ソフトウェアのしくみ

放送大学 岡部 洋一

まえがき

コンピュータが出現したころ，その仕事の大部分は四則演算であった。三角関数の計算も微分方程式を解くのも，すべて四則演算の積み上げで近似的に実行できるため，弾道計算，惑星の軌道計算などに利用されてきた。

しかしその後，コンピュータで文字列が扱えるようになると，文書作成，表計算，データベースなどという新たな領域を獲得した。また，音（音声や音楽）や画像（図や写真や動画）などを扱えるようになると，いわゆるマルチメディアであるゲームや芸術にも利用されるようになり，さらに Web の発明により，情報の伝達は世界に拡がったのである。

このように，コンピュータが生活のすみずみにまで入り込むようになったのは，ハードウェアとソフトウェアの圧倒的発展による。まずキーボードやディスプレー，スピーカーにマイク，カメラなどが取り付けられ，入出力の情報が巨大化し，さらにこれらを収納できるメモリーやハードディスクなどの周辺機器の大容量化と高性能化がはかられた。これらハードウェアの速度と規模の増大とともに，ソフトウェアが容易に開発できるようになり，マルチメディアやデータベースなどの情報量の大きなデータを扱えるような環境が整備されたのである。これに加え，ネットワークという別のハードウェアの発展も，大量情報の世界への発信と獲得を促進している。

ハードウェアがいわば量としての進展を促進したのに対し，質の進展はソフトウェアが果たしてきたというのが私の見解である。それも当初は機械語といってコンピュータにしか理解できなかった命令で作成していたのであるが，高級言語を使うようになり，ソフトウェアの開発力は

一気に上がったのである。また，すべてのソフトウェアを一人で開発していたのに対し，場合によっては千人もの開発者で開発できるような改良もなされてきた。

本書では，ソフトウェアがどのようにハードウェアを動かすのか，ソフトウェアの基本的構造や概念を説明し，応用ソフトウェアの構成法，多人数でソフトウェアを作成する際の手法などについて述べる。なお，特定の言語に大きく依存しないような記述には心掛けたが，できれば一つぐらいの言語を知った上での学習を勧める。そのほうが，自分の修得している言語の特長，相互の違いなどが正確に把握できるからである。

本書にはまた，多くの用語や概念が出現してくる。各概念に対し，なるべく，詳細な説明を行うように心掛けたが，それでもある程度基礎的あるいは常識的な概念については，説明が簡略化されている場合もあろう。また，私の説明だけでは理解しづらい場合もあろう。そうしたときに，Web の利用が好ましい。ぜひとも，実際にコンピュータの触れる環境にあり，自分で Web などで内容を確かめながら，さらには，簡単なプログラムを作成しながら，本書を読み進めていただくことを切に期待する。

著者

目　　次

放送とのおおよその対応表を掲載する。

放送	本テキスト
第 1 回	第 1 章，第 2 章
第 2 回	第 3 章，第 4 章 4.2 節まで
第 3 回	第 4 章残り
第 4 回	第 5 章 5.3 節まで
第 5 回	第 5 章残り，第 6 章
第 6 回	第 7 章
第 7 回	第 8 章 8.2 節まで
第 8 回	第 8 章残り
第 9 回	第 9 章
第 10 回	第 10 章
第 11 回	第 11 章 11.3 節まで
第 12 回	第 11 章残り
第 13 回	第 12 章
第 14 回	第 13 章
第 15 回	第 14 章

1 ソフトウェアとは

《目標&ポイント》ソフトウェアとは何か，ハードウェアとの比較，ハードウェアとの関連の説明から始め，コンピュータというハードウェアに対するソフトウェアであるプログラムについて概説する。

《キーワード》ソフトウェア，ハードウェア，コンピュータ，プログラム，機械語，アセンブラ言語，高水準プログラム言語

1.1 ソフトウェアとハードウェア

ハードウェア (hardware) という言葉を聞くことがあろう。ウェアとは道具といった意味があるので，ハードウェアとは硬い物，つまり機械のことを意味する。本書で取り上げる情報あるいはコンピュータの世界では，**パソコン** (personal computer)，**マイクロプロセッサ** (micro-processor) といった**コンピュータ** (computer) を指す。マイクロプロセッサーとは，家電製品とか製造機械のようなハードウェアの中で，これらを制御するために置かれた専用コンピュータであり，キーボードとかディスプレーは持たないものが多い。

これらに対し，**ソフトウェア** (software) とは柔らかい物という意味で，狭義には機械を動かす側のしくみ，具体的には**プログラム** (program) を意味する。さらに広義にはコンピュータシステムを運用する人や組織まで含むこともある。本書では，ソフトウェアといえば，狭義のプログラムを指すものとする。

他の例をあげてみよう。例えば，電力システムでいうと，発電機，変圧器，送電線といったものがハードウェアである。これらに対し，発電電力を制御したり，周波数を制御したり，変電所の供給先を切り替えたりするシステムがソフトウェアである。もともと，これらの多くは人間が行ってきたが，長い年月をかけて，リレーシステム，コンピュータシステムへと変遷してきた。人間で処理していた時代には，人間そのものがソフトウェアとして働いていたのである。ただ，人間なので，ときどきその手順を無意識あるいは故意に変える可能性がある。これを補うのがマニュアルである。その意味で，マニュアルがソフトウェアといえる。さて，人間の代わりに，リレーシステムやコンピュータシステムが使われるようになると，リレーやコンピュータはハードウェアであるので，ソフトウェアとはこれらを指令するプログラムということになる。現在は，ほとんどあらゆるシステムがコンピュータで制御されるようになってきているため，ソフトウェアの重要性はますます高くなってきているのである。

ハードウェアとしてのコンピュータは電子回路であるので，それに対する指令も電気信号の形で与えらえる。電子回路には連続的な量を扱うアナログ回路と不連続な量を扱うディジタル回路があるが，コンピュータはいうまでもなくディジタル回路である。さらに，不連続といっても，「0」と「1」（今後 0/1 と記載する）しかない 2 値のディジタル量を扱う電子回路である。1 以上の大きな数を扱う場合には，その大きな数を 2 進数に変換し，何桁かで表現される 0/1 の集合体として処理する。また，近年，アナログ量もアナログ–ディジタル変換回路を通して，2 進数に変換してから処理することが多くなってきている。なお，2 進数 1 桁は**ビット** (bit) という単位で呼ばれる。

コンピュータに与える指令は**命令** (instruction) と呼ばれ，やはり 0/1 の

集合で与えられる。0/1 の集合体は**コード** (code) ともいわれるので，**命令コード** (instruction code) とも呼ばれる。コンピュータに与えられる命令コードはコンピュータ内の**メモリー** (memory) に蓄えられている。メモリーは一定のサイズのものが多数並んだ形で構成され，それぞれ 0 番から順に**アドレス** (address) が付けられている。アドレスは**番地** (address) とも呼ばれる。本書では，説明文中ではアドレスと呼ぶが，図中などで短縮した文字列を使う際は番地という。

コンピュータは，これらの指令をアドレスの低いほうから順番に読み込んでいって，その指示通りの作業をこなしていくことにより，複雑な計算などの情報処理をしていくのである。

1.2 プログラムの種類

0/1 の羅列である命令コードは**機械語** (machine language) と呼ばれる。一例をあげれば，

```
0000 0000 0001 0010
```

のような形をしており，人間には極めて読みにくく，また指令書を作るのも容易ではない。これを

```
ADD X, Y, Z
```

などと記載すると，「X レジスタの内容と Y レジスタの内容を加えて Z レジスタに入れよ。」という意味であることが比較的容易にわかるようになる。なお，レジスタとは計算部分のすぐ傍に置かれた小さなメモリーである。こうした記述によるプログラム表現を**アセンブラ言語** (assembler

language) と呼ぶ。アセンブラ言語は文字[1)]を使っているが，簡単に機械語に変換できることが特長である。もちろん，こうした文字列から 0/1 の組み合わせである機械語を作り出すアセンブラというプログラムも必要となる。

さらに，上記の命令を

```
Z = X + Y
```

などと[2)]書くことができると，もっと人間にとって読みやすい，あるいは書きやすい表現となる。こうした，より人間的なプログラム表現を**高水準プログラム言語** (high level program language) と呼ぶ。COBOL，BASIC，C，Pascal，LISP，Java などという言葉を聞いたことがあるかも知れないが，こうしたものはすべて高水準プログラム言語の名称である。

演習問題 1

問題 1.1　次の用語を理解したかどうか確認せよ。

1）アドレス
2）命令コード
3）機械語
4）アセンブラ言語
5）高水準プログラム言語

1）0/1 しか扱えないコンピュータが文字をどうやって扱うかは，後に述べる。
2）多くの高水準プログラム言語で，‘=’ は左右が等しいという意味ではなく，右の計算結果を左へ代入するという意味である。したがって ‘X = X + 1’ もあり得る。

2 コンピュータのしくみ

放送大学 岡部 洋一

《目標＆ポイント》ソフトウェアであるプログラムによって動作を行うコンピュータの機能と概念を理解し，プログラムとどう関わりうるのかを学習する。
《キーワード》コンピュータ，命令，命令コード，演算命令，移動命令，ジャンプ命令，蓄積プログラム方式

2.1 コンピュータと電卓

ソフトウェアの働きを理解するには，その対象であるハードウェアのしくみを理解しておく必要がある。我々の場合にはコンピュータである。コンピュータは**図** 2.1 に示すように，**中央処理装置** (central processing unit, CPU) を中心とし，その周りにプログラムやデータを格納しておく**メモリー** (memory)，また，キーボード，ディスプレーなどの**周辺装置** (peripheral unit) を配した構造となっている。コンピュータが発達するにつれ，メモリー以外に周辺装置の一つとして，ハードディスクなどの**大容量蓄積装置** (mass storage) も使われるようになっているため，メモリーはこれらと区別するために，**主メモリー** (main memory) とも呼ばれるようになってきている。大容量蓄積装置については，改めて議論することとする。

コンピュータを動かすには，メモリー上にあらかじめプログラムを用意しておき，メモリー上の所定のアドレスからメモリーの内容を読み込み，そこに記載された命令にしたがって，次々とその命令を実行してい

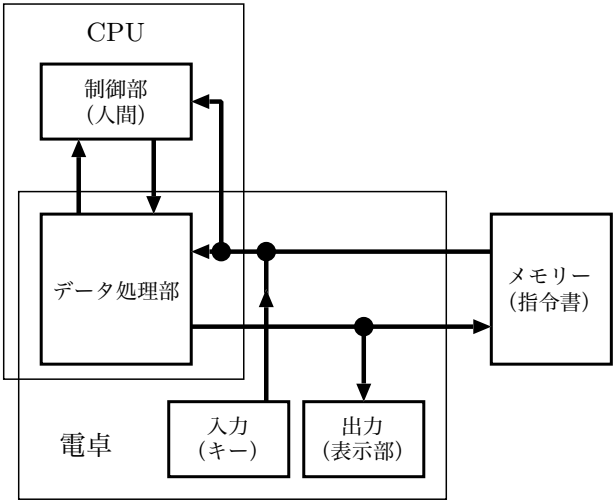


図 2.1　コンピュータの構成（括弧内は人間が電卓を使う場合との対応）

くしかけになっている。といってもなかなかピンとこないかも知れない。コンピュータの構成は，人間が電卓を使う場合と類似性が高い。図では，コンピュータの中の CPU をさらに**データ処理部** (data processing unit) と**制御部** (control unit) と二つに分けて描いてある。

データ処理部とは，簡単にいうと電卓のようなものと理解してよい。厳密には，データ処理部とは，電卓からキーボードと表示装置を取り除いたようなものである。データ処理部は，四則演算に必要なあらゆる演算を処理する機能が納められている。四則演算，符号反転といった算術演算以外に，ビットごとの論理演算などを行う機能もある。この部分は**算術論理回路** (arithmetic logic unit, ALU) と呼ばれる。また，全ビットを左右にシフトするような機能もある。乗除算は，加減算とシフト機能で実現できるので，これを省いたデータ処理部もある。コンピュータは，も

ともと膨大な算術計算をするように設計されたが，通常の電卓が持っていない論理演算もできるため，その後，データの整理などの各種の用途に使われるようになった。

さらに，データ処理部には，複数の**内部メモリー** (internal memory) が用意されている。通常の電卓はメモリーが一つのものが多いが，コンピュータの CPU の内部メモリーは数個から十数個と数が多い。これら複数のレジスタを有効に使い回すことにより，複雑な作業をこなしているのである。コンピュータで，単にメモリーというと，CPU の外に置かれていて，プログラムやプログラムが使うデータを格納する**外部メモリー** (external memory) のことを指すため，内部メモリーは特に**レジスタ** (register) と呼ぶことになっている。以下，本書でも内部メモリーはレジスタ，外部メモリーは単にメモリーと呼ぼう。

電卓は人間が使うものであるが，コンピュータは制御部という電子回路が利用するものであるため，これらはすべて電気信号でやりとりするようになっている。つまり，データ処理部には何本かの入力があり，ある入力線は数値データを送り込むためのものだったり，ある線は「+」とかいった命令を送り込むために使われる。ディジタルの世界では，信号は 0/1 しかないので，電卓にあるように複数のボタンと対応をとるためには，何本かの線の束で，どのボタンを押したかを知らせる必要がある。この押しボタンに対応する電気信号のことを**命令** (instruction) と呼ぶ。

また，電卓には表示装置が付いていて，人間に計算結果やエラーが発生したことを通知してくれる。これも，コンピュータでは，データ処理部から何本かの電気信号を伝える線が出ていくことになる。ある出力線は計算結果を送出するためだったり，ある線はデータ処理部の計算がうまく遂行されたか，エラーがあったかなどを知らせるために使われる。これらデータ処理部から制御部へ送られる電卓のエラーに相当する情報

は，**フラグ** (flag) と呼ばれる。コンピュータの場合には，電卓より詳細な情報を送出するため，フラグ情報用の線数も多い。

2.2　命令コード

コンピュータのデータ処理部に作業を指示するのは，制御部と呼ばれる電子回路である。したがって，命令は 0/1 の組み合わせで構成される。その例を**図** 2.2 に示す。この 0/1 の命令コードを覚える必要はまったくない。まず，命令コードは CPU ごとにまったく異なる場合が多いし，ここで記載したものも，私が仮想的に想定した CPU のものだからである。この表で「対応するアセンブラ命令」は，ある程度，CPU 依存性があるものである。しかし，それでも，読者がアセンブラ命令を使うことはほとんどないので，この表は最初の数章で必要なだけの参考として見てほしい。

まず，命令コードは，原則，一つのアドレスに納まるように設計されている。しかし，命令の中でアドレスを指定するような場合には，どうしても命令が長くなるため，二アドレスや三アドレスにわたる命令も必要となる。表中二行で表現されている命令は，このような二語命令である。

電卓と同じように，加算や減算などの**演算命令** (arithmetic instruction) は当然持っているが，それ以外に，（外部）メモリーから内部メモリーであるレジスタへデータを読み込んだり，逆にレジスタの内容をメモリーへデータを書き出す**移動命令** (move instruction) と呼ばれるものがある。これは，電卓のアナロジーでいえば，指令書に書かれたデータをキーボードから電卓へ入れたり，逆に表示部の内容を指令書へ書き写すことに対応する。

また，命令はメモリー上に行単位で作業順に書かれているのが通常で

対応するアセンブラ命令 (説明)		命令コード
演算命令	(算術や論理演算を行う)	
ADD i, j, k;	(Reg. i と Reg. j の加算結果を Reg. k に入れる)	0000 (i) (j) (k)
SUB i, j, k;	(Reg. i から Reg. j を減算した結果を Reg. k に入れる)	0001 (i) (j) (k)
AND i, j, k;	(Reg. i と Reg. j のビットごとの AND をとり, 結果を Reg. k に入れる)	0100 (i) (j) (k)
OR i, j, k;	(Reg. i と Reg. j のビットごとの OR をとり, 結果を Reg. k に入れる)	0101 (i) (j) (k)
EOR i, j, k;	(Reg. i と Reg. j のビットごとの EOR をとり, 結果を Reg. k に入れる)	0110 (i) (j) (k)
SHL i, k, n;	(Reg. i を n ビット左シフトした結果を Reg. k に入れる)	1000 (i) (k) (n)
SHR i, k, n;	(Reg. i を n ビット右シフトした結果を Reg. k に入れる)	1001 (i) (k) (n)
MOV i, k;	(Reg. i の内容を Reg. k に入れる)	1100 0000 (i) (k)
NEG i, k;	(Reg. i の補数を Reg. k に入れる)	1100 0001 (i) (k)
ADD1 i, k;	(Reg. i に 1 を加えた結果を Reg. k に入れる)	1100 0010 (i) (k)
SUB1 i, k;	(Reg. i から 1 を引いた結果を Reg. k に入れる)	1100 0011 (i) (k)
NOT i, k;	(Reg. i の各ビットの NOT を Reg. k に入れる)	1100 0100 (i) (k)
移動命令	(メモリーや周辺装置とのデータのやり取り)	
LD i, n;	(メモリーの n 番地から Reg. i 番へロード)	1110 0000 0000 (i) (n)
ST i, n;	(Reg. i からメモリーの n 番地へストア)	1110 0000 0001 (i) (n)
ジャンプ命令	(無条件ジャンプ, 条件ジャンプなど)	
JP n;	(n 番地 (次の実行命令の番地) へ飛ぶ)	1111 0000 0000 0000 (n)
JPZ n;	(零フラグが立っていればメモリーの n 番地へジャンプ, そうでないときには次の番地へ移動)	1111 0000 0000 0001 (n)
JPN n;	(負フラグが立っていれば n 番地へジャンプ, そうでないときには次の番地へ移動)	1111 0000 0000 0010 (n)
JPC n;	(MSB より上位へキャリーがあれば n 番地へジャンプ, そうでないときには次の番地へ移動)	1111 0000 0000 0011 (n)
JPO n;	(オーバフローフラグが立っていれば n 番地へジャンプ) そうでないときには次の番地へ移動)	1111 0000 0000 0100 (n)
HLT;	(動作を停止する (自分自身のアドレスへジャンプする))	1111 1111 1111 1111

図 2.2　命令セットの例（Reg. はレジスタの略）

あるが，場合によっては次の行の命令でなく，離れた行へ跳んでいってそこからの命令を実行できるようなしかけがあるほうが便利なので，行を跳ぶための**ジャンプ命令** (jump instruction) と呼ばれる命令も用意されている。ジャンプ命令は，次に実行するアドレスを収納している**プログラムカウンタ** (program counter, PC) を書き換えることで，実行される。

ジャンプ命令のうち，JP は**無条件ジャンプ** (unconditional jump) 命令と呼ばれ，そこへ来ると指定のアドレスへ必ずジャンプする。HLT はプログラムを停止させる命令であるが，自分自身へジャンプする命令でも達成できるため，本書では無条件ジャンプ命令に分類した。しかし，HLT は純粋なジャンプ命令ではないという立場をとる書もあり，そうした書では，通常のジャンプ命令と HLT をあわせて，**制御命令** (control instruction) と呼ぶ場合もある。

条件が満足されればジャンプし，そうでなければジャンプしない JPZ，JPN，JPC，JPO といった**条件ジャンプ** (conditional jump) 命令は，それぞれ対応するフラグを見てジャンプするかどうかを決定する。必ずしも，その詳細を理解する必要はないが，簡単に説明をしておこう。

フラグ (flag) とは演算命令の結果の概要を知らせてくれる警報のようなものである。JPZ は演算結果が 0 の場合に出る零フラグが立っているときにジャンプする。これと減算演算を利用すると，二つのレジスタの内容が等しいかどうかを判断できる。JPN は演算結果を符号あり整数とみなしたときに負，つまり，ビット幅の最上位が 1 のときに発生する負フラグでジャンプする。JPC は一つ前の演算で，ビット幅以上にキャリーの出入りがあった場合，つまり，符号なし整数とみなして演算を行った結果，オーバフローを起こした場合に立つキャリーオーバフラグによりジャンプを行う。JPO は一つ前の演算で，符号あり整数とみなして演算を行った結果が正しくないときに立つオーバフローフラグによってジャ

ンプする。

なお，ここで符号つき整数とは，負数を 2 の補数[1] で表すことにより正負の両方を表現できるように定義された数で，例えば幅 4bit の場合，-8 から 7 までの整数を定義できる。一方，符号なし整数とは，同じビット幅で正数のみを扱うため，幅 4bit であると，0 から 15 までの整数を定義できる。符号あり整数に対して定義された ADD や SUB は，符号なし整数に対しても共用できる。ただし，いずれの場合も結果も同じビット幅で表現しようとすると，正しくない結果を出すことがある。これをオーバフローという。オーバフローフラグは符号あり整数に対して，ADD や SUB の結果が正しくないときに警報を出すように設計されているため，符号なし整数に対しては正しい警報とはならない。しかし，符号なし整数のオーバフロー検知はキャリーオーバフラグで知ることができる。その他，いろいろなフラグがあり，それに対応した条件ジャンプ命令があるのが普通であるが，本書では，ここで示したジャンプ命令のみで説明を行う。

電卓の指示書ではこうしたジャンプ命令が使われることは少ないし，やってほしいことが必ずしも順序立てて記載されているとも限らない。ジャンプ命令の代わりに，指示書中に矢印で指示されることもあるし，作業手順は口頭で伝えられることすらある。しかし，コンピュータの指示書であるメモリー上の命令群は，必ずやってほしいことを順に並べることになっている。第 1 行の命令の次に行うのは第 2 行の命令といったように，行に対応するメモリーのアドレスの順に記載される。ただし，1 命令 1 行とは限らない。一つの命令を記載するのに，1 アドレスでは足らず，複数アドレスを要する長命令と呼ばれる命令がある。例えば，**図** 2.2 に

1) ビット幅 n の場合，絶対値の同じ正負の数を加えたときに 2^{n+1} となるように定義される補数。

示した命令セットでも，移動命令である LD，ST，および HLT を除くジャンプ命令は番地を扱っているが，番地指定だけで 1 アドレスが必要なため，どうやっても，命令を記載するために 2 アドレス分が必要となる。厳密にいうと，長命令の次の命令は，直後のアドレスではなく，さらにその一つ先となるため，プログラムカウンタを二つ増やす作業が必要となる。ジャンプ命令は，何アドレスか先のアドレスへジャンプすることを指示する命令なので，これらも当然，順番を崩すこととなる。

