lfloor x \rfloor$ は、x を越えない最大の整数を表し、床関数と呼びます。たとえば、$\lfloor 4.8 \rfloor = 4$, $\lfloor 9 \rfloor = 9$ などです。(この関数は、手続き floor という名前で用意されています。)

```
(let ((PI #f)    ; 円周率を蓄えてゆくレジスタ
      (Ak #f)    ; k 番目の項を保持するレジスタ
      (n1 #f)    ; atan(1/5) を計算するのに必要な繰り返し回数
      (n2 #f)    ; atan(1/239) を計算するのに必要な繰り返し回数
      (size #f)) ; レジスタの桁数

  ⟨ 手続き loops⟩
  ⟨ 手続き register-size⟩
  ⟨ 手続き log10⟩

  ;; PI:= 16atan(1/5) を計算する
  (define (loop-16atan1_5 k)
    (if (> k n1)
      (begin
        ;   PI := (- PI (/ 956 57122))
        (Ak ':=1!)       ; 4atan(1/239) の初項の計算のために
        (Ak '*i! 956)    ; 一時的に Ak を使う
        (Ak '/i! 57122)  ;
        (PI '-! Ak)
        ; この時点で Ak には (956/57122) が入っている
        ;   Ak := (* (/ 956 57122) (/ 2 3)(/ 1 57122))
        (Ak '*i! 2)
        (Ak '/i! 3)
        (Ak '/i! 57122)
        (loop-4atan1_239 1))
      (begin
        ;   PI := PI + Ak
        (PI '+! Ak)
        ;   Ak := (* Ak (/ (* 2 (+ k 1)) (+ (* 2 (+ k 1)) 1))
        ;          (/ 1 26))
        (Ak '*i! (* 2 (+ k 1)))
        (Ak '/i! (* (+ (* 2 (+ k 1)) 1)))
        (Ak '/i! 26)
        (loop-16atan1_5 (+ k 1)))))

  ;; PI := PI - 4atan(1/239) を計算する
  (define (loop-4atan1_239 k)
    (if (> k n2)
      'done
      (begin
        ;   PI := (- PI Ak)
        (PI '-! Ak)
        ;   Ak := (* Ak (/ (* 2 (+ k 1)) (+ (* 2 (+ k 1)) 1))
        ;              (/ 1 57122))
        (Ak '*i! (* 2 (+ k 1)))
        (Ak '/i! (+ (* 2 (+ k 1)) 1))
        (Ak '/i! 57122)
        (loop-4atan1_239 (+ k 1)))))

  ;; PI の本体
  ; ループ回数、桁数を求める
  (set! n1 (loops D 5))
  (set! n2 (loops D 239))
  (set! size (max (register-size D 5) (register-size D 239)))
  ; レジスタを作る
  (set! PI (make-register size))
  (set! Ak (make-register size))
```

```
  ; 円周率の計算を開始
  ;   PI := (/ (* 15 5) 26)
  (PI ':=1!)
  (PI '*i! (* 16 5))
  (PI '/i! 26)
  ;   Ak := (* 16 (/ 5 26) (/ 2 3) (/ 1 26))
  (Ak ':=1!)
  (Ak '*i! (* 16 5 2))
  (Ak '/i! (* 26 3 26))
  (loop-16atan1_5 1)
  ;  円周率を表示する
  (PI 'show)))
```

小数点以下 1000 桁まで円周率を求めてみましょう。

```
> (pi 1000)
000003.
1415926535 8979323846 2643383279 5028841971 6939937510
5820974944 5923078164 0628620899 8628034825 3421170679
8214808651 3282306647 0938446095 5058223172 5359408128
4811174502 8410270193 8521105559 6446229489 5493038196
4428810975 6659334461 2847564823 3786783165 2712019091
4564856692 3460348610 4543266482 1339360726 0249141273
7245870066 0631558817 4881520920 9628292540 9171536436
7892590360 0113305305 4882046652 1384146951 9415116094
3305727036 5759591953 0921861173 8193261179 3105118548
0744623799 6274956735 1885752724 8912279381 8301194912
9833673362 4406566430 8602139494 6395224737 1907021798
6094370277 0539217176 2931767523 8467481846 7669405132
0005681271 4526356082 7785771342 7577896091 7363717872
1468440901 2249534301 4654958537 1050792279 6892589235
4201995611 2129021960 8640344181 5981362977 4771309960
5187072113 4999999837 2978049951 0597317328 1609631859
5024459455 3469083026 4252230825 3344685035 2619311881
7101000313 7838752886 5875332083 8142061717 7669147303
5982534904 2875546873 1159562863 8823537875 9375195778
1857780532 1712268066 1300192787 6611195909 2164201989
3517
#<unspecified>
```

上の結果を本当の値と比べると最後の 3 桁が違っていますので、小数点以下 1001 桁は正しい値が求まっています。

まとめ

本実習では円周率を高精度で計算するプログラムを作りました。逆正接関数の値の計算に、その級数を基にして加減乗除だけを使いました。その他、$\sin, \cos$ などの関数も、加減乗除を使って計算できます。

最終的に作ったプログラムは複雑で少し長めです。しかしいきなり目的のプログラムを作らず、計算方式を確かめながら、順々に複雑なものにしていきました。そのために、さ

ほど大きな苦労やプログラムの間違いで苦しむことはありませんでした。作るプログラムが大きくなってくると、部分ごとに動作テストをする方法が非常に有効となってきます。

本実習で紹介した方法以外にもいくつか円周率を計算する方法がありますので、本で調べて実際に試してみるとよいでしょう。

練習問題

1. オイラーの公式を使い、円周率を高精度で計算するプログラムを作りなさい。$\tan^{-1}$ の計算はオイラー級数を使いなさい。

2. ガウスの公式を使い、円周率を高精度で計算するプログラムを作りなさい。$\tan^{-1}$ の計算はオイラー級数を使いなさい。

3. 本実習で示した手続き `pi` を、グレゴリー級数で計算するように変更してみなさい。

4. レジスタオブジェクトが、オブジェクト同士のかけ算をできるようにしなさい。

5. レジスタオブジェクトが、オブジェクト同士の割り算をできるようにしなさい。

6. 上で示したレジスタオブジェクトは正の数しか取り扱えません。負の数も取り扱えるよう変更しなさい。

7. 上で示したレジスタオブジェクトでの値の表示は、それがもつ桁すべてを表示してしまいます。指定した桁数だけを表示するように変更しなさい。

8. ** 任意の桁数の整数を取り扱える機能はほとんどの Scheme 処理系で用意されていて、**巨大整数** (bignum) と呼ばれています。本実習で使った方法を参考にして、任意の桁数を持つ整数を取り扱える整数用のレジスタオブジェクトを作りなさい。1000 の階乗 $1000! = 1000 \cdot 999 \cdot 998 \cdots 2 \cdot 1$ を計算するプログラムを書きなさい。

9. ある定数 $e = 2.71828\cdots$ に対して関数 $f(x) = e^x$ を考えます。この $f(x)$ は、微分しても変化しない、すなわち $\frac{d}{dx}f(x) = f(x)$ という性質があります。また i を虚数単位 ($i^2 = -1$) とすれば、$e^{i\pi} = -1$ という、摩可不思議な関係式が成立します。この定数 e は自然対数の底と呼ばれていて、次の級数で計算できます。

$$\begin{aligned} e &= \sum_{k=0}^{\infty} \frac{1}{k!} \\ &= 1 + \frac{1}{1} + \frac{1}{1\cdot 2} + \frac{1}{1\cdot 2\cdot 3} + \cdots \end{aligned}$$

この級数に基づいて、自然対数の底を (Scheme の実数データとして) 求めなさい。

10. * 自然対数の底を小数点以下 1000 桁まで求めるプログラムを作り、実行させてみなさい。

11. *** Scheme インタプリタで円周率の高精度計算プログラムを実行するのは、たいへん時間がかかります。より高速に計算するために、本実習で示したプログラムを他のプログラミング言語 (たとえば C 言語、Fortran, Pascal など) で書き換え、コンパイラを使ってコンパイルし、実行してみなさい。

12. 円周率が何桁まで計算できるかをめぐって、熾烈な競争が繰り広げられています。現在の円周率の計算競争では、より高速な計算公式である、ガウス·ルジャンドルの方法が使われています。ガウス·ルジャンドルの方法とは次のものです。

    (a) 4 つの変数 A, B, T, X の初期値を、$A = 1$, $B = 1/\sqrt{2}$, $T = 1/4$, $X = 1$ とします。

    (b) A と B の差が、必要としている精度になるまで、以下の動作を繰り返し実行します: ($x \leftarrow y$ は、x 計算結果を y に代入することを表します。)

        i. $Y \leftarrow A$

        ii. $A \leftarrow (A + B)/2$

        iii. $B \leftarrow \sqrt{B \cdot Y}$

        iv. $T \leftarrow T - X \cdot (Y - A)^2$

        v. $X \leftarrow 2 \cdot X$

    (c) $\frac{(A+B)^2}{4T}$ が、求まった円周率です。

    (金田康正著, “πのはなし” より)

    ガウス·ルジャンドルの方法に基づき、実数データを使って円周率を計算する手続きを作りなさい。

13. ***** ガウス·ルジャンドルの方法に基づき、任意の桁数で円周率を計算する手続きを作りなさい。(それぞれの変数の初期値は、求めようとする円周率の桁数よりも多い桁数で計算しておく必要があります。)

    大型コンピューターまたはスーパーコンピューターを使い、Fortran あるいは C 言語などで上記のプログラムを上手に書けば、円周率の世界記録も狙えます。(計算機の使用料は高額になるでしょうから、くれぐれも気をつけましょう)

参考文献

- 大野栄一, “パソコンで挑む円周率　πの歴史から計算まで,” ブルーバックス B-889, 講談社, 1991 年 1 月.
- 金田 康正, “πのはなし”, 東京図書, 1991 年 4 月.

- 有馬哲, 浅枝陽, “詳解演習 微積分 I,” 東京図書, 1981 年 4 月.
- 有馬哲, 浅枝陽, “詳解演習 微積分 II,” 東京図書, 1981 年 9 月.
- 伊理正夫, 藤野和建, “数値計算の常識,” 共立出版, 1985 年 6 月.

11.2 プログラミング言語処理系の基礎 — 式の計算

本実習では、与えられた式を計算するプログラムを作ります。入力の式がどのようなものかを調べ、それに応じた動作させます。与える式はコンピューターに与える一種のプログラムとみなすことができます。これから作るプログラムは、少々大げさですが、「式」というプログラミング言語に対する言語処理系です。実行するものを逐次解釈しながら実行するので、インタプリタ方式の言語処理系といえます。

入力となる式は (1 + 2) のような形で与えられるものとします。Scheme の文法に合っていないため、この式をそのまま Scheme に与えて計算させるわけにはゆきません。入力がどうなっているかを調べ、それに応じた動作をさせて計算するプログラムを作る必要があります。

最終的には ((6 - 3) + (3 + (2 + 2))) といった複雑な式が計算でき、しかも電卓についているようなメモリー機能を持ったプログラムを作ります。いきなりそのようなプログラムを作るのは大変ですので、最低限の機能を持ったものから、徐々に機能を付け加えてゆきます。取り扱える演算は加算と減算だけにします。それ以外の演算子を取り扱えるように改造するのは簡単なので、読者への問題とします。

11.2.1 第一段階: 単純な式の計算

最初は (2 + 4) や 3 のように、(⟨ 数$_1$⟩ ⟨ 演算子 ⟩ ⟨ 数$_2$⟩) または単に⟨ 数 ⟩という 2 つの形に限定した式を計算する手続き calc-exp0 を作ります。これから作る calc-exp0 は 1 引数の手続きで、(calc-exp0 '(2 + 4)) として呼び出すと評価結果 6 が返されるようにします。また単に (calc-exp0 '2) として呼び出すと、評価結果 2 が返されるものとします。

calc-exp0 の仮引数を exp とします。(calc-exp0 '(2 + 4)) として呼び出すと仮引数 exp の値はリスト (2 + 4) となります。次の考えに従えば、式を計算することができます。

- exp が数の場合 (6 など)
  — その数そのものを返します。
- exp がリストの場合 ((2 + 4) など)
  1. exp の第一要素を取り出し、これを n_1 とおきます。
  2. exp の第二要素を取り出し、これを op とおきます。
  3. exp の第三要素を取り出し、これを n_2 とおきます。
  4. A op が + の場合
     $n_1 + n_2$ を計算し、その結果を calc-exp0 の値とします。
     B op が - の場合
     $n_1 - n_2$ を計算し、その結果を calc-exp0 の値とします。

以上の考えに基づいて calc-exp0 を作ると、次のようになります。

```
(define (calc-exp0 exp)
  (cond
    ((number? exp)
     exp)
    ((list? exp)
     (case (length exp)
       ((3)
        (let* ((n1 (list-ref exp 0))
               (op (list-ref exp 1))
               (n2 (list-ref exp 2)))
          (if (and (number? n1) (number? n2)
                   (or (eq? op '+) (eq? op '-)))
            (case op
              ((+) (+ n1 n2))
              ((-) (- n1 n2)))
            (error "syntax error"))))
       (else (error "syntax error"))))
    (else (error "syntax error"))))
```

計算方法のアイディアの部分では、入力に間違いがある場合は書いていませんでした。calc-exp0 では、与えられる式が正しい形をしているかどうかをチェックし、変な入力の時はエラーとしています。人間がデータを入力したりする場合は、必ずといっていいほど入力間違いをします。データが正しいかどうかを必ずチェックするようにしましょう。

では、この手続きを実行させてみましょう。

```
> (calc-exp0 '2)            — ⟨数⟩を与えた場合
2
> (calc-exp0 '(2 + 7))      — 加算をする式 2 + 7 = 9
9
> (calc-exp0 '(32 - 6))     — 減算をする式 32 − 6 = 26
26
> (calc-exp0 'abc)          — 文法違反の入力
ERROR: syntax error
> (calc-exp0 '(2 + 7 + 5))
            — (⟨数1⟩⟨演算子⟩⟨数2⟩) の形でない
ERROR: syntax error
```

```
> (calc-exp0 '(+ 4 10))
```

— これも (⟨ 数$_1$ ⟩ ⟨ 演算子 ⟩ ⟨ 数$_2$ ⟩) の形でない

```
ERROR: syntax error
```

入力に (calc-exp0 '(2+7)) (2 と+、+と 7 の間にスペース文字がないことに注意) として呼び出すと、エラーとなります。

```
> (calc-exp0 '(2+7))
ERROR: syntax error
```

これはスペース文字がなく続けて書いてあるので、Scheme は 2+7 をひとつのデータとみなすからです。

calc-exp0 は十数行の小さなプログラムですが、その背景にはいろいろな計算機科学の概念があります。以下では、文法に関することと入力が持つ意味について説明します。

まず入力の**文法** (grammer) について説明します。入力としてどのようなものを取り扱うか (入力が満たさないといけない文法) は、作ろうとするプログラムの仕様の一部です。そのため、文法を明確に決めておく必要があります。(上では直観的な理解のため、文法はいいかげんな説明で済ませていました。本当はプログラムよりも文法を決めるのが先です。) プログラミング言語の用語では、言語の文法的な側面のことを**構文論** (syntax) と呼んでいます。

規則 G0-1: ⟨ 式 ⟩ := ⟨ 数 ⟩ | (⟨ 数 ⟩ ⟨ 演算子 ⟩ ⟨ 数 ⟩)
規則 G0-2: ⟨ 演算子 ⟩ := + | -
規則 G0-3: ⟨ 数 ⟩ := 0 | 1 | 2 | ··· | -1 | -2 | ···

図 11.6: 文法 G0

第一段階で取り扱える式は、図 11.6 に示した文法 G0 を満たすものに限ります。ここで、文法の記述方法について説明します。文法の記述は、以下に示す規則に従っています[4]。

- :=
  左辺のものは右辺のものに置き換えられる、ということを表しています。

[4]本当は ⟨ 数 ⟩ に対して厳密な文法規則を決める必要があります。(たとえば +10 や -10 は許しても +-10 は許さない、ということの根拠となる規則) これは与えるデータは Scheme でのリストとしているので、使っている Scheme 処理系がデータの入力を受け付ける段階で Scheme データとしての文法チェックがされます。このことにより、わたしたちが作る手続きは厳密な文法チェックをしないで済んでいます。もし入力が文字列で与えられる場合は、簡単な文法チェックでは不十分となります。たとえば、 "(5 + 8" が与えらた場合、閉じ括弧) がないことを検出し、エラーにする必要があります。

- |
  選択を表し、どれかひとつ、ということを表しています。

- タイプライタ体 ((や + の形の書体) で書かれた部分
  その文字そのものを表しています。

この文法の記述方法は、**BNF 記法** または**バッカス·ナウア形式** (Backus-Naur form) といいます。

この規則に従って図 11.6の文法 G0 を解読すると、次のようになります。⟨ 式 ⟩ というものは⟨ 数 ⟩ もしくは (⟨ 数 ⟩ ⟨ 演算子 ⟩ ⟨ 数 ⟩) どちらか一方の形をしたものをいいます。⟨ 数 ⟩ は数字のならびで、20 や -189 などのものです。⟨ 演算子 ⟩ は + または - です。

文法 G0 に従えば、その文法に合うものを導き出すことができます。

⟨ 式 ⟩ ⇒ (⟨ 数$_1$⟩ ⟨ 演算子 ⟩ ⟨ 数$_2$⟩) — (規則 G0-1)
⇒ (4 ⟨ 演算子 ⟩ ⟨ 数 ⟩) — ⟨ 数$_1$⟩ を 4 に (規則 G0-3)
⇒ (4 + ⟨ 数 ⟩) — ⟨ 演算子$_1$⟩ を + に (規則 G0-2)
⇒ (4 + 10) — ⟨ 数$_2$⟩ を 10 に (規則 G0-3)

もうひとつ例を見てみましょう。

⟨ 式 ⟩ ⇒ ⟨ 数 ⟩ — (規則 G0-1)
⇒ 4 — ⟨ 数 ⟩ を 4 に (規則 G0-3)