一般には，ジャンプ命令の中にジャンプするアドレス先そのものを記載する**直接アドレス指定** (direct addressing) と，記載されたアドレスにあるレジスタの内容を加えた結果がジャンプするアドレスになるという**相対アドレス指定** (relative addressing) がある。これはジャンプ前にアドレスの加算という作業をすれば済むのであるが，後に述べるサブルーチンの実行ではたびたび登場するため，あらかじめ，命令セットの中に組み込まれている場合が多い。

また，**間接アドレス指定** (indirect addressing) という概念もある。これは移動命令やジャンプ命令で指定されたアドレスに別のアドレスが書いてあり，そのアドレスに対して作業を行うというものである。間接アドレス指定とは，ジャンプなどのアドレス欄にジャンプ先の参照が書かれていると思って作業することである。これもよく用いられるため，命令セットの一部に組み入れられていることもある。

機械語のプログラムは，これらの命令をメモリー上に並べることで構成される。

2.3 蓄積プログラム方式

人間が電卓を扱う際，すべての作業を頭に入れておくのも一つの方法であるが，複雑な作業を行うには，どのボタンをどの順番で押すかといった作業手順を書いたメモを用意するのがよいであろう。また，入力すべきデータ，途中のデータ，最終結果などのデータも，メモ用紙に記載するのが便利である。コンピュータの中では，こうした作業命令やデータはメモリー上に記載する。

命令とデータに異なるメモ用紙を使う方法もあるが，現在のコンピュータでは，命令とデータを混載してよいようになっている。こうした命令とデータを同じメモリーに混載して作業を行う方式を**蓄積プログラム方式** (stored program concept) と呼ぶ。もちろん，命令を用意する際，どれが命令でどれがデータであるのか，命令を用意する人，つまりプログラマはきちんと理解していなければならない。ここをいい加減にすると，データを命令だと思って実行し，パソコンを暴走させてしまうことすらあるのである。まだ，詳細は理解しなくてよいが，ある命令まで実行して，その先にデータが存在し，さらに先にまた命令があるような際，前の命令群の最後に無条件ジャンプ文を置くことで，命令の連続性が保持できることぐらいは理解しておくとよいかも知れない。

実は，この蓄積プログラム方式の導入により，コンピュータは大変大きな機能増大が図られたのである。それは高水準プログラムという概念である。**図** 2.3 に基づいて説明しよう。高水準プログラムとは人間に親和性のある文字の集合である。これをコンピュータのわかる機械語に翻訳しないと，コンピュータはその指示通りに動くことはできない。そこで，コンパイラと呼ばれる翻訳ソフトウェアが活躍する。このソフトウェアは，高水準プログラムをテキストの入力データとして機械語プログラ

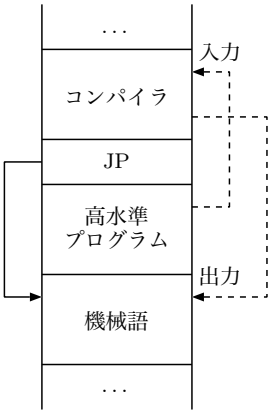


図 2.3　蓄積プログラム方式による高水準プログラム実行の動作原理

ムを出力とする。なお，コンパイラの入力は**ソースプログラム** (source program)（源の意味），出力は**目的プログラム** (object program) とも呼ばれる，その意味で，出力データはあくまでも 0/1 からなるデータである。これが断固データであると思っていると，それ以上，何も話は進まないが，これを機械語の命令だと思えば，命令の実行ができるのである。蓄積プログラム方式では，メモリー上に書かれているのが命令かデータかの区別は何もなく，その区別はプログラマが知っているだけである。そこで，例えば翻訳/通訳の作業が終了した時点で，この機械語データの領域にジャンプすると，そこから先はこれをデータとは理解せず，命令だと思って作業してくれることになる。

人間に親和性の高いテキストで書いた高水準プログラムが，なぜ実行可能になるのかは，こうしたしかけがあるからなのである。

演習問題 2

問題 2.1 次の用語を理解したかどうか確認せよ。

1) 命令コード
2) 演算命令
3) 移動命令
4) ジャンプ命令
5) フラグ
6) 蓄積プログラム方式

問題 2.2 ビット幅が 3bit である場合，000 から 111 までの 8 種類の符号なし整数（10 進数で 0 から 7）の加算を，すべての組み合わせに対し行い，その結果の下 3bit の結果を 8×8 の表にしてみよう。このうちで，3bit の範囲で結果が期待通りになっていないものに × を付せ。この × のついた結果に対してはキャリーオーバフラグが立つ。

問題 2.3 同様のことを，000 から 111 が符号あり整数（0 から 3 と-4 から-1）に対して行い，表を作成してみよう。このうちで，3bit の範囲で結果が期待通りになっていないものに × を付せ。この × のついた結果についてはオーバフローフラグが立つ。

3 プログラム

放送大学 岡部 洋一

《目標&ポイント》プログラムの概念を理解する。機械語プログラム，アセンブラプログラムと高水準プログラムとの関係を理解すると共に，アセンブラ，コンパイラ，インタプリタの存在を知る。

《キーワード》機械語，アセンブラ言語，アセンブラ，高水準プログラム言語，コンパイラ，インタプリタ，応用ソフトウェア

3.1 機械語プログラム

プログラム (program) とは，ある特定のルールにしたがった**プログラム言語** (program language) で書かれた指令書のことである。そのルールのことを**文法** (grammar) という。プログラムを書く人を**プログラマ** (programmer) と呼ぶ。電卓でいえば，プログラムとは，機械を制御するための指令書のようなものである。機械が理解できる 0/1 の組み合わせで表されるプログラムは，CPU にもっとも密着したもので，**機械語プログラム** (machine language program) と呼ばれる。この 0/1 の組み合わせを**機械語** (machine language) という。

機械語プログラムの一例として，0x0010 番地と 0x0011 番地にある二つの数を合計して，結果を 0x0012 番地へ書き込むプログラムを**図** 3.1 左に示そう。本当は 0/1 の組み合わせであるが，それでは桁が増えるので，16 進表現命令部分については，**図** 2.2 に示した対応するアセンブラ言語の略号を用いて記載している。なお，0x とは以後の数字が 16 進表現

図 3.1　二つの数の和を計算する機械語プログラム（16 進表現している）とアセンブラプログラム

であることを示す。例えば，図のアドレス 0x0000 番地の内容は 0xE000 となっている。これは 10 進表現（‘10’,‘11’,...,‘15’），つまり 2 進表現（‘1010’,‘1011’,...,‘1111’）が 16 進表現では（‘A’,‘B’,...,‘F’）になることに着目すれば，2 進表現では ‘1110 0000 0000 0000’ である。**図** 2.2 の移動命令にある ‘LD 0, ...’ に対応することがわかるであろう。以下，2 進表現の命令コードのままではわかりづらいので，図右のアセンブラプログラムと書かれた部分に示したものを参考にして欲しい。

プログラムの流れを説明しておこう。0x0010 番地には，事前に被加数

を入れておく。0x0011 番地には，事前に加数を入れておく。0x0012 番地には，加算の結果が入る。まず，0x0010 番地と 0x0011 番地の内容を，LD（格納）命令を使って，R0 および R1 へコピーする。続いて，ADD（加算）命令を使って，R0 および R1 の内容を加算し，その結果を R0 へ代入する。最後に，ST（蓄積）命令を使って，R0 の内容を 0x0012 番地へコピーし，HLT（停止）命令で，作業を停止[1]　する。

改めて強調するが，機械語は使うコンピュータの種類，もっといえば CPU の種類ごとにまちまちである。ここでは ADD のようなアセンブラ命令を併記したが，実は ‘ADD 0，1，0’ という命令の具体的実現である 0/1 のコードも CPU の種類ごとに異なるのである。例えば，ここで仮定した仮想的な CPU の場合には，0x0010 つまり 2 進表現では ‘0000 0000 0001 0000’ となる。しかし，これを見ても，一般の人には極めて理解が難しい。コンピュータがわかるだけである。このような理由から，現在，機械語のプログラムを直接書く人はほぼ皆無であるといってもよいであろう。

3.2　アセンブラプログラム

機械語のプログラムとほぼ同一内容であるが，それを多少，読みやすくしたプログラム言語を**アセンブラ言語** (assembler language)，それを使って記載されたプログラムを**アセンブラプログラム** (assembler program) という。改めて**図** 3.2 に**図** 3.1 に対応するアセンブラプログラムを示す。**図** 3.1 右もアセンブラプログラムであるが，若干異なっている。データの格納されているメモリーの具体的アドレスが書いてなく，代わりに**ラベル** (label) と呼ぶ Data0，Data1，Data2 を使って書かれている。ラベルは英数字を組み合わせた単語で，行の先頭から始まるものとする。特

1）停止してしまったコンピュータは，通常リセットボタンを押すことで，再び指定されたアドレスからプログラムを再実行することができる。

```
        LD 0, Data0;   // 被加数のロード
        LD 1, Data1;   // 被加数のロード
        ADD 0, 1, 0;   // 加算
        ST 0, Data2;   // 結果のストア
        HLT;           // 停止
Data0:  0x0005;        // 被加数
Data1:  0x0006;        // 加数
Data2:  0x0000;        // 結果
```

図 3.2　二つの数の和を計算するアセンブラプログラム

に，特定のアドレスを指定したいときには，数字の組み合わせ（10 進数）または 0x から始まる 0-9，A-F（16 進数）を置く。また，行頭の空白もしくはラベル後の空白の後ろは，命令またはデータとする。

実はアセンブラプログラムは**アセンブラ** (assembler) という翻訳プログラムによって機械語プログラムに変換されるのである。この際，アルファベットで書かれた命令は 0/1 の機械語に翻訳される。さらに，Data0 などが具体的にどのアドレスを指すかは，アセンブラが勝手に決めてくれるのである。確かに，プログラムの書かれたメモリーアドレス領域と重ならなければ，Data0 は 0x0010 番である必要はない。どの位置でもよいのである。アセンブラにメモリーの割り付けを頼めるだけ，プログラマの負担は軽減されるのである。

アセンブラプログラムは，ほぼそのまま機械語に変換されるため，CPU 依存性は相変わらず高い。したがって，CPU が異なれば，別の命令セットを使って記載しなければならない。しかし，優秀なプログラマが無駄の少ないプログラムを書けば，動作速度も速く，メモリー領域もあまりとらない機械語が生成できる。したがって，動作速度やメモリーの制限

が非常に厳しい場合には，アセンブラプログラムが使われることもある。しかし，ほとんどのプログラマは次に示す高水準言語を使って，プログラムを記載することが多い。

3.3　高水準プログラム

人間との親和性の高いプログラム言語である**高水準プログラム言語** (high level program language) を使って記載されたプログラムが**高水準プログラム** (high level program) である。C，C++，Java，FORTRAN，BASIC，Perl，Ruby など，よく 耳にするプログラム言語はほとんど，高水準プログラム言語である。また，これらの多くは CPU に依存せず，多様な機械で使える。それは，機械に依存しないプログラムから，機械に依存するアセンブラプログラムを作り出すコンパイラやインタプリタと呼ばれるソフトウェアがあるからである。

コンパイラ (compiler) は翻訳者の意味で，高水準プログラム全体を一気に翻訳してアセンブラ言語にするプログラムである。翻訳のことを**コンパイル** (compile) ともいう。コンパイルされた結果は，アセンブラにより機械語のプログラムに変換され，CPU はそれを実行することになる。C，C++，Pascal，FORTRAN といった高水準プログラム言語は，コンパイラ方式である。

コンパイラのプログラム自体は，ビルのタワークレーンの立ち上げと同じような手法で開発される。タワークレーンの場合には，最初は小さなタワークレーンを人力で運びあげて作る。次にその小さなクレーンを使って，より大きなクレーンを作りあげる。というように，順に大きなクレーンを実現していくのである。コンパイラの場合には，まずアセンブラ言語を使って，より簡単な高水準プログラム言語のコンパイラを開

発し，それを使ってより高次のコンパイラを作り上げるというサイクルを続けていくのである。

インタプリタ (interpreter) とは通訳の意味で，高水準プログラムを行単位（もう少し大きい単位の場合もある）に機械語になおして実行するプログラムである。まさに通訳のように，ちょっとずつ翻訳していく。したがって，少しずつプログラミングし，テストをしていくという方法が可能である。一方で，完成したプログラムといえども，行単位の翻訳をしながら実行するため，高速な動作は期待できない。プログラム開発は楽であるが，速度の出ない手法であるため，短いプログラムに適している。なお，この欠点を補う方法として，コンパイルもできるインタプリタもある。BASIC，Perl，Python，Ruby，Lisp，Prolog，Smalltalk などはインタプリタ方式である。

なお，初期の Pascal や Java は，コンパイラによって比較的アセンブラ言語に近い形の中間コードと呼ばれるものを生成し，それをインタプリタによって実行する。しかし，アセンブラ言語も一種の中間コードであるとみなすこともでき，コンパイラとインタプリタの境は昔ほど明確ではなくなってきている。このため本書でも，これらの区別をしないで単にコンパイラと呼ぶこともある。

高水準プログラム言語は多数あるが，一例として，今まで示した加算のプログラムを C 言語で記述したものを**図** 3.3 に示しておこう。C では，すべてのプログラムを関数形式で記載することになっている。しかも，一番最初に実行してほしいプログラムには main という関数名を付けることになっているため，第 1 行と最終行で ‘main() {...}’ としたが，ここではその詳細を知る必要はない。第 2 行からが本質的なプログラムである。まず，内部で使う変数 x，y，z を整数型として宣言している。変数とは，いろいろな作業を行う際，一時的に値を格納しておく領域であ

```
main(){
    int x, y, z;
    x = 4;
    y = 5;
    z = x + y;
    print z;
}
```

図 3.3　二つの数の和を計算する C プログラム

り，アセンブラプログラムの Data0，Data1，Data2 に相当する。第 3 行から第 5 行にわたって等号 '=' が現われるが，これは右辺の計算結果を左辺へ代入するという意味であり，左辺と右辺が等しいことを示してはいない。

機械語プログラムやアセンブラプログラムと異なり，高水準プログラムの多くは，作業が終わった際，どこにデータが残っているかを教えてくれないし，その領域の保全をしてくれない。そのため必要な情報をユーザに伝えるような命令を実行する必要がある。作業第 6 行の 'print z;' はそのためのものである。この命令により，プログラムはディスプレーを経由してデータをユーザに示す。なお，C では print という命令はなく，近い意味を持つ別の命令が用意されているが，わかりやすくするために，単に print とした。

3.4　応用プログラム

何らかの利用を目的として，ベンダやユーザが開発したプログラムを**応用プログラム** (application program) と呼ぶ。これらは通常，コンパイルされて提供されるが，多くの場合**応用ソフトウェア** (application software)

と呼ばれる。応用ソフトウェアという言葉は厳密には応用プログラムも含む広い概念であるが，すぐに使えるようになったものを指すことが多い。この章に示した小さなプログラムも一種の応用プログラムであるが，社会常識的にはもう少し大きな事務処理用とか，科学技術計算用などのプログラムを指すことが多い。アセンブラ，コンパイラ，インタプリタも広い意味では応用ソフトウェアの一種であるが，これらも応用プログラム開発用ツールとして除外されることが多い。

演習問題 3

問題 3.1 次の用語を理解したかどうか確認せよ。

1) プログラム
2) 機械語
3) 16 進表現
4) アセンブラ言語，アセンブラ
5) 高水準プログラム言語
6) コンパイラ，インタプリタ

問題 3.2 ラベル Data0 に入っている内容からラベル Data1 に入っている内容を引いた結果をラベル Data2 へストアするアセンブラプログラムを書け。

問題 3.3 ラベル Data0 に符号あり整数が入っているものとし，その絶対値をとるアセンブラプログラムを書け。正負の判断は NEG という命令を実行し，その結果，負（つまり元は正）ならば JPN で終了点へジャンプし，それ以外の場合はもう一度 NEG を実行し，終了すればよい。

4 制御構造と構造化プログラミング

放送大学 岡部 洋一

《目標&ポイント》ジャンプ命令で実現される分岐，ループといった作業と，それを読みやすくするための工夫である構造化プログラミング，さらに構造化されたいくつかのプログラム構造である制御構造について述べる。

《キーワード》制御構造，ジャンプ命令，ジャンプ，分岐，ループ，構造化プログラミング

4.1 ジャンプ命令

電卓で，計算の途中で絶対値を使うような場合を考えよう。これは値が正だったらそのまま，負だったら，符号反転キーを押すことで達成できる。符号反転キーがなければ，-1 を掛けてもよい。このように，条件によって作業手順を変えることはしばしばあるが，コンピュータのプログラムでは，プログラムをメモリー上に 1 次元的に並べることになっているため，作業手順を変えるには何らかのジャンプが必要である。つまり，通常は 1 次元的に配列された命令を上から順にこなしていけばよいが，異なる作業をしなければならないときには，その作業の記載された場所へジャンプするのである。

実際のコンピュータでは，内部に**プログラムカウンタ** (program counter, PC) と呼ばれるところがあり，そこには次に実行すべき命令のアドレスが記憶されている。したがって，まずプログラムカウンタに書かれているアドレスの命令を持ってくる。この命令を CPU に持ってくる作業を**フェッ**

チ (fetch) という。次に，通常の命令では，プログラムカウンタの値を 1 増しておく。そして命令を実行する。プログラムカウンタに書かれたアドレスから，次の命令をフェッチする。以下，これを繰り返すことで，命令をメモリーの順番に実行することができる。一つのアドレスに収容できないような長い命令もある。こうした長命令については，プログラムカウンタをそのぶん余計に増加しておく必要があるが，本書では，説明の簡略化のため，長命令の場合でも 1 アドレスに収容されているような表現を使うこととする。

ジャンプ命令は，命令の後ろに書かれたジャンプ先のアドレスをプログラムカウンタに設定する。すると，次にフェッチすべき命令は一つ先ではなくなるため，ジャンプが可能となるのである。ジャンプを行うには，**図** 2.2 に示した**ジャンプ命令** (jump instruction) と分類される命令を用いる。演算命令，移動命令に次ぐ第三の種類の命令群である。これには**無条件ジャンプ** (unconditional jump) と**条件ジャンプ** (conditional jump) とがある。無条件ジャンプは，いかなる場合でも必ずジャンプ先のアドレスへジャンプさせるというものである。アセンブラ言語では，しばしば，JP とか GOTO の後ろにジャンプ先を付けて記載する。

一方，条件ジャンプとは，ALU などの結果によって，次に実行する命令をジャンプ先にしたり，すぐ次の行にしたりする命令である。例えば

```
    JPZ     JP_PNT;
```

は，ALU の内容が 0 ならば，次の実行命令はラベル JP_PNT のついたアドレスに存在していることを示す。これを JP_PNT へジャンプするという。ALU の内容が 0 でなければ，この命令の直後に書かれた命令を実行する。つまり，条件が満たされたときにのみ，ジャンプすると覚えておけばよい。

この他，ALU が負の場合にジャンプする JPN とか正の場合にジャンプ

アドレス	命令	説明
	LD 0, Data0;	// R0 へ Data0 の内容を設定
	NEG 0, 0;	// 符号反転
	JPN Label0;	// R0 が負ならば Label0 へジャンプ
	JP END;	// 正だったら終了処理
Label0;	NEG 0, 0;	// 正だったら符号反転
END:	ST 0, Data1;	// 結果を Data1 へ代入
	HLT;	// 停止
Data0:	0x0004;	// 絶対値をとりたい数
Data1:	0x0000;	// 結果

図 4.1　絶対値を求めるアセンブラプログラム

する JPP とか，演算がオーバフローした際にジャンプする JPO などがある。**図** 2.2 に示した命令セットでは命令数を著しく限ったため，JPP はなく，JPZ と JPN を連続して使うことで実現する。

例として，変数 X の内容の絶対値をとるアセンブラプログラムを**図** 4.1 に示そう。条件ジャンプを行うには，ALU で何らかの演算をした結果でフラグが立つので，ここでは符号反転してみる。結果が負なら，Label0 へ条件ジャンプし，もう一度符号反転して正とし，そのまま END へ入って結果を Data1 へ代入する。結果が正なら，END へ無条件ジャンプし，結果を Data1 へ代入する。ジャンプ命令によって形成されるプログラムの流れの構造を**制御構造** (control structure) と呼ぶ。

なかでも，条件ジャンプ命令は多くの可能性を生み，これを巧みに使うことにより，いくらでも複雑な制御構造を作ることができる。とはいえ，**図** 4.1 のプログラムを見て制御構造がさっと頭に入る人は少ないと思わ

れる。まず，JPN とか JP などを発見し，そのジャンプ先のラベル（アドレスといっても良いだろう）を確認する。続いてどのような条件でジャンプするかを調べる。こうしたことで，ようやくプログラムの制御構造が見えてくる。

もっとも簡単な制御構造には分岐とループがある。**分岐** (branch) とは，条件によって作業を二つに分け，それぞれの作業が終了すると，再び合流して同じ作業を行う。**図** 4.1 の制御構造は分岐である。一方，**ループ** (loop) とは，同じ作業を繰り返す可能性のある制御構造である。二つの可能性があり，一つはプログラムのある点まで来たら，条件ジャンプでその点より前の点へジャンプするものである。もう一つはある点まできたら無条件ジャンプで前の点へ戻るものである。そうなると無限に同じことを繰り返してしまい，ループを脱出できなくなるので，通常はループ内に条件ジャンプが入り，条件を満たすと，無条件ジャンプした点の次へ脱出することができる。

4.2 構造化プログラミング

分岐とループ以外にも，いろいろ面白い制御構造が構成できる。複雑な処理をしようと思えば思うほど，多くのループを構成し，本流からループ内へジャンプしてみたり，ループ内から別のループ内へジャンプしたり，複数のループを絡ませたものや，その制御構造はどんどん複雑になっている。複雑になりすぎたプログラムは，絡み合った具合を例えて，**スパゲティプログラム** (spaghetti program) と呼ばれる。第三者がそのプログラムを読もうと思っても，とてもフォローしきれないばかりか，時間が経つと製作者本人すら理解できなくなってしまう。今や，スパゲティプログラムとは，その複雑性に対する称賛ではなく，蔑視の言葉にすら

なっている。

こうした制御構造の複雑さを回避すべきであるということで提唱されたのが，**構造化プログラミング** (structural programming) という概念[1)]である。これは，分岐やループ構造間のジャンプを禁じるものである。分岐やループがあっても，1 回ごとに必ず本流に戻られる保証があるため，プログラムは大変読みやすくなる。なお，構造化プログラミングといえども，こうした構造の入れ子構造，つまり分岐の中にループが入るとか，ループの中にループが入るなどの構造までは禁止していない。後ろの節で述べるが，幸いにして，どんなに複雑な制御構造でも必ずこうした構造化プログラミングが可能である。

以後，構造化プログラミングを前提にした制御構造の話を進めるが，アセンブラ言語ではわかりづらいので，高水準プログラム言語である C や Java などで使われている用語の助けを借りて話を進めよう。

4.3　分岐文

分岐構造を高水準プログラム記載するには，if **文** (if statement) と呼ばれる分岐文を使う。`if-then-else` 文，あるいは `if-else` 文ともいう。例えば，C や Java などでは次のように記載する。

```
if (条件式) {
    then ブロック
} else {
    else ブロック
}
```

1) Edsger Wybe Dijkstra, "A Case against the GO TO Statement", (1968)

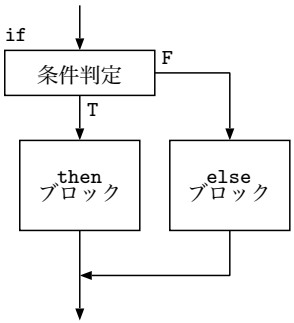


図 4.2　if 文のフローチャート

条件が満たされれば then ブロックを実行し，満たされなければ else ブロックを実行し，いずれの場合にも実行後は本流に戻る構造である。なお，ブロックとは複数の命令文からなる複合文であり，単一文も含む。C 言語の場合は，単一文とは何らかの命令に ‘;’ を付けたもの，複合文とはこれらの集合を { } で囲んだものである。

この流れをしばしば，**図** 4.2 のように書く。こうした，制御の流れを見やすく記載したものを**フローチャート** (flow chart) という。この描き方以外にも，O>’PAD’ と呼ばれる種々の描き方がある。

これをアセンブラプログラムに翻訳すると**図** 4.3 のように，まさに分岐構造となっていることが理解できよう。if 文のうち else ブロックや then ブロックのない構造も許されている。else ブロックがない場合には，‘else {...}’ すべてを省けばよい。then ブロックがない場合には，最初の ‘{...}’ をすべて省けばよい。if 文の then ブロックや else ブロックの中にさらに他の制御構造を入れることも可能である。else ブロックが丸ごと，他の条件式による if 文であることもある。その場合，

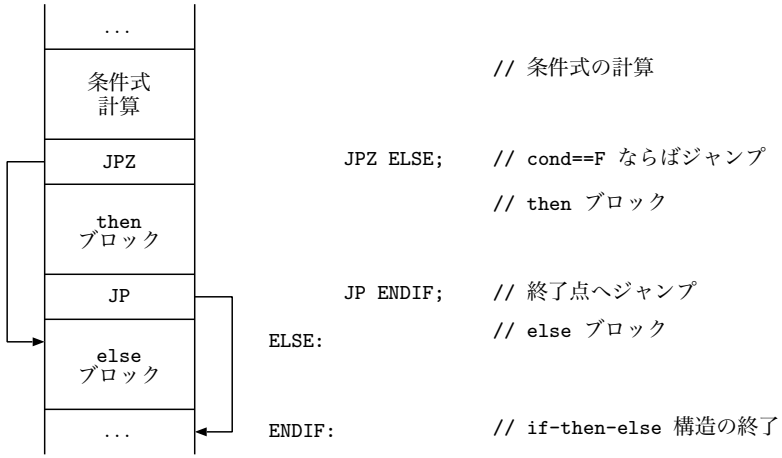


図 4.3　if 文の構造とアセンブラプログラム

言語によっては，else と if を重ねて elseif とか elsif とか略すことがある。

4.4　前置判定ループ文

ループ構造の一つは**前置判定ループ文** (pre-decided loop statement) といい，ループの頭で条件判定をし，条件の成立する限り，ループを繰り返すというものである。while **文** (while statement) といい，条件が不成立であると，ループを実行せず，while 文の次の文へ飛んでいく。例えば，C や Java などでは次のように書く。

```
while（条件式）{
    while ブロック
}
```

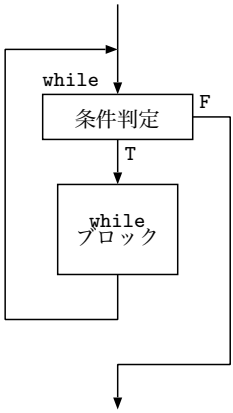


図 4.4　while 文のフローチャート

この流れをしばしば，**図** 4.4 のようなフローチャートで描く。

算術計算などでは，ある変数を次々に増加または減少させながら，同じ計算を行う局面が多い。例えば，1 から 10 までの加算などは，ループの回数ごとに i なる変数を 1 から順に増していき，それをある変数，例えば s に毎回加えていくことで達成される。データが，あらかじめ，あるアドレスを先頭に順にいくつかのメモリーに収納されているとき，その合計を計算するような場合にも，先頭アドレスに i（0 から）を加えたアドレス内のデータを次々と変数 s に加えることで計算できる。およそ，表計算ソフトの計算のかなりが，このような計算であることを考えると，こうしたループ構造は重要であり，for **文** (for statement) と呼ばれている。例えば，C や Java などでは次のように書く。

```
for（初期化式；前実行式；後実行式）{
-- 次ページへ続く --
```

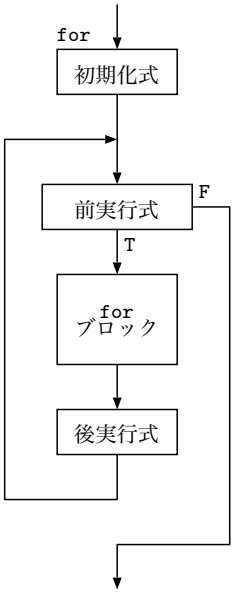


図 4.5　for 文のフローチャート

```
    for ブロック
}
```

このフローチャートを描いてみると，**図** 4.5 のようになっており，本質的には次に示した while 文と同等であることがわかる。

```
初期化式;
while（前実行式）{
    for ブロック
    後実行式;
}
```

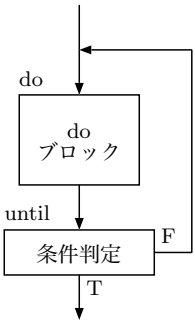


図 4.6　do 文のフローチャート

したがって for 文はこれを単に読みやすくしただけであるが，この構造は多用されるため，種々のプログラム言語に定義されている。

4.5　後置判定ループ文

ループ構造のもう一つは**後置判定ループ文** (post-decided loop statement) と呼ばれるもので，ループの終わりで条件判定をし，条件が成立するまで，ループを繰り返し，do ブロックを実行するというものである。これを do **文** (do statement) と呼び，C や Java では次のように書く。

```
do {
    do ブロック
} until (条件式);
```

この流れをしばしば，**図** 4.6 のようなフローチャートで描く。実は C や Java では until の代わりに while を使っている。単に，前者は条件成立

でループを離脱するのに対し，後者は条件不成立でループを離脱するだけの差である。

do 文は while 文で書き換えることができる。ただし，do ブロックは，いかなる条件でも do ブロックを最低一回は実行するので，while 文の前にも do ブロックを置く必要がある。逆に while 文を do 文に置き換えるには，条件成立の際には do ブロックを一回も実行しないように，if を do の前もしくは後に入れて，do ブロックの実行を回避させる必要がある。

4.6　goto 文の使用禁止

以上述べた分岐文とループ文以外に，どのプログラムにも goto **文** (goto statement) というのが用意されている。goto 文は，機械語やアセンブラ言語における無条件ジャンプ文に対応するものである。分岐文やループ文の中でも使うことができる。また，else ブロックを持たない if 文の then ブロックに goto 文を置くと，条件ジャンプを行うこともできる便利な文である。以下，これを if-goto 文と記載することにする。

しかし，これを多用すると，まさにスパゲティプログラムとなってしまう。一方，構造化プログラミングとは，ここまでに述べた分岐文とループ文の利用は許すが，goto 文の利用を禁じたものである。構造化の特長は，分岐とかループの作業が終わると必ず構造化文の終了点へ到達できることなのである。それ故に，プログラムの流れが掴みやすくなっているのであるが，それが，goto 文で任意のところへのジャンプが許されるようになると，プログラムの流れを掴むことは大変難しいことになってしまうのである。もちろん，構造化といっても，条件文の中に条件文があるとか，条件文の中にループ文がある，ループ文の中に条件文がある，ループ文の中にループ文があるといった**入れ子** (nesting) 構造は許されて

いる。入れ子構造があっても，プログラムの流れは文ごとに文終了点へ来ることは明らかであるので，流れはつかみやすい。

すべての制御構造が構造化プログラミングの方法で実現可能かどうかは，大いに気になるところである。例えば，while ループの途中から if-goto 文で while 文の上のほうへ戻るような制御構造を考える。

```
...
LABEL0:
ブロック 1
while（条件 1）{
    ブロック 2
    if（条件 2）{goto LABEL0;}
    ブロック 3
}
...
```

構造化するとなるとループの途中におけるジャンプは許されていないので，まず if-goto 文の条件が成立した際には，早々に while 文の残りをスキップする必要がある。これは，条件 2 を否定した else ブロックのない if 文で実現できる。さらに，while 条件に関らず，while 文から脱出できなければならないため，while 文の条件式を若干書き直す必要が生じる。なお，以下‘&&’は条件と条件の AND，‘||’は条件と条件の OR，‘!’は条件の否定を表している。こうして，if-goto 文を while 文の外，それも while 文の直後へ出す。

```
...
flag2=0;
LABEL0:
ブロック 1
flag1=条件 1;
while (条件 1 && !flag2) {
    ブロック 2
    flag2=条件 2;
    if (!条件 2) {
        ブロック 3
    }
}
if (flag1 && flag2) {goto LABEL0;}
...
```

flag2 という変数があるが，これは条件 2 を代入しているので，基本的には条件 2 と読みなおしてもよさそうである。しかし，同じ条件 2 を多用する場合には，いくつかのブロックを経由するうちに，条件 2 が変化する可能性があり，どの時点での条件 2 を利用するかが重要となる。そのため，代入式の置かれた時点での条件 2 を，他の場所でも使うという意味がある。また，while 文に入る前にたまたま条件 2 が 1（真）であると，while 文に入ることができなくなってしまうために，事前に flag2 を 0（偽）にしておくことによりこれを回避することもできる。

最後の上に戻る if-goto 文は，do 文によって実現できるので，次のような形にすることができ，構造化が完成する。

```
...
flag2=0;
do {
    ブロック 1
    flag1=条件 1;
    while (条件 1 && !flag2) {
        ブロック 2
        flag2=条件 2;
        if (!条件 2) {
            ブロック 3
        }
    }
} while (flag1 && flag2);
...
```

こうすると，三重の入れ子にはなるが，文の中から外へのジャンプは一切無くなっており，分岐文やループ文の終了ごとに，流れは必ず終了点を通過していることが確認できよう。

一般に，分岐文やループ文の途中から `if-goto` 文により脱出したいという場合には，まず `if` 文でこれらの文内の最後の点までジャンプする。特に，これらの文が `do` 文である場合には，元の `if-goto` 文の条件式と同一条件でループを脱出できるように，`do` 文の条件式を直す必要が生じる。そうして，`if-goto` 文を分岐文やループ文の外へ出す。`goto` 文のジャンプ先が下に位置する際は `if` 文を使えばよいが，戻りジャンプの場合には `do` 文を使うことになる。なお，`while` 文の絡む場合には，前節で述べた方法で `do` 文としてからここに述べた方法で変更するのが楽である。

逆に外の if 文から，if，while，do 文内へジャンプしてくることもある。この場合には，まずこれらの文の直前へジャンプさせ，これらの文の条件式を少し修正して文中へ導入し，文内の頭で改めて if 文を作成し，文内の所望の点までジャンプさせる。下からジャンプしてくる際は，do 文に変更することは，前述の場合と同様である。こうして，いかなる場合でも，構造化できることがわかる。

なお，goto 文を禁じるといっても，goto 文を入れるとプログラムが極めてすっきりするとか，わかりやすい場合にまで禁止しているわけではない。ある程度，臨機応変が必要である。

演習問題 4

問題 4.1　while 文の機械語の構造を**図** 4.3 を参考にして描け。

問題 4.2　for 文の機械語の構造を**図** 4.3 を参考にして描け。

問題 4.3　do 文の機械語の構造を**図** 4.3 を参考にして描け。

問題 4.4　次の then ブロックの途中に if-goto 文の入ったプログラムを構造化せよ。

```
...
if（条件 1）{
    ブロック 1
    if（条件 2）{goto LABEL0;}
    ブロック 2
}
-- 次ページへ続く --
```

```
ブロック 3
LABEL0:
...
```

問題 4.5 次の if-goto 文から do 文へジャンプするプログラムを構造化せよ。

```
...
do {
    ブロック 1
    LABEL0:
    ブロック 2
} while (条件 1);
ブロック 3
if (条件 2) {goto LABEL0;}
...
```

5 サブルーチン

放送大学 岡部 洋一

《目標＆ポイント》ジャンプ命令で実現される制御構造と並ぶもう一つの重要な概念がサブルーチンである。ある程度まとまった仕事，特に同じような仕事を繰り返すような場合，その仕事を主な流れから括り出してブロック化したものであり，将来，オブジェクト指向言語などにつながっていく重要な概念である。

《キーワード》サブルーチン，関数，引数，戻り値，ポインタ，参照，主プログラム

5.1 サブルーチンとは

プログラムの中で，いくつかの箇所で同じような作業を行うことがある。こうした場合，**サブルーチン** (subroutine) なるものを定義し，プログラムからそこへジャンプし，また決まった作業が終わると，元へ戻るようなしくみがあるとよい。また，主プログラムが長くなりすぎ，作業内容が不鮮明になってしまうような際，その中である程度まとまった作業を取り出し，サブルーチンとして使う方法もある。

サブルーチンへジャンプすることを，サブルーチンの**呼び出し** (call) という。また，サブルーチンを終了して呼び出し側に戻ることを**戻り** (return) という。また，戻る際に関数から呼び出し側に与えられる値のことを**戻り値** (return value) または**返り値** (return value) という。サブルーチンは主プログラムから呼び出す場合だけではなく，別のサブルーチンから呼

```
// 呼び出し側プログラム
int x = 5;
int y = 3;
int z;

z = add(x, y);       // サブルーチン呼出し
print z;
exit;

// サブルーチン
int add(int u, int v) { // 呼び出されるサブルーチン
    int w;
    w = u + v;
    return w;        // 戻り値
}
```

図 5.1　サブルーチンの例（値渡し）

び出すこともあるので，以後，サブルーチンを利用しているプログラムを主プログラムとは呼ばないで，呼び出し側プログラムと呼ぶことにする。

サブルーチンが戻り値を持っている場合，**関数** (function) と呼ぶ場合もある。ちなみに，C ではサブルーチンをすべて関数と呼び，戻り値のない場合も void（空）を返す関数としている。一方，Pascal では戻り値あるサブルーチンを関数，ないものを**宣言** (procedure) としている。

C によるサブルーチンの具体例を**図** 5.1 に示そう。例えば，二つの値の和を求めることが多い場合，add というサブルーチンを定義しよう。本

当は，この程度の計算であると，サブルーチン化する意味はあまりないのであるが，もう少し複雑な計算をたびたびする場合には，サブルーチンを作成することは，実用上も大きな意味がある。なお，図中，‘//’は**コメント** (comment) であり，それ以後の部分はプログラムの実行上は無視される。文中に説明を入れる場合によく用いられる。

5.2　引数と戻り値

引数(ひきすう) (argument) というのは，**図** 5.1 に示すように，サブルーチン名の後に括弧で囲んだいくつかの変数を指す。呼び出し側でサブルーチン名に引き続き括弧で囲った変数を，**実引数**(じつひきすう) (real argument) といい，サブルーチン側でサブルーチン名の宣言に続く括弧内の変数を**仮引数**(かりひきすう) (dummy argument) という。

仮引数に相当するメモリーは，サブルーチン側のメモリー領域に確保され，実引数で与えた値はそこにコピーされた後に，サブルーチンで利用される。したがって，サブルーチン側でこの変数の値を変えても，呼び出し側には何の影響も生じない。したがって，実引数は定数でも構わない。仮引数や，サブルーチン内で用いられる変数名は，サブルーチン内だけで有効である。したがって，**図** 5.1 のサブルーチン側の仮引数の名称を，呼び出し側の実引数 x や y と同じにしても，コンパイラは別の変数としてみなす。こうした引数の値の受け渡し法は**値**(あたい)**渡し** (call by value) と呼ばれる。

呼び出し側プログラムとサブルーチンの間で，値を渡すもう一つの方法がある。それは**大域変数**(たいいきへんすう) (global variable) と呼ばれる変数を利用することである。ある変数を例えば `global` などと宣言しておくと，呼び出し側プログラムであろうとサブルーチン内であろうと，いたるところで，この変数名を持つ変数は同じ対象として扱われる。サブルーチン内では

同じ変数名を使っても，呼び出し側とは独立に扱われるという局所ルールが適用されないのである。したがって，サブルーチン内で大域変数を変更すれば，その変更結果は呼び出し側にも及ぶのである。しかし，この方法は呼び出し側プログラムだけを読むと，変数の内容の変更があったことが，明示的に示されていない，つまりサブルーチンの独立性が欠けているために，わかりづらいプログラムとなってしまうため，あまり推薦されない方法である。

最後に，もう一つ，参照渡しという方法が存在する。その説明をする前に，ポインタという概念を説明しておこう。例えば変数 x のアドレスを&x と記載する。こうしたアドレスを代入する変数を**ポインタ** (pointer) と呼ぶのである。ポインタの内容としてのアドレスを，**参照** (reference) とも呼ぶ。例えば，x を整数型変数とし，こうした整数型変数の参照を格納するポインタを p_x としよう。この p_x に x のアドレスを格納するには，‘p_x = &x;’ と書く。一方，ポインタの指すアドレスの内容は，*p_x のように，ポインタの前に ‘*’ を付けることで，知ることができる。

参照渡し (call by reference) とは，アドレスを引数としてサブルーチンを呼び出す方法である。例えば，変数 x と y の内容を入れ替えるサブルーチンを考えてみよう。**図** 5.2 を見てみよう。サブルーチンの仮引数は int*という型である。これは，整数を指すポインタという意味である。これら二つのポインタに，呼び出し側では，二つの変数 x と y のアドレスを与えている。したがって，*p_u と*p_v は，変数 x と y の値ということになる。それを変数 w を使って以下の三行で入れ替えている。この結果，*p_u の内容，つまり x の値は y の値になる。同様に，*p_v の内容，つまり y の値は x の値となることになる。しかし，このように参照渡しを使って，引数のアドレスの内容を変更することは，大域変数と同様，可視性や独立性を欠くため，その利用は極力避けるべきであろう。あくま

```
// 呼び出し側プログラム
int x = 5;
int y = 3;

swap(&x, &y);
print x, y;
exit;

// サブルーチン
void swap(int* p_u, int* p_v) {
    int w;
    w = *p_u;
    *p_u = *p_v;
    *p_v = w;
}
```

図 5.2　サブルーチンの例（参照渡し）

でも，情報をサブルーチンに渡すときにだけ利用することを推奨する。

となると，このようなサブルーチンでは，二つの値を戻り値で戻さなければいけなくなる。本書では詳細は示さないが，そのような場合，**構造体** (structure) と呼ばれるデータの集合を，塊として渡す手法が可能である。また，戻り値をポインタにして，明示的にアドレスを戻すのはよい方法である。

なお，大域変数や参照渡しの説明は，C に準拠して説明したが，他の言語でも，何らかの似たような考え方があることを理解してほしい。

5.3 実装

サブルーチンは，主プログラムからだけではなく，他のサブルーチンから呼び出す可能性もあるので，呼び出し側プログラムからは分離し，独立した位置に置かれるのが普通である。呼び出し側のプログラムからサブルーチンに単にジャンプするだけではなく，引数や戻り値をどう渡すかを考慮しなければならない。仮引数はサブルーチンに置くのは当然として，そのアドレスは呼び出し側に伝えなければならない。仮引数は一つとは限らないため，先頭アドレスを共有することになる。

同様に，戻り値や呼び出し側への戻りアドレスなどは両方のプログラムが共有すべき情報である。戻り値についても，そのサイズは戻すべきデータの種類によって変化する。整数と小数，文字など，皆，異なるサイズを持っているし，まして，前節に述べた構造体は明らかに異なるサイズを持っている。したがって，引数同様に，先頭アドレスを共有することになる。

以後，**図** 5.3 にしたがって話を進めよう。図では文字数を減らすため，アドレスを番地と表記した。まず仮引数の先頭番地の情報であるが，これは，サブルーチンの先頭からみてわかりやすい位置に記載しておくのがよいであろう。ここでは，サブルーチン先頭のジャンプ命令の直後に置いてあり，呼び出し側は，サブルーチンの先頭番地から計算して推定することになる。サブルーチンの一番最初の行は，サブルーチンの実体へのジャンプ命令とする。

同様に，戻り値の先頭番地（以下，単に戻り値番地と記載）を，呼び出し側にある戻り番地に置かれた最初の命令の次に置くことにすれば，サブルーチン側ではその番地を戻り番地から簡単に推定できる。戻り番地には呼び出し側の命令の継続部分へのジャンプ命令を記載しておく。