このようにして導き出すことのできるものはすべて、文法 G0 をみたすものです。このように、⟨⟩をまったく含まないように文法規則より導き出すこと、あるいは導き出されたものを**導出** (derivation) といいます。手続き calc-exp0 の実行例でエラーとなった (2 + 7 + 5) は、文法 G0 から導出できないことは直観的に分かると思います。

与えられたものが文法 G0 の導出であるかどうかを判断することもできます。たとえば (4 + 10) の場合を以下に示します[5]。

1. (4 + 10) はリストです。
2. よって、⟨ 式 ⟩ := (⟨ 数 ⟩ ⟨ 演算子 ⟩ ⟨ 数 ⟩) の規則が使われています。(規則 G0-1)
3. すると、リストの第一要素と第三要素は数で、第二要素は + か - でないといけません。(4 + 10) をみれば、確かに合致しています。(規則 G0-3 と 規則 G0-2)

[5]本当は形式言語理論に基づいた入力の解析が必要になりますが、これは本書の内容を越えています。本書の範囲内ではあまり問題にならないので、ここではいい加減な方法で済ませます。

このように、文法のどの規則を使ってその入力が導出されたかを解析することを、**構文解析** (parsing) といいます。どのようにして入力が導出されたかが分かれば、あとはその言語の意味の定義に合わせた動作をします。これにより与えた入力の意図に合った結果を得ることができます。

これまでは文法について説明しましたが、実は文法を決めるだけでは不十分です。与えられた入力がどのような意味を持つかを決めないといけません。プログラミング言語の用語では、入力とその意味との対応を決める規則のことを**意味論** (semantics) と呼んでいます。

入力が `(4 + 10)` であるとしましょう。これは単に `4`, `+`, `10` のリストにしかすぎません。文法とは別に、入力 `(4 + 10)` は $4+10$ (書体の違いに注意して下さい) であり、これを計算の計算結果の 14 を表す、ということを決めてやらないといけません。いま取り扱っている文法は式に関するものなので、その意味はほとんど明らかです。しかしまぎらわしさを避けるためには、意味をきちんと記述しないといけません[6]。

規則 S0-1: $\mathsf{Eval}(\langle\, \text{数} \,\rangle) = \langle\, \text{数} \,\rangle$が表す数値
規則 S0-2: $\mathsf{Eval}(\,(\ \langle\, \text{数}_1 \rangle\ \texttt{+}\ \langle\, \text{数}_2 \rangle\ )\,) = \mathsf{Eval}(\langle\, \text{数}_1 \rangle) + \mathsf{Eval}(\langle\, \text{数}_2 \rangle)$
規則 S0-3: $\mathsf{Eval}(\,(\ \langle\, \text{数}_1 \rangle\ \texttt{-}\ \langle\, \text{数}_2 \rangle\ )\,) = \mathsf{Eval}(\langle\, \text{数}_1 \rangle) - \mathsf{Eval}(\langle\, \text{数}_2 \rangle)$

図 11.7: 意味規則 S0

文法 G0 に合った「プログラム」(入力) の意味を、図 11.7に示します。図での $\mathsf{Eval}(\mathrm{x})$ は x の意味、すなわち x の計算結果を表す記号として使っています。

`(10 + 4)` の意味は、この意味規則に従って調べることができます。

$$
\begin{aligned}
\mathsf{Eval}(\texttt{(4 + 10)}) &= \mathsf{Eval}(\texttt{4}) + \mathsf{Eval}(\texttt{10}) && \text{— (規則 S0-2)} \\
&= 4 + \mathsf{Eval}(\texttt{10}) && \text{— (規則 S0-1)} \\
&= 4 + 10 && \text{— (規則 S0-1)} \\
&= 14
\end{aligned}
$$

このことから、`(4 + 10)` の意味は 14 となります。

以上のことより、文法規則と意味規則のふたつが決まって、はじめてプログラミング言語が決まることが分かったかと思います。

[6]意味論の記述方法には操作的意味論 (operational semantics), 表示的意味論 (denotational semantics)、公理的意味論 (axiomatic semantics) などがありますが、これもまた本書の内容を越えるので説明しません。本書ではわりと直観的な方法で説明します。

11.2.2 第二段階: 式の入れ子

第二段階では（(6 + 3) - 5）のように、入れ子となった式を書くことができるように拡張します。文法規則 G1 を図 11.8に示します。G0 では（⟨数⟩⟨演算子⟩⟨数⟩）というように、加算や減算のときは数しか演算の対象として許していませんでした。G1 ではこれを変更して⟨式⟩ := （⟨式⟩⟨演算子⟩⟨式⟩）のように、演算の対象に式を書くことを許しています。

規則 G1-1: ⟨式⟩ := ⟨数⟩ | （⟨式⟩⟨演算子⟩⟨式⟩）
規則 G1-2: ⟨演算子⟩ := + | -
規則 G1-3: ⟨数⟩ := 0 | 1 | 2 | ··· | -1 | -2 | ···

図 11.8: 文法 G1

私達がふだん括弧のある式の計算をするときと同じように、入れ子の式が現れたら入れ子の部分を先に計算し、その結果に置き換えて計算をします。この考えに従って決めた意味規則 S1 を、図 11.9に示します。

規則 S1-1: $\mathsf{Eval}(\langle \text{数} \rangle) = \langle \text{数} \rangle$が表す数値
規則 S1-2: $\mathsf{Eval}(\text{(} \langle \text{式}_1 \rangle \text{ + } \langle \text{式}_2 \rangle \text{)}) = \mathsf{Eval}(\langle \text{式}_1 \rangle) + \mathsf{Eval}(\langle \text{式}_2 \rangle)$
規則 S1-3: $\mathsf{Eval}(\text{(} \langle \text{式}_1 \rangle \text{ - } \langle \text{式}_2 \rangle \text{)}) = \mathsf{Eval}(\langle \text{式}_1 \rangle) - \mathsf{Eval}(\langle \text{式}_2 \rangle)$

図 11.9: 意味規則 S1

この文法規則 G1 と意味規則 S1 を満たす、式の計算手続きの実現に移ります。演算対象が入れ子の式のときはさらにそれを (再帰的に) 計算してやらないといけない、ということが第一段階と違う点です。これの機能を持たせるのは以外と簡単です。以下に実現例 calc-exp1 を示します。

```
(define (calc-exp1 exp)
  (cond
    ((number? exp)
     exp)
    ((list? exp)
```

```
(case (length exp)
  ((3)
   (let* ((n1 (calc-exp1 (list-ref exp 0)))   ;*
          (op (list-ref exp 1))
          (n2 (calc-exp1 (list-ref exp 2))))  ;*
     (if (and (number? n1) (number? n2)
              (or (eq? op '+) (eq? op '-)))
         (case op
           ((+) (+ n1 n2))
           ((-) (- n1 n2)))
         (error "syntax error"))))
  (else (error "syntax error"))))
(else (error "syntax error"))))
```

calc-exp0 と違う部分は ;* をつけた場所だけです。calc-exp0 での (n1 (list-ref exp 0)) は、calc-exp1 では (n1 (calc-exp1 (list-ref exp 0))) となっています。calc-exp0 では演算の対象は数であると決まっていましたが、calc-exp1 では演算の対象となる部分をさらに再帰的に計算しています。このほんのわずかな変更により、入れ子となった式が現れてもちゃんと計算できるようになります。

ではこれを実行させてみましょう。

```
> (calc-exp1 '(2 + 2))
4
> (calc-exp1 '((2 + 3) + 4))
9
> (calc-exp1 '((2 + 3) + (1 + 2)))
8
> (calc-exp1 '((2 + 3) + (1 + (1 + 1))))
8
> (calc-exp1 '((2 + 3) + ((2 + 2) + ((3 + 3) + 1))))
16
```

この手続きの動きはすこし複雑なので、((2 + 3) + 4) をデータとして与えたときを例として、どのように計算が進んでゆくかを眺めてみましょう。

$$
\begin{aligned}
\texttt{((2 + 3) + 4)} &= \boxed{\texttt{(2 + 3)}} + \texttt{4} && \texttt{(2 + 3)} \text{ の値を再帰的に計算します} \\
&= \boxed{\boxed{\texttt{2}} + \texttt{3}} + \texttt{4} && \texttt{2} \text{ の値を再帰的に計算します} \\
&= \boxed{\texttt{2 + 3}} + \texttt{4}
\end{aligned}
$$

$$
\begin{aligned}
&= \boxed{2+\boxed{3}} + 4 && \text{3 の値を再帰的に計算します}\\
&= \boxed{2+3} + 4 \\
&= \boxed{2+3} + 4 && \text{2 + 3 を計算します}\\
&= 5+\boxed{4} && \text{4 の値を再帰的に計算します}\\
&= 5+4 && \text{5 + 4 を計算します}\\
&= 9
\end{aligned}
$$

11.2.3 第三段階: メモリー機能

本実習の最終段階では、電卓についているようなメモリー機能を追加した手続き calc-exp2 を作ります。メモリーに保持されている数を、数の代わりとして使うことができるようにします。

メモリー関連の機能は次の機能を持たせることにします。

- AC
  メモリーの値をゼロにします。この式の値は 0 とします。

- (Min ⟨ 式 ⟩)
  ⟨ 式 ⟩の計算結果をメモリーに入れます。(Min ⟨ 式 ⟩) の値は、⟨ 式 ⟩の計算結果とします。

- MR
  メモリーに保持されている数を表します。

これらの記号は、電卓のボタンに書かれている記号にちなんでいます。

たとえばこれらの機能は、次のようにして使います。

```
(calc-exp2 'AC)              — メモリーの値をゼロにします
(calc-exp2 '(Min (3 + 4))    — 3 + 4 = 7 をメモリーに入れます
(calc-exp2 '(MR + 6))        — メモリーの値と 6 を加えます
(calc-exp2 '(Min (MR + 4))
                  — メモリーの値に 4 を加えた結果をメモリーに入れます
```

第三段階での文法 G2 と意味 S2 を、それぞれ図 11.10と図 11.11 に示します。

ここで使うメモリーは、プログラミング言語での変数とみなすことができます。使えるメモリーはひとつなので、ひとつの変数しか使えないことになります。メモリー操作のそれぞれの機能は、変数の立場から次のように見ることができます。

Min	変数への代入
MR	変数の値の参照
AC	(Min 0) の簡略記法

規則 G2-1: ⟨ 式 ⟩ := ⟨ 数 ⟩ | (⟨ 式 ⟩ ⟨ 演算子 ⟩ ⟨ 式 ⟩)
| AC | MR | (Min ⟨ 式 ⟩)
規則 G2-2: ⟨ 演算子 ⟩ := + | -
規則 G2-3: ⟨ 数 ⟩ := 0 | 1 | 2 | ··· | -1 | -2 | ···

図 11.10: 文法 G2

規則 S2-1: Eval(⟨ 数 ⟩) = ⟨ 数 ⟩が表す数値
規則 S2-2: Eval(AC) = 0
(メモリーの値を 0 にする)
規則 S2-3: Eval(MR) = メモリーの値
規則 S2-4: Eval((⟨ 式$_1$⟩ + ⟨ 式$_2$⟩)) = Eval(⟨ 式$_1$⟩) + Eval(⟨ 式$_2$⟩)
規則 S2-5: Eval((⟨ 式$_1$⟩ - ⟨ 式$_2$⟩)) = Eval(⟨ 式$_1$⟩) − Eval(⟨ 式$_2$⟩)
規則 S2-4: Eval((Min ⟨ 式 ⟩)) = Eval(⟨ 式 ⟩)
(Eval(⟨ 式 ⟩) の値をメモリーに代入する)

図 11.11: 意味規則 S2

メモリーの値を保持するために、ひとつの Scheme 変数 *memory* を使うことにします。calc-exp0 や calc-exp1 では、入力が数ならばその数をそのまま返していました:

```
(define (calc-exp1 exp)
  (cond
    ((number? exp)
     exp)
    ((list? exp)
      ... 省略...  )))
```

それにより、⟨ 数 ⟩ の場合での式の値を求めています。

これと同じようにして、AC と MR に対する動作を追加できます。もし入力の式が AC のときはメモリーの値をゼロにし、0 を値として返します。入力の式が MR のときは、メモリーの値を返します。

これに関連した部分を以下に示します。(手続き calc-exp3 の完全なプログラムリストはあとで示します。) ;* をつけた場所が、新しく追加された部分です。

```
(define *mameory* 0)    ;* メモリーの値を保持します。
(define (calc-exp3 exp)
  (cond
    ((eq? exp 'AC)      ;* 式が AC のときは、
     (set! *memory* 0)  ;*   メモリーをゼロにし、
     0)                 ;*   値として 0 を返します。
    ((eq? exp 'MR)      ;* 式が MR のときは、
     *memory*)          ;*   メモリーの値を返します。
    ((number? exp)
     exp)
    ((list? exp)
      ... 省略...  )))
```

あとは、(Min ⟨ 式 ⟩) に対する動作を追加するだけです。入力の式が (Min ⟨ 式 ⟩) の形をしていたら、calc-exp3 に ⟨ 式 ⟩ を与えて再帰的に呼び出します。そしてその計算結果を *memory* に代入し、(Min ⟨ 式 ⟩) の結果として ⟨ 式 ⟩ の計算結果を返します。

では calc-exp2 の全リストを以下に示します。calc-exp1 より新しく加わった部分には ;* の印をつけています。

```
(define *memory* 0)          ;*
(define (calc-exp2 exp)
  (cond
    ((eq? exp 'AC)           ;*
     (set! *memory* 0)       ;*
     *memory*)               ;*
    ((eq? exp 'MR)           ;*
     *memory*)               ;*
    ((number? exp)
     exp)
    ((list? exp)
     (case (length exp)
       ((2)                                      ;*
        (cond                                    ;*
          ((eq? (car exp) 'Min)                  ;*(Min ⟨ 式 ⟩) の計算
           (set! *memory* (calc-exp2 (cadr exp))) ;*
           *memory*)                             ;*
          (else (error "illegal expression"))))  ;*
       ((3)
        (let* ((n1 (calc-exp2 (list-ref exp 0)))
               (op (list-ref exp 1))
               (n2 (calc-exp2 (list-ref exp 2))))
          (if (and (number? n1) (number? n2)
                   (or (eq? op '+) (eq? op '-)))
            (case op
```

```
            ((+) (+ n1 n2))
            ((-) (- n1 n2)))
          (error "syntax error"))))
     (else (error "syntax error"))))
  (else (error "syntax error"))))
```

これを実行してみましょう。

```
> (calc-exp2 'AC)              — メモリーの値をゼロにします
0
> (calc-exp2 'MR)              — メモリーの値をみてみます
0
> (calc-exp2 '(Min (3 + 4)))   — 3 + 4 をメモリーに代入します
7
> (calc-exp2 'MR)              — メモリーの値を確認してみます
7
> (calc-exp2 '(Min (MR + (10 - 5))))
    — メモリの内容と (10 − 5) を加えたものを、メモリーに代入します
12
> (calc-exp2 'MR)              — メモリーの値をみてみましょう
12
```

他にも色々入力して、動作を試してみて下さい。

11.2.4 まとめ

本実習では、与えられた式を解釈して、その式を計算する手続きを作りました。その手続きは、式という単純な言語の言語処理系です。入力データ (式) は Scheme のリストで与えていたために、eq?, car, cdr などの手続きを使うことで簡単に構文解析をすることができました。

このように、 リストや対という Scheme のデータを使うことで、構造を持ったデータの入出力や、その構造の解析が驚くほど簡単になっています。C, Fortran, Pascal などのプログラミング言語を知っている読者には、特にそのことを強く実感できるものと思います。

練習問題

1. calc-exp2 が乗算 * を取り扱えるよう、改造しなさい。
2. calc-exp2 が除算 / を取り扱えるよう、改造しなさい。
3. * 上の 2 つの問題での改造に合わせて、文法 G2 と意味 S2 の定義も修正しなさい。

4. 円周を何度も計算するときは、円周率 3.14 を何度も入力しないといけません。この労力を省くために、calc-exp2 に記号定数の導入を考えます。これは、ある記号が現れると、その記号に対する値に置き換える機能です。たとえば、pi という記号があらわれると、3.14 に置き換えます。

```
> (calc-exp2 'pi)
3.14
> (calc-exp2 '(10 * pi))
31.4
> (calc-exp2 '(2 * pi))
6.28
```

calc-exp2 を改造して、この機能を追加しなさい。そのほか便利な定数をいくつか登録しなさい。

5. 正弦関数 $\sin x$ を (sin ⟨ 式 ⟩) の形で使えるよう、calc-exp2 を改造しなさい。

```
> (calc-exp2 '(sin 0))
0.0
> (calc-exp2 '(sin (pi / 6)))
499.770102643102e-3
```

6. ***** (この問題は構文解析についての基礎理論を必要とし、本書の内容を越えています。) 本実習では、入力となる式はリストや数などで与えていました。これを変更して、文字列として与えるようにします。

```
> (calc-str-exp "2")
2
> (calc-str-exp "(2 + 7)")
9
> (calc-str-exp "((3 + 2) - 4)")
1
```

この手続き calc-str-exp を実現しなさい。

C、Pascal, Fortan などのプログラミング言語を使って本実習と同じ内容のプログラムを作るのは、この問題と同程度の複雑な内容を含んでいます。calc-exp-str と calc-exp2 のプログラムは、それぞれ何行になったか数えてみなさい。

参考文献

- 武市 正人, “プログラミング言語”, 岩波講座ソフトウエア科学第 4 巻, 岩波書店, 1994 年.
- M. Hennessy (荒木 啓二郎, 程 京徳 訳), “プログラミング言語の意味論入門,” Information&Computing-76, サイエンス社, 1993 年.
- J. E. Hopcroft, J. D. Ullman (野崎 昭宏, 高橋 正子, 町田 元, 山崎 秀記 訳), “オートマトン 言語理論 計算論 I,” Information&Computing-3, サイエンス社, 1984 年.