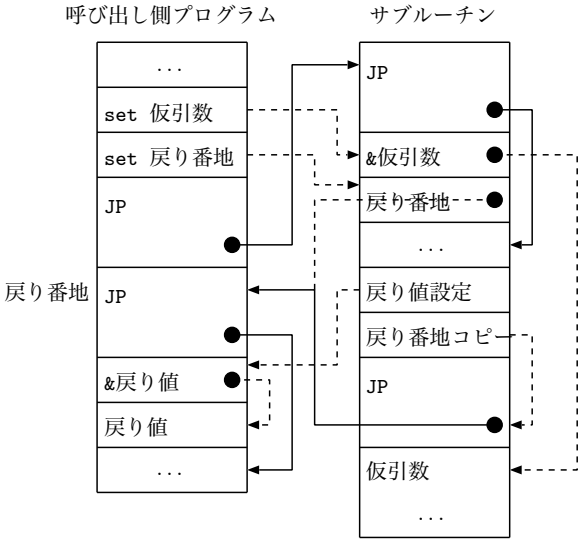


図 5.3　サブルーチンの実装（&は番地を示す）

このような配置を前提としてプログラムの流れを追ってみよう。まず，呼び出し側のプログラムでは，サブルーチンの先頭番地+2 のところにある仮引数の置かれた番地から順に，実引数の値をコピーする。続いて，サブルーチンの先頭番地+3 の位置に，戻り番地を書き込む。これらの処理が終わると，サブルーチンの先頭へジャンプすることになる。

サブルーチンの先頭アドレスには JP 命令が置かれており，これら共有情報の書かれている領域を迂回した後，サブルーチンの本体のプログラムを実行する。サブルーチンでは，コピーされた仮引数の内容を利用しながら，処理を行う。サブルーチン内で新たに利用するいろいろな変数はサブルーチン内に置くものとする。サブルーチンとして，すべての処理が終了すると，戻り値を計算し，戻り番地から推定した呼び出し側の戻り番地の 2 番先に記載された戻り値番地を利用して，戻り値を呼び出

し側に記載する。そして，最後に戻り番地へジャンプする。

呼び出し側に戻ると，まず JP により戻り値の書かれた領域を迂回し，必要に応じ戻り値を利用しながら，呼び出し側本来のプログラムの続きを実行することになる。

5.4 動的メモリー確保

次節の話では，ヒープ領域とスタック領域という用語を多用するため，まずこれらについて，説明しておこう。

昔のプログラムは，使うデータ領域もそれほど大きくなく，また，プログラマはあらかじめ利用するデータのサイズを自分で制御していた。しかし，最近のプログラムは大きなデータを扱い，かつ，利用するデータの量も，そのときの仕事に依存するものが多くなってきている。例えば表計算ソフトウェアでも，ほとんど無制限なサイズの表を扱うことができる。

また，現在のコンピュータでは複数のプログラムが同時平行に動作する。同時といっても，本当に同時に動作するには，プログラムの個数だけの CPU を必要とするため，実際には，**時分割処理** (time sharing processing) といって，あるプログラムを少しだけ実行し，また別のプログラムを少しだけ実行し，...といった作業を行う。メモリーについても，全メモリー領域を複数のプログラムが分けあって利用する。さらに，メモリーが不足になる場合には，ハードディスク領域へ一時退避も行う。こうした資源利用の制御はすべて後ろの章に述べる OS が行ってくれる。

各プログラムはスタートする前に，OS に対し，プログラムが使用するメモリーの総量（上限値）などを申告する。その値まではメモリーを自由に使えるわけである。動的なメモリーを使う場合には，その自由領域

内で**ヒープ領域** (heap area) を利用する。ヒープ領域は，複数のプログラムで共用するため，通常，メモリーの余裕分をすべて割り当てるので，極めて広大な領域である。あるプログラムがここを利用したいときには，そのつど，後ろに説明する OS に対して利用を申請する。利用が認められると，OS から使ってよいメモリー領域のアドレスが与えられる。利用が終了すると，OS に対し，利用の終了したことを申請する。こうして，常に必要最小限のメモリーが用意される。

ヒープ領域は通常連続した一つの領域であることが多いが，だからといって，その内部を連続的に利用するとは限らない。例えば，後から申請したメモリーはまだ使用中であるが，先に申請したメモリーはもう利用が終了したような場合，メモリーの利用は飛び飛びとなる。こうした場合，後述するリストという構造を利用することとなるが，今のところ，OS がすべてうまくやってくれると理解してほしい。

これとは別に，**スタック領域** (stack area) という OS により管理されているメモリー領域も存在する。スタック領域は単にスタックとも呼ぶ。スタックは，メモリー領域が連続しているということを除くとヒープ領域と同様に，自由に確保できる領域である。スタックとは本を積み上げるような感覚で，新たな情報を追加する際には，一番上，メモリーでいえば，一連の記憶の最前部にしか追加できない。この情報を追加する作業を**プッシュ** (push)，あるいは上に積むという。したがって，一番最初の記憶はスタック領域の一番最後のアドレスに置くことになり，積めば積むほど，アドレスの若い方へ伸びていくことになる。

以下，スタックという場合の上下は，本のイメージで用いることとする。また，積み上げた本の下のほうが簡単には見られないように，スタックの場合にも，通常はスタックの最上部付近の情報にしかアクセスしない。また，最後に追加したメモリー領域を解放することを**ポップ** (pop)

と呼ぶ。ある応用ソフトウェアで積んだスタックは，そのソフトウェアの責任で元へ戻さなければならない。ただし，途中でトラブルのためにそのソフトウェアが予期せぬ突然終了をしてしまう可能性も想定し，OS は応用ソフトウェア終了ごとにスタックを元に戻してくれる。

スタックは，サブルーチンへのジャンプなどで多用される。サブルーチンとはプログラム内で利用するプログラム内プログラムのようなもので，主プログラムからジャンプすることもあれば，サブルーチンから，それも場合によっては再帰的に自身と同じサブルーチンから，ジャンプすることもあるなど，非常に複雑な流れとなることがある。こうした際，サブルーチンジャンプが発生するたびに，戻りアドレスを記憶しておくことで，プログラムの流れをきちんと把握することができる。複雑な流れといっても，プログラムの流れは 1 次元的であるので，こうしたプロセス管理のための情報も 1 次元的でよく，そのためにスタックが多用される。

5.5 再帰呼び出し

サブルーチンから，同じサブルーチンを呼び出すことを**再帰呼び出し** (recursive call) と呼ぶ。例えば 10!(=1×2×3×⋯×10) の計算をする際，再帰呼び出しを使って階乗を計算するには，10!=10×9!と定義し，徐々に小さな値の階乗に責任を押し付けていく。最後に 1!となったら，1 を返すようにすればよい。

そこで，factorial というサブルーチンを次のように定義する。

```
int factorial(int x) { // サブルーチン
    if (x == 1) {      // x が 1 ならば 1 を返す
        return 1;
```

```
    } else {            // それ以外は x*(x-1)!を返す
        return x * factorial(x - 1);
    }
}
```

プログラムの記述としては，簡単であるが，これを実装しようとすると，結構面倒な現象が発生する。それは，9 の階乗を計算中，まだ 10 の階乗の計算のサブルーチンプログラムは残しておかなければいけないことである。10 の階乗のプログラムは 9 の階乗の計算結果が戻ってきてから，再び，命令の実行を続けていく必要があるからである。以下同様，最大，10 個の factorial のサブルーチンを実在させなければならない。それが常に 10 個と限られていれば簡単であるが，100!の計算を要求される場合には，最大 100 個のサブルーチンを実在させる必要があるのである。要するに，再帰的サブルーチンは処理プログラムが動的に生成されるようにしなければならない。

このようなサブルーチン実行中に自身が利用される可能性を**リエントラント** (reentrant)，つまり再入可能という。対象となるサブルーチンから別のサブルーチンが呼び出され，そこから再び対象となるサブルーチンが呼び出されるような場合も，広い意味の再帰呼び出しであり，リエントラント性が確保されていなければならない。先に述べたサブルーチンの実装法では，残念ながらリエントラント性は確保されていない。

リエントラント性を確保するには，先の実装で，サブルーチンそのもの，あるいはせめてサブルーチン側に置かれていた変数のすべてを動的に確保する必要がある。サブルーチンの先頭に置かれていた戻りアドレスや仮引数の先頭アドレスも動的に確保する必要がある。これらが静的であると，再帰呼び出しにより，それらの内容が書き換えられてしまい，戻ったときに，正しい動作をしなくなるからである。

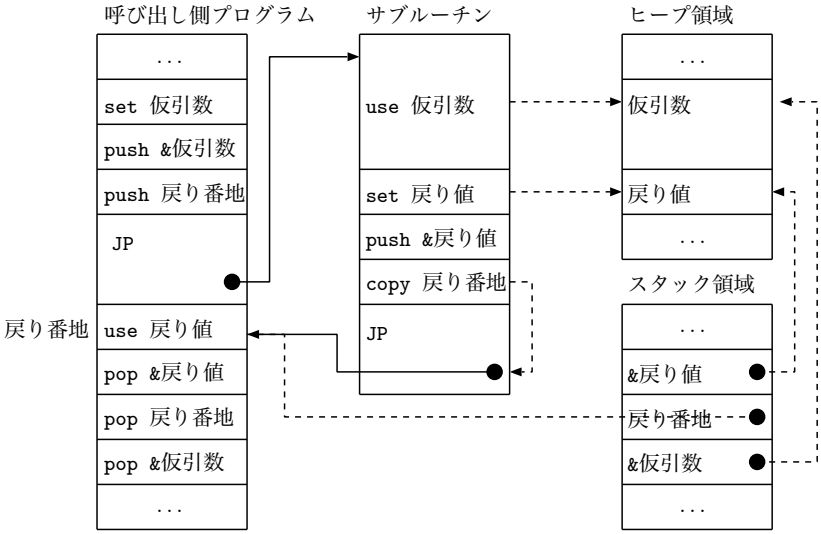


図 5.4　リエントラントサブルーチンの実装

再帰呼び出しでは，対象となるサブルーチンから，自身への呼び出しも含め，必ずサブルーチン呼び出しが存在する。その際，戻り値も動的なメモリー確保の対象となるので，注意してほしい。

通常，サブルーチンの先頭の方に置かれた戻りアドレスと仮引数の先頭アドレスはスタック領域に積まれ，サブルーチンの終了とともに解放される。サブルーチンで利用されるその他の変数や，さらにその先の呼び出しサブルーチンからの戻り値を収納するメモリーはヒープ領域に確保され，これらもサブルーチン終了とともに解放される。階乗の計算のように，同じサブルーチンが何度も呼ばれると，スタック領域もヒープ領域もどんどん利用が増えていくことになる。そして，すべてが終わると，これらはまた縮小することになる。

リエントラントなサブルーチンの実装には何種類もの方法があるが，

ここではその一つを**図** 5.4 に示す。前述のように，戻り値，仮引数，サブルーチンで使う変数などは，ヒープ領域に確保している。さらに，これらのポインタである戻り番地，戻り値番地，仮引数番地などは，スタック領域に確保している。なお，本図には明示していないが，サブルーチン内で利用される変数も，スタック領域に確保する。リエントランス性を確保するのは，このようにやや面倒であることから，多くの高水準プログラム言語では，再帰呼び出しを禁じている。

演習問題 5

問題 5.1　次の用語を理解したかどうか確認せよ。

1）主プログラム，サブルーチン，関数
2）実引数，仮引数
3）戻り値
4）ポインタ，参照
5）ヒープ領域，スタック領域
6）再帰呼び出し，リエントラント

問題 5.2　言語によっては，戻りアドレスをサブルーチンの中にデータ領域をとり，そこに記載することが行われる。再帰呼び出しを行った場合，この方式ではどんな不具合が生じるであろうか。

問題 5.3　言語によっては，戻りアドレスの番地を呼び出し側のスタックではなく，サブルーチンの中にスタックに記憶することが行われる。再帰呼び出しを行った場合，この方式ではどんな不具合が生じるであろうか。

6 アルゴリズム

放送大学 岡部 洋一

《目標＆ポイント》あることを達成するためのプログラムの書き方にはいろいろあるが，実際にプログラミングする際には，計算速度，記憶の占有量，読みやすさなど，いろいろなことを意識して作成する必要がある。こうしたプログラミング作成の基礎的な流れであるアルゴリズムについて解説する。

《キーワード》アルゴリズム，計算速度，ステップ数，構造化プログラミング，プロファイラ

6.1 アルゴリズムとは

プログラムの手順を**アルゴリズム** (algorithm)[1] という。プログラムの流れあるいは構成といってもよいだろう。同じ言語で同じ作業をするプログラムでも，その処理のしかた，書き方，考え方はプログラマによって千差万別である。同じ結果は得られるが，恐ろしく時間のかかるプログラム，バグがあることがわかっても，後からそれがどこにあるのか容易に発見できないプログラムなど，まったくいろいろである。

良いプログラムとは次のような条件を満たすものである。

- 計算速度が速いこと
- メモリーをあまり使わないこと
- 考え方がすっきりしていること
- 読みやすいこと

1) インド数学をイスラム圏へ紹介した数学者の名前が起源とされている。

このうち，特に前 3 項が，よいアルゴリズムの条件である。どちらかというと上位ほど優先される。

第 1 項は，同じ結果を得るために，いかに手数を少なくできるかという工夫である。この詳細は，次節で述べる。

第 2 項は，昔，計算機の搭載メモリーが少ないころに特に重視されていたが，近年の計算機では十分なメモリーが搭載されるようになり，比較的気にならなくなりつつある。しかし，メモリーが無限にあることを前提にしたようなプログラムは今でも許されてはいない。

また，現在，複数のプログラムを同時平行的に実行（厳密には時分割処理）するようになってきているが，大きなメモリー領域を必要とするプログラムは他のプログラムが動作する際，一時的にハードディスクに退避させられる。これを**スワップ** (swap) と呼ぶが，スワップには相当の時間を有するため，結果として実行時間が伸びることになる。いずれにせよ，使用メモリーは少ないほうがベターである。

第 3 項の考え方がすっきりしていると，一般には計算速度も速いことが多い。しかし，考え方がすっきりしているからといって，必ずしも速いとは限らない。やはり，計算速度が速いことが最大優先である。

第 4 項は，膨大なプログラムになると意味を持ってくる。多くのプログラミングでは，一度で完璧な動作をするプログラムに至ることは珍しい。いくつかのバグと呼ばれる間違いが入るものである。また，仮に間違えがなくても，計算のやり方に新しい工夫が入ったり，入出力すべき内容が変わったりして，プログラムの変更が生じる。こうした場合，膨大なプログラムであると，一度書いたものを読みなおすのが，大変な労力となる。最初のプログラミングのときから，読みやすいプログラムを書くように努力する必要があるのである。

6.2 よいアルゴリズムの例

第1項について，詳細を述べよう。例えば，与えられた数Nが素数かどうかを判定するプログラムを考えてみよう。素数とは約数として自分自身と1しか持たない数である。この原理原則にしたがえば，与えられた数の一つ小さい数であるN-1から順に割り算をしていき，除数が2に達するまで割りきれないことを確認していけばよい。最悪，N-2回の割り算を実行することになる。この計算はほぼN^1に比例する時間を要することから，オーダー1の計算と呼ばれる。

しかし，このアルゴリズムはかなり無駄が多いことに気付くかも知れない。そもそも，Nがどんな数であろうが，N/2までの割り算では割り切れることはないのである。そこでN/2を越えない整数から2までのチェックに変更すれば，割り算の回数はN/2-1に縮小できる。計算のオーダーはやはり1である。

77が素数であるかどうかを調べるには，11で割ってみなくても，7で割ってみるだけで確認できる。11は明らかに77/2よりも小さい。それでは，どのくらい小さい数までのテストで済むかというと，除数と商が同じ値になるところの$\sqrt{N}$を越えない数から下のテストで十分であることがわかる。九九の折り返し点といってもよいかも知れない。したがって，この計算はたかだか$N^{1/2}$-1回の割り算で済み，オーダー1/2に縮小できる。

計算の順番も重要である。今まで，大きいほうの除数から割り切れるかをテストしてきたが，小さいほうからテストするほうが，かかる時間の平均値は少ない。最初に2で割り切れるかをテストするだけで，50%の確率で素数でないかをチェックできてしまう。以下同様に小さい数で割るほど，多くの可能性を潰すことができるからである。しかし，与えた

数がたまたま素数の場合には，どちらから計算しても割り算の総数は変わらない。つまり，最後に示した改良は，平均計算回数は減らすことに寄与しても，最大計算回数は変わらないことになる。

このように，計算時間を減らすにはいろいろな工夫がある。こうした工夫がアルゴリズムの改良なのである。

6.3　ユーザから見たよいアルゴリズム

このプログラムを利用するユーザから見ると，プログラムが読みやすくできているかとか考え方がエレガントかどうかはどうでもよいことである。最大関心事は計算時間とか操作性である。懇切丁寧なマニュアルよりは，マニュアルなしで操作できることであろう。そこで，ユーザの立場から見たよいアルゴリズムは，例えば次のようなものになる。

- 最大計算速度が速いこと
- 平均計算速度が速いこと
- メモリーをあまり使わないこと
- 操作性がよいこと
- 可能ならばマニュアルを読まなくてもすむこと

プログラマはこうした点にも配慮しながら，プログラムを組む必要があるのである。

6.4　プログラムの保守

初歩のプログラマの陥りやすい穴は，プログラムがうまく動くようになった瞬間，ほっとして後をフォローしないことである。およそどんなプログラムでも，後からの改良や，作った当初には気付かなかったようなバグが発見され改修が必要になる。ところが，作って一か月も経つと，

自分の書いたプログラムですら，その流れや考え方を読むのは容易ではない。

このために，当初から次のような点に留意してプログラム作成を行うことをすすめる。

- 変数名をわかりやすくする
- コメントなど説明を多くつける
- 構造化プログラムとする
- プログラムがなるべくブロックになるように心掛け，適切なブロックをサブルーチンとする

まずは，第 1 節でも述べたように，当初からプログラムそのものを読みやすく書く努力をする。変数名にわかりやすい具体的な名前を付けることは必須である。単なる変数名ではなく，何を扱っているかをわかるようにする。`pointer_x` とか `OutputForPrinter` などのように，‘_’（アンダースコアは多くの処理系でアルファベット並みに扱われる）を使うとか，大文字小文字をうまく組み合わせる。現在の多くの言語では，変数名として何十字にもおよぶ相当長いものが許されている。しかも，コンパイルしてしまうと，変数名の長さは実行ファイルのサイズや計算速度には何の影響もないことを知っていてほしい。変数名やサブルーチン名をすべて人名にした私の学生がいたが，一か月もすると，本人ですらそのプログラムを読めないという悲劇が発生した。

また，可能な限り多くのコメントを付けておく。どんな言語でも，プログラム中にコンパイラがまったく無視してくれる文章を入れることが可能になっている。こうしたプログラムとしては無視される文章をコメントと呼ぶ。C 言語では，各行の ‘//’ のあとに記載された文章はすべてコメントとして無視される。また ‘/*’ と ‘*/’ で囲まれた行も無視される。こうしたコメントを多量に入れることにより，流れや考え方を思い

出すことができるのである。

別の文書を作成して，プログラムと一緒にしておくのも一つの手法である。こうした添付文書はしばしば readme という名前を付されている。よくダウンロードしたフリーソフトなどに，readme とか readme.txt などがあるのは，こうした備忘録である。ただし，別のファイルにすると，それを無くしてしまう危険性が高い。プログラム自身に長いコメントとして書き込んでおくほうが安全である。ただ，こうして作成したソフトウェアを出荷や配布する場合には，実行ファイルのみを配ることが多いので，その場合，ユーザに知らせたい内容だけは readme にすることが多い。

スパゲティプログラムは厳禁である。必ず構造化プログラミングの作法にしたがって記載する。スパゲティではないが，凝った短縮記述も厳禁である。例えば，for 文は第 1 行の条件括弧内に，初期化式，前実行式，後実行式と三つの式を記載することができる。この三つの式を巧みに書くことにより，while 文だろうと do-while 文だろうと何でも記述可能なのである。後実行文に for ブロックの内容を入れてしまえば，先頭の 1 行だけでありとあらゆるループ文が記載できるのである。だからといって，このようにして作成したプログラムは決して読みやすいものではない。多少，効率が下がる場合でも読みやすい記述をするというのが，CPU の機能が上がった現在における戦略である。

プログラムをなるべく作業ごとにまとまるようブロック化し，可能なものはサブルーチンとする。サブルーチン名も長くてよいので，わかりやすいものとする。こうすることにより，主プログラムから処理の詳細が消えるため，流れがつかみやすくなる。また，大域変数や参照渡しを避け，引数と戻り値によって情報の受け渡しをするようにするのがよい。

6.5 プロファイラ

計算のオーダーを小さくするなど，種々の工夫はまず必要である。しかし，いくらオーダーが小さくても，毎回の繰り返しごとに長時間を使ってしまえばやはり時間はかかってしまう。どこでどのくらい時間を使うのかを知ることは，プログラムの実行速度を上げる上で，極めて重要な情報となる。

プログラムの各部分の処理にどのくらいの時間を要するか，を計測するソフトウェアがある。これを**プロファイラ** (profiler) といい，こうした問題点を探す作業を**プロファイリング** (profiling) という。プロファイラとは，プログラムの各行の実行時間を計測し，プログラム終了時点でその結果を教えてくれる。特に，実行時間の長い順に通知してくれる機能があるので，どの行の実行に手間取っているかがただちに判明する。プログラムの中に繰り返し文があれば，その中の各行の実行時間の累積時間が提示される。いずれにせよ，どういう作業が時間を使うのかが，たちどころに把握できる。

全体の計算時間を縮めるためには，著しく実行時間を使っている行を丁寧に調べ，同じ結果を及ぼす他の命令に差し替えるとか，何か能率の悪いことをやっていないかをチェックし，改良する。プロファイラを使うようになると，どんな命令が時間を使うかがよく見えてくるようになり，普段のプログラミングでも，そうした命令を避けるようになり，プログラマとしての技量も大いに上がるため，ぜひ利用してほしい。