この比、一期の藝能の定まる初めなり。さるほどに、稽古の堺なり。聲も既に直り、體も定まる時分なり。されば、この道に二つの果報あり。聲と身なりけり。これ二つは、この自分に定まるなり。年盛りに向ふ藝能の生づる所なり。

世阿弥 「風姿花伝」
昭和三十八年 岩波書店刊

第 12 章

Scheme インタプリタ

本章では Scheme インタプリタを作ります。Scheme がどのようにしてプログラムの実行を進めてゆくかを学ぶことで、Scheme に対する理解がより深くなると思います。

Scheme インタプリタは評価すべき式を解釈し、それに対応する動作をすることでプログラムが実行されます。Scheme インタプリタにとっては、実行しようとしているプログラムは、ただのデータにしかすぎません。ですがわたしたちには、Scheme プログラムが実行されているように見えます。

ここで作る Scheme インタプリタは Scheme で書きます。もしインタプリタを実行する処理系がインタプリタなら、インタプリタ上で (いまから作る) インタプリタプログラムが実行され、その上で Scheme プログラムを実行することになります。Scheme 自身で Scheme を記述するという点で巡回的ですが、混乱をしないように注意して下さい。

本実習で作る Scheme 処理系の名前は、TS (Tiny Scheme — 超小形 Scheme) とします。言語の仕様とインタプリタの概要を説明したあと、インタプリタの構成部分のそれぞれを順に作ってゆきます。

本実習では、TS で書かれたプログラムやデータは、大文字のサンセリフ体 (SANS SERIF) で書きます。TS の基盤となっている Scheme プログラムやデータは、従来通りタイプライタ体 (`typewriter`) で書き表します。

12.1 インタプリタの概要

12.1.1 言語仕様

TS の言語仕様について説明します。Scheme 処理系の核となる部分だけを作り、処理系をコンパクトにします。いろいろな機能の拡張は練習問題とします。

取り扱えるデータの型は、ブール値、整数、記号、対、手続きだけとします。

特殊形式には、以下のものを作ります。

- (QUOTE ⟨ データ ⟩)
  データの評価をしないための特殊形式です。

- (IF ⟨ 条件式 ⟩ ⟨ 式$_1$⟩ ⟨ 式$_2$⟩)
  条件式の値によって、評価する式を選択します。(IF ⟨ 条件式 ⟩ ⟨ 式$_1$⟩) という形は使えません。

- (BEGIN ⟨ 式$_1$⟩ ⟨ 式$_2$⟩ $\cdots$)
  ならんだ式を順に評価し、最後の式の評価結果を返します。

- (DEFINE ⟨ 変数 ⟩ ⟨ 式 ⟩)
  変数の定義をします。手続きの定義をするときは、(DEFINE ⟨ 変数 ⟩ (LAMBDA $\cdots$)) とします。簡略記法の (DEFINE (⟨ 変数 ⟩ ⟨ 引数 ⟩) ⟨ 本体 ⟩) という書き方は許されていません。

- (SET! ⟨ 変数 ⟩ ⟨ 式 ⟩)
  変数の値を書き換えます。

- (LAMBDA ⟨ 引数リスト ⟩ ⟨ 式$_1$⟩ ⟨ 式$_2$⟩ $\cdots$)
  複合手続きデータを作ります。

基本手続きは、NULL?, EQ?, CONS, CAR, CDR, PAIR?, +, −, =, >, QUIT を用意します。手続き QUIT は、TS を終了させる TS 独自の手続きです。

以上の言語仕様に従って、いくつかの Scheme プログラムを書いてみましょう。Scheme の標準的な手続き LIST? と LENGTH の実現は次のようになります。TS の言語仕様にはエラーを発生させる手続きがないので、正しくない入力のときは偽 #F を返すようにしています。

```
(DEFINE LIST?
  (LAMBDA (S)
    (IF (NULL? S)
      #T
      (IF (PAIR? S)
        (LIST? (CDR S))
        #F))))
(DEFINE LENGTH
  (LAMBDA (S)
    (DEFINE LENGTH-LOOP
      (LAMBDA (REST LEN)
        (IF (NULL? REST)
```

```
        LEN
        (LENGTH-LOOP (CDR REST) (+ LEN 1)))))
  (IF (LIST? S)
    (LENGTH-LOOP S 0)
    #F)))
```

このように、基本的な機能を組み合わせることでより複雑な機能をつくり、言語の機能を拡張してゆくことができます。

12.1.2 インタプリタの構成

本実習で作る Scheme インタプリタは、大きく分けて以下のものより構成されます。(図12.1参照)

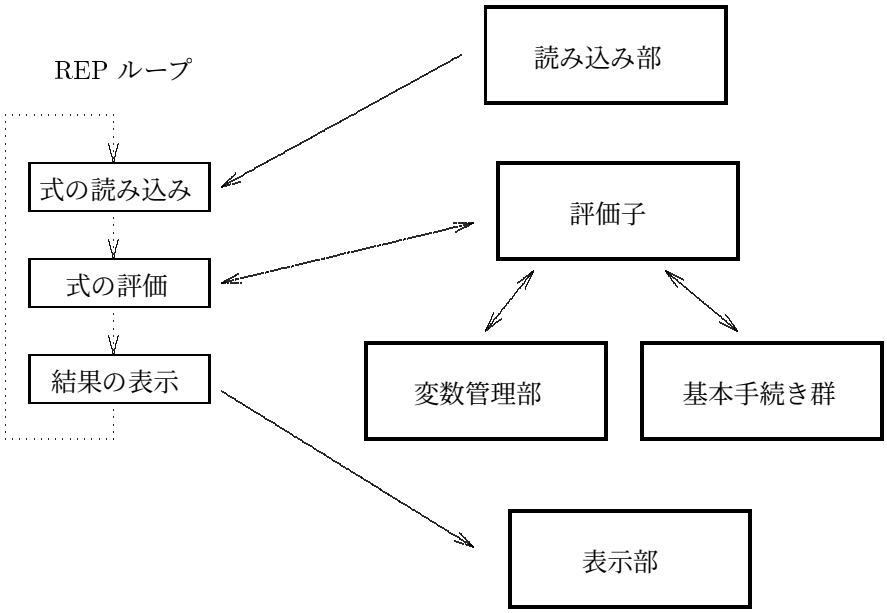


図 12.1: インタプリタの構造

1. **読み込み-評価-表示ループ**
   式の読み込み (Read)、 評価 (Evaluate)、結果の表示 (Print) を永遠に繰り返すループです。Read-Evaluate-Print の頭文字をとって、 REP ループともよばれます。式を読み込むときは、インタプリタ内部でのデータ表現に変換します。

2. **変数の管理機構**
   変数名とその値の組を保持する表を用意します。変数値を参照するときはその表を調べます。局所変数を取り扱う機構も必要です。

3. **評価子**
   実際に式を評価する部分です。評価するデータの種類を調べ、データ型に従って評価をします。たとえば手続き呼び出しのときは、引数を評価して手続きを呼び出します。

4. **基本手続き**
   cons, car, + などの基本的な手続きは、インタプリタがあらかじめ用意する**基本手続き** (primitive procedure) として準備します。

12.2 インタプリタプログラム

12.2.1 インタプリタの初期化と REP ループ

手続き ts を呼び出すことで、インタプリタが始動するものとします。手続き ts は、環境 (変数表) の初期化と基本手続きの登録をし、REP ループを呼び出します。

```
;;; 手続き ts --- インタプリタを始動する
(define (ts)
  (newline)
  (display "TS (TINY SCHEME IN SCHEME)")
  (newline)
  (ts:reset-environ)                    ; 環境を初期化する
  (ts:intern-primitive-procedures)      ; 基本手続きの登録をする
  (ts:read-eval-print-loop)             ; REP ループに入る
  #t)
```

次は REP ループです。これは図 12.1での「REP ループ」に相当します。

TS で実行するプログラムに間違いがあったときは、「エラー」として実行を中断する必要があります。TS のトップレベルに戻ってエラーメッセージを表示し、別の式の入力を待ちます。手続き ts:read-eval-print-loop は、トップレベルに戻るための継続を引数にして手続き ts:read-eval-print を呼び出します。インタプリタを終了する TS の手続き QUIT が呼ばれるときに限って、継続呼び出しの値は偽#f を返すようにします。その場合はインタプリタを終了します。それ以外のときは REP を繰り返し、REP ループを実現します。

```
; トップレベルへの継続（エラー時などに、非局所的な手続き脱出に使われる）
(define *ts:top-level-continuation* #f)
;;; REP ループ
(define (ts:read-eval-print-loop)
```

```
  (letrec
    ((loop    ; REP ループ
       (lambda ()
         ; 手続き ts:read-eval-print を継続付きで呼び出す
         (if (call-with-current-continuation ts:read-eval-print)
           (loop)       ; さらに、REP をする。
           (begin       ; #f が返されると、インタプリタを終了する
             (display "GOOD BYE.")
             (newline))))))
    (loop)))  ; REP ループに入る
```

手続き ts:read-eval-print は一回だけ REP をする手続きです。変数 *ts:top-level-continuation* に、エラーからの脱出のための継続を代入します。そして、以下のことをします。

1. 手続き ts:prompt により、プロンプトを表示します。
2. 手続き ts:read-user-input により、式を読み込みます。
3. 手続き ts:eval により、式を評価します。
4. 手続き ts:print-value により、評価結果を表示します。

以下で使われている手続き ts:top-environ は、トップレベルでの環境を返す手続きです。手続き force-output は、処理系によっては用意されていないことがあります。その場合は、変数*ts:scheme-system-has-force-output* の値を#f にして下さい。(force-output について、139ページも参照して下さい。)

```
(define (ts:read-eval-print cont)
  (set! *ts:top-level-continuation* cont)
  (ts:prompt)  ; プロンプトを表示する。
  (let ((user-input (ts:read-user-input)))  ; 式を読み込む
    (let ((val (ts:eval user-input (ts:top-environ))))
                                  ; トップレベルの環境で式を評価する
      (ts:print-value val)        ; 評価結果を表示する
      #t)))
(define *ts:scheme-system-has-force-output* #t) ; 処理系に合わせる
(define (ts:prompt)
  (display "] ")
  (if *ts:scheme-system-has-force-output*
```

```
      (force-output))
  #t)
```

エラーが生じたときは、次に示す手続き ts:error を呼び出します。この手続きはエラーメッセージを表示し、TS のインタプリタのトップレベルに戻ります。ts:error は任意の数の引数を受けとることができるよう、(lambda msg ...) という形にしていることに注意して下さい。

```
(define ts:error
  (lambda msg
    (letrec
      ((loop
         (lambda (s)
           (if (null? s)
             (begin (newline) (newline)
               (*ts:top-level-continuation* #t)) ; トップレベルへ
             (begin
               (display (car s)) (display " ")
               (loop (cdr s)))))))
      (newline)
      (display "TS ERROR")
      (newline)
      (loop msg))))
```

12.2.2 インタプリタ内部のデータ表現

TS は Scheme で記述しているので、実現の土台となっている Scheme と実現しようとしている Scheme (TS) とを混同してしまいがちです。これ以降では、実現の土台となっている Scheme でのデータを「Scheme データ」と呼び、実現しようとしている Scheme (TS) でのデータを「TS データ」と呼ぶことで、それら 2 つを区別することにします。

ここでは、 TS データをどのようにして Scheme データとして実現するかを説明します。TS データは、それがどの型のデータであるかを表す「印」を持っています。これは 5.1「有理数計算手続きの製作」で使った技法です。空リスト、ブール型、整数、記号は、(cons ⟨ 型の印 ⟩ ⟨Scheme データ ⟩) として TS データを作ります。他の型についても、car 部はデータの印となるようにします。そうすることで、データの型は印を見るだけで分かります。

たとえば TS データ 2001 は、Scheme データで (TS-TAG:INTEGER . 2001) と表されます。TS データ #T は、(TS-TAG:BOOLEAN . #t) となります。

以下にそれぞれの型の実現を示します。

空リスト

変数 *ts:null-obj* は、TS データとしての空リストを保持します。手続き ts:null? は、TS での空リストかどうかを判定します。

```
(define *ts:null-obj*     (cons 'TS-TAG:NULL '()))
(define (ts:null? ts-obj) (eq? ts-obj *ts:null-obj*))
```

ブール型

変数 *ts:true* と *ts:false* はそれぞれ、TS データの真と偽を保持します。手続き ts:make-boolean は、Scheme データの真/偽から TS データの真/偽を作ります。

```
(define *ts:true*         (cons 'TS-TAG:BOOLEAN #t))
(define *ts:false*        (cons 'TS-TAG:BOOLEAN #f))
(define (ts:make-boolean scheme-obj)
  (if scheme-obj
    *ts:true*
    *ts:false*))
```

手続き ts:boolean? は、TS でのブール型のデータかどうかを判定します。

```
(define (ts:boolean? ts-obj) (eq? (car ts-obj) 'TS-TAG:BOOLEAN))
```

整数

手続き ts:make-integer は、Scheme データの整数から TS データ整数を作ります。手続き ts:integer? は、TS での整数型のデータかどうかを判定します。

```
(define (ts:make-integer n)  (cons 'TS-TAG:INTEGER n))
(define (ts:integer? ts-obj) (eq? (car ts-obj) 'TS-TAG:INTEGER))
```

記号

手続き ts:make-symbol は、Scheme データの記号から TS データの記号を作ります。手続き ts:symbol? は、TS での記号型のデータかどうかを判定します。

```
(define (ts:make-symbol scheme-obj)
  (cons 'TS-TAG:SYMBOL scheme-obj))
(define (ts:symbol? ts-obj)
  (eq? (car ts-obj) 'TS-TAG:SYMBOL))
```

以上が TS での基本的な TS データです。TS データの Scheme データとしての値が必要な場合は、TS データの cdr 部を調べればいいことになります。次の手続き ts:get-scheme-value は、基本的な TS データが表す Scheme データとしての値を取り出します。

```
(define (ts:get-scheme-value ts-obj) (cdr ts-obj))
```

対

手続き ts:cons は、2 つの引数を対にします。手続き ts:pair? は、TS データが対かどうかを判定します。手続き ts:car と ts:cdr は、ts:cons による対の作り方に合わせて、car 部と cdr 部を取り出します。これらは (ts:car (ts:cons s1 s2)) = s1 かつ (ts:cdr (ts:cons s1 s2)) = s2 が成り立つように作らないといけません。

```
(define (ts:cons s1 s2)   (cons 'TS-TAG:PAIR (cons s1 s2)))
(define (ts:pair? ts-obj) (eq? (car ts-obj) 'TS-TAG:PAIR))
(define (ts:car s)    (cadr s))
(define (ts:cdr s)    (cddr s))
```

この他、ts:car と ts:cdr を組み合わせた手続きも作っておきます。

```
(define (ts:cadr s)   (ts:car (ts:cdr s)))
(define (ts:cddr s)   (ts:cdr (ts:cdr s)))
(define (ts:caddr s)  (ts:car (ts:cddr s)))
(define (ts:cadddr s) (ts:cadr (ts:cddr s)))
```

複合手続き

手続き ts:make-compound-procedure によって、λ式 (LAMBDA (X1 X2 ···) EXP1 EXP2 ···) に対する複合手続きデータが作られます。

手続き ts:make-compound-procedure には、ts-parameter-list, body, env の 3 つの仮引数があります。仮引数 ts-parameter-list はλ式の引数リスト (X1 X2 ···) が、仮引数 body はλ式の本体 (EXP1 EXP2 ···) が与えられます。env は、λ式が評価される環境です。環境とは変数の値を決めるものです。手続きデータにλ式の本体だけでなく環境も含めることで、

- 局所変数を持つオブジェクト (9.1「自動販売機のシミュレート」参照)
- 静的有効域則

が実現できます。(これらは後の節で詳細に説明します。)

以下に示す手続きでは、引数渡しを簡略にするために、引数リストを逆順にして複合手続きデータに保持させています。手続き呼び出しのときには、与えられた実引数の数とλ式で定義された仮引数の数が合っていないといけません。このチェックのために、複合手続きデータにその引数の数も記録しておきます。

```
(define (ts:make-compound-procedure ts-parameter-list body env)
  (letrec
    ((param-rev-loop
       (lambda (ts-params rev-params nargs)
         (if (ts:null? ts-params)
           (list 'TS-TAG:COMP-PROC rev-params nargs body env)
           (param-rev-loop
             (ts:cdr ts-params)
             (cons (ts:car ts-params) rev-params)
             (+ nargs 1))))))
    (param-rev-loop ts-parameter-list '() 0)))
```

複合手続きデータかどうかを判定する手続きは次の通りです。

```
(define (ts:compound-procedure? ts-obj)
  (eq? (car ts-obj) 'TS-TAG:COMP-PROC))
```

この他、複合手続きデータからその手続きの引数リストを取り出す手続きなどを作ります。

```
(define (ts:get-arglist-compound-procedure proc)
  (list-ref proc 1))
(define (ts:get-nargs-compound-procedure proc)
  (list-ref proc 2))
(define (ts:get-body-compound-procedure proc)
  (list-ref proc 3))
(define (ts:get-env-compound-procedure proc)
  (list-ref proc 4))
```

基本手続き

手続き ts:make-primitive-procedure は、TS の基本手続きを作る手続きです。この手続きは、TS の基本手続きを実現する Scheme 手続き、TS での登録名、そして引数の数を引数としています。

```
(define (ts:make-primitive-procedure proc name nargs)
  (list 'TS-TAG:PRIM-PROC proc name nargs))
```

次の手続きにより、引数に与えられたデータが基本手続きデータであるかどうかを判定します。

```
(define (ts:primitive-procedure? ts-obj)
  (eq? (car ts-obj) 'TS-TAG:PRIM-PROC))
```

以下の手続きはそれぞれ、基本手続きデータから基本手続きを実現する Scheme 手続き、TS での登録名、そして引数の数を取り出す手続きです。

```
(define (ts:get-body-primitive-procedure proc)   (cadr proc))
(define (ts:get-name-primitive-procedure proc)   (caddr proc))
(define (ts:get-nargs-primitive-procedure proc)  (cadddr proc))
```