もちろんプロファイラは言語と深く関わっているので，言語ごとに異なるプロファイラが用意されている。プロファイラが用意されていない場合には，自分で要所ごとに出力文を置き，そのときの時刻を出力させるとよい。

演習問題 6

問題 6.1　1 から N までを，for 文を使って加算するプログラムのオーダを求めよ。

問題 6.2　n を 1 から N まで変えて，n^2 の和を求めるプログラムのオーダを求めよ。

問題 6.3　最大 N までの九九のような積の表を作成するプログラムのオーダを求めよ。

問題 6.4　地図上 N 個の地点をすべて回って元に戻る経路の最短時間を求める問題は，巡回セールスマン問題と呼ばれている。すべての各 2 地点間の必要時間はわかっているものとして，特別な手法を使わないで，総当たりで時間を計算するプログラムのオーダを求めよ。

問題 6.5　統計で使われる標準偏差などの計算で分散を求める際，N 個の各データを x[i]，平均を ave として，$(\sum$(x[i]-ave)^2)/N でも計算できるし，$(\sum$x[i]^2)/N-ave^2 でも計算できる。どちらの式を採用する方がアルゴリズム的に見て良いかを検討せよ。

7 高水準プログラム言語

放送大学 岡部 洋一

《目標＆ポイント》高水準プログラム言語にもいろいろな種類がある。どのような種類の言語があるか，また，それを実行するしかけについて説明する。

《キーワード》高水準プログラム言語，手続き型プログラム言語，オブジェクト指向プログラム言語，関数型プログラム言語，論理型プログラム言語，コンパイラ，インタプリタ，ライブラリー，ロード，実行プログラム

7.1 高水準プログラム言語の種類

高水準プログラム言語は，ややいいすぎかも知れないが，毎年 1 件のペースで提案されているともいわれている。それほど，多くのプログラム言語があるのも事実である。それを一つ一つ解説していっても意味がないのと，本書はソフトウェアの概念を与えるのが目的であって，言語を教育するわけではないので，代表的な言語に限っても，ある特定の言語の詳細がわかるような解説は行わない。

多くの言語が提案されているが，いくつかに分類することができる。

- 手続き型プログラム言語
- オブジェクト指向プログラム言語
- 関数型プログラム言語
- 論理プログラム言語
- その他，問い合わせ型言語やスクリプト言語など

これらの言語の概要と考え方を順に述べていくこととする。なお，最後

の問い合わせ型言語やスクリプト言語は，データベースソフトへの問い合わせとか，OS に対する指令などで使われるものであり，大きなプログラムを作成するためのものではないため，説明を省略する。

7.2　手続き型プログラム言語

C, FORTRAN, BASIC などという大部分のプログラム言語は，コンピュータにやらせたい作業を順に記載していく**手続き型プログラム言語** (procedural program language) と呼ばれるものである。プログラマのやりたいことはおよそ何でもできるが，気を付けないと，他者にとって読みづらい，いや自分でも後からは読みづらい保守の難しいプログラムとなりやすい。

これを避けるために，いくつかの推奨事項がある。

- なるべく構造化すること
- 機能を分解し，適切な大きさのモジュール（サブルーチン）に分離すること
- 大域変数（グローバル変数）を多用しない
- モジュール内で値を変更する場合，その変更が戻り値などに反映するように記述する

これは推奨事項であるので，必ずしも守らないプログラマも多い。それをある程度強制的に守らせるために考え出されたのが，オブジェクト指向プログラム言語や関数型プログラム言語であるといってもいいすぎではないであろう。

最初の構造化については第 4 章ですでに述べたが，要はジャンプ文があると，プログラムの構造がいくらでも複雑になるので，可能な限り簡単な構造にせよというものである。このため，`if` 文，`while` 文，`do` 文などが導入された。その他の推奨事項については，以下の各節において説

明する。

7.3 オブジェクト指向プログラム言語

人とか物に作業を頼むといったような形式でプログラムを行うのが**オブジェクト指向プログラム言語** (object-oriented program language) である。また, そのプログラムを**オブジェクト指向プログラム** (object-oriented program, OOP) という。こうしたプログラム言語の代表的なものに, C++, Smalltalk, Java, Python, Ruby, Perl などがある。C++は C もその概念に含むため, オブジェクト指向の使い方のみならず, 手続き型の使い方も可能である。実は, C++は C にオブジェクト指向の概念を拡張して作られている。したがって, 比較的近年開発された言語の多くは, 手続き型でもオブジェクト指向型のプログラムでも書けるようになっているものが多い。

この人とか物に対応するものが**オブジェクト** (object) である。OOP は Mac や Windows で GUI（グラフィカルユーザインタフェース）が使われ出してから急速に発展した。「窓 A さん, 現われてください」,「窓 A さん, 移動してください」,「窓 B さん消滅してください」といった命令が頻発するのである。OOP は現在のソフトウェア開発にとって, もっとも主要な言語の一つである。また, いくつかの新しい概念も入っているため, 章を改めて説明したい。

7.4 関数型プログラム言語

手続き型プログラム言語の大きな問題の一つは, サブルーチンの中で何でもできることである。サブルーチンの中で, 主プログラムの変数の内容をこっそり変えることもできるし, 引数がポインタであると, そのポ

インタの指す変数の内容を変えることもできる。そうした何でもできる機能を良いと思う人もいるが，サブルーチンの行った結果はすべて戻り値として確認できるようにすべきであるという主張の人もいる。戻り値が引数からだけ一意に決まることを**参照透明性** (referential transparency) と呼ぶ。なお，関数の呼び出しにより変更される変数は一つとは限らない。そのような場合には，変数の塊（リストという）を戻り値とすることで，目的を達成する。

逆に，サブルーチンが自身の戻り値以外に，他のサブルーチンや主プログラムに影響を与えることもある。こうした戻り値に反映されない影響は，しばしば**副作用** (side-effect) と呼ばれる。少なくとも，サブルーチンを作った人でない他人が主プログラムだけ見て，戻り値に反映されていない変化があるのを推定するのは非常に難しい。そのため，副作用という汚名が付けられているのである。また，戻り値を計算する際，引数以外の変数を利用すると，これも他人が見るとわかりづらいプログラムとなる。

さらに，サブルーチンの引数として，他あるいは自身のサブルーチンの結果を `f(g(x))` のような形で引用できる場合，こうしたサブルーチンは**第一級関数** (first class function) という。

参照透明性が確保されていること，副作用がないこと，第一級関数であること，の三つの要件が守られるように作られた言語を**関数型プログラム言語** (functional program language) と呼ぶ。手続き型プログラム言語では，戻り値のあるサブルーチンを関数と呼ぶ場合がある。しかし，関数型プログラム言語における関数とは，リエントラント可能であることに加え，参照透明性があり，かつ副作用がないことが保証されている場合に限るのである。

代表的な言語は `LISP` である。あまり普及していないので，知らない人

も多いかと思われるが，LISP のプログラムは，サブルーチンだけで構成されているといえよう。サブルーチンの引数も別のサブルーチンか，定数が入ってくるというようなイメージである。実際の LISP は括弧を多用して記述するため，C などのプログラミングに慣れた人には非常に読みづらい。また，通常の算術式のような記法も許されていない。例えば，2+3×4+1 を LISP で記載することを試みよう。まず，和と積を算術記号ではなく，ADD および MUL という関数（サブルーチン）で記載しよう。

```
ADD(2, MUL(3, 4), 1)
```

ただし，これは通常の C などの言語による書き方である。LISP ではこれをさらに一捻りした記載をする。関数[1)] を引数を同じ括弧の中へ移動し，その先頭に書くのである。したがって，

```
(ADD 2 (MUL 3 4) 1)
```

となる。さて，サブルーチン MUL は，3 と 4 から計算できる答えを返すだけで，他の変数をこっそり変化させたりすることはできない。というのは，サブルーチンの定義もすべて副作用のないサブルーチンを組み合わせて構成されるからである。

LISP などの多くの関数型プログラム言語でも，関数によっては若干の副作用が残る。例えば，データの内容を出力する PRINT 関数のようなものは，ディスプレーに文字列を表示するという副作用がある。したがって，関数型プログラム言語という用語は手続型プログラム言語に対する比較として使われると考えてもよいであろう。なお，副作用のまったく無いような厳格な関数型プログラム言語も存在する。

---

1) 実際の LISP では ADD は ‘+’，MUL は ‘*’ と書かれるが，ここでは読みやすいように，一般の関数の名称に近い形とした。

7.5　論理プログラム言語

昔のプログラムは，ほとんど，膨大な算術計算を行うために作られてきた。しかし，近年，コンピュータの利用先はどんどん拡張され，人間の行える作業は何でも実現してみようという方向に向かっている。特に論理的思考をまねてみようという方向が，人工知能と呼ばれる領域を形作ったのである。いわゆる三段論法のようなことをコンピュータにやらせようという目的で作られたのが**論理プログラム言語** (logic program language) である。有名な例として，

```
人間(ソクラテス).
死ぬ(X) :- 人間(X).
?- 死ぬ(ソクラテス).
```

上の 2 行から推論して第 3 行で yes と解答するのが，論理型プログラム言語である。

論理的に整合性のあるデータから，すべての可能性のあるデータを引き出すことができることから，大量のデータから所望のデータを取り出すデータベースの分野での用途も期待されてきた。一方で，厳格な論理性を有するため，近い答えを排除してしまう。これが，人間の持つ曖昧な判断に対し，劣るとされ，利用はやや限定的である。もっとも，人間の持つ曖昧性が優れているかというと，必ずしもそうとはいえない。むしろ，人間との親和性に劣ると言い替えるべきかも知れない。

LISP も，リストが扱えるため，論理型プログラム言語として機能した。それを発展させた Prolog が比較的よく知られている。

7.6 Web に適合した言語

最近のプログラムは Web 上で動作するものが多く，そのために文字列の扱いが楽な言語が好まれて利用される。高水準プログラム言語開発当初によく用いられた Fortran や COBOL が最近特殊な分野でないと見られなくなったのも，またその後，急速に発展した C があまり使われなくなったのも，それが理由である。

これらに代わって使われ出したのが，Perl，Python，Ruby，PHP，Java，JavaScript，Adobe Flash などである。これらは，言語の種類としては手続き型プログラム言語またはオブジェクト指向プログラム言語であるが，文字列に対する処理能力が高いことに特色がある。

当初は，サーバ側でプログラムを実行することにより，静的な Web ページと同様な文字列を返す形の CGI と呼ばれる技術が使われた。これらには Perl，Python，Ruby，Java などが使用される。

しかし，通信がネックになりやすい内容を表示するには，ユーザ側でプログラムを実行する方が有利なため，最近はこうしたリッチインタネットアップリケーションと呼ばれるものが増えてきている。これらには JavaScript，Adobe Flash，PHP などが使われる。

これらの言語については詳細を示さないが，Web で使われることもあり，使い方については Web 上に大量に掲載されているので，必要に応じ利用してほしい。

7.7 コンパイラとインタプリタ

高水準プログラム言語で作成されたプログラムは，機械語しかわからないコンピュータにとっては文字の羅列にすぎない。それをコンピュータ

にとって理解可能な機械語に変換するソフトウェアが必要である。**コンパイラ** (compiler) とは翻訳者の意味で，プログラム全体ができ上がった時点で，**ソースプログラム** (source program) と呼ばれる高水準プログラム言語の文字列を入力とし，機械語である**目的プログラム** (object program) を出力するソフトウェアである。また，翻訳する作業を**コンパイル** (compile) という。

コンパイルはすべてのプログラムを完成させた後でないと，実施できない。例えばプログラムが書きかけであると，コンパイルエラーが発生する。途中までででもプログラムを実施してくれないかという思いのあるユーザも少なくない。最初から，1 行打ち込むたびに，その行を実行してくれるような高水準プログラム言語も存在する。行単位の実行でも，行単位の機械語を生成する必要がある。この変換を行うプログラムを**インタプリタ** (interpreter) という。通訳という意味であり，コンパイラの翻訳との意味の違いが理解できよう。

しかし，Java に見られるように，コンパイルによって中間コードを出力し，それをインタプリタで解釈しながら実行するものが出てくるなど，コンパイラとインタプリタの区別は判別しづらくなってきている。また，インタプリタの動作も本質的にはコンパイラとほぼ同じであるので，以後は両者の区別をあまりしないでコンパイラという用語でまとめて議論を行うものとする。

コンパイラは，二つのフェーズで翻訳を行うものが多い。最初のフェーズでは，機械的に翻訳しやすいデータ構造を反映した**中間表現** (intermediate representation, IR) を出力する。例えば，通常の算術計算を行う際に書き下す括弧などの入った算術式をソースプログラムとすれば，逆ポーランド記法と呼ばれるものが中間表現といえる。逆ポーランド記法とは後置記法ともいわれ，2+3 を ‘2 3 +’ と記載する方法である。ENTER 記号は

本質ではないが，2 桁以上の数字を入れるのと区別するために使う。さらに，2+3×4 などは ‘2 3 4 × +’ とする。後置記法では，演算子が出てきた場合には，必要な数字はその直前から順に利用することになっているので，‘×’ は ‘3 4’ を対象とする。その結果，式は ‘2 12 +’ となる。続いて，‘+’ は ‘2 12’ を対象とするため，14 が得られることとなる。‘-’ については，減算と符号反転で対象とする数字の数が異なるため，異なる演算子として扱うこととなる。こうした後置記法は，コンピュータの命令の順番に密着しているため，機械語に翻訳しやすいのである。

まず高水準プログラムという文字列を構文解析する。まず，その文字列が文法違反をしていないかがテストされる。その際，利用されるのが，正規表現と文脈自由文法である。まず正規表現であるが，例えば，多くの高水準プログラム言語では，変数はアルファベットで始まる連続した英数字と定義されている。これを `['A'-'Z''a'-'z']['A'-'Z''a'-'z''0'-'9']*` と記載する。最初の ‘[’ から ‘]’ はアルファベット 1 文字，続く ‘[’ から ‘]’ は英数字 1 文字，‘*’ は英数字の 0 回以上の繰り返しを示す。こうした正規表現を用いて，変数などに用いられている単語以外にも，`if` や `while` などのように制御のために使われている予約語，演算記号や括弧もすべて分離し，トークンというものに分離する。正規表現への適用アルゴリズムについて，もし興味があれば Web などで調べてほしい。こうすることで，変数の候補がすべて揃うことになる。

続いて文脈自由文法にしたがっているかをチェックする。プログラム言語の構文は一般に，字句解析用の正規表現と構文解析用の文脈自由文法，例えば**バッカス・ナウア記法** (Backus-Naur Form) で定義される。**図** 7.1 は LISP に関する単純な構文の定義であり，LISP のプログラムは `list` でなければならないことになっている。詳細は省くが，`list` の右辺は `expression` の 0 回以上の組み合わせ，`symbol` はアルファベットの 0 回

```
list   ::= '(' expression* ')'
expression ::= atom | list
atom   ::= number | symbol
number   ::= [+-]?['0'-'9']+
symbol   ::= ['A'-'Z''a'-'z'].*
```

図 7.1　LISP に対するバッカス・ナウア記法による文法

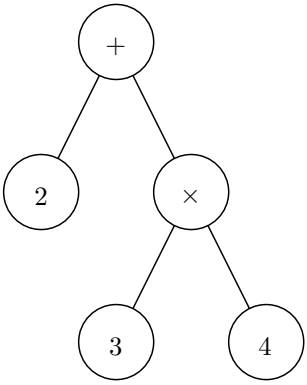


図 7.2　2+3×4 の構文木

以上の組み合わせ，number は ‘+ -’ のいずれかが 0 または 1 個あり，それに数字が 1 回以上続く，また，expression や atom の右辺の ‘|’ はいずれかという意味である。

この作業を行いながら，プログラムの構造を反映した**構文木**（こうぶんぎ） (syntax tree) というものを作成する。構文木というのは簡単にいえば，**図** 7.2 に示すように，演算子の下に必要な要素を並べたものである。この際，構文解析したものが，意味的に問題がないかをチェックする。例えば if の後ろに条件式が存在しているかなどのチェック，変数の型がおかしくないかなどのチェックである。

構文木が完成すると中間表現を生成する。構文木の下の演算が完了しないと上の演算が実施できないことを考慮する。この木の場合には，前述のように ‘2 3 4 × +’ が出力される。

第二のフェーズでは中間表現から機械語を生成する。この際，ジャンプを伴う流れの解析，簡単な計算の最適化を行い，アセンブラ言語に翻訳し，ただちに機械語に翻訳する。中間表現は，CPU の具体的な命令群には依存しないため，Windows マシンであろうと，Mac マシンであろうと関係なく生成できる。したがって，コンパイラを種々のマシンに合わせて作成しようとすると，第二フェーズの部分を書き直すだけで事足りる。

文法解析しながらアセンブラ言語への翻訳を行う。例えば if 文などを翻訳すると，アセンブラレベルでは goto 文が必要となるが，そのジャンプ先のアドレスは if ブロック（場合によると else ブロックも）の翻訳が終了しないと決定できない。このような時に，仮ラベルを付けておき，後からきちんとしたアドレスを与えるほうがはるかに作業しやすい。アセンブラ言語では，もともとラベルの使用を許しているので，その意味でアセンブラ言語を経由させるのは意味があるのである。ここまでの作業がコンパイルである。

出力サブルーチン，三角関数，指数関数などといったいろいろなプログラムで比較的よく使われるサブルーチンはすでにアセンブラー化されたものが，**ライブラリー** (library) という形で用意されていることが多い。これらのうちで，ユーザプログラムで利用されるものだけを取り出して，付ける作業を**リンク** (link) と呼ぶ。リンクが終わると，アセンブラ言語を機械語に変換しつつ，メモリー上に展開する。その際，ラベル名も実アドレスに変換される。そうして，その最初のアドレスにジャンプすると，機械語となったプログラムが実行されることになる。これを**実行プログラム** (executable program) と呼ぶ。

7.8　デバッグ

文法は守られているが，何かおかしなことがあると，実行はうまくいかない。あるいは一見正常に動作するが，期待通りの結果が得られない場合，プログラムには何らかのミスや欠点があることになる。文法上のミスも含め，あらゆる間違いを**バグ** (bug) と呼ぶ。虫つまり害虫の意味である。よほど短いプログラムならば，一度で完全なものが書けるかも知れないが，多くの場合，バグは存在するのが普通である。このバグを減らしていく作業を**デバッグ** (debug)，虫取りという。

昔は，出力文をあちこちに挿入し，変数などに期待通りの値が設定されるかを調べることにより，デバッグを行った。現在は，多くの高水準プログラム言語ごとに**デバッガ** (debugger) と呼ばれる専用のデバッグ支援ソフトウェアがある。これは，プログラムのエディタのような画面上で任意の行をマークでき，実行がそこに達すると，そこで実行を中断し，その時点における変数の値やそれまでに要した CPU 占有時間を示してくれるソフトウェアである。もちろん，そこから実行を再継続もできるし，中止もできる。

広義のデバッグの作業には，プログラムの効率化，高速化も含まれる。その場合には p.68 で述べたプロファイラを利用することになる。こうした支援ソフトウェアを利用して，徐々にデバッグを行っていくことになる。

演習問題 7

問題 7.1 次の用語を理解したかどうか確認せよ。

1) 手続き型プログラム言語
2) オブジェクト指向プログラム言語
3) 関数型プログラム言語
4) 論理型プログラム言語
5) アセンブラ言語
6) 高水準プログラム言語
7) コンパイラ
8) デバッグ

問題 7.2 LISP プログラム（ADD 2（MUL 3 4）1）が**図** 7.1 に示したバッカス・ナウア記法に合致しているかを調べてみよう。

8 オブジェクト指向プログラム言語

放送大学 岡部 洋一

《目標&ポイント》プログラムを擬人化して，相互に分担する仕事を分類し，互いに仕事を頼み合うという形で作成するオブジェクト指向プログラム言語についてその概念を説明する。

《キーワード》オブジェクト指向プログラム言語，OOP，オブジェクト，クラス，メソッド，is-a 関係，has-a 関係，継承，カプセル化，多態性

8.1 オブジェクト

本章では，**オブジェクト指向プログラム言語** (object-oriented program language)，および，それを用いた**オブジェクト指向プログラム** (object-oriented program, OOP) と呼ばれるプログラム言語を，Ruby という言語を使って学んでいこう。なぜ，Ruby なのかというのは，現在広く使われている高水準プログラム言語の中で，開発者が珍しく日本人[1)] であることに加え，表現が簡素であり，それにもかかわらず，OOP の形式をきちんと踏んでいるからである。ここでは Ruby を使ったが，それ以外の OOP でも本章に述べる概念はそれほど異なっていない。

OOP は，一つのソフトウェアを作成するのに多くの人数がかかわるようになり，他人の作ったプログラムを使うことが頻発するようになったため，他人の作ったプログラムの詳細に可能な限り立ち入らないで利用しようという考えから考案された。まず，他人のプログラムを使う場合に

1) 「まつもと ゆきひろ」氏

は，人に作業を頼むような形とする。機械的な処理をするものでも，擬人化して考えるとわかりやすい。この頼む相手を**オブジェクト** (object)，オブジェクトに頼む命令を**メッセージ** (message) という。

例えばパソコンの画面に散りばめられたアイコンであるが，ある特定のアイコン（とりあえず aIcon とでも名付けておこう）をクリックすると「aIcon さん，アプリケーションを起動してください」という作業依頼がなされ，aIcon をそれを受けて，あらかじめ設定されたアプリケーションを立ち上げるのである。この一つ一つのアイコンがオブジェクトである。それぞれのアイコンは位置の情報，画面上に示すアイコン図形，起動すべきアプリケーションといった情報を持っている。これら情報を格納している変数を**インスタンス変数** (instance variable) という。また，アプリケーションを起動する以外にも，位置を移動するとか，画面に現われるとか，画面から消えるとか，いくつかの共通な実行可能な作業項目を持っている。これらを**メソッド** (method) という。