Scheme データから TS データへの変換

以下に示す ts:scheme-obj->ts-obj を使って、Scheme データから TS データへの変換をします。与えられた Scheme データの型に従って、TS データを作る手続きを呼び出しています。対の場合は、 car 部と cdr 部それぞれに対して、再帰的に TS データに変換します。

```
(define (ts:scheme-obj->ts-obj scheme-obj)
  (cond
    ((null? scheme-obj)    *ts:null-obj*)
    ((boolean? scheme-obj) (ts:make-boolean scheme-obj))
    ((integer? scheme-obj) (ts:make-integer scheme-obj))
    ((symbol? scheme-obj)  (ts:make-symbol  scheme-obj))
    ((pair? scheme-obj)
      (ts:cons (ts:scheme-obj->ts-obj (car scheme-obj))
               (ts:scheme-obj->ts-obj (cdr scheme-obj))))
    (else (ts:error "ILLEGAL INPUT:" scheme-obj))))
```

12.2.3 式の読み込みと表示

式を読み込む手続き ts:read-user-input は次の通りです。これは図 12.1での「読み込み部」に相当します。手続き read を呼び出すことで TS が評価すべき式を Scheme のデータとして読み込み、それを TS の内部データ表現に変換します。

```
(define (ts:read-user-input)
  (let ((scheme-obj (read)))   ; まず Scheme データとして読み込む
    (if (eof-object? scheme-obj)
      ; 入力が終りならトップレベルへ #f を返す。(ts が終了する)
      (*ts:top-level-continuation* #f)
      ; 入力があれば、データ変換したものを返す
      (ts:scheme-obj->ts-obj scheme-obj))))
```

データの表示について説明します。これは図 12.1での「表示部」に相当します。手続き ts:print-exp を呼び出して式を表示し、改行します。手続き ts:print-exp は、データの型を調べ、それぞれの型に応じて表示をします。もし数などの基本的なデータの場合は、TS データから Scheme データを取り出して表示します。対の場合は、car 部と cdr 部それぞれに対して再帰的に表示します。

```
(define (ts:print-value ts-exp)
  (ts:print-exp ts-exp)
  (newline))
(define (ts:print-exp ts-exp)
  (cond
    ((or (ts:boolean? ts-exp) (ts:integer? ts-exp)
       (ts:symbol? ts-exp) (ts:null? ts-exp))
     (display (ts:get-scheme-value ts-exp)))
    ((ts:primitive-procedure? ts-exp)
     (display "<primitive-procedure:")
     (display (ts:get-name-primitive-procedure ts-exp))
     (display ">"))
    ((ts:compound-procedure? ts-exp)
     (display "<compound-procedure>"))
    ((ts:pair? ts-exp)
     (display "(")
     (ts:print-exp (ts:car ts-exp))
     (display " . ")     ; ドット記法による表示
     (ts:print-exp (ts:cdr ts-exp))
     (display ")"))))
```

12.2.4 評価子 (その 1)

ではいよいよ評価子の作成に入ります。これは図 12.1での「評価子」に相当します。式の評価の方法の基本的な考え方は、11.2「プログラミング言語処理系の基礎 — 式の計算」で学んだ通りです。ここでは新たに、変数への参照、特殊形式の実行、手続き呼び出しが加わります。

評価子を実現する手続き ts:eval には、2 つの仮引数 exp と env があります。仮引数 exp には、評価しようとしている式が与えられます。仮引数 env には、式の評価をする「環境」が与えられます。環境とは、変数がどの値を持っているかを表すものです。環境については、次節で詳しく説明します。

評価子は式の型で場合分けをし、型ごとに対応した動作をします。空リスト、ブール値、

整数は自己評価的データなので、与えられたものをそのまま返します。記号の場合は、手続き ts:lookup-binding によって環境 env での変数値を取り出し、その値を評価値とします。対の場合は、特殊形式か手続き呼び出しです。

手続き ts:special-form? により、特殊形式かどうかを判定します。手続き ts:do-special-form は特殊形式を実行し、手続き ts:do-application は手続きを呼び出します。これらの手続きは後の節で説明します。

```
(define (ts:eval exp env)
  (cond
    ((ts:null? exp)          ;** 空リスト
     exp)                    ; 自己評価的データなので、それ自身を返す
    ((ts:boolean? exp)       ;** ブールデータ
     exp)                    ; 自己評価的データなので、それ自身を返す
    ((ts:integer? exp)       ;** 整数
     exp)                    ; 自己評価的データなので、それ自身を返す
    ((ts:symbol? exp)        ;** 記号
     (ts:lookup-binding      ; 束縛を調べて、変数値を見つける
       exp env))
    ((ts:pair? exp)          ;** 対 （特殊形式または手続き呼び出し）
     (if (ts:special-form? exp)
       (ts:do-special-form exp env)   ; 特殊形式
       (ts:do-application exp env)))  ; 手続き呼び出し
    (else                    ; 出てくるはずのないデータ
      (ts:error "ILLEGAL OBJECT:" exp))))
```

12.2.5 変数とその値 (原理)

特殊形式と手続き呼び出しについて説明する前に、変数の管理と環境について説明します。ここでは TS での変数の管理方法の概念について説明し、実現方法は後の節で示します。この部分は図 12.1での「変数管理部」に当たります。

変数の値は A リストに似た、変数と値の組をリストにして保持します。たとえば変数 X, Y の値がそれぞれ 1, 2 のときは、（（X . 1）（Y . 2））と表現します。このように変数名が car 部、値が cdr 部である対をリストにして、複数の変数の値を保持します。変数名と値の対を **束縛** (binding)、束縛のリストを **枠組** (frame) とそれぞれ呼ぶことにします。

しかしこれだけでは不十分です。たとえば次の Scheme プログラムを見て下さい。

```
(DEFINE X 1)
(DEFINE Y 2)
(DEFINE F
```

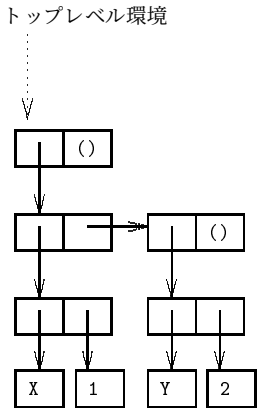


図 12.2: トップレベル環境の例

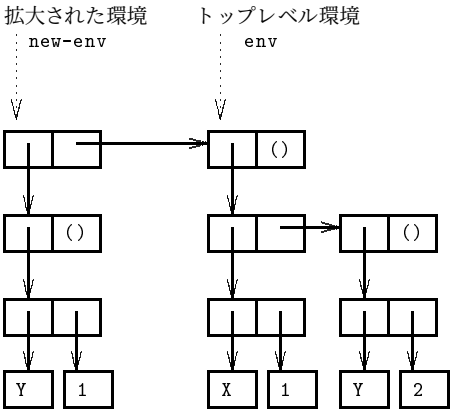


図 12.3: 手続き F の本体の評価のために拡大した環境

(LAMBDA (Y) (+ Y 2000)))

この例では、3 つの変数 X, Y, F がトップレベルで定義されています。(F は手続きです。) この状態での枠組は、((X . 1) (Y . 2) (F . ⟨ 手続き ⟩)) となっています。このことから分かるように、枠組が変数の値を決めています。

ここで式 (F 1) の評価を考えます。手続き F に与える実引数は 1 で、仮引数 は Y となっています。そのため手続き F の中では、変数 Y の値は 1 でないといけません。このため、F の本体の評価で使う環境に、束縛 (Y . 1) が必要となります。さらに、手続き F の実行が終ったあとでは、変数 Y の値は 1 でないといけません。(Y の値を変更しっぱなしの方法ではいけません。)

ここでは枠組のリストにすることで問題を解決します。このように変数の値を決定するものを、**環境** (environment) と呼びます。

トップレベルで環境の例を、図 12.2に示します。(実際には多くの基本手続きがトップレベルで定義されていますが、ここでは省略しています。)

手続き F が呼ばれるときは、F の仮引数と実引数の値による新しい枠組を作ります。その新しい枠組が最初の要素となるよう、環境に加えます。すなわち、(⟨F の引数の枠組 ⟩ ⟨ トップレベルでの枠組 ⟩) であるリストにします。今の例では、⟨F の引数の枠組 ⟩ は ((Y . 1)) です。このように、環境に新しい枠組を加えることを**環境の拡大**と呼びます。図 12.2での環境を拡大した環境を、図 12.3に示します。

手続き F の本体の式は、拡大された環境で評価されます。変数の値の検索は、まず最初に環境の最初の枠組を探します。もし見つからなければ、環境の二番目の枠組を探します。このように、変数が見つかるまで枠組を順に検索します。そのため本体の式にある変数 Y の値は 10 となり、トップレベルでの定義である 2001 とはなりません。

式 (+ Y 2000) の評価を考えてみましょう。図 12.3での環境 `new-env` でこの式を評価す

れば、結果は 2001 になります。一方、環境 env で評価すれば、結果は 2002 になります。これらから分かるように、どの環境にあるかによって変数の値は違います。このことが、評価子 ts:eval に与える引数は評価する式だけでなく、式を評価する環境も必要である理由です。

環境の操作と変数参照に関して、次の手続きを用意します。実現については、後の節で示します。

- (ts:reset-environ)
  トップレベル環境を、何も定義されていない状態にします。

- (ts:top-environ)
  トップレベル環境を返します。

- (ts:lookup-binding var env)
  環境 env での変数 var の値を返します。環境の最初の枠組から順番に探索されます。もし見つからなければ、エラーとなります。

- (ts:define-var var val env)
  環境 env で、変数 var を値 val として定義します。環境の最初の枠組で、変数は定義されます。

- (ts:set-var! var val env)
  環境 env での変数 var の値を、 val に変更します。変数は環境の最初の枠組でなくてもかまわず、見つかった枠組での束縛を変更します。

- (ts:extend-environ vars vals env)
  環境の拡大をし、拡大した環境を結果として返します。この手続きはまず、変数のリスト vars とその値のリスト vals のそれぞれの第 i 要素を組にして枠組を作ります。その枠組 env の先頭に付けて、拡大した枠組を返します。これにより、新しい枠組が新しい環境の第一要素となります。この手続きは、複合手続きが呼び出されたとき、引数を渡すために使われます。

12.2.6 評価子 (その 2 特殊形式)

手続き ts:special-form? は、与えられた式が特殊形式かどうかを調べます。これは式の car 部が特殊形式のキーワードとなっているかどうかで判定をします。

```
(define (ts:special-form? exp)
  (case (ts:get-scheme-value (ts:car exp))
    ((QUOTE IF BEGIN DEFINE SET! LAMBDA) #t)
    (else                                #f)))
```

手続き ts:do-special-form にて特殊形式の実行をします。それぞれ、次の動作をします。

- (QUOTE ⟨ データ ⟩)
  ⟨ データ ⟩ そのものを返します。これにより、⟨ データ ⟩ を評価しない、ということが実現されます。

- (IF ⟨ 条件式 ⟩ ⟨ 式$_1$⟩ ⟨ 式$_2$⟩)
  ⟨ 条件式 ⟩ の評価結果が偽でなければ ⟨ 式$_1$⟩を評価し、その結果を IF の評価結果とします。それ以外のときは ⟨ 式$_2$⟩ を評価し、その結果を IF の評価結果とします。

- (BEGIN ⟨ 式$_1$⟩ ⟨ 式$_2$⟩ ···)
  式のならび ⟨ 式$_1$⟩, ⟨ 式$_2$⟩ ···を順番に評価し、最後の式の評価結果を BEGIN の評価結果とします。式のならびを評価するのに手続き ts:eval-begin を使います。

- (DEFINE ⟨ 変数 ⟩ ⟨ 式 ⟩)
  ⟨ 式 ⟩ の評価結果が ⟨ 変数 ⟩ の値となるよう、現在の環境の最初の枠組で定義します。DEFINE の結果として変数名を返します。

- (SET! ⟨ 変数 ⟩ ⟨ 式 ⟩)
  ⟨ 式 ⟩ を評価し、その結果を ⟨ 変数 ⟩の新しい値として代入します。⟨ 変数 ⟩ の束縛を環境の中から見つけて、値を書き換えます。そのため、⟨ 変数 ⟩ はすでに定義されている必要があります。DEFINE の結果として新しい値を返します。

- (LAMBDA ⟨ 引数リスト ⟩ ⟨ 式$_1$⟩ ⟨ 式$_2$⟩ ···)
  ts:make-compound-procedure を呼び出して、複合手続きデータを作ります。これに与える第一引数は手続きの仮引数のリスト、第二引数は手続きの本体、第三引数はこのλ式が評価された環境です。複合手続きデータに作られたときに環境も一緒に持たせることで、局所変数や静的有効域則が実現されます。

```
(define (ts:do-special-form exp env)
  (case (ts:get-scheme-value (ts:car exp))
    ((QUOTE)                    ;** (QUOTE OBJ) の場合
     (ts:cadr exp))
    ((IF)                       ;** (IF CON EXP1 EXP2) の場合
     (let ((con (ts:eval (ts:cadr exp) env)))
       (if (not (eq? con *ts:false*))
         (ts:eval (ts:caddr exp) env)
         (ts:eval (ts:cadddr exp) env))))
    ((BEGIN)                    ;**(BEGIN EXP1 ...) の場合
```

```
     (ts:eval-begin (ts:cdr exp) env))
    ((DEFINE)                    ;**(DEFINE VAR EXP) の場合
     (ts:define-var
       (ts:cadr exp) (ts:eval (ts:caddr exp) env) env)
     (ts:cadr exp))
    ((SET!)                      ;**(SET! VAR EXP) の場合
     (let ((val (ts:eval (ts:caddr exp) env)))
       (ts:set-var! (ts:cadr exp) val env)
       val))
    ((LAMBDA)                    ;**(LAMBDA (VAR) EXP1...) の場合
     (ts:make-compound-procedure
       (ts:cadr exp) (ts:cddr exp) env))
    (else                        ; 出てくるはずのないデータ
      (ts:error "ILLEGAL SPECIAL FORM:" exp))))
```

次は (BEGIN $\langle \exp_1 \rangle \cdots$) の本体を順に評価する手続き ts:eval-begin を作ります。仮引数 exps は式のならび $\langle \exp_1 \rangle \cdots$です。これから式を順に取り出し、ts:eval を呼びます。

一番最後の式の場合は特別扱いし、直接 ts:eval を呼び出しています。(この手続きは ts:do-application-compount-procedure の中で、複合手続きの本体の実行のためにも呼び出されます。)

```
(define (ts:eval-begin exps env)
  (letrec
    ((begin-loop
       (lambda (rest last)
         (if (ts:null? (ts:cdr rest))
             (ts:eval (ts:car rest) env)
             (begin-loop
              (ts:cdr rest)
              (ts:eval (ts:car rest) env))))))
    (if (ts:null? exps)
        *ts:null-obj*
        (begin-loop exps *ts:null-obj*))))
```

12.2.7 手続き呼び出し

ここでは手続き呼び出しについて説明します。手続きには 2 種類あります。利用者が (LAMBDA $\cdots$) によって作る複合手続きと、インタプリタにあらかじめ組み込まれている

基本手続きです。手続き ts:do-application は最初、後で説明する手続き ts:eval-args を呼び出して引数を評価します。そして、基本手続きか複合手続きかによって、対応した手続き呼び出しをする手続きを呼び出します。実際に呼び出しをする手続きの詳細は、後の節で示します。

```
(define (ts:do-application exp env)
  (let ((proc (ts:eval (ts:car exp) env)) ; 手続きデータを取り出す
        (args (ts:eval-args (ts:cdr exp) env))) ; 引数を評価する
    (cond
      ((ts:compound-procedure? proc)      ;** 複合手続きの場合
       (ts:do-application-compound exp env proc args))
      ((ts:primitive-procedure? proc)     ;** 基本手続きの場合
       (ts:do-application-primitive exp env proc args)))))
```

手続き ts:eval-args は、手続きへの引数を評価します。たとえば、式 (FOO EXP1 EXP2 EXP3) の評価では、ts:eval-args の仮引数 args に引数のリスト (EXP1 EXP2 EXP3) が与えられます。この手続きは、args のそれぞれの要素を評価したものを Scheme のリストにして返します。返された結果のリストでの評価結果の順番は、仮引数 args での順番の逆にしていることに注意して下さい。上の例の場合では、(⟨EXP3 の評価結果 ⟩ ⟨EXP2 の評価結果 ⟩ ⟨EXP1 の評価結果 ⟩) が返されることになります。

```
(define (ts:eval-args args env)
  (letrec
    ((ev-arg-loop
       (lambda (args results)
         (if (ts:null? args)
           results             ; 反転したままの結果のリストを返す
           (ev-arg-loop
             (ts:cdr args)
             (cons (ts:eval (ts:car args) env) results))))))
    (ev-arg-loop args '())))
```

12.2.8 基本手続きの呼び出し

基本手続きの呼び出しをするための手続き ts:do-application-primitive を作ります。apply を使い、環境と引数を「Scheme で実現された TS の基本手続きを実現する手続き」の引数として呼び出します。そのため環境の拡大はしません。

```
(define (ts:do-application-primitive exp env proc args)
```

```
(let ((proc-nargs (ts:get-nargs-primitive-procedure proc)))
  (if (or
        (eq? 'any proc-nargs)      ; 引数の数の整合性をチェック
        (= (length args) proc-nargs))
    (apply                         ; 引数の数が合っていれば、
                                   ;    基本手続きを呼び出す。
      (ts:get-body-primitive-procedure proc)
      (reverse args))
    (ts:error
      "ILLEGAL NUMBER OF ARGS TO"
      (ts:get-name-primitive-procedure proc)))))
```

12.2.9 複合手続きの呼び出し: 局所変数と文面的有効域則

局所変数を持ったオブジェクトと、文面的有効域則が実現できる理由をここで説明します。たとえば次の手続きを見て下さい。これは 9.1「自動販売機のシミュレート」(186ページ) で紹介した、財布オブジェクトを作る手続きと同等なものです。ここでは手続き定義での DEFINE の簡略表記 (DEFINE (PROC ARG) ···) は使わずに、(DEFINE PROC (LAMBDA (ARG) ···)) の形で書いています。

```
(DEFINE MAKE-WALLET
  (LAMBDA (MONEY)
    (LAMBDA (AMOUNT)
      (IF (> AMOUNT MONEY)
        "NOT ENOUGH MONEY"
        (BEGIN
          (SET! MONEY (- MONEY AMOUNT))
          MONEY)))))
```

この手続きに対して、式 (MAKE-WALLET 100) を評価したときに返される値は、(LAMBDA (AMOUNT) (IF ···)) が評価された結果の複合手続きデータです。(DEFINE W1 (MAKE-WALLET 100)) によって複合手続きデータを変数 W に定義したときの様子を、図 12.4に示します。(枠組の詳細は簡略化して書いています。) 複合手続き型の実現の所で説明したように、W1 には λ式が評価されたときの環境もパックされています。

ここで、式 (W1 10) の評価を考えてみましょう。W1 の値は複合手続きデータです。まず複合手続きデータから環境を取り出し、その環境に対して引数渡しによる環境の拡大をし、拡大された環境で λ式の本体 (今の例では (IF (>AMOUNT MONEY) ···)) を評価すればいいことになります。

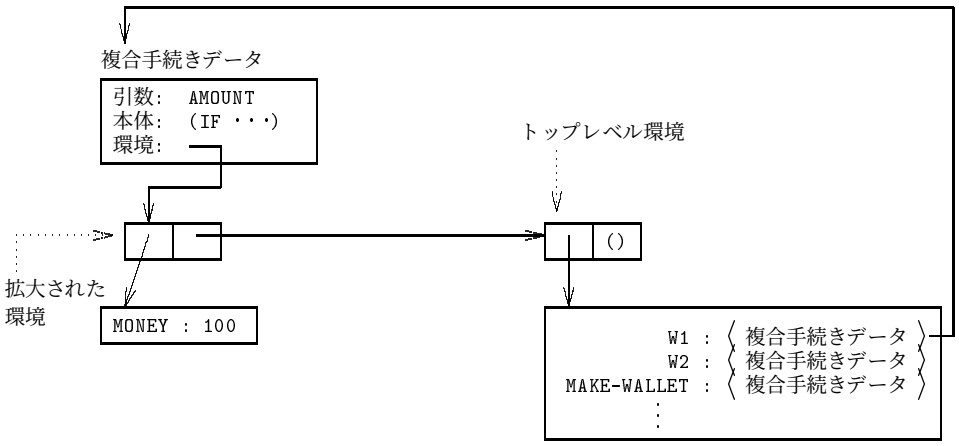


図 12.4: (DEFINE W1 (MAKE-WALLET 100)) を実行

複合手続きを呼び出すための手続き ts:do-application-compound を、上で説明したことに基いて作ります。これはまず最初、仮引数と実引数との枠組を作ります。次に複合手続きが作られたときの環境を拡大します。そして手続きの本体を、拡大した環境で評価します。

```
(define (ts:do-application-compound exp env proc args)
  (if (= (length args)                ; 引数の数の整合性をチェック
         (ts:get-nargs-compound-procedure proc))
      (let ((new-env                  ; 引数の数が合っていれば、
             (ts:extend-environ       ;   環境を拡大し、
               (ts:get-arglist-compound-procedure proc)
               args (ts:get-env-compound-procedure proc))))
        (ts:eval-begin                ;   手続きの本体を評価する
          (ts:get-body-compound-procedure proc) new-env))
      (ts:error
        "ILLEGAL NUMBER OF ARGS TO"
        (ts:get-name-primitive-procedure proc))))
```

次に環境の拡大をする手続き ts:extend-environ を示します。この手続きは (ts:extend-environ vars vals env) の形で呼ばれます。この手続きの仮引数 vars は複合手続きの仮引数のリストで、仮引数 vals は複合手続きに与える (評価済みの) 実引数のリストです。

vars を (VAR1 VAR2 ・・・)、vals を (VAL1 VAL2 ・・・) とします。するとこの手続きは、枠組 ((VAR1 . VAL1) (VAR2 . VAL2) ・・・) を作り、環境 env にこの枠組を追加することで環境を拡大します。この手続きは拡大した環境を返します。

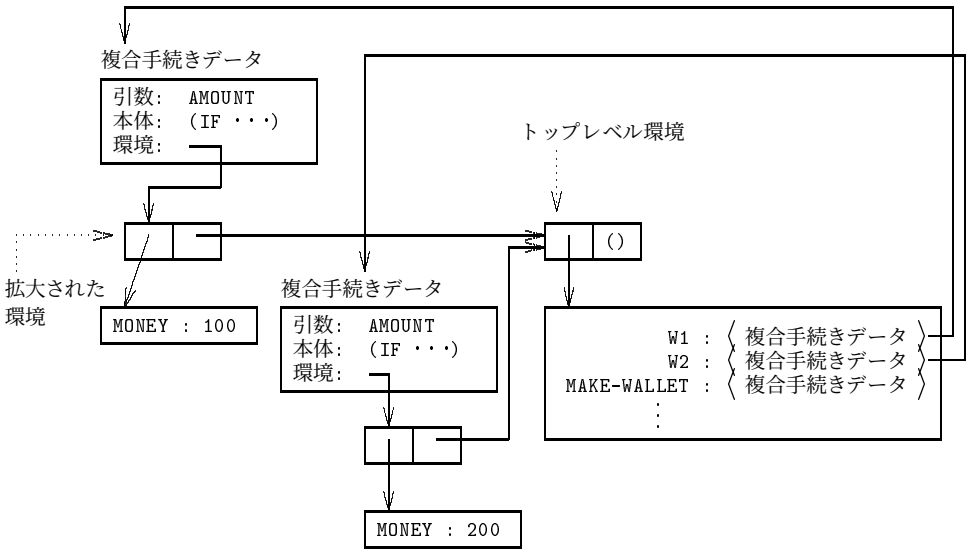


図 12.5: 続いて (DEFINE W2 (MAKE-WALLET 200)) を実行

```
(define (ts:extend-environ vars vals env)
  (letrec
    ((loop
       (lambda (vars vals bindings)
         (if (null? vars)       ; すべての変数に対して束縛を作れば
           (cons bindings env) ;    枠組を ENV 加えて、返す。
           (loop (cdr vars)    ; 束縛を枠組に加えてゆく。
                 (cdr vals)
                 (cons (cons (car vars) (car vals))
                       bindings))))))
    (loop vars vals '())))
```

手続き MAKE-WALLET が呼ばれるたびに、新しい枠組が作られてトップレベルの環境が拡大されます。2 つの式 (DEFINE W1 (MAKE-WALLET 100)) と (DEFINE W2 (MAKE-WALLET 200)) が評価された直後の様子を、図 12.5に示します。W1, W2 ともに、環境の拡大のために加えられた枠組は別のものです。そのため、それぞれで SET! によって同じ名前の変数 MOMEY を書き換えても、違う束縛に対する書換えなので互いに影響は与えません。この理由により、オブジェクトに固有な局所変数を持たせることができるわけです。

次に文面有効域則について説明します。(文面有効域則は、静的束縛とも呼ばれます。) 次のプログラムにおいて、式 (F2 1) の評価の計算過程を考えてみましょう。

```
(DEFINE A   2001)
```

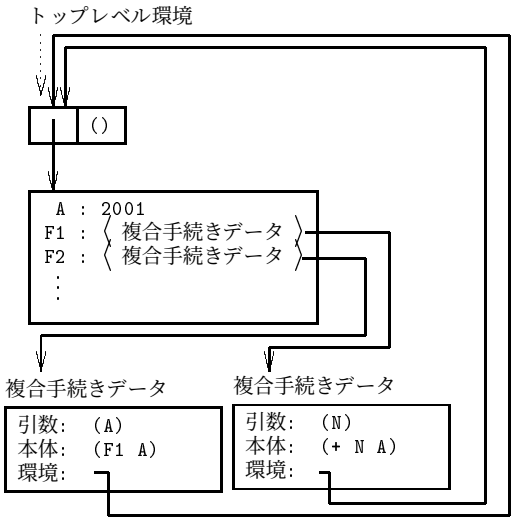


図 12.6: 手続き F1 と F2

```
(DEFINE F1  (LAMBDA (N) (+ N A)))
(DEFINE F2  (LAMBDA (A) (F1 A)))
```

F1, F2 の値はそれぞれ複合手続きです。それらを作り出した λ式はトップレベル環境で評価されているので、複合手続きデータの持つ環境部はトップレベル環境です。この様子を図 12.6に示します。(枠組は簡略化して図示しています。)

手続き F2 に引数 1 が与えられて呼び出されると、A の値が 1 である枠組が作られ、トップレベル環境を拡大した環境が作られます。この拡大された環境で、F2 の本体は評価されます。(図 12.7参照) F2 の本体は (F1 A) なので、A の値 1 を引数にして手続き F1 を呼び出します。

手続き F1 に引数 1 が与えられると、N の値が 1 である枠組が作られます。手続き F1 (厳密には変数 F1 に値である複合手続きデータ) の環境部の値はトップレベル環境となっています。このため、トップレベル環境に対して引数渡しによる枠組が追加されて、環境が拡大されます。この拡大された環境で F1 の本体が評価されます。すると F1 の中にある変数 A は、トップレベル環境で定義されているものを表すことになります。従って A の値は、 2001 となります。このときの様子を図 12.8に示します。以上のような理由により、文面有効域則が実現されます。

12.2.10 変数とその値 (実現)

環境での変数の値を調べたり、環境に変数を定義するための手続きを作ります。

手続き `ts:reset-environ` はトップレベル環境を、なにも定義されていない状態にします。何も定義されていない環境を、図 12.9に示します。

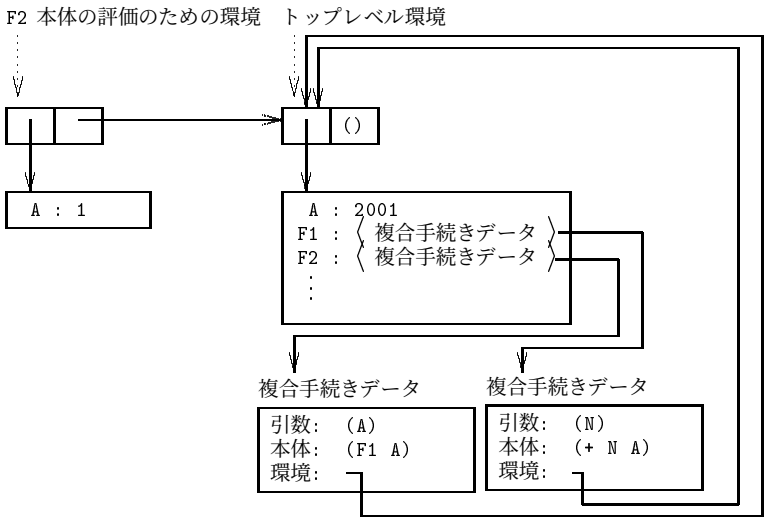


図 12.7: F2 の本体を評価するために拡大された環境

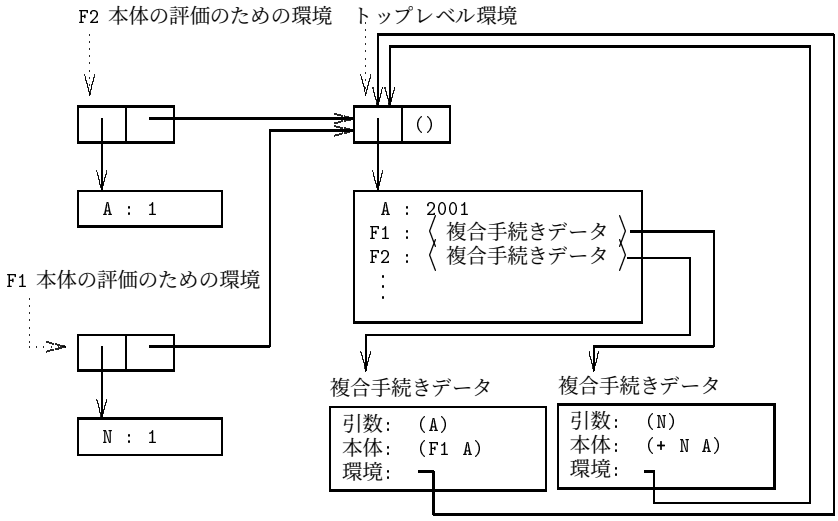


図 12.8: F1 の本体の評価のために拡大された環境

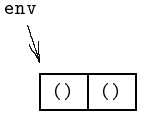


図 12.9: 環境の初期状態

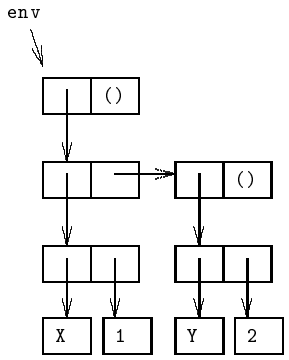


図 12.10: 束縛の追加前

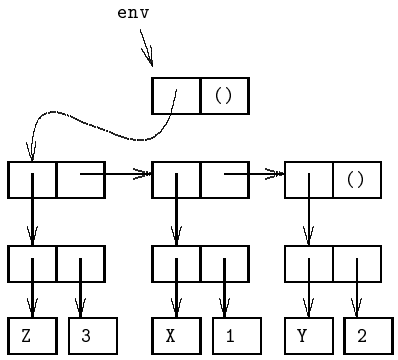


図 12.11: 束縛の追加後

トップレベル環境は変数*ts:bindings* に持たせておきます。手続き ts:top-environ はトップレベル環境を返します。

```
(define *ts:bindings*       #f)
(define (ts:reset-environ)  (set! *ts:bindings* (list '())))
(define (ts:top-environ)    *ts:bindings*)
```

以降ではいろいろな手続きを作りますが、それらでは次の「下請け」手続きを使っています。

- (ts:find-binding var env)
  環境 env での変数 var の束縛を探し、対 (⟨変数⟩ . ⟨値⟩) を返します。見つからないときは #f を返します。

- (ts:find-binding2 var env)
  環境 env の最初の枠組で変数 var の束縛を探し、対 (⟨変数⟩ . ⟨値⟩) を返します。もし見つからないときは #f を返します。

これら 2 つの手続きの実現は後で示します。

まず最初に、変数の値を調べる手続き ts:lookup-binding を示します。手続き ts:find-binding を使って目的の変数の束縛を見つけ、束縛の変数値の部分を取り出しています。もし束縛が見つからなければ手続き ts:error を呼んで、エラーメッセージを表示して実行を中断します。

```
(define (ts:lookup-binding var env)
  (let ((binding (ts:find-binding var env)))
    (if binding
      (cdr binding)
```

```
      (ts:error "UNBOUND VARIABLE:"
        (ts:get-scheme-value var)))))
```

次は変数を定義するための手続き ts:define-var を示します。この手続きは、(ts:define-var var val env) の形で呼ばれます。手続き ts:define-var は環境 env の中に変数 var のための新しい束縛を作り、その値が val であるようにします。図 12.10 は変数を定義する前の環境を、図 12.11は変数を定義した後の環境を図示しています。もしすでに環境 env の中に変数 var の束縛があれば、その値を val に書き換えます。

```
(define (ts:define-var var val env)
  (let* ((binding (ts:find-binding2 var env)))
    (if binding
      (set-cdr! binding val)
      (set-car! env (cons (cons var val) (car env)))))
  val)
```

次は変数値の書換えをする手続き ts:set-var! を作ります。この手続きは (ts:set-var! var val env) の形で呼ばれます。この手続きはまず手続き ts:find-binding を使って、環境の中から変数 var の束縛を見つけます。もし見つかれば、束縛の値の部分を書き換えることで変数の値を変えます。もし見つからないときは ts:error を呼び、エラーメッセージを表示させて実行を中断します。(これが定義済みの変数にしか SET! が使えない理由です。)

```
(define (ts:set-var! var val env)
  (let* ((binding (ts:find-binding var env)))
    (if binding
      (set-cdr! binding val)
      (ts:error "UNBOUND VARIABLE:"
        (ts:get-scheme-value var)))))
```

最後に環境の中から変数 var の束縛を見つける手続き ts:find-binding と ts:find-binding2 を示します。

手続き ts:find-binding は変数の束縛 (変数名が car 部、変数値が cdr 部である対) を返します。もし束縛が見つからないときは #f を返します。束縛の検索は、まず最初の枠組を調べ、もしなければ次の枠組を調べ、ということを繰り返します。

```
(define (ts:find-binding var env)
  (letrec
    ((loop
       (lambda (env)
```

```
        (if (null? env)
          #f                    ; 見つからなかったら #f を返す
          (let ((binding (assoc var (car env))))
            (if binding
              binding                ; 見つかれば、その束縛を返す
              (loop (cdr env))))))))); 次の枠組の中を探す
    (loop env)))
```

手続き ts:find-binding2 は、現在の環境の最初の枠組に目的の変数の束縛があるかどうかを調べます。(env の最初の枠組を取り出し、その中だけで束縛を探します。) 返す値は手続き ts:find-bindings と同じです。

```
(define (ts:find-binding2 var env)
  (assoc var (car env)))
```

12.2.11 基本手続きの実現

これは図 12.1での「基本手続き群」に相当します。

手続き ts:intern-prim-proc により、基本手続きを登録します。トップレベル環境で変数を定義することで、基本手続きの登録を実現しています。第一引数 name には、基本手続きの名前を Scheme データとして与えます。第二引数 proc には、「Scheme で書かれた、TS 基本手続きを実現する手続き」を与えます。第三引数 proc には、登録する基本手続きの引数の数を与えます。もしこの値が any なら、任意の引数をとることを表します。

```
(define (ts:intern-prim-proc name proc nargs)
  (ts:define-var
    (ts:scheme-obj->ts-obj name)
    (ts:make-primitive-procedure proc name nargs)
    (ts:top-environ)))
```

手続き (ts:intern-primitive-procedures は、基本手続き群を登録します。この手続きは初期化のときに一度だけ呼ばれます。

```
(define (ts:intern-primitive-procedures)
  (ts:intern-prim-proc  'NULL?   tsp:null?  1)
  (ts:intern-prim-proc  'EQ?     tsp:eq?    2)
  (ts:intern-prim-proc  'CONS    tsp:cons   2)
  (ts:intern-prim-proc  'CAR     tsp:car    1)
  (ts:intern-prim-proc  'CDR     tsp:cdr    1)
  (ts:intern-prim-proc  'PAIR?   tsp:pair?  1)
```

```
  (ts:intern-prim-proc  '+       tsp:+      'any)
  (ts:intern-prim-proc  '-       tsp:-      2)
  (ts:intern-prim-proc  '=       tsp:=      2)
  (ts:intern-prim-proc  '>       tsp:>      2)
  (ts:intern-prim-proc  'QUIT    tsp:quit   0))
```