ここで述べたインスタンス変数やメソッドを有する共通概念を定義するものを**クラス** (class) という。オブジェクトは何らかのクラスに属し，クラスに共通なメソッドとインスタンス変数を有するが，インスタンス変数の値はそれぞれ独立であるため，異なる位置に存在したり，異なるアイコン図形を持つことができる。オブジェクトとは元々「物」のことを意味し，OOP ではメッセージを送る対象であるやや抽象化された概念である。特定のクラスを意識して，その**メンバ** (member) としてのオブジェクトは**インスタンス** (instance) と呼ばれる。OOP とは，いろいろなクラスのインスタンスであるオブジェクトにメッセージを送るという形で書いていくプログラミングであるといえる。インスタンスに仕事を頼むときには「**インスタンス. メソッド** ()」の形[2] でメッセージを送る。

2） Ruby では引数のない場合，括弧を省略することも可能である。

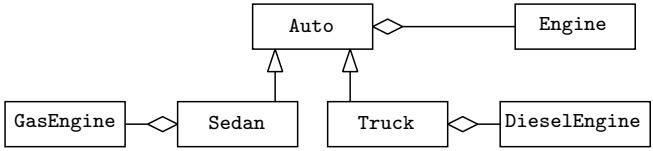


図 8.1　クラスの関係（ is-a 関係:三角矢印，has-a 関係:菱形矢印）

さて，オブジェクトに作業を頼むようになると，オブジェクトがきちんと頼まれたことを処理さえしてくれれば，それが具体的にどのように実施されていくのかについては，無関心でいられるようになる。例えば，アイコンをクリックしたらアプリケーションソフトウェアが立ち上がってさえくれれば，アイコンがどのようなしくみでアプリケーションを立ち上げているのかはどうでもよくなる。こうして，プログラムの独立性が高まり，複数のプログラマが関与することが容易になるのである。

OOP では一般に，多くのクラスが登場する。そこで，まずクラスをどのように設計していくかが重要となってくる。例として，種々の自動車を使ったプログラムを考えよう。自動車は Automobile であるが，紙面節約のため，Auto と記載する。Auto には色々な車種があるが，そのうちセダン（ Sedan）やトラック（ Truck）を考えよう。色々な車種を持つ Auto ではあるが，それは多くの部品から構成されている。車輪，客席，荷台などを考慮すればより具体的になろうが，そのうち，エンジン（ Engine）のみを扱おう。簡単のために，Sedan は必ずガソリンエンジン（ GasEngine）を，Truck はディーゼルエンジン（ DieselEngine）を搭載していることにしよう。こうしたいくつかの概念はすべて，それぞれ，クラス Auto，クラス Sedan，クラス Engine などのクラスを構成する。これらのクラスは，**分類** (classification) と **分解** (decomposition) という二つの方法で関係付けられている。Auto を分類していくと，Truck と

Sedan に分けられる。一方，Auto も Sedan も Truck もそれぞれ分解してみると，種々のエンジンなどの部品を持っている。これらの関係を**図** 8.1 に示すが，分類の方向も，分解の方向も矢印とは逆向きになっているので注意してほしい。

「分類」の関係は is-a **関係** (is-a relationship) と呼ばれる。「Sedan is a Auto.」などと書けるからである (英語的には an Auto)。つまり，「Sedan は Auto の一種」ということである。このとき，分類元を**スーパクラス** (super class)，分類先を**サブクラス** (sub class) という。Sedan や Truck をサブクラスとすると，Auto がこれらのスーパクラスとなる。動物を分類し哺乳類，鳥類などに分け，哺乳類を分類して人間，犬などに分ける行為も is-a 関係で結ばれている。

一方,「分解」の関係は has-a **関係** (has-a relationship) と呼ばれる。「Auto has a Engine.」などと書けるからである。このように，クラスの関係をよく分析してからプログラムを作成すべきである。

8.2 is-a 関係の構築

まず，Auto 関係のクラスが has-a 関係で利用する Engine 関係のクラスから検討しよう。エンジンにはガソリンエンジンやディーゼルエンジンなどがある。これらのクラス間にもたまたま「GasEngine is a Engine.」など is-a 関係が成立するので，まずはエンジン関連のクラスによって is-a 関係の実現法を示そう。

Engine をスーパクラスとし, サブクラスとして GasEngine と DieselEngine を置く。サブクラスに共通概念があれば，それをスーパクラスで定義し，サブクラスはスーパクラスを継承するという形をとるのがよい。**図** 8.2 に示す Engine クラスから見ていこう。クラスの定義は「class クラス

```
# クラスの定義
  # クラス Engine の定義
  class Engine
    def initialize(disp="")
      @disp = disp     # 排気量
    end
    def howDisp()      # 排気量の問合せ
      puts("排気量 #{@disp} cc")
    end
    def start()        # エンジンの起動
    end
    def stop()         # エンジンの停止
    end
  end
  # クラス GasEngine の定義
  class GasEngine < Engine
    def start()        # エンジンの起動
      puts("#{@disp} cc の G エンジン起動")
    end
    def stop()         # エンジンの停止
      puts("G エンジン停止")
    end
  end
  # クラス DieselEngine の定義 (略)
```

図 8.2　Engine 関係クラスの定義（ engine.rb）

名」で始まり対応する end で終了する。#以後行末までは Ruby におけるコメント[3] であり，プログラムの実行上は無視される。

次はメソッドの設計である。後に具体例を示すが，あるクラスのメンバーであるインスタンスに活動してもらうには，まずインスタンスを「クラス名.new()」で**生成** (construct) しなければならない。最初に書かれているメソッド initialize(...) は，クラスに属するインスタンスの生成の際に実行されるやや特殊なメソッドである。その際，引数があると，new() も引数を持たせて生成しなければならない。このようにインスタンスを生成する際に実行されるメソッドは特殊であり，特に**コンストラクタ** (constructor) と呼ばれ，Ruby では必ず initialize という名称で定義される。なお，言語によっては，インスタンスの利用を終えるとき，**デストラクタ** (destructor) といって，インスタンスを消し去るメソッドを用意しなければならないものもある。

このクラスの initialize は disp（排気量 displacement の略）という引数を持つ。「disp=""」とあるのは，new(disp) に何も与えられなかった際には，disp に空の文字列""を与えるという意味である。引数のデフォールト値であると理解してほしい。メソッドの定義はすべて「def メソッド名」で始まり，end で終了する。メソッド initialize() の定義をみると，@disp とあるが，Ruby では「@」で始まる文字列は**インスタンス変数** (instance variable) であり，外部からは読み書きできないインスタンスの中だけで密かに利用される変数を意味する。直接は見えないがこのクラスのメソッドを通じてのみ読み書きはできる。このため，initialize の引数 disp を代入することで@disp に値を与えている。

次の howDisp() は逆に@disp の内容を読み出すメソッドである。puts()

3) やや面倒なことにコメントの記載法は言語ごとに異なる。

は引数の内容を文字列にし，改行コードを付して出力する。変数の内容を出力したい場合は，文字列の中に「#{ 変数 }」を入れる。

次の start() および stop() は，エンジンを起動および停止するメソッドである。ただし，Engine はガソリンエンジンやディーゼルエンジンなどを統括するものであり，それぞれのエンジンにより起動法や停止法は異なるため，具体的な作業は記載していない。

特に別途指定しない限り，Ruby ではインスタンス変数（この場合@disp）はインスタンスの外部からは直接アクセスできないように隠蔽されている。このクラスのインスタンスの一つを例えば aEngine とするとき，aEngine.@disp を利用することはできないということである。一方，メソッドは外部からアクセス可能となっている。aEngine.start() などは利用可能なのである。ただし，メソッドでも initialize だけは new によってのみ利用できるが，やはりインスタンス変数と同じ基準で隠蔽されているため，aEngine.initialize(disp) は利用できない。

隠蔽 (hiding) とはいうが，別に人間が読めないという意味ではない。実際，新しいクラスを設計する際，良く作られたクラスを参考にすることはよくある。単に，プログラム内で，インスタンスの外部（設定によってはクラス，あるいはサブクラスの外部）から利用が可能かどうかを言っているのである。デフォールトで与えられているこうした隠蔽の制限を変更することも可能であるが，これらデフォールトの制限を利用する方が，一般的には利用しやすいコードとなるのである。こうした制限の結果，クラス Engine の利用者は，このクラスにどのようなメソッドが用意されていて，どんな結果が生じるかだけを知れば，その内部の具体的処理状況を知る必要はなくなるのである。こうした概念を**カプセル化** (encapsulation) と呼ぶ。このようにコードが隠蔽などにより独立してくることが，複数人によるプログラム開発を容易にした要因の一つなのである。

続いてクラス GasEngine の定義を見てみよう。GasEngine は Engine で定義されたインスタンス変数やメソッドをすべて使えるのが便利である。それにはクラス宣言のところで「< Engine」を記載する。こうしたスーパクラスの資産を利用することを，**継承** (inheritance) という。GasEngine のコードを見ると，initialize() がないが，これは Engine のコンストラクタをそっくり利用しているからである。

GasEngine に対しても，start()，stop() といったメソッドが必要であろう。例では，これらのメソッドはスーパクラス Engine のものとは異なる定義がなされている。これを**オーバライド** (override) という。本来は，これらのメソッドはハードウェアである実物のガソリンエンジンと結びついていて，実際にエンジンを起動したり停止したりするものであろうが，便宜上「G エンジン起動」などのメッセージを出すことにとどめたい。また start() では，スーパクラス Engine で定義されたインスタンス変数「@disp」を，自分のクラスのインスタンス変数のように利用している。

8.3 has-a 関係の構築

次に**図** 8.3 によって，Engine 関係のクラスを has-a 関係で利用する Auto 関係のクラスを設計しよう。第 1 行の「require_relative "engine"」は engine.rb を利用するという宣言であり，もし**図** 8.2 の内容が同じファイルに書かれている場合には不要な宣言である。本書ではセダン Sedan は必ずガソリンエンジンを使うものとして，クラスを設計している。自動車のスーパクラス Auto は has-a 関係で一般的エンジンを利用しているが，その場合，Auto の initialize で，クラス Engine のインスタンスである@aEngine を Engine.new(disp) で生成して利用している。new() の

```
require_relative "engine"
# クラス Auto の定義
class Auto
  def initialize(disp="")
    @aEngine = Engine.new(disp)
  end
  def start()         # 自動車のスタート
    puts("ハンドブレーキを緩める")
    @aEngine.start()
  end
  def stop()          # 自動車のストップ
    puts("ハンドブレーキを締める")
    @aEngine.stop()
  end
end
# クラス Sedan の定義
class Sedan < Auto
  def initialize(disp="")
    @aEngine = GasEngine.new(disp)
  end
end
# クラス Truck の定義 (略)
```

図 8.3　Auto 関係クラスの定義（auto.rb）

引数には，initialize() の仮引数 disp を代入している。他の言語でも，has-a 関係のクラスとは，このような形で連携を保つことが多い。クラス Engine を継承して連携することも可能であるが，is-a 関係にないクラスを継承するのは混乱を招くことがあるので，推奨しない。メソッド start() では，自動車を動かすための準備をした後，@aEngine.start() で一般的なエンジンを起動している。stop() も同様である。

クラス Sedan はクラス Auto のサブクラスとして定義しているため，そのクラス宣言で「< Auto」としている。initialize はスーパクラスの同名のメソッドとは異なるためオーバライドしており，@aEngine を GasEngine のインスタンス変数として生成している。それ以外のメソッドは，Auto のメソッドをそのまま継承している。

さて，start() であるが，あちこちに start() が定義されている。どの start() が使われるかは，それを使うオブジェクトの種類による。GasEngine のインスタンス変数として生成されたオブジェクトは，クラス GasEngine で定義された start() を使うこととなる。このように，同じ名称のメソッドを定義できることを**多態性** (polymorphism) という。これまでに示した**継承**，**カプセル化**，**多態性**を OOP の三本柱という。

8.4 オブジェクトへのメッセージ送信

作成したクラス Auto を使ってみよう。OOP におけるプログラムは基本的には，種々のクラスのインスタンスにメッセージを送る，つまり，それぞれのクラスのメソッドを使うことになる。**図** 8.4 では，クラス Auto や Sedan のインスタンスを生成し，それらにメッセージを送っている。Auto のインスタンス aAuto に start() というメッセージを送ると，右のコメント欄に示した出力からわかるように，ブレーキ操作のみ行われ

```
  require_relative "auto"
# プログラム
  aAuto = Auto.new()        # Auto のインスタンス aAuto の生成
  aAuto.start()             # 出力: ハンドブレーキを緩める

  aSedan = Sedan.new(1000) # Sedan のインスタンス aSedan 生成
  aSedan.start()            # 出力: ハンドブレーキを緩める
                            #       1000 cc の G エンジン起動
  aAuto = aSedan            # aAuto に aSedan を代入
  aAuto.start()             # 出力: ハンドブレーキを緩める
                            #       1000 cc の G エンジン起動
```

図 8.4　クラス Auto を使ったプログラム（test.rb）

る。クラス Engine のインスタンス aEngine を start() するが，それは**図** 8.2 のクラス Engine の start() で示されているように，何も出力しないからである。

次に Sedan のインスタンス aSedan を，排気量 1000 cc のエンジンを有するとして生成し，start() してみる。すると，今度はブレーキ操作の後，無事に 1000 cc の G エンジンが起動する。もし他の排気量を持つインスタンスを生成したければ，bSedan のような別名のインスタンスを生成し，種々の作業を行わせればよい。

三番目は少し面白い試みで，既に生成された aAuto に aSedan を代入している。aAuto は Auto 型の変数であり，aSedan は Sedan 型変数である。そもそも型が異なるものを代入してよいのかという疑問が湧くが，これは許されているのである。そして，代入以降，aAuto は Sedan 型の

インスタンスとして動作することになる。したがって，aAuto.start() は aSedan.start() と同じ応答をするのである。

OOP について駆け足の説明をしたが，Web 検索でいくらでも詳細を知ることができる。例えば「C++ アクセス制限」とすると C++のアクセス制限のしかたが，また，「言語名 入門」や「言語名 マニュアル」で，各言語のプログラミングのしかたや仕様を知ることができる。さらに「言語名 インストール」とすると多くの言語が無償で利用できるので，プログラムを作成しながら，内容を理解していくことも可能である。これが，もっとも速くかつ深く言語を理解する手法である。

演習問題 8

問題 8.1 次の用語を理解したかどうか確認せよ。

1) OOP
2) オブジェクト
3) クラス
4) メソッド
5) 継承，多態性，カプセル化

問題 8.2 クラス DieselEngine および Truck を作成せよ。さらに，これらを利用して，2000 cc のトラックをスタート，ストップし，出力を予想してみよう。

9 マルチメディア

放送大学 岡部 洋一

《目標＆ポイント》音声，静止画，動画といったマルチメディアの情報をどう扱うかを解説する。また，印刷機とのやりとり，電子ブックなどのしくみについても説明する。

《キーワード》データ圧縮，音，画像，静止画，動画，印刷，電子ブック

9.1 マルチメディアとは

当初，コンピュータは計算をする機械であった。日本語で計算機といっているのも，英語で compute する者というのも，すべてそのことを指している。しかし，今，コンピュータというと，Web を見たり，種々の音楽を聞いたり，静止画や動画といった画像を見たり，本を読んだりと，その内容は計算とはほど遠いところにあり，いろいろなメディアの処理という概念に変わってきたように思われる。こうした五感に訴えるようなコンピュータの入出力は**マルチメディア** (multi-media) と呼ばれる。実はマルチメディアという用語は元来，文章，音楽，静止画，動画，といったいろいろなメディアが統合されたものを指す。しかし，最近は本書で示すように，個々のメディアであっても，ディジタル化によって，統合的に扱えるようになったものを指すようになってきたため，本書でも，その意味で用いることとする。

こうしたことができるようになったのは，

- CPU の速度が速くなった。

- 内部メモリーが大容量になった。
- ハードディスクなどの外部記憶装置が大容量になった。
- インタネットの速度が増大した。

の 4 つの要因が上げられる。前二者はマルチメディアのような情報量の多いものを，ちゃんと処理して人間の負担にならない程度に迅速に提示するために必要である。三番目は情報量の大きなものを自分のコンピュータに残しておくために必要であり，最後の項目は，それを他人と共有するために必要な技術革新である。

もともとメディアとは情報媒体であり，新聞，放送，書籍といった，情報を載せる媒体のことを指した。しかし，コンピュータが出現し，種々の形式の情報の塊をコンピュータという一つの機械で扱えるようになってきてから，メディアという言葉が変容し，音，画像といったさまざまな種類の情報を指すようになってきた。その意味で本書でマルチメディアという場合には，それらさまざまな形態の情報を指すこととする。

マルチメディアの特徴は，前述のように，単純なテキストなどと比べ，情報量が多い。一般的な意味での情報量も多いが，コンピュータにとってより深刻なのは，使うビット数という意味の情報量が多いのである。このため，蓄積しておくにもメモリーを多くとるし，それを送受信するための負担も大きい。コンピュータの世界では，こうした負担を少しでも軽くするために，**データ圧縮** (data compression) という技術を発展させた。例えば，モノクロの静止画を送る際，白の連続や黒の連続する場合が多いことを考慮し，"白黒黒白白白白白白白白白黒"などとは送らず，"白黒 2 白 10 黒"といったような形式で送るなどの工夫をしている。圧縮されたデータは，利用する前に元に戻さなければならない。これを解凍あるいは伸長という。なおデータ圧縮という概念には，人間の感覚などが鋭敏でない部分の情報を削減して，圧縮する方法も含んでいる。こう

した方法により圧縮した場合には，解凍しても完全には元に戻らない。

マルチメディアの話となると，音，画像といった情報をどう外から取り込んだり，どうユーザの感覚に結び付けるのかという入出力の話に加え，どのようにデータを圧縮するのかという話の双方が必要となる。

9.2　ワープロ文書

ワードプロセッサで作成したフォント指定，章節などの位置など組版もされた文書，いわゆるデザインされたワープロ文書をマルチメディアと呼ぶには，ちょっと抵抗があるかも知れない。しかし，コンピュータが現れた当初は，コンピュータへの入出力はすべて 0/1 のビット列であった。やがて，そこに文字の列が加わる。大きさも飾りもない単なる文字の組み合わせ，いわゆる，テキストである。入力はキーボードからのアルファベットと数字と若干の記号のみ，出力も大きさも一定のたった一種類の**等幅フォント** (monospaced font) である。その後，この文字を少しずらして二重写しにすることにより太字，文字の上ほど右に表示することで斜体といった多少の文字修飾が可能となる。それに合わせ，簡単なワープロ文書作成ソフトウェア，つまり**ワードプロセッサ** (word processor)，いわゆるワープロ [1)] が開発されるようになってきた。この時期，作成されたファイルは，テキスト部分と，その後部に，どの位置の文字をどのように修飾するかを示した部分とで構成された。

その後，**フォント** (font) の種類も急増し，文字ごとに異なる最適な幅を持つ**プロポーショナルフォント** (proportional font)，また，大きさが自由に変更できる**スケーラブルフォント** (scalable font) といったものが開発され，修飾の可能性は大幅に拡張されるにいたり，本格的なワードプ

1) 実は，ワープロという言葉が出始めたころのワープロには，こうした文字修飾などのデザイン機能はなかった。

ロセッサの発展に繋がったのである。こうした歴史を見ると，やはりデザイン付き文書というものは，もっとも基本的な文字列であるテキストデータを跳び出したメディアという意味でマルチメディアの端緒となったメディアと言えよう。また，そのソフトウェアの技術を見ると，他の音とか画像といったマルチメディアの基本構造を構成しているのである。

さて出力デバイスの代表であるディスプレーであるが，その表示は例えば 1200×675 のような**ピクセル** (pixel) と呼ばれる有限個の格子の交点をどのくらい光らせるかで行っている。このピクセルに対応して，コンピュータ内には VRAM (video RAM) と呼ばれるメモリー領域が設定されており，そこへ書き込まれた値に比例して，ピクセルが光るようになっている。さらにカラーであると，各点は三原色で構成されるため，3 倍の VRAM 領域が確保されることになる。

原始的なフォントである**ビットマップフォント** (bitmap font) の場合，**ビットマップ** (bitmap) と呼ばれる VRAM に対応した文字の大きさのメモリー領域に，文字の各点の明かるさを書き込んだもので定義されている。ディスプレーに文字を現すには，そのフォントに対応するビットマップを VRAM の対応領域にコピーすることで達成される。**プロポーショナルフォント** (proportional font) も文字ごとにビットマップの大きさが異なるだけで，本質的にはビットマップのコピーで表示される。**スケーラブルフォント** (scalable font) などの場合には，文字の縁を表す直線や曲線の集合で構成される。そして，VRAM 上に必要なサイズに縮小あるいは拡大された文字の縁の作図を行い，これらで囲まれた領域を黒くする。

ワープロとは，デザインのないテキストを作成できることと，そのテキストをデザイン付きにできること，さらにそれをデザイン付きで表示できる機能を持ったソフトウェアである。ワープロ文書をファイルとして保存する際，画面に対応する VRAM の情報をそのままセーブすればよ

さそうであるが，ファイルサイズが巨大になり過ぎてしまう。また，再編集したいときなど，VRAM の情報から元のテキストや修飾の状態を取り出すのは大変な作業となる。このため，テキストと修飾状態が簡単に取り出すことのできるような形でファイル化する。一つは各ソフトウェアに固有なバイナリーファイルと呼ばれるもので，まだ CPU の機能があまり高くなかったころ，可能な限り CPU 負担の少ない形式で，CPU と情報のやりとりを行えるよう構成されたものが多い。その結果，バイナリー形式は第三者には解読が難しいものが多い。したがって，ワープロソフトウェアが変わると，完全には元のワープロ文書が復元できない場合もある。

これに対し，近年，使われるようになってきたのが，**タグ** (tag) によって制御された文書である。基本的にはテキストであるが，修飾したい文字列の前後などにタグと呼ばれる特別なテキストを挿入するものである。タグも通常の文字列であるので，第三者が修飾の状況を把握するのも容易である。タグであることを示すために，通常，**制御文字** (control letter) あるいは**エスケープ文字** (escape letter) と呼ばれる特別な文字，例えば ‘\’ を頭に付けて通常のテキストと区別する。制御文字自体を通常のテキストの中で使いたいときは ‘\\’ などのようにする。この代表は Microsoft Word などでも使われる RTF ファイルなどである。さらに，数式などを多用する理工系の研究者に利用されている TeX（テフ）もその代表である。

米軍が膨大な数のマニュアルに利用することから普及が進んだ SGML (standard generalized markup language)，Web で使われる HTML (hypertext markup language)，また現在のワープロソフトウェア [2] の標準となりつつある XML (extensible markup language)，さらには現在急速に普及

2）Microsoft Word の場合は Word2007 より XML になった。

しつつある**電子ブック** (electronic book) の標準である EPUB (electronic publication) や iBook (iBook) などは，‘< >’ で囲ったテキストをタグとした文書である。ちなみに，markup とは文書に付ける「しるし」のことで，タグと同じ意味である。また，markup language とは，こうしたタグで制御する文書の記述言語という意味である。

こうしたファイルからワープロ文書を復元するには，再び同じワープロソフトウェア，あるいはそのタグを理解できるビューアを使う必要がある。単純なテキストのように，その復元内容が簡単に見られるわけではない。これらのファイル形式はいずれもデータ圧縮という概念が存在していることに気付くかも知れない。データ圧縮には，元データをあるルールに基づいて圧縮する。復元側もそのルールを知らないと，復元できないのである。圧縮側と復元側が共通に持つルールが必要であることを理解して欲しい。

印刷機にデータを送る場合にも，昔は印刷イメージ全体のビットマップを送っていた。しかし，大部分のデータは文字である。したがって，文字列のテキストに加え，文字のフォントやサイズや色などの情報を送れば，大幅なデータ圧縮が可能となる。写真などのデータのみビットマップで送ればよい。もちろん，その場合には印刷機側に多数のフォントに対応した文字の情報などが必要となる。これも，かつては，各フォントごとに文字をビットマップで保存してきたが，そうすると文字のサイズごとに情報が必要となるため，最近はこれをベジェ曲線などの縮小拡大に強いスケーラブルフォントという形で保存するようになり，印刷機側のデータ圧縮がはかられてきている。

こうしたフォント情報などを，ある標準規格にしたがって通信することにすると，印刷機器ごとに異なる規格を用意する必要がなくなり，便利である。その一つの試みが**ポストスクリプト** (post script) というもの

であるが，今もって各プリンタごとに異なる規格が用いられており，それぞれプリンタごとに異なるデバイスドライバを必要としているのが現状である。なお，PDF (portable document format) という方式は，ポストスクリプトの技術を継承している。

9.3　音

音は空気の振動であるから，スピーカに波形を送ればよい。しかし，コンピュータはディジタルなので，時間方向も振幅方向も離散している。実際の音の波形は激しく振動はしているが，時間方向を拡大してみれば，その振幅は滑らかに変化している。これを一定の時間間隔で切り出し，その刻々の振幅を送出する，つまりパルスと呼ばれる幅の狭い棒グラフのような形の振幅を送出することにすれば，時間方向の離散化には対応できそうである。この時間方向の離散化を**標本化** (sampling) と呼ぶ。もちろん，時間間隔が大きいと高さの差分に相当したブツブツとした音が聞こえてしまうが，耳に聞こえないほどの高い周波数に相当する短い間隔の棒グラフを送り出せば，滑らかな波形と変わりない音が聞こえるのである。

振幅も一定高さの整数倍の高さを発生することにすれば，一定高さが十分小さいという条件で，滑らかな波形とほとんど同じ形を発生することができる。こうして時間方向にも振幅方向にも離散化することが可能となる。このような離散化作業は**量子化** (quantization) とも呼ばれる。

振幅は，多くのビット幅を持つ 2 進数で表す。これを棒グラフのような高さに変換するには，DA **変換器** (DA convertor) を用いる。DA 変換回路の詳細については省略するが，高い桁にあるビットに対しては大きな振幅を，低い桁にあるビットに対しては小さな振幅の電圧を出す電圧発

生器を用意し，これらを合計する回路により，ディジタル信号をアナログ信号へ変換する。

時間方向の離散化については**サンプリング定理** (sampling theorem) と呼ばれる理論的裏付けがあり，それによると，再現したい波形の持つ周波数の 2 倍の周波数で波形を送出すると，完全に波形を再現できることが知られている。現在，人間の耳はおよそ 20kHz まで聞こえることがわかっているが，これを少し低目にした 11kHz までに限っても，音楽などはあまり遜色ないため，この周波数の 2 倍の 22kHz で波形を送出する。また，音声などはもっと低い周波数で送出しても問題ないため，4kHz とか 2kHz での送出も行われることがある。

プログラムでは，最大 22kHz，つまり 1 秒間に 22,000 回のペースで，DA 変換回路に整数を送ればよい。通常，DA 変換回路およびその後ろにあるスピーカは，特定アドレスを持つメモリーのように見せかけてある。したがって，この特定アドレスにデータを書き込めば，音が出ることになる。

音楽のデータ圧縮については，電子楽器から出発した MIDI (musical instrument digital interface) という概念がある。電子楽器では同じキーボードを叩いても，ピアノ，オルガンに代表される弦楽器，管楽器，打楽器などの音を出すことができる。これらの音は**音源** (musical source) と呼ばれ，ディジタル的に合成されている。そこで，音源の種類と高さ，大きさといった情報だけを蓄積し，PC 間と音源を持った機器間をやりとりすれば，生の波形と比較し，蓄積情報も情報伝達もずっと楽になる。このやりとりの規格が MIDI である。

もちろん，MIDI では大演奏家の微妙なムードなどを伝えることはできないが，何しろ大量のデータ圧縮ができるため，この技術を使った音楽は急速に発展したのである。最近は PC にもいくつかの音源を搭載するこ

とが可能となり，電子楽器がなくても，PC だけでも音楽を楽しむことができるようになってきている。

9.4　画像

静止画や動画といった**画像** (image) は，通常，PC のディスプレーで見ることが多いので，それを前提に話を進める。カラーを扱う場合，VRAM の各ピクセルには，三原色に対応した三つの値を与える必要があるが，これらは RGB **信号** (red-green-blue signals, RGB signals) と呼ばれる。各 RGB 信号の強度を何段階で表示できるかはディスプレー側の回路に依存する。かつては，256 段階であった。したがって（255，0，0）は赤，（0，255，0）は緑，（0，0，255）は青，さらに（0，0，0）は黒，（255，255，0）は黄，（0，255，255）はシアン，（255，0，255）は紫，（255，255，255）は白などとなる。表現できる色の種類は 256×256×256=16,777,216 色となる。最近は各 RGB 信号を 65,536（256^2）段階まで，ほぼ滑らかに表示できるディスプレーが一般的になり，表現可能色は約 281 兆（256^6）色にもなる。

まず**静止画** (still image) の扱いであるが，**ドロー** (draw) と呼ばれる製図系のものと，**ペイント** (paint) と呼ばれる絵画系のものがある。日本語ではどちらも静止画であるが，ドロー系は直線や曲線により構成されるもので，例えば円は中心と半径と内部の色だけ指定すればよいので，大幅なデータ圧縮が可能である。曲線も**スプライン曲線** (spline curve) や**ベジェ曲線** (Bézier curve) のように，いくつかの点を指定し，その点を結ぶ滑らかな曲線を再現するルールだけ決めればよいので，数点の座標の指定に色情報を与えるだけでよくなる。

これに対しペイント系は，画家（ユーザ）の意思により，いくらでも

詳細変更が可能であるため，**ビットマップ** (bitmap) と呼ばれるピクセルごとの色データを確保しなければならず，大変な情報量が必要であった。しかし，こうしたペイント系のデータ圧縮も可能となった。一つの技術は FAX などで取り入れられた技術であるが，横に走査したデータを見ると，ある線のデータと次の線のデータの相関が極めて高いので，これを利用し，異なるデータだけを蓄積したり送ったりすると大幅なデータ圧縮が可能なのである。

もう一つの手法は JPEG (joint photographic experts group) と呼ばれる方法である。名前の由来は，画像圧縮方法を検討していた技術者集団のグループ名であったが，そこの制定した圧縮法の名前として定着したものである。画面を 8×8 画素からなる小さな正方形で分割し，それを周波数変換し，低周波成分は詳細に，高周波成分は粗い情報を割り当てることでデータを圧縮する。人間の視覚情報処理がゆっくりした明るさの変化に敏感で，急峻な変化には鈍いという特長を利用したものといえよう。

動画 (moving image) は一言でいえば，静止画をすばやく変更していくことで実現する。動画のデータ圧縮は連続する静止画間の相関が極めて高いことを利用する。どの圧縮方式でも，まず，直前の静止画との差分をとり，その差分情報だけを送信する。これだけでも相当のデータ圧縮が達成される。MPEG (moving picture experts group) と呼ばれる方式では，さらにその差分に対し，JPEG のような処理をすることにより，一層のデータ圧縮を達成するのである。

9.5 マルチメディアに関するプログラム

以上のように，マルチメディアを相手にしたプログラムといっても，周辺装置に接続された擬似メモリーもしくはポートに対して，対応する

ビットの書き込みをするだけでよいので，原理的には，特に複雑なことはない。しかし，ほとんどのマルチメディアソフトウェアではデータ圧縮を行うため，これに対応する必要がある。しかも，その圧縮法によっては，かなり複雑な処理を行っているため，その処理法を正しく理解し，プログラム化しなければならない。

演習問題 9

問題 9.1　単純テキストのファイルも，実はディスプレーに表示される文字像と文字コードの間に，一種のデータ圧縮を行っている。テキスト作成プログラムとそのテキストを見るプログラムがどのようなルールを共有しているか考えてみよ。

問題 9.2　サンプリング定理について，一定周波数の正弦波を種々の周波数のパルス列でサンプルした結果を，図を描くことで確かめてみよう。

1）正弦波の周波数に対し，非常に高い周波数でサンプルすると，サンプル結果は同じような値が比較的連続することを確かめよ。具体的には 10 倍程度の周波数で図を描いてみよう。

2）サンプリング周波数を下げていき，正弦波の周波数の 4 倍程度であると，正の値あるいは負の値の続く回数がおよそ 2 回ずつとなり，頻繁に上下が入れ替わるようになる。

3）サンプリング周波数が正弦波の周波数の 2 倍とほぼ同じであると，正負の値がほぼ交互に出現する。

4）サンプリング周波数が正弦波の周波数とほぼ同じであると，サ

ンプリングの結果は同じような値が継続し，1) の結果と区別が難しい。

これらのことから，サンプリング周波数が正弦波の周波数の 2 倍以下であると，問題が生じることが理解できよう。

問題 9.3　比較的変化の少ない波形は低い周波数成分しか持たず，変化の激しい波形は高い周波数成分を持つことが知られている。このことから，ある画面の x 方向の空間周波数が低く，y 方向の空間周波数が高い場合，画面の明るさの変化はどのようになっているか推定せよ。

10 オペレーティングシステム

放送大学 岡部 洋一

《目標&ポイント》現在のコンピュータでは，複数のプログラムが置いてあり，また，それを同時平行的に動かすことも可能である。また，仕様の異なる入出力装置と接続することも可能である。こうした仕事を行っているオペレーティングシステムについて解説する。

《キーワード》オペレーティングシステム，OS，デバイスドライバ，ファイルシステム，メモリー管理，プロセス管理，ユーザインタフェース

10.1 OSとは

個人やベンダの作ったソフトウェアを**応用ソフトウェア** (application software) と呼ぶ。現在は，こうした応用ソフトウェアであるプログラムをメモリーの先頭アドレスから格納しておき，それを直接実行するようなことは，現在のコンピュータではあり得ない。そのようにすると，一つのプログラムが終了するごとに，新しいプログラムを何らかの方法でメモリー上に記憶させ，毎回コンピュータを起動しなおす必要が生じるからである。

現在は，複数のプログラムを同時に動作させることが普通である。また，それぞれのプログラムごとにディスプレー上に**窓** (window) を持ち，プログラム間の独立性が保証されている。

また，種々の大容量記憶媒体，ディスプレー，キーボード，音響装置などは，コンピュータの製品ごとに異なる会社の部品が取り付けられてい

る。それに対応して異なる応用ソフトウェアを作るのは大変である。そこで，ハードウェアの差異を隠して，共通の呼び出しで，これらのハードウェアを使えるようにする必要がある。

そこで用意されたのが，応用ソフトウェアの裏方となってこれらの動作を支え，コンピュータ全体を統括する**システムソフトウェア** (system software) である。**基本ソフトウェア** (basic software) とも呼ばれる。システムソフトウェアが行う作業はおよそ以下のようなものである。

- ハードウェアとの仲介
- ファイルシステム
- メモリー管理
- プロセス管理

このうち，ハードウェアとの仲介をするソフトウェアを**オペレーティングシステム** (operating system, OS) と呼ぶ。しかし，オペレーティングシステムという用語は，広義には，しばしばシステムソフトウェアと同義に使われる。本書でも，オペレーティングシステムを広義に使い，応用プログラムを支えるシステムソフトウェア全体を指すこととしよう。代表的な広義な意味でのオペレーティングシステムは，サーバなどで使われている `Unix`，それに近い `Linux`，`PC` を中心に発達した `Microsoft` 社の `Windows`，`Apple` 社の `iOS` などである。

10.2 ハードウェアインタフェース

また，キーボード，ディスプレー，ハードディスクなどの大容量記憶媒体，音響装置，さらにネットワークへの通信装置などは，コンピュータの製品ごとに異なる会社の部品が取り付けられている。これら `CPU` や外部装置の制御をいちいち応用ソフトウェア側でプログラミングするの

は大変であるので，応用ソフトウェアに対して装置に依存しない形の呼び出しを可能にするような工夫も必要である。さらに，ディスプレー画面に出す内容は，すべて応用ソフトウェアが行ってきた。しかし，マルチタスクの概念に合わせて，各応用ソフトウェアはそこに開かれた窓と対応をとるようになってきた。こうした窓についても，簡単な呼び出しでアクセスできると便利である。

これら CPU や外部装置の制御をいちいち応用ソフトウェア側でプログラミングするのは大変であるので，応用ソフトウェアに対して装置に依存しない形の呼び出しを可能にするような工夫も必要である。そのために，利用している外部装置ごとに**デバイスドライバ** (device driver) というソフトウェアが開発され，OS との間を取り持っている。デバイスドライバも OS の一部としてよいが，デバイスドライバは通常，外部装置の製造会社に作成を依頼することが多い。

10.3　大容量蓄積装置の管理

次節で述べるメモリー管理は，OS の重要な仕事であるが，最近は周辺装置として置かれているハードディスクなどの大容量蓄積装置との関連が深くなっているので，まず大容量蓄積装置の管理について説明しよう。

p.14 で**メモリー** (memory) と記載した**主メモリー** (main memory) は，一般に，読み書きの速度が速いがやや容量が小さく，かつ電源が切れると内容が消えてしまう**揮発メモリー** (volatile memory) であることが多い。これに対し，**周辺装置** (peripheral unit) として置かれているハードディスクなどの**大容量蓄積装置** (mass storage) とは GB，TB（1B=1byte=8bit）といった大きな容量を持つメモリーであるが，電源が切れても書き込んだ内容が消えることのない**不揮発メモリー** (nonvolatile memory) である。

特にハードディスクは容易に書き換え可能である特長を持つ。CPU の中に置かれているレジスタと呼ばれる小さくかつ極めてアクセスの速いメモリー，主メモリー，ハードディスクと順に，アクセス速度が下がり，容量が大きくなっていく様子をしばしば**メモリーチェーン** (memory chain) と呼ぶ。また，これらのメモリーの特性から，多くの応用プログラムは大容量蓄積装置に置かれ，特定のプログラムを実行したいときに，そのプログラムを主メモリー上にコピーしてから実行するようになってきている。

ハードディスクは，ある程度の領域をまとめて読み書きするのは多少速いが，飛び飛びに読み書きするとアクセス時間が著しく増大する。このため，応用プログラム全体とかデータ全体をファイルというまとまった形で読み書きする。さらに，ディスクの任意のところから読み書きするのではなく，**セクタ** (sector) と呼ばれる同じサイズの領域に分割し，この単位で読み書きをする。したがって，小さなファイルでも一つのセクタを使ってしまう。一方，大きなファイルでは複数のセクタ，それも場合によっては飛び飛びに使うことになる。このため，セクタが使われているか否かやセクタの繋がり状況を管理する必要が出てくる。また，ファイルの数が多くなってくると管理が困難になるため，任意の数のファイルを**ディレクトリー** (directory) と呼ばれる小箱のような概念に格納することが行われる。こうしたセクタやファイルやディレクトリーの管理ソフトウェアは**ファイルシステム** (file system) と呼ばれ，OS の重要な仕事の一つである。ハードディスクのメーカに依存する部分であるデバイスドライバと協力して，管理を行っている。

なお，CD (compact disc) や DVD (digital versatile disc) などの媒体は書き込みのできるものもあるが，書き込み速度はハードディスクに比較すると極めて遅いため，基本的には読み出し専用である ROM (read only

memory) 型の大容量蓄積装置として扱われる。つまり，セクタやディレクトリーという概念が存在する。書き込みについても，基本的にはハードディスクと同じであるが，ハードディスクのように頻繁に読み書きを実施しないことを前提に，全体に一気に書き込むとか，たかだか数回に分けて書き込んでいくような専用ソフトウェアが用いられる。

近年，フラッシュメモリーを用いた SSD (Solid State Drive) が使われるようになってきている。磁気的なハードディスクと比較し，可動部分がないため，機械的ショックに強く，また価格的にもハードディスクに接近しつつある。半導体ではあるが，書き込み速度はハードディスクに近いため，メモリーチェーンの比較的後ろに置かざるを得ない。こうした特徴から，本来は連続書き込みといった概念は不要なデバイスであるにもかかわらず，ハードディスクと同じようにセクタ，ディレクトリーといった概念によるハードディスクに見せかけたデバイスとして提供されている。

ディレクトリーは一種のファイルとして扱われる。もちろん，ただのファイルではなくディレクトリーであるという標識が必要である。また，ファイルと異なり，データとして，自身のディレクトリー名，その中に置かれているファイルや子ディレクトリーへのポインタが格納されている。

1 セクタの上限サイズを越えたファイルは，当然，他のセクタも使うようになるが，連続性は保証されない。このため，セクタとセクタの繋がりの管理情報も，このハードディスクの中に収容されている。ハードディスクの容量がだんだん大きくなってくると，セクタの総数が多くなり，セクタの番地を表すビット数を変更する必要が生じてくる。また，セクタそのもののサイズも変更が必要となる。こうした際，ハードディスクの基本仕様ががらっと変わり，新しい規格が導入されることになる。

また，何度も情報を書き換えたハードディスクでは，セクタ間の連続性

がどんどん低下していく。これを**フラグメンテーション** (fragmentation) という。ハードディスクは，もともとセクタが不連続になっても，利用可能であるが，アクセス速度はどんどん落ちていくため，定期的に断片的になったセクタを連続的になおす**デフラグメンテーション** (defragmentation) という作業が必要となる。

10.4 メモリー管理

主メモリーはハードディスクとは異なり，読み書きの速度が速く，かつランダムに読み書きしても速度が落ちないという特長があり，1 アドレスごとに利用されることが多い。したがって，このメモリーの使用にあたっては，所定のアドレスが使われているかどうかを統括的に管理する必要がある。これを**メモリー配置** (memory allocation) 管理といい，OS の重要な仕事の一つとなっている。

あまりに多くのプログラムの同時実行を要求されると，各プログラムが必要とする格納領域の総和が，搭載メモリーの上限を越えてしまうことがある。このような場合には，データを含むプログラムの一部をハードディスク上に退避させながら実行する必要が生じる。退避させた情報は，必要があればまたメモリー上に戻さなければならないので，メモリーとハードディスク間の入れ換え作業つまり，**スワップ** (swap) という作業が必要となる。このようなことをするには，各プログラムが動的に確保するヒープ領域の管理も矛盾なく行わなければならない。このようにして，利用可能な見かけのメモリー領域を増加する手法を**仮想メモリー** (virtual memory) と呼ぶが，この作業も OS の重要な仕事の一つである。

また，p.56 の「サブルーチン」のところで述べたヒープ領域やスタック領域の管理も，OS の仕事である。

10.5　プロセス管理

また，プログラムの実行も，一つのプログラムを実行するだけでなく，複数のプログラムを同時平行的に実行することが多くなってきている。といっても，実際には，CPU が一つの場合には，本当に同時に複数のプログラムが動くわけではない。主メモリーには複数のプログラムを搭載しておくが，あるプログラムをある程度実行したら，次のプログラムを実行するなど，少しずつ断片的に複数のプログラムを実行する**マルチタスク** (multi-task) という方法が一般的である。

さらに，近年は一つのコンピュータが複数の CPU を持つことも多くなってきている。こうした場合，切れ切れに分離したプログラムの実行を CPU に割り振る仕事も必要となってくる。

10.6　ユーザインタフェース

現在のパソコンでは，ディスプレー上に，各応用プログラムに対応した複数の窓を表示している。また，いくつかのアイコンや背景も表示することができる。こうした機能は，一足飛びに実現できたわけではない。

まず，コンピュータの出力としてディスプレーが使えるようになった当初は，ディスプレーには文字を表示することしかできなかった。次の段階で，図形を描くことが可能となった。さらに，マルチタスクの実現と合わせて，ディスプレー上に窓を開けて，その中に各プログラムごとのテキストや図形の出力を出せるようになった。これが，ウィンドウシステムの始まりである。当初は，窓を作成するのも，各応用ソフトウェアのプログラマの責任であった。現在は，窓の作成も，また応用ソフトウェアの出力をその中にきちんと，出すのも，すべて OS に任せることができ，窓操作のプログラミングは非常に楽になっている。

まず，OS に対し，所定の場所へ所定のサイズの窓を開けることを要請する。その際，窓に何らかの臨時の名前を付けておく。以後，その窓名を使って，所望の窓とのデータのやりとりが可能になるのである。窓にテキストや図を書き込めば，必ずその窓内に納まるように，テキストや図が表示される。ユーザが窓に書き込んだデータは窓名を使って，プログラムに取り入れることもできる。ユーザがマウスなどを使って，窓を移動したり，窓のサイズを変更しても，プログラマはそのことを意識する必要はない。そうした際に窓内の図形などの移動やはみ出しの処理はすべて OS がやってくれるのである。最後に，必要としなくなった窓を閉じることで，臨時の名前の役割も終了するのである。

窓以外にも，最近は，ディスプレー内に並んだアイコンをクリックすることで，所望のプログラムを起動したり，アイコンがデータを代表するものの場合には，そのデータを扱える応用ソフトウェアが起動され，そのデータを読み込んでくれる。こうしたユーザにとってわかりやすい図形によるコンピュータ制御法を，**グラフィカルユーザインタフェース** (graphical user interface, GUI) と呼ぶ。こうした作業を，応用ソフトウェア個々ではなく，一つ上の，いわばメタソフトウェアである OS に任せることは，応用ソフトウェアの開発者の手間を減らすという効果だけではなく，コンピュータの動作環境を統合的にするという観点からも，現在，決して省略することのできない OS の仕事となっているのである。

演習問題 10

問題 10.1　次の用語を理解したかどうか確認せよ。

1）システムソフトウェア
2）OS（狭義，広義）
3）応用ソフトウェア
4）デバイスドライバ
5）ファイルシステム
6）メモリー管理
7）プロセス管理
8）ユーザインタフェース

問題 10.2　スタック領域の管理プログラムの概要を考えてみよ。スタックにプッシュ要求があった場合，ポップ要求があった場合，どんな作業をしたらよいかを考えてみよう。

問題 10.3　窓の管理プログラムは，オブジェクト指向言語との親和性がよい。どんなインスタンス変数を用意したらよいか，どんなメソッドを用意したらよいかを考えてみよう。

11 リスト構造とデータベース

放送大学 岡部 洋一

《目標＆ポイント》ある程度多くのデータを処理するとき，これらのデータを探しやすくし，かつ効率よく記憶しておくための工夫が必要である。こうした場合に威力を発揮するのが，リスト構造である。また，適切に管理されたデータの集合はデータベースと呼ばれる。これらについて解説する。

《キーワード》データベース，配列，ハッシュ関数，スタック，キュー，線形リスト，ツリー，二分木，ドット対，関係データベース

11.1 データベースとは

かつて，コンピュータの仕事の大部分は，四則演算の延長であるいわゆる計算であったが，最近は大量のデータを扱うことが多くなってきた。その背景には，ハードディスクなどの大量蓄積装置の発達がある。このため，四則演算とは異質の大量データをどう扱うかという技術が必要となる。やや特異な技術なので，本章ではこうした技術に特化して説明を行う。

プログラムはしばしばデータの集合を扱うことがある。例えば，三次元の位置などのベクトル的な量を表すには（x，y，z）の三つの量をメモリー上にまとめて連続的に配置し，三つを一緒に移動したりコピーしたりできるほうが便利であることは容易に想像できよう。 さらに，コンピュータの規模が大きくなるにつれ，扱うデータもどんどん大きくなり，ある概念に沿った大量のデータの集合である**データベース** (database,

key0	name0, sex0
key1	name1, sex1
...	...

key0	name0	sex0
key1	name1	sex1
...	...	...