以下に、「Scheme で書かれた、TS 基本手続きを実現する手続き」を示します。これらはどれも、第一引数には呼び出されたときの環境が与えられます。第二引数以降に、基本手続き呼び出しのときに与えられた引数が与えられます。

```
(define (tsp:cons s1 s2)
  (ts:cons s1 s2))
(define (tsp:car s)
  (ts:car s))
(define (tsp:cdr s)
  (ts:cdr s))
(define (tsp:null? s)
  (ts:make-boolean (ts:null? s)))
(define (tsp:eq? s1 s2)
  (ts:make-boolean
    (eq? (ts:get-scheme-value s1) (ts:get-scheme-value s2))))
(define (tsp:pair? s)
  (ts:make-boolean (ts:pair? s)))
(define tsp:+
  (lambda args   ; 引数がすべてリストになって、args に与えられる。
    (letrec
      ((loop
         (lambda (nums sum)
           ; nums は引数の数の残り、sum は和の中間結果
           (if (null? nums)
             (ts:make-integer sum)
             (loop (cdr nums)
               (+ (ts:get-scheme-value (car nums)) sum))))))
      (loop args 0))))
(define (tsp:- n1 n2)
  (ts:make-integer
    (- (ts:get-scheme-value n1) (ts:get-scheme-value n2))))
(define (tsp:= n1 n2)
```

```
  (ts:make-boolean
    (= (ts:get-scheme-value n1) (ts:get-scheme-value n2))))
(define (tsp:> n1 n2)
  (ts:make-boolean
    (> (ts:get-scheme-value n1) (ts:get-scheme-value n2))))
```

上に示した TS での加算手続き + は、任意の数の引数をとることができます。(減算手続き – は、2 つの引数に固定されています。) + を実現する Scheme 手続き tsp:+ 自体が任意の数の引数を受けとることができるようにしてあり、引数の数に応じた動作をするようにしてあります。

最後に QUIT を実現する手続きを示します。継続に偽 #f を引数として呼び出すことでトップレベルへ偽 #f を返し、TS を終了させます。

```
(define (tsp:quit env)
  (*ts:top-level-continuation* #f))
```

12.3 インタプリタの実行

インタプリタを実行させてみましょう。上に示した手続きをすべて Scheme 処理系で定義した後で式 (ts) を評価すれば、Scheme インタプリタ TS が動き始めます。基盤となっている Scheme インタプリタは上に示した TS のプログラムを実行していて、TS は与えられた Scheme プログラムを解釈して実行します。

```
  > (ts)


  TS (TINY SCHEME IN SCHEME)
  ] ▉
```

プロンプト] が表示され、TS は入力を待っています。

1 から n までの総和を計算する手続き SUM を定義します。

```
  ] (DEFINE SUM
      (LAMBDA (N)
        (IF (= N 0)
          0
          (+ N (SUM (- N 1))))))
  SUM
```

この手続きは再帰的な定義をしています。引数渡しのメカニズムの動作を確認するため、SUM の仮引数と同じ名前の変数 N をトップレベルで定義しておきます。

```
] (DEFINE N (QUOTE POOH))
N
```

では、SUM を実行してみましょう。式 (SUM 100) により、1 から 100 までの総和を計算します。SUM を実行したあと、トップレベル変数 N の値も調べてみます。

```
] (SUM 100)
5050
] N
POOH
```

今度は「財布」オブジェクトを使って、局所変数の機能を確かめてみましょう。

```
] (DEFINE MAKE-WALLET
    (LAMBDA (INITIAL-MONEY)
      (DEFINE MONEY INITIAL-MONEY)
      (DEFINE EXTRACT
        (LAMBDA (AMOUNT)
          (SET! MONEY (- MONEY AMOUNT))
          MONEY))
        EXTRACT))
MAKE-WALLET
] (DEFINE W1 (MAKE-WALLET 100))
W1
] (DEFINE W2 (MAKE-WALLET 200))
W2
] (W1 10)
90
] (W2 10)
190
```

ほかにもいろいろと試してみて下さい。

12.4 その他の話題

ここでは TS のプログラムでは取り扱わなかった、Scheme インタプリタに関するいくつかの話題について述べます。

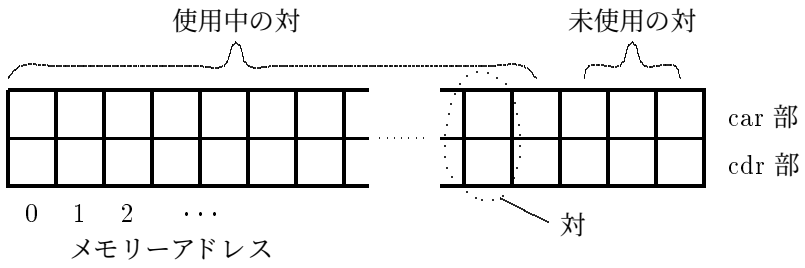


図 12.12: Scheme での主記憶管理の概念図

12.4.1 主記憶の管理とガベージコレクション

手続き cons を使うことで、新しい対をつくり出すことができます。これは主記憶の使われていない部分の一部を取り出して、対として使っています。(図 12.12に概念図を示します。実際の処理系では、もっと複雑な管理手法を取ることが多いです。) そのために、いつかはすべての主記憶を使い果たして、ひとつも対をつくり出すことができなくなってしまいます。

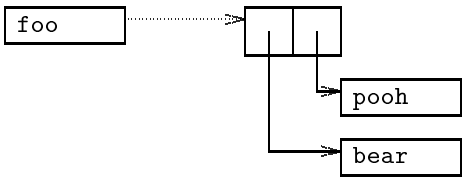
対をつくり出せなくなったからといっても、主記憶を完全に使い果たしたわけではありません。次の例を見てみましょう。

```
> (define foo (cons 'pooh 'bear))
#<unspecified>
> (set! foo (cons 'piglet 'pig))
#<unspecified>
```

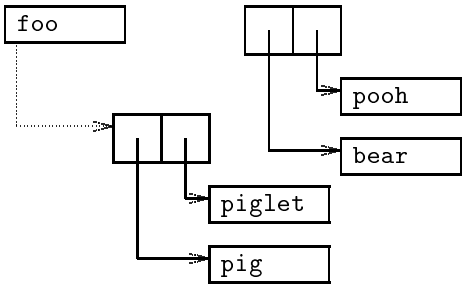
まずはじめに、変数 foo の値は (pooh . bear) となります。このとき cons によって、新しい対が作られています。ところがその後、変数 x の値は (piglet . pig) と変更されました。すると最初の対はどの変数の値でもなく、他のリストの一部にもなっていません。言いかえれば、完全に捨て去られた対となっています。そのような対はまったく使われていないので、再利用できることが分かります。主記憶が足りなくなると、Scheme 処理系は使われていない主記憶の部分を見つけ、それらを再利用できるようにします。このことを**ガベージコレクション** (garbage collection) といいます。

ガベージコレクションにはいくつもの手法が知られていますが、ここでは単純な手法の概略を説明します。対として使っている記憶セルの 1 ビットを、マーク用のビット (マークビット) とします。このマークビットはガベージコレクションのためだけに使われ、他の部分では使いません。対が足らなくなると、ガベージコレクションを開始します。その手順は以下の通りです。

1. 使用中のデータにすべてマークをつける。



(a) (define foo (cons 'pooh 'bear)) の後



(b) 変数 foo を書き換えた後

図 12.13: 使われていない対

— 環境中のそれぞれの変数に対し、その値となっているデータのマークビットを 1 にします。データが対なら、その car 部や cdr 部の値となっているデータに対しても、さらにマークを付けます。ベクトルデータなら、それぞれの要素の値となっているデータにも、さらにマークを付けます。このように、使われているデータから手繰れるデータすべてにマークを付けます。

2. 主記憶を走査し、マークのついていない部分を回収する。
   — 主記憶の最初から終りまでひとつひとつの記憶セルを調べ、マークがついていなければその部分は捨て去られた部分なので、回収します。回収した部分はガベージコレクション後、再利用されます。

Scheme 処理系でプログラムを実行しているときに、ときどき Garbage collecting... と表示されます。このときにガベージコレクションが実行されています。

12.4.2 継続

継続データには、継続が作られた時点での「計算過程の状態」が保持されています。実をいうと、本章で作った TS 処理系で継続を実現するのは簡単ではありません。というのも、計算過程の状態は基盤となっている Scheme 処理系での変数と実行時のスタックなどに分散していて、計算を継続するのに必要な情報を集めたり、元に戻すことが難しいからです。ここでは継続の実現のアイディアを簡単に述べます。

第1章の 1.2での説明で分かるように、計算過程の状態はレジスタとスタックに保持されています[1]。継続データを作るとときは、レジスタの内容とスタックの内容とをパックし、それを継続データとします。

継続データの表す計算の状態に戻るには、継続データからレジスタの値とスタックの内容を戻すことで、継続データが作られたときの状態に戻ることができます。このようにして実行の再開が可能となります。

練習問題

1. 乗算をする * を基本手続きとして追加しなさい。 ただし引数の数は 2 とします。
2. 除算をする / を基本手続きとして追加しなさい。ただし引数の数は 2 とします。(整数しか数として許していないので、除算の結果が整数でない場合は切捨てをするなどして、整数にしなさい。)
3. 手続き – に任意の数の引数を与えることができるようにしなさい。
4. 手続き – に与えられる引数の数が 1 の場合、引数の符合を変えたものを返すようにしなさい。
5. 手続き – にひとつも引数が与えられなかったとき、– はどのような振舞いをすべきかを考察しなさい。* が任意の数の引数をとるようにしたとき、* の場合についても同じ問題について考察しなさい。
6. 表示をする手続き DISPLAY と、改行する手続き NEWLINE を基本手続きとして追加しなさい。
7. 絶対値を返す ABS を基本手続きとして追加しなさい。
8. 商を計算する QUOTIENT と、剰余を計算する MODULO を、を基本手続きとして追加しなさい。
9. LIST? を基本手続きとして追加しなさい。
10. LENGTH を基本手続きとして追加しなさい。
11. LIST-TAIL を基本手続きとして追加しなさい。
12. REVERSE を基本手続きとして追加しなさい。

[1]主記憶にも保持されていますが、Scheme プログラムの実行に関してはほとんどがスタックに置かれています。

13. APPEND を基本手続きとして追加しなさい。

14. LIST-REF を基本手続きとして追加しなさい。

15. SET-CAR! と SET-CDR! を基本手続きとして追加しなさい。

16. BOOLEAN? を基本手続きとして追加しなさい。

17. NOT を基本手続きとして追加しなさい。

18. 文字列型を追加しなさい。それにともなって、文字列に対する基本手続きをいくつか作りなさい。

19. ベクトル型を追加しなさい。それにともなって、ベクトルに対する基本手続きをいくつか作りなさい。

20. 実数型を追加しなさい。それにともなって、実数に対する基本手続きをいくつか作りなさい。ただし、以下の点に注意しなさい。整数と実数を足した結果は実数でないといけません。$4/2$ は整数であっても、 $4/3$ は実数で値を返さないといけません。

21. 有理数型を追加しなさい。それにともなって、実数に対する基本手続きをいくつか作りなさい。

22. 特殊形式 IF が (IF ⟨ 条件 ⟩ ⟨ 式 ⟩) の形も取り扱えるようにしなさい。

23. 特殊形式 AND を追加しなさい。

24. 特殊形式 OR を追加しなさい。

25. 特殊形式 COND を追加しなさい。

26. 特殊形式 CASE を追加しなさい。

27. 手続き定義のための簡略記法 (DEFINE (⟨ 変数 ⟩ ⟨ 仮引数のならび ⟩) ⟨ 本体 ⟩) を取り扱えるようにしなさい。なおこの記法は、(DEFINE ⟨ 変数 ⟩ (LAMBDA (⟨ 仮引数のならび ⟩) ⟨ 本体 ⟩)) と等価です。

28. 次の手続き SUM について考えます。

```
(DEFINE SUM
  (LAMBDA (N)
    (IF (= N 0)
      0
      (+ N (SUM (- N 1))))))
```

(SUM 2) の評価で、仮引数 N の値が 1 になったときの環境の様子を図示しなさい。さらに 0 のときの環境の様子も図示しなさい。

29. 手続きの中で局所変数を用意する特殊形式 LET を追加しなさい。なお LET 構文

(LET ((⟨ 変数$_1$⟩ ⟨ 式$_1$⟩)
(⟨ 変数$_2$⟩ ⟨ 式$_2$⟩)
⋮)
⟨ 本体 ⟩)

を、

((LAMBDA (⟨ 変数$_1$⟩ ⟨ 変数$_2$⟩ ···)
⟨ 本体 ⟩)
⟨ 式$_1$⟩ ⟨ 式$_2$⟩ ···)

として実現しなさい。

30. 手続きの中で局所変数を用意する特殊形式 LET* を追加しなさい。なお LET* 構文

(LET* ((⟨ 変数$_1$⟩ ⟨ 式$_1$⟩)
(⟨ 変数$_2$⟩ ⟨ 式$_2$⟩)
⋮)
⟨ 本体 ⟩)

を、

((LAMBDA (⟨ 変数$_1$⟩)
((LAMBDA (⟨ 変数$_2$⟩)
···
⟨ 本体 ⟩)
···
⟨ 式$_2$⟩)
⟨ 式$_1$⟩)

として実現しなさい。

31. LOAD を基本手続きとして追加しなさい。

32. TS の起動時に、ある特定のファイルをロードするようにしなさい。そのファイルに頻繁に使う手続きの定義を書いておけば、いちいち手で入力したりロードさせる手間が省けます。

33. **** 本章と同じ機能を持った Scheme インタプリタを、C 言語、Pascal、Fortran のいずれかで書いてみなさい。

34. ***** 上の問題で作ったインタプリタに、ガベージコレクション機能を加なさい。

ときどき婦人たちが手紙を書きにくるが、其處に私がゐるのを見ると、何も言はないでそのまま引きかへす。中には私の原稿をのぞいて行くものもあるが、何を書いてゐるんだか解るまいと高をくくつて、私は知らん顔をしてゐる。

堀辰雄 「美しい村」
『美しい村』収録 昭和三十一年 清水書院刊

Appendix A

ASCII 文字集合

以下に ASCII 文字集合の表を示します[1]。アルファベット文字や数字を計算機上で取り扱うためには、それぞれの文字に整数を割り当てて、その整数によって文字を指定する方法を使います。そして文字と整数との対応関係を定めたものが文字集合です。

以下の表で文字 c の文字コードを調べるには、次のようにします。まず表の中に文字 c を見つけます。そしてその行の左端に書かれた数と、その列の上端に書かれた数とを加え合わせることで、その文字の文字コードを得ます。たとえば、文字 E の文字コードは $64 + 5 = 69$ となります。

	+0	+1	+2	+3	+4	+5	+6	+7
0	nul	soh	stx	etx	eot	enq	ack	bel
8	bs	ht	nl	vt	np	cr	so	si
16	dle	dc1	dc2	dc3	dc4	nak	syn	etb
24	can	em	sub	esc	fs	gs	rs	us
32		!	"	#	$	%	&	’
40	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7
56	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G
72	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W
88	X	Y	Z	[	\	]	^	_
96	‘	a	b	c	d	e	f	g
104	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	del

表 A.1: ASCII 文字集合表

なお、文字コード 32 の文字は空白文字 (スペース文字) です。文字コードが 0 から 31 までと 127 の文字は、制御文字 (control character) **制御文字** (control character) と呼ばれます。これら

[1] ASCII とは、American Standard Code for Information Interchange の略で、‘アスキー’ と読みます。ASCII 文字集合の他に、EBCDIC 文字集合がありますが、これは主に IBM の大型計算機で使われています。

の内のいくつかは頻繁に使われるため、特別に以下のような別名が付けられています。(括弧内は、Scheme でそれぞれの文字を名前で指定するときの書き方です。)

- 文字コード 0 — 空文字 (null)
- 文字コード 8 — 消去文字、またはバックスペース文字 (backspace)
- 文字コード 9 — 桁飛ばし文字、またはタブ文字 (tab)
- 文字コード 10 — 改行文字 (newline)
- 文字コード 12 — 改頁文字 (page)
- 文字コード 13 — 復改文字 (return)
- 文字コード 32 — 空白文字、あるいはスペース文字 (space)

制御文字は `#\⟨` 文字の名前 `⟩` で指定することができます。たとえば `#\⟨soh⟩`、`#\⟨page⟩` は、それぞれ文字コード 1, 12 の文字を表します。

Appendix B

Scheme 参照マニュアル

- **変数参照**

◇ ⟨variable⟩
— 変数 ⟨variable⟩ の値.

- **リテラル式**

◇ ⟨constant⟩
— 定数. (文字列や数など.)

◇ ’⟨datum⟩
— (quote ⟨datum⟩) と等価.

◇ `(quote` ⟨datum⟩`)`
— ⟨datum⟩そのものを表す.

- **手続き呼出し**

◇ (⟨op⟩ ⟨arg_1⟩ ···)
— ⟨arg_1⟩ ···を引数にし、手続き⟨op⟩を呼び出す.

- **λ式**

◇ `(lambda` ⟨formals⟩ ⟨body⟩`)`
— 仮引数を⟨formal⟩、本体を⟨body⟩とする手続きを生成.

- **定義**

◇ `(define` ⟨var⟩ ⟨exp⟩`)`
— 変数定義.

△ `(define (`⟨var⟩ ⟨formals⟩`)` ⟨body⟩`)`
— `(define` ⟨var⟩ `(lambda (`⟨formals⟩`)` ⟨body⟩`))` と等価.

△ `(define (`⟨var⟩ . ⟨formal⟩`)` ⟨body⟩`)`
— `(define` ⟨var⟩ `(lambda` ⟨formal⟩ ⟨body⟩`))` と等価.

- **条件式**

◇ `(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)`
— 条件式.

△ `(if` ⟨test⟩ ⟨consequent⟩`)`
— 条件式.

◇ (cond ⟨clause$_1$⟩ ⟨clause$_2$⟩ ···)
— 条件式.
◇ (case ⟨key⟩ ⟨clause$_1$⟩ ···)
— 場合分け.
◇ (and ⟨test$_1$⟩ ···)
— 左から順に、結果が#f になるまで評価する.
◇ (or ⟨test$_1$⟩ ···)
— 左から順に、結果が非#f になるまで評価する.

• 代入

◇ (set! ⟨var⟩ ⟨expr⟩)
— 変数への代入.

• 局所変数

◇ (let ⟨bindings⟩ ⟨body⟩)
— 局所変数を持つブロックで⟨body⟩を評価.
△ (let* ⟨bindings⟩ ⟨body⟩)
— 局所変数を持つブロックで⟨body⟩を評価. (順次環境を拡大する)
◇ (letrec ⟨bindings⟩ ⟨body⟩)
— 局所変数を持つブロックで⟨body⟩を評価. (相互再帰のための環境を作る)

• 逐次実行

◇ (begin ⟨expr$_1$⟩ ⟨expr$_2$⟩ ···)
— 逐次評価し、最後の式の値を返す.

• 繰り返し

△ (do ((⟨var$_1$⟩ ⟨init$_1$⟩ ⟨step$_1$⟩) ···) (⟨test⟩ ⟨exp$_1$⟩ ···) ⟨body⟩)
— 局所変数を作り、⟨test⟩が真の間 ⟨body⟩の評価を繰り返す.
△ (let ⟨var⟩ ⟨bindings⟩ ⟨body⟩)
— 名前付き let.

• 遅延評価

△ (delay ⟨expr⟩)
— 遅延オブジェクトの生成.
△ (force *promise*)
— 遅延オブジェクトの評価.

• 準引用

◇ (quasiquote *obj* ···)
— quote と似ているが、unquote の付いたものは評価される.
◇ ‘*obj*
— (quasiquote *obj*) と等価.
◇ (unquote *obj* ···)
— (quasiquote 内) *obj*を評価する.
◇ ,*obj*
— (unquote *obj*) と等価.
◇ (unquote-splicing *obj* ···)
— (quasiquote 内) *obj*の評価をし、リストを連結する.
◇ ,@*obj*
— (unquote-splicing *obj* ···) と等価.

• ブール型

◇ (not *obj*)
— *obj*の論理否定.
◇ (boolean? *obj*)
— *obj*がブール型か?

• 等価性判定

◇ (eqv? obj_1 obj_2)
— obj_1 と obj_2 は等しいか?
◇ (eq? obj_1 obj_2)
— obj_1 と obj_2 は等しいか?
◇ (equal? obj_1 obj_2)
— obj_1 と obj_2 は等しいか?

• 対

◇ (pair? *obj*)
— 対か?
◇ (cons obj_1 obj_2)
— 対を作る.
◇ (car *pair*)
— 対の car 部.
◇ (cdr *pair*)
— 対の cdr 部.
◇ (set-car! *pair obj*)
— 対の car 部を置き換える.
◇ (set-cdr! *pair obj*)
— 対の cdr 部を置き換える.
◇ (caar *pair*)
— (car (car *pair*)) と等価.
◇ (cadr *pair*)
— (car (cdr *pair*)) と等価.
⋮
◇ (cdddar *pair*)
— (cdr (cdr (cdr (car *pair*)))) と等価.
◇ (cddddr *pair*)
— (cdr (cdr (cdr (cdr *pair*)))) と等価.

• リスト

◇ (null? *obj*)
— 空リストか?
◇ (list? *obj*)
— リストか?
◇ (list *obj* ···)
— *obj* ···より成るリスト.
◇ (length *list*)
— *list* の長さ.
◇ (append *list* ···)
— リストの連結.
◇ (reverse *list*)
— リストの反転.

△ (`list-tail` *list k*)
— *list*の最終要素.
♢ (`list-ref` *list k*)
— *list*のk番目の要素.
♢ (`member` *obj list*)
— *list*のトップレベルに *obj*があるか? (比較は `equal?`で行なう.)
♢ (`memq` *obj list*)
— *list*のトップレベルに *obj*があるか? (比較は `eq?`で行なう.)
♢ (`memv` *obj list*)
— *list*のトップレベルに *obj*があるか? (比較は `eqv?`で行なう.)
♢ (`assoc` *obj alist*)
— 連想リストの検索. (比較は `equal?`で行なう.)
♢ (`assq` *obj alist*)
— 連想リストの検索. (比較は `eq?`で行なう.)
♢ (`assv` *obj alist*)
— 連想リストの検索. (比較は `eqv?`で行なう.)

• 記号

♢ (`symbol?` *obj*)
— 記号か?
♢ (`symbol->string` *sym*)
— 文字列から記号を作る.
♢ (`string->symbol` *str*)
— 記号の名前の文字列を得る.

• 数

♢ (`number?` *obj*)
— 数か?
♢ (`complex?` *obj*)
— 複素数か?
♢ (`real?` *obj*)
— 実数か?
♢ (`rational?` *obj*)
— 有理数か?
♢ (`integer?` *obj*)
— 整数か?
♢ (`exact?` z)
— 厳密数 (整数または有理数) か?
♢ (`inexact?` z)
— 非厳密数か?
♢ (`=` z_1 z_2 z_3 $\cdots$)
— $z_1 = z_2 = ...$ か?
♢ (`<` x_1 x_2 x_3 $\cdots$)
— $z_1 < z_2 < ...$ か?
♢ (`>` x_1 x_2 x_3 $\cdots$)
— $z_1 > z_2 > ...$ か?
♢ (`<=` x_1 x_2 x_3 $\cdots$)
— $z_1 \leq z_2 \leq ...$ か?

◇ (>= x_1 x_2 x_3 $\cdots$)
— $z_1 \geq z_2 \geq$ ... か?
◇ (zero? z)
— 零か?
◇ (positive? x)
— 正か?
◇ (negative? x)
— 負か?
◇ (odd? n)
— 奇数か?
◇ (even? n)
— 偶数か?
◇ (max x_1 x_2 $\cdots$)
— 最大値.
◇ (min x_1 x_2 $\cdots$)
— 最小値.
◇ (+ z_1 $\cdots$)
— 加算.
◇ (* z_1 $\cdots$)
— 乗算.
◇ (- z_1 z_2)
— 減算.
△ (- z_1 z_2 $\cdots$)
— 減算.
◇ (- z)
— 符号の反転.
◇ (/ z_1 z_2)
— 除算.
△ (/ z_1 z_2 $\cdots$)
— 除算.
◇ (/ z)
— 逆数.
◇ (abs x)
— 絶対値.
◇ (quotient n_1 n_2)
— 商.
◇ (remainder n_1 n_2)
— 余り.
◇ (modulo n_1 n_2)
— n_2 を法とした n_1.
◇ (gcd n_1 $\cdots$)
— 最大公約数.
◇ (lcm n_1 $\cdots$)
— 最小公倍数.
△ (numerator q)
— 分子.
△ (denominator q)
— 分母.

◇ (floor x)
— xを越えない最大の整数. (床関数)
◇ (ceiling x)
— xより小さくない最小の整数. (天井関数)
◇ (truncate x)
— 絶対値がxの絶対値以下の、最もxに近い整数
◇ (round x)
— xに一番近い整数
△ (rationalize x y)
— x に y より離れていない有理数でもっとも単純なもの.
△ (exp z)
— $\exp z$
△ (log z)
— $\log z$ (底は e)
△ (sin z)
— $\sin z$
△ (cos z)
— $\cos z$
△ (tan z)
— $\tan z$
△ (asin z)
— $\mathrm{asin}\ z$
△ (acos z)
— $\mathrm{acos}\ z$
△ (atan z)
— $\mathrm{atan}\ z$
△ (atan y x)
— $\mathrm{atan}\ (y/x)$
△ (sqrt z)
— $\sqrt{z}$
△ (expt y z)
— y^z
△ (make-rectangular x y)
— 複素数 $x + yi$ を作る.
△ (make-polar r t)
— 複素数 re^{it}を作る.
△ (real-part z)
— zの実部.
△ (imag-part z)
— zの虚部.
△ (magnitude z)
— zの絶対値.
△ (angle z)
— zの偏角.
△ (exact->inexact z)
— 厳密数から非厳密数を作る.
△ (inexact->exact z)
— 非厳密数から厳密数を作る.

◇ (number->string n)
— 数から文字列を作る.
◇ (number->string n $radix$)
— 数から文字列を作る. (基数を $radix$ とする)
◇ (string->number str)
— 文字列から数を作る.
◇ (string->number str $radix$)
— 文字列から数を作る. (基数を $radix$ とする)

- **文字**

◇ (char? obj)
— 文字か?
◇ (char=? ch_1 ch_2)
— 文字が等しいか?
◇ (char<? ch_1 ch_2)
— $ch_1 < ch_2$ か?
◇ (char>? ch_1 ch_2)
— $ch_1 > ch_2$ か?
◇ (char<=? ch_1 ch_2)
— $ch_1 \leq ch_2$ か?
◇ (char>=? ch_1 ch_2)
— $ch_1 \geq ch_2$ か?
◇ (char-ci=? ch_1 ch_2)
— (大/小文字を区別せずに) $ch_1 = ch_2$ か?
◇ (char-ci<? ch_1 ch_2)
— (大/小文字を区別せずに) $ch_1 < ch_2$ か?
◇ (char-ci>? ch_1 ch_2)
— (大/小文字を区別せずに) $ch_1 > ch_2$ か?
◇ (char-ci<=? ch_1 ch_2)
— (大/小文字を区別せずに) $ch_1 \leq ch_2$ か?
◇ (char-ci>=? ch_1 ch_2)
— (大/小文字を区別せずに) $ch_1 \geq ch_2$ か?
◇ (char-alphabetic? ch)
— アルファベット文字か?
◇ (char-numeric? ch)
— 数字か?
◇ (char-whitespace? ch)
— 空白文字か?
◇ (char-upper-case? $letter$)
— 大文字か?
◇ (char-lower-case? $letter$)
— 小文字か?
◇ (char->integer ch)
— 文字の文字コードを得る.
◇ (integer->char n)
— 文字コードに対応する文字を得る.
◇ (char-upcase ch)
— 大文字にする.

◇ (char-downcase *ch*)
— 小文字にする.

• 文字列

◇ (string? *obj*)
— 文字列か?
◇ (make-string *k*)
— 長さ *k*の文字列を作る.
◇ (make-string *k ch*)
— *k*個の *ch* よりなる文字列を作る.
◇ (string *ch* ···)
— *ch* ···のならびである文字列を作る.
◇ (string-length *str*)
— 文字列の長さ.
◇ (string-ref *str k*)
— 文字列の*k*番目の文字.
◇ (string-set! *str k ch*)
— 文字列の *k*番目の文字の置換.
◇ (string=? str_1 str_2)
— str_1=str_2か?
◇ (string<? str_1 str_2)
— str_1<str_2か?
◇ (string>? str_1 str_2)
— str_1>str_2か?
◇ (string<=? str_1 str_2)
— str_1≤str_2か?
◇ (string>=? str_1 str_2)
— str_1≥str_2か?
◇ (string-ci=? str_1 str_2)
— (大/小文字を区別せずに) str_1=str_2か?
◇ (string-ci<? str_1 str_2)
— (大/小文字を区別せずに) str_1<str_2か?
◇ (string-ci>? str_1 str_2)
— (大/小文字を区別せずに) str_1>str_2か?
◇ (string-ci<=? str_1 str_2)
— (大/小文字を区別せずに) str_1≤str_2か?
◇ (string-ci>=? str_1 str_2)
— (大/小文字を区別せずに) str_1≥str_2か?
◇ (substring *str start end*)
— 部分文字列.
◇ (string-append *str* ···)
— 文字列の結合.
◇ (string->list *str*)
— 文字列から文字のリストを作る.
◇ (list->string *ch* ···)
— 文字のリストから文字列を作る.
△ (string-copy *str*)
— *str* と同じ内容の文字列を作る.

△ (string-fill! *str ch*)
— *str* のすべての文字を *ch* に書き換える.

• **ベクトル**

◇ (vector? *obj*)
— ベクトルか?

◇ (make-vector *k*)
— 大きさ *k* のベクトルを作る.

△ (make-vector *k fill*)
— 大きさ *k* のベクトルを作る. (各要素の初期値を *fill* とする.)

◇ (vector *obj* ···)
— (各要素の値を指定して) ベクトルを作る.

◇ (vector-length *vect*)
— ベクトルの大きさ.

◇ (vector-ref *vect k*)
— ベクトルの要素の参照.

◇ (vector-set! *vect k obj*)
— ベクトルの *k*番目の要素を *obj* に書き換える.

◇ (vector->list *vect*)
— ベクトルからリストを作る.

◇ (list->vector *list*)
— リストからベクトルを作る.

△ (vector-fill! *vect e*)
— ベクトルのすべての要素を置き換える.

• **制御**

◇ (procedure? *obj*)
— 手続きか?

◇ (apply *proc list*)
— *list* を実引数のリストとし *proc*を呼び出す.

△ (apply *proc* arg_1 ... *list*)
— (apply *proc* (append (list arg_1 ···) *list*)) と等価.

◇ (map *proc* $list_1$ $list_2$ ···)
— 各 $list_j$ の要素を *proc*に 作用させ、その結果をリストにする.

◇ (for-each *proc* $list_1$ $list_2$ ···)
— $list_j$ のそれぞれに対し *proc* を作用させる.

△ (force *promise*)
— 評価の遅延をしていた *promise* を評価する.

• **継続**

◇ (call-with-current-continuation *proc*)
— 現在の継続を引数にし、*proc* を呼び出す.

• **入出力**

◇ (call-with-input-file *str proc*)
— *str*を読み込みオープンし、ポートを *proc*の引数にして呼び出す.

◇ (call-with-output-file *str proc*)
— *str*を書き込みオープンし、ポートを *proc*の引数にして呼び出す.

◇ (input-port? *obj*)
— 入力ポートか?

◇ (output-port? *obj*)
— 出力ポートか?
◇ (current-input-port)
— 現在入力ポートを得る.
◇ (current-output-port)
— 現在出力ポートを得る.
◇ (with-input-from-file *str thunk*)
— 現在出力をファイル *str*にして *proc*を呼び出す.
◇ (with-output-to-file *str thunk*)
— 現在入力をファイル *str*にして *proc*を呼び出す.
◇ (open-input-file *filename*)
— ファイルの読み出しオープンをする.
◇ (open-output-file *filename*)
— ファイルの書き込みオープンをする.
◇ (close-input-port *port*)
— 入力ポートをクローズする.
◇ (close-output-port *port*)
— 出力ポートをクローズする.
◇ (read [*port*])
— (ポート *port*より) 式を読む.
◇ (read-char [*port*])
— (ポート *port*より) 1 文字読む.
◇ (peek-char [*port*])
— (ポート *port*より) ファイルポインタを進めずに 1 文字読む.
◇ (eof-object? *obj*)
— ファイル終端子か?
△ (char-ready? [*port*])
— (ポート *port* に) 読むべき文字があるか?
◇ (write *obj* [*port*])
— (ポート *port* へ) *obj*を書き込む.
◇ (display *obj* [*port*])
— (ポート *port* へ) *obj*を表示.
◇ (newline [*port*])
— (ポート *port* へ) 改行.
◇ (write-char *char* [*port*])
— (ポート *port* へ) *char*を書き込む.

- **その他**

◇ (load *filename*)
— Scheme プログラムのロード
△ (transcript-on *filename*)
— メッセージをファイルにも書出す
△ (transcript-off)
— メッセージのファイルへの書出しを止める

- **その他 (SCM 特有のもの)**

√ (quit [*n*])
— ngscm の終了 (*n*はエラーコード)

√ (error obj_1 obj_2 ···)
— エラーメッセージを表示し、read-eval-print ループに戻る.
√ errobj
— この変数に実行エラーの原因が代入される.
√ (abort)
— read-eval-print ループに戻る.
√ (defined? *sym*)
— *sym*は定義されているか?
√ (gc)
— ガベージコレクションを行なう.
√ (terms)
— GNU General Public Licese の表示.
√ (list-file *str*)
— ファイルの内容の表示.
√ (system *str*)
— システムコマンドの実行.
√ (getenv *str*)
— 環境変数の値の取得.
√ most-positive-fixnum
— システムで取り扱える最大の正整数.
√ most-negative-fixnum
— システムで取り扱える最大の負整数.
√ internal-time-units-per-second
— 1 秒当たりの内部時計の単位数.
√ (get-internal-run-time)
— ある時刻から経過した、内部時計での実行時間.
√ (get-internal-real-time)
— ある時刻から経過した、内部時計での実時間.
√ (get-decode-time)
— 現在の時刻を、秒、分、時、日、月、年、曜日、正月からの日数、夏時間フラグの 9 つを要素とするベクトルで得る. それぞれベクトルの第 0 要素から第 8 要素に入る. 第 8 要素は、非ゼロのとき夏時間であることを表す.
√ (get-universal-time)
— 1970 年 1 月 1 日 0 時 0 分 0 秒 GMT から経過した秒数.
√ (decode-universal-time *time*)
— (get-universal-time) で得られた時刻を、get-decode-time で得られる形式のデータに変換する.
√ (read-line [*port*])
— (ポート *port* から) 1 行読み込む.
√ (read-line! *str* [*port*])
— (ポート *port* から) 1 行読み込み、*str* に書き込む.
√ (write-line *str* [*port*])
— (ポート *port* へ) *str* を 1 行として書き込む.
√ (file-exists? *filename*)
— ファイル *filename* は存在するか?
√ (delete-file *filename*)
— ファイル *filename*の削除

√ (`rename-file` $filename_1$ $filename_2$)
— ファイル $filename_1$ の名前を $filename_2$ に変更

√ (`stat` *port*) または (`stat` *filename*)
— ポート *port* の示すファイルまたは *filename* で指定されたファイルの、ファイルに関する情報をベクトルで返す. 返される情報は、ベクトルの第 0 要素から第 11 要素まで、順に以下の通り: 入出力装置の識別子 (st_dev)、inode 番号 (st_ino)、ファイルモード (ファイルの型、属性、アクセス制御ビットなど) (st_mode)、リンクカウント (st_nlink)、ファイルの所有者のユーザー識別子 (st_uid)、ファイルのグループ識別子 (st_gid)、入出力装置の識別子 (st_rdev)、バイト数で数えたファイルの大きさ (st_size)、最後にアクセスされた時間 (st_atime)、最後に変更された時間 (st_mtime)、最後に状態が変更された時間 (st_ctime). (括弧内は、C 言語での stat 構造体.) における対応したメンバ名.

√ (`file-position?` *port*)
— ポート *port* のファイルポインタを得る.

√ (`file-set-position` *port* *n*)
— ポート *port* のファイルポインタを *n* バイト目に移動.

√ (`force-output` [*port*])
— (ポート *port* の) 未出力の出力を書き出す

√ (`isatty?` *port*)
— ポート *port* は非ファイル装置に対するポートか?

√ (`chdir` *str*)
— カレントディレクトリを *str*に変更.

Appendix C

NGSCM 編集機能参照カード

- **起動**

ngscm	ngscm の起動
ngscm *filename*	起動時にファイルを読み込む
ngscm -s *filename*	編集機能を使わない起動法
ngscm -a *filename*	Scheme 処理系の初期化をせずに起動

- **NGSCM の終了方法**

NGSCM の終了	C-x C-c
NGSCM の中断 (一時的な中断)	C-z

- **エラーからの回復**

コマンドの途中入力または実行を中断する	C-g
カーソルを中央にし, 画面を描く	C-l
乱れた画面を描き直す	M-x redraw-display

- **ヘルプ機能**

文字列を含んだコマンドの表示 (apropos)	C-h a
キーバインドを表示する	C-h b
キーによって実行されるコマンドを表示する	C-h c
ヘルプ機能のオプションの表示	C-h C-h
オンラインで編集機能を学ぶ (tutorial)	C-h t

本「NGSCM 編集機能参照カード」は、GNU Emacs Version 18 とともに配布されている参照カードを元にし、NGSCM の参照カードに修正したものです. この参照カードの複写ならびに配布は、以下の注意書きに従って下さい.

- **引数**

数引数を与えることで, 繰り返して編集指令を実行することができる. C-u は, それだけだと 4 を指定したことになる. 繰り返し C-u を押すと, 引数の値は 4 倍ずつ増える.

数引数の指定	M-0, M-1,..., M-9
数引数の指定	C-u
負の数引数の指定	M- -

- **ファイル**

ファイルを NGSCM に読み込む	C-x C-f
ファイルをセーブする	C-x C-s
変更されたファイルをセーブする	C-x s
バッファに他のファイルの内容を挿入する	C-x i
バッファの内容を指定したファイルに書き込む	C-x C-w
ディレクトリエディタ Dired を起動する	C-x d

- **探索**

前方へのインクリメンタル検索	C-s
後方へのインクリメンタル探索	C-r
正規表現のインクリメンタル探索	C-M-s
前回の探索と同一のパターンで探索	M-x search-again
前方検索	M-s
後方探索	M-r

インクリメンタル探索では, 以下のキーが定義されている:

最後の文字の効果を取り消す	DEL
インクリメンタル探索の終了	ESC
探索を中断します	C-g
次の候補の前方探索	C-s
次の候補の後方探索	C-r
文字 c そのものを探索パターンに加える	C-q c

どちらの方向に対しても, 同じパターンでの探索は C-s または C-r で行なう.

C-g を入力した時, 探索が成功している時であればカーソルを元の位置に戻し探索を中断する. もし探索が失敗していれば, 探索の失敗している部分が取り消される.

- **入力**

文字 c の挿入	c
文字 c そのものの挿入	C-q c

C-q は, 制御文字を入力するのに用いられる.

- **移動**

カーソルの移動:

移動の単位	**後方**	**前方**
文字	C-b	C-f
語	M-b	M-f

行	C-p	C-n
行の始め (終り) への移動	C-a	C-e
段落	M-[	M-]
ページ	C-x [	C-x]
バッファの始め (終り) への移動	M-<	M-<

画面移動:

次の画面へスクロールします	C-v
前の画面へスクロールします	M-v

- **削除と消去**

削除の単位	**後方**	**前方**
文字 (消去であり, 削除ではない)	DEL	C-d
語	M-DEL	M-d
行 (終りまで)	M-0 C-k	C-k

リージョンの削除	C-w
削除されたテキストのヤンク	C-y

- **マーク**

マークの設定	C-@ または C-SPC
カーソルとマークの入れ換え	C-x C-x

- **対話置換**

テキスト文字列を対話的に置換	M-%
正規表現を用いての対話置換	M-x query-replace-regexp

置換モードでの入力:

置換を行ない, 次の候補を探す	SPC
置換を行なわず, 次の候補を探す	DEL
現在の候補を置換し, 置換を終了	.
残りの候補全てを置換	!
対話置換を終了	ESC

- **マルチウインドウ**

他のウインドウを消去	C-x 1
現在のウインドウを消去	C-x 0
ウインドウを垂直に 2 分割	C-x 2
他のウインドウをスクロール	C-M-v
他のウインドウに切替える	C-x o
ウインドウを小さくする	M-x shrink-window
ウインドウを大きくする	C-x ^
他のウインドウでファイルを読み込む	C-x 4 f

- **清書**

カーソルの後に改行を挿入	C-o

カーソル周辺の空行を消去	C-x C-o
カーソル位置にちょうど 1 つのスペースをおく	M-SPC
段落の詰め込み	M-q
詰め込みする桁の設定	C-x f
詰め込み接頭辞の設定	C-x .

- **大文字/小文字の変更**

語を大文字にする	M-u
語を小文字にする	M-l
大文字で始まる語にする	M-c
リージョンを大文字にする	C-x C-u
リージョンを小文字にする	C-x C-l

- **バッファ**

他のバッファの選択	C-x b
バッファの一覧を表示	C-x C-b
バッファの消去	C-x k
他のウインドウでバッファを選択	C-x 4 b
バッファの変更フラグをオフ	M-~

- **キーボードマクロ**

キーボードマクロの定義の開始	C-x (
キーボードマクロの定義の終了	C-x)
キーボードマクロの実行	C-x e
キーボードマクロの定義への追加	C-u C-x (

- **ミニバッファ**

ミニバッファ (エコー領域) でのキーの定義:

可能な限り補間する	TAB
1 語まで補間する	SPC
補間し, 実行する	RET
編集命令の中断	C-g

- **Scheme モード / Scheme Interaction モード**

バッファを Scheme モードにする	M-x scheme-mode
Scheme 処理系の始動	M-x run-scheme

入力履歴 (Scheme Interaction モードのみ)

1 つ前の入力の式	M-p
1 つ後の入力の式	M-n
1 つ前の入力にカーソルを移動	C-p
1 つ後の入力にカーソルを移動	C-n

式の評価

エコー領域から式を読み込み, 評価	M-ESC

カーソルの直前の式の評価 (Scheme Interaction モード)	C-j
カーソルの直前の式の評価	C-c C-e
リージョン内の式の評価	C-c C-r

リストと式

式を 1 つ越えて先へ移動	C-M-f
式を 1 つ越えて逆方向へ移動	C-M-b
右の S 式を削除	C-M-k

Defuns

現在の defun の先頭に移動	C-M-a
次の defun の先頭に移動	C-M-e
現在の defun にリージョンを設定	C-M-h

プログラムの字下げ

現在行の字下げ	TAB
改行し, 字下げする	LFD または C-j

• **dired モード**

ディレクトリエディタ dired の起動	C-x d

ディレクトリエディタ dired でのキーの定義:

ファイル消去フラグを立てる	C-d または d
ファイルのコピー	c
ファイルを読み込む	e または f
次の行	SPC または n
前の行	p
ファイル名の変更	r
フラグを降ろす	u
フラグを降ろす	DEL
消去を実行する	x

• **正規表現**

正規表現においては, 以下のものは特別の意味を持っている:

任意の 1 文字	. (dot)
0 回もしくはそれ以上の繰り返し	*
1 回もしくはそれ以上の繰り返し	+
0 回または 1 回の繰り返し	?
与えられた文字集合内の文字	[...]
与えられた文字集合にない文字	[^ ...]
行の先頭	^
行の末尾	$
文字 c の引用	\c
代替 (“or”)	|
グループ化	\(... \)
n番目のグループ	\n
バッファの先頭	\‘

バッファの末尾	\'
語の先頭または末尾	\b
語の先頭または末尾以外	\B
語の先頭	\<
語の末尾	\>
語を構成する任意の文字	\w
語を構成する文字以外	\W

- **その他**

文字の交換	C-t
シェルコマンドの実行	M-!
カーソル位置の情報を表示	C-x =
バックアップファイルの作成, 非作成の切替え	M-x make-backup-files
emacs のバージョンを表示する	M-x emacs-version
ng のバージョンを表示する	M-x ng-version

Appendix D

Scheme 処理系の入手方法

本書で説明したいろいろな Scheme プログラムを実際に動作させるには、Scheme 処理系を入手する必要があります。幸いなことに、多くの処理系がフリーソフトウエアとなっています。フリーソフトウエアとは、開発者の善意により無料で配られているソフトウエアです[1]。入手した処理系を使ったり再配布するときは、附属の注意書きの内容をよく読み、守って下さい。

D.1 Scheme 処理系の一覧

1996 年 1 月現在での、Scheme 処理系のリストを示します。(以下に示したものの他にも、いろいろとあります。) Scheme 処理系の多くは、活発に改良されています。Scheme の処理系に関する最新情報は、インターネットでのネットニュースのニュースグループ comp.compliers にときどき投稿されています。それと同じ内容が WWW でも公開されています。URL は以下の通りです。

```
http://www.idiom.com/free-compilers/
```

これを、FTP でも入手できます:

```
ftp://ftp.idiom.com/pub/compilers-list/free-compilers
```

FTP とは File Transfer Protocol の略で、コンピューター間でファイルを転送するための方法のひとつです[2]。

ELK (Extension Language Kit)
最新版: 2.2
概要: 言語の拡張が容易にできる、Scheme 処理系です。X-Window インターフェース、動的ローディング、Unix システムコールインターフェースなどの機能があります。
作者: Oliver Laumann
動作要件: Unix, Ultris, MS-DOS など。

[1]無料だからといって粗悪なものばかりとは限りません。フリーソフトウエアには、市販ソフトウエアよりも遥かに品質の高いものも多くあります。ですが、市販ソフトウエアには用意されている電話質問などのサポートは、ありません。(開発者はフリーソフトウエアで生計を立てているわけではないので、当然です)

[2]本来の意味は、ファイル転送のためのプロトコルの名前です。FTP に従ったファイル転送をするプログラム名が ftp だったことにより、「ファイル転送」の意味でも使われるようになっています。

入手先: 以下の匿名 FTP にて公開されています。(無料)
ftp://fpt.x.org.contrib/elk-2.2.tar.gz

MIT Scheme
最新版: 7.2
概要: インタプリタ、コンパイラ、Scheme で書かれた Emacs 系エディタなど、多くの機能を持つ、巨大な Scheme 処理系です。
作者: マサチューセッツ工科大学 Scheme チーム
動作要件: 68000 CPU マシン (HP9000, Sun3, NeXT), MIPS CPU マシン (Decstation, Sony, SGI), i386 CPU マシン (MS-DOS, Windows, Unix) など
入手先: 以下の匿名 FTP にて公開されています。(無料)
ftp://altdorf.ai.mit.edu/archive/scheme-7.2

NGSCM
最新版: 3.3.1
概要: Scheme 処理系 SCM バージョン 4e1 を、テキストエディタ Ng バージョン 1.3L に組み込んだ Scheme システムです。
作者: 角川裕次 (SCM は Aubrey Jeffer 氏、 Ng は 吉田茂樹氏らによる)
動作要件: Unix (SunOS 4.1.x, Solaris 2.4, FreeBSD 2.x), MS-DOS. (ただし MS-DOS 版は、NGSCM バージョン 2 のみ利用可能)
入手先: 以下の匿名 FTP にて公開されています。(無料)
ftp://gull.se.hiroshima-u.ac.jp/pub/ngscm/

PC-Scheme
最新版: 3.03
概要: インタプリタ、コンパイラ、Emacs 系エディタなどの機能を持ちます。
作者: Texas Instruments 社
動作要件: i286, i386 CPU を持つ、IBM PC とそのコンパチブルマシンの MS-DOS 上で動作します。
入手先: $95 で販売されています。詳細は電子メイルで rww@ibuki.com にお問い合わせ下さい。

SCM
最新版: 4e1
概要: 多くのオペレーティングシステムで動作する、コンパクトで高速な Scheme インタプリタです。
作者: Aubrey Jeffer
動作要件: Amiga, Atari-ST, Macintosh, MS-DOS, OS/2, NOS/VE, Unicos, VMS, Unix
入手先: 以下の匿名 FTP にて公開されています。(無料)
ftp://nexus.yorku.ca/pub/oz/scheme/new/*

SOID (Scheme In One Defun)
最新版: 3.0
概要: 非常に小さな Scheme 処理系です。
作者: George Carrette
動作要件: Unix, VMS, Amiga, Macintosh, Windows NT, OS/2 など。
入手先: 以下の匿名 FTP にて公開されています。(無料)
ftp://ftp.cs.indiana.edu/pub/gjc/

VSCM

最新版: V0r3
概要: 非常に可搬性の高い Scheme 処理系です。処理系は C 言語の標準的な機能だけを使って書かれています。そのため、多くの機種で処理系のコンパイルが可能となっています。
作者: Matthis Blume
動作要件: Unix, Macintosh
入手先: 以下の匿名 FTP にて公開されています。(無料)
ftp://ftp.cs.indiana.edu/pub/scheme-repository/imp/vscm*.tar.gz

D.2 FTP の方法

Unix の場合での FTP の方法を示します。インターネットに読者の使っているコンピューターが接続されていないと、FTP はできません。最近はパソコン通信のホストコンピューターもインターネットに接続され、利用者に FTP 機能を公開しているものも多くあります。

インターネットの FTP サーバーでは、ソースコードの形で Scheme 処理系を公開している場合が多くあります。その場合は、自分でコンパイルしないといけません。パソコン通信のホストには MS-DOS 用の実行形式を置いていることがあります。コンパイルするのがめんどくさいときは、パソコン通信からダウンロードすることをおすすめします。

以下に NGSCM を入手する例を示します。ここでは、インターネットに接続された Unix ワークステーションから、FTP サーバー gull.se.hiroshima-u.ac.jp に接続する場合です。

1. FTP コマンドの起動
   接続先を指定して、FTP コマンドを起動します。

```
% ftp gull.se.hiroshima-u.ac.jp RET
Connected to gull.se.hiroshima-u.ac.jp.
220 gull FTP server (Version wu-2.4(10) Sun Apr 24 11:22:55 JST 1994)
ready.
```

2. ユーザー名を聞かれています。ftp と入力し、リターンキーを押します。

```
Name (gull.se.hiroshima-u.ac.jp:kakugawa): ftp RET
331 Guest login ok, send your complete e-mail address as password.
```

3. あなたの電子メイルアドレスを聞かれています。あなたの電子メイルアドレスを入力し、リターンキーを押します。(入力しても画面には表示されません。)

```
Password: ⟨あなたの電子メイルアドレス⟩ RET
```

   すると、 FTP サーバーからのメッセージが表示されます。(このメッセージは、 FTP サーバーによって大きく違います。)

```
230-WELCOME TO OUR FTP SERVER!!
230-
230-This ftp server provides the following software:
    ⋮ 省略
230 Guest login ok, access restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

   以上により、サービスを受ける権利のあるユーザーとして認証されたことになります。これ以降で FTP サービスを受けることができます。

4. ファイルの転送モードをバイナリモードにします。

```
ftp> type binary RET
200 Type set to I.
```

5. NGSCM のディレクトリに移動します。

```
ftp> cd pub/ngscm RET
250 CWD command successful.
```

6. 置かれているファイルの一覧を見ます。

```
ftp; dir RET
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 3074
drwxr-xr-x 2 1226 1000     512 May  7  1994 MS-DOS-2.22
-rw-r--r-- 1 1226 1000     769 Nov 16 06:19 PATCH-3.2.6-3.2.7
-rw-r--r-- 1 1226 1000  404661 Dec 27  1993 ngscm-1.12.tar.gz
-rw-r--r-- 1 1226 1000  506958 Jul 29  1994 ngscm-2.23.1.tar.gz
-rw-r--r-- 1 1226 1000  601809 May  7  1994 ngscm-2.23.tar.gz
-rw-r--r-- 1 1226 1000  521253 Nov 13 03:37 ngscm3.2.6-Ng1.3L+SCM4e1.tar.gz
-rw-r--r-- 1 1226 1000  521665 Nov 16 06:26 ngscm3.2.7-Ng1.3L+SCM4e1.tar.gz
-rw-r--r-- 1 1226 1000  521971 Jan 11 02:19 ngscm3.3.1-Ng1.3L+SCM4e1.tar.gz
226 Transfer complete.
```

7. 目的のファイルを、手元のコンピューターに転送します。遠くはなれたところにある FTP サーバーに接続している場合や通信速度の遅い回線を使っている場合は、転送に時間がかかることがあります。そのときは気長に待ちましょう。

```
ftp> get ngscm3.3.1-Ng1.3L+SCM4e1.tar.gz RET
local: ngscm3.3.1-Ng1.3L+SCM4e1.tar.gz remote: ngscm3.3.1-Ng1.3L+SCM4
e1.tar.gz
200 PORT command successful.
150 Opening BINARY mode data connection for ngscm3.3.1-Ng1.3L+SCM4e1.
tar.gz (521971 bytes).
226 Transfer complete.
521971 bytes received in 6.7 seconds (76 Kbytes/s)
```

8. 転送が済んだので、 FTP を終了します。

```
ftp> quit RET
```

9. インストールします。
ほとんどすべての Scheme 処理系はソースコードの形で配布されているので、ソースコードをコンパイルする必要があります。

参考文献

- Steven Allen Robenalt, “Catalog of Free Compilers and Interpreters,” `http://www.idiom.com/free-compilers/`, 1995/05/04.

索引

あ

か

さ

た

な

は

ま

や

ら

わ

記号

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z