図 11.1　値が複数ある場合の考え方

DB) と呼ばれるようなものも急増してきている。例えば，学籍簿のようなものは，学生に対する氏名，性別，住所，入学時期，授業料納入状況などの大量の情報を，すべての学生に対して蓄えている。

データの集合は，いずれはそれらを利用するために作成することが多い。利用するからには，簡単に探しやすい工夫をする必要がある。必要なデータを探すことを**検索** (search) と呼ぶ。もっともよく使われるのが，データを代表する**キーワード** (keyword) を使う方法である。Web などの検索ページでは，**全文検索** (full-text search) といって，データ内に出現するあらゆる単語をキーワードとして，検索可能としている。

所望のデータにアクセスするためのこうしたキーワード的な短い長さの「きっかけ」を**キー** (key) と呼ぶ。キーは単語だけとは限らない。学生証番号のような数字とアルファベットの集合とか，数字だけのキーもある。また，データを集合に取り込んだ際の順番，つまり**配列番号** (array number) もキーの一種と理解することができる。ただし，一般のキーと異なり，配列番号はメモリー上の位置と深く関係しているのが普通である。つまり配列のスタートアドレスに配列番号を加える（配列の要素が大きい場合には，そのサイズ分を書ける必要がある）と，所望データにアクセスできる。

例えば，配列番号をキーとするような場合，一つのキーに対し，データが一つしか存在しない。このような場合には，キーを与えるだけで対応

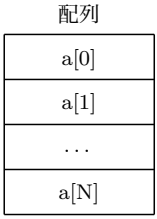


図 11.2　配列の構造

データが確定する。こうした場合, そのキーに対応するデータを**値** (value) と呼ぶ。学籍簿のように，値が一つの概念ではなく，氏名，性別など集合になっている場合も多く見られる。こうした場合，**図** 11.1 に示すように，値は氏名，性別などの複合した一つの塊であるとみなす場合と，一つのキーに対して値 1，値 2 と複数の値があるとみなす場合の二つの考え方がある。

こうしたいろいろなデータの集合に対し，いろいろなメモリーの配置法が提案されている。どれが本命であり，どれが優位というのは，こうした集合データをどういう使い方をするのかに大きく依存する。そのため，いろいろな提案があるのであり，またいずれもまだ生き残っているのである。本章では，そのうち代表的なものについて，紹介する。

11.2　配列，ハッシュ関数

配列 (array) とは複数のデータを記憶しておく手法としてもっとも基本的な構造であり，**図** 11.2 に示すように，データをメモリー上に一次元的に配置したリストである。ただし，データのサイズはすべて同じ長さ，もしくは一定の与えられたサイズに納めるようにし，各データの先端アドレスは，配列全体の先端アドレスから容易に算出できるようになって

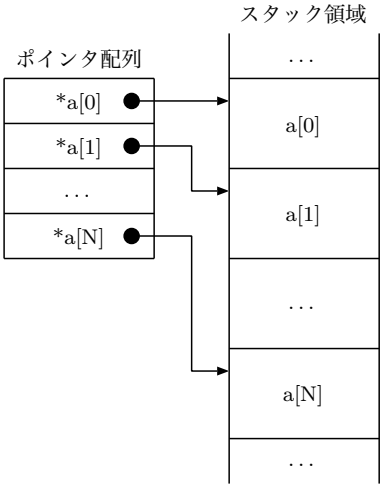


図 11.3　ポインタ配列の構造

いることとする。

この場合，キーは配列番号であり，特定の配列番号のデータを探し出すためには，配列番号 × データサイズを，配列の先端アドレスに加えて得られたアドレスにアクセスすればよい。配列全体の先端アドレスに一番最初のデータを置き，以後それに続いてデータを置いていくことが多いが，その場合，配列番号は 0 から始まることになる。多数の数字データの和，平均，標準偏差などを計算するような場合，こうした計算は数字の出現順番に無関係であるので，配列の利用は極めて有効である。

データが不定長だったりする場合には，データの置かれているアドレスの計算の手間が面倒になる。こうした場合，ポインタという概念を使うと便利である。**ポインタ** (pointer) とはデータの格納されているアドレスのことをいい，データを指すものという意味である。つまり，**図** 11.3 に示すように，生のデータはある程度勝手なアドレスに拡張し，配列に

は，そのアドレス，つまりポインタをデータとして格納しておくのである。このようにポインタは多用されるが，以下の説明では，すべてのデータは `1word`（メモリーのもともとの幅）であるとして，説明しよう。複数 `word` や不定長 `word` のデータはすべて，ポインタを介してアクセスされると理解してほしい。

キーとなる数字が一連の並んだものではなく，飛び飛びでかつ大きな値を持つものが多いなど，広い範囲に拡がっている場合，これらをそのまま配列の順番の数字に対応させると，膨大な配列領域を用意しなければならなくなり，無駄の多い構造となる。ならば，飛んでいるところを詰めて，キーと値をセットで配置すればよさそうであるが，そうなると，検索の場合に手間がかかることとなる。

こうした場合，**ハッシュ関数** (hash function) という関数が利用される。ハッシュ関数とは，大きな範囲に拡がった数字を決められた範囲の数字に変換する関数のことで，指数関数のような連続的な関数ではなく，簡単な数字処理プログラムにより，もともと大きな数字範囲に飛び飛びに存在していたキーとデータの組を，ほぼデータ数ぐらいの大きさの配列に納めるものである。ハッシュ関数により変換された配列番号を**ハッシュ値** (hash value)，ハッシュ値を配列番号とするデータの配列を**ハッシュ表** (hash table) という。

どのようなハッシュ関数を選ぶのかが一つのポイントになる。ハッシュ値がなるべくランダムに分散し，ハッシュ値が重複しないことが望ましい。しかし，それでもそれを完全に避けることはできない。このハッシュ値の**衝突** (collision) をどう避けるかについては，例えば衝突した際に，ハッシュ表を構成しているまだ空いている配列要素に割り当てる方法がある。これを**オープンアドレス法** (open addressing) という。例えば，すぐ下を見て，空いていればそこを利用する。駄目ならば，さらにその先をチェッ

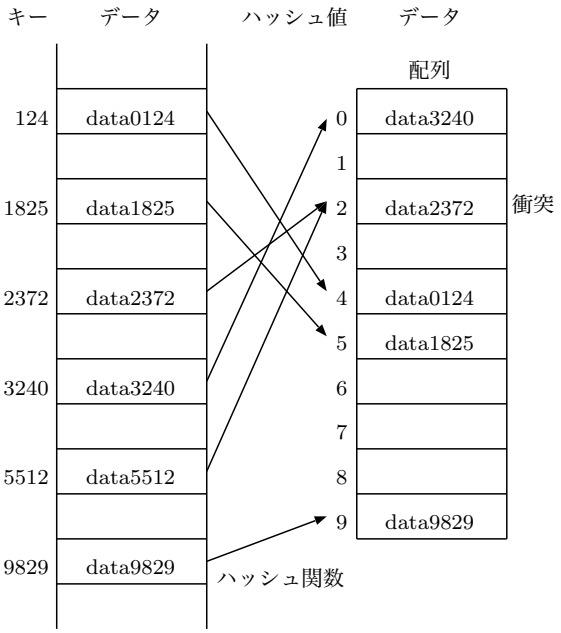


図 11.4　ハッシュ関数を利用して広い範囲に散ったキーを短い配列へ収容する方法

クするという方法である。しかし，この方法は，ハッシュ表に団子状態を作りやすいという問題を生じる。団子ができると，その団子の中に別のキーのハッシュ値が割り当てられる可能性が高くなり，ちょうど渋滞がさらなる渋滞を呼ぶような問題につながる。そこで，再びハッシュ関数を利用して，空いた先を探すなどの工夫がとられる。

また，衝突が発生した場合，同じハッシュ値にいくつかのデータを割り当てる**連鎖法** (chaining) と呼ばれる方法もある。その場合には，次節で述べる線形リストの手法により，ポインタを使って複数のデータを繋いで配置する。渋滞のような問題は生じないが，ポインタ操作はそれな

りに時間がかかるため，あまりデータが長くならないような配慮が必要となる。

さてハッシュ関数であるが，前述のように，キーがちょっとでも異なると，対応するハッシュ値がランダムに散るような関数が望ましい。キーが近いとハッシュ値も近くなるような関数であると，衝突が発生しやすくなるからである。関数と言っても，ランダムに近い数値を作り出す関数なので，実際には関数型のサブルーチンである。さらに，計算時間が短くないと，速い検索ができなくなり意味がなくなる。例えば，**除算法** (division method) と呼ばれるものは，キーを配列の総数で割り，その余りをハッシュ値とするものである。キーが連続していると，ハッシュ値も連続した値となるが，これにより衝突の可能性が高くなるわけではないので，有効な方法である。ただし，団子はやや発生しやすくなる。この他，**中央二乗法** (mid-square method) や**折畳み法** (folding method) などが知られている。

ハッシュ関数のもう一つの重要な特長は，検索が速いことである。単純にキーのリストから検索語と一致するキーを探そうとすると，いちいち順に一致をチェックしていかなけらばならない。ところがハッシュ関数を使うと，ハッシュ値として配列の番号が定まるため，キーの場所が直ちにわかり，一致のテストを行う必要がない。ハッシュ関数が多用されるのは，実はこの高速性の方が主な理由であるといってよいだろう。

11.3　スタック，キュー

データの取り込みと，データの処理がある程度独立に行われるシステムでは，データを一時的に溜め込んでおく必要がある。窓口処理のように，先に入ったデータを先に処理するキューと呼ばれるデータの蓄積法が思い浮かぶが，p.56 のサブルーチンの処理の際使われたスタックのよ

うに，後のデータを処理しないと，前のデータが処理できないような蓄積法も必要である。 ここでは，まず構造の簡単なスタック，続いてキューの話を行う。

11.3.1　スタック

データをどんどん入れていき，最後に入れたデータを最初に見られるようにした格納法が**スタック** (stack) である。データのうち，最初に入れた方を**終端** (tail)，もっとも新しい方を**先端** (head)[1] と呼ぼう。スタックとはこの先端しか見られないようにしたデータ構造である。皿を置いていくと，底が徐々に沈んでいくような皿置きのイメージである。各皿が格納したい情報に対応する。また，情報を増やすには，新しい情報を先端に積んでいく。この作業を**プッシュ** (push) と呼ぶ。積まれた皿の一番上を見るのは容易であるが，途中の皿を見るのは極めて難しい。つまり，最上段の情報にしかアクセスできない。情報を廃棄するのも上から順にしか廃棄できない。一番上の情報を取り出しながら廃棄することを**ポップ** (pop) と呼ぶ。最後に入れた情報が最初に出てくることから LIFO (last-in-first-out) と呼ばれることもある。あるいは，最初に入れた情報が最後に出てくることから FILO (first-in-last-out) とも呼ばれる。

順番に並んだ皿に対応させて，これを**図** 11.5 に示すように，配列で実現することも可能である。プッシュやポップの際，メモリーの内容をいちいち 1 個ずつ移動するのは大変であるので，底の固定された皿置きのように，情報が入っただけ，先端の上に積み上げていく。配列は上らから下へ確保していくものなので，スタックの場合にも積めば積むほど番地の高い方 (図示するときは下) へ伸びるもの[2] として説明する。つま

1) ここでは最初に取り出せる方を先端としたが，書によっては逆の定義もある。
2) p.56 では逆に，下位のアドレスへ伸びるスタックを説明した。

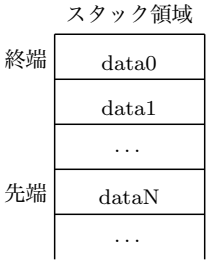


図 11.5　配列によるスタックの実現

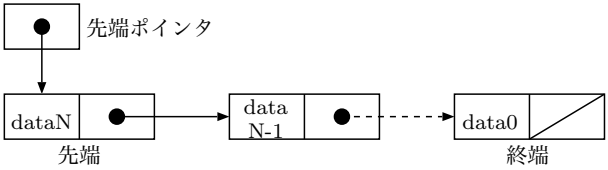


図 11.6　線形リストによるスタックの実現（斜線は `Null` ポインタ）

り，終端が低い番地に，先端が高い番地に配置されることが多い。従ってプッシュは，確保されたスタックされたメモリー領域の一番下に連続して配列を伸ばし，そこへデータを書き込むことであり，一方，ポップは一番下のデータを読んでから配列を一つ短くすることになる。

多くの言語では，配列はあらかじめサイズを決めて確保する。このため，スタックのように，全サイズが確定できないものに利用するのは，あまり得策とはいえない。そこで，通常は，**図** 11.6 のように，データをポインタを利用して，一次元的に繋いでいくことにより実現する。データとポインタから成る要素を**ノード** (node) と呼ぶ。スタックでは，各ノードは一つのポインタしか持たず，各ノードをポインタを使って繋いで直線的に並べた**線形リスト** (linear list) として実現できる。また，線形リストの管理用に，線形リストとは独立に先端ノードのアドレスを格納

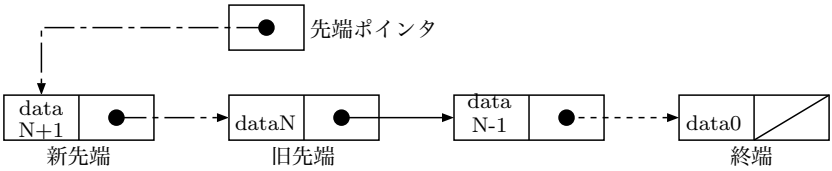


図 11.7　スタックのプシュ

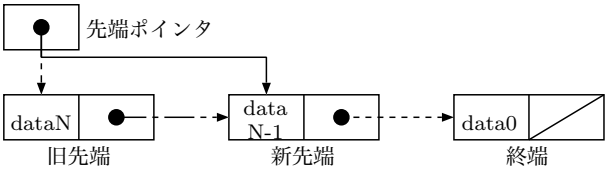


図 11.8　スタックからノードを削除

した**先端ポインタ** (head pointer) が必要である。先端ポインタは**ヘッダ** (header) ともいう。

スタックの各ノードは，データと一つのポインタの組み合わせでできており，先端ノードのポインタは次のノードのアドレスを指している。以後，次のノードのポインタはさらに次のノードのアドレスを指し，最後は終端ノードで終了する。終端ノードのポインタは，`Null`[3] と呼ばれるポインタとしては意味のないあらかじめ定めた値が設定される。図では，ポインタの箱に斜線を描くことで表している。

プッシュの際は，**図** 11.7 に示すように，まず新ノードをメモリー上に確保し，そこに新しいデータと先端ポインタの値をコピーする。さらに，先端ポインタの値を，この新しいノードのアドレスに置き換える。

3）`Null` は `nil` とも言い，実装上は `0x0000 0000` とすることが多い。`0x0000 0000` 番地は，`OS` で使われ，多くの応用プログラムでは使われないため，特殊なポインタとしての意味を持たせることができる。見掛け，特別なノードを指しているようにも見えることから，終端ノードのポインタにはダミーノードを指しているとも表現する。

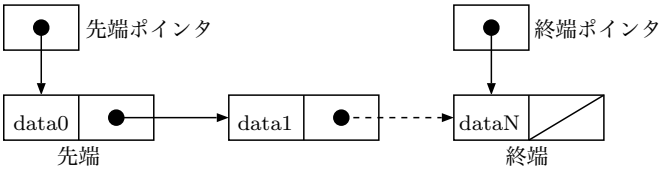


図 11.9　キュー

ポップの際は，**図** 11.8 に示すように，旧先端ノードのデータを読み出し，旧先端ノードのポインタの内容を先端ポインタへコピーした後，旧先端ノードのメモリーを解放する。ポップを繰り返していき，ノードが最後の一つになったとき，さらにポップすると先端ポインタには `Null` が代入される。したがって，線形リストにノードが残っているかどうかは，先端ポインタが `Null` でないかを確認すればよい。

11.3.2　キュー

一列の切符売場の行列のように，最初に入った人が最初に処理されるようにアレンジされたデータの格納法を**キュー** (queue) と呼ぶ。キューとは待ち行列を意味する。最初に入れた情報が最初に出てくることから FIFO (first-in-first-out) と呼ばれることもある。基本的に，行列の一番前のデータを読むことはできるが，行列内や最後のデータを読むことはできない。キューは配列で実現されることは少ない。ほとんどの場合，スタックと同様な**図** 11.9 に示す線形リストで実現される。先に読むことのできる行列の先頭をリストの先端，データを追加できる行列の末尾を終端とする。キューでは，管理用に先端ポインタに加え，最後のノードを指す終端ポインタも利用される。

キューに新しいデータを追加することを**エンキュー** (enqueue) と言い，データは常に終端ノードの後に追加される。まず，新ノードのメモリー

を確保し，そこに新しいデータを書き込む。新ノードのポインタは `Null` としておく。これまでの終端ノードの置かれている場所は終端ポインタからわかるので，旧終端ノードのポインタと，終端ポインタの双方に，新ノードのアドレスを書き込む。この作業により，線形リストには新しいノードが追加される。

データの読み出しは**デキュー** (dequeue) と呼ばれ，先端ノードに対してのみ行われ，通常，読み出しと共に，先端のノードは処理が終了したものとして消失する。つまりスタックのポップと同じ処理が行われる。

11.3.3　両端キュー

スタックとキューは共に線形リストで実現できるため，これを一体化した**両端キュー** (deque) というものもある。これは，線形リストの先端と終端のいずれにも新ノードを加えたり，いずれからもノードを削除できる必要がある。これを行うには，先端ポインタと終端ポインタの双方が必要であることはいうまでもないが，さらに，終端側からも線形リストがたどれるように，各ノードは二つのポインタを持ち，一つは終端側の隣接ノードのアドレス，もう一つは先端側の隣接ノードのアドレスを保持する必要がある。この構造を**双方向連結リスト** (doubly-linked list) と呼ぶ。

線形リストや線形双方向連結リストの終端を先端に繋ぐ，つまり先端ノードのアドレスを終端ノードのポインタに代入すると，環状の連結ノードとなる。見張りが先端ポインタだけでよくなるという利点があるが，リストの尽きた場合の検出がやや複雑になる。これらを，**環状リスト** (circular list) と呼ぶ。

これまでに紹介した線形や環状の構造をまとめて**リスト** (list) という。その属性として，**線形** (linear) か**環状** (circular)，**片方向** (singly-linked)

か**双方向** (doubly-linked) といった区別がある。したがって，スタックやキューは厳密には線形片方向連結リストということになる。

11.4　ツリー

線形リストや環状リスト以上に使われるのが，二次元的にリンクを拡げたツリーである。中でも，二分木と呼ばれるものは，データベースと絡んで積極的に利用される。その他，近い概念であるマルチウェイツリーやドット対についても説明を行う。

11.4.1　二分木

配列，スタック，キューなどはいずれも直線的な長いリストである。これに対し，ノードを二次元的に接続した構造を**バイナリーツリー** (binary tree) もしくは**二分木**（にぶんき） (binary tree) という。さらに，ツリーを作る際，将来の検索が早くなるように意識して作成したものを**バイナリーサーチツリー** (binary search tree) と呼ぶ。ただし，バイナリーツリーもしくは二分木と言えばバイナリーサーチツリーを意味することが多く，以下本書では二分木ということで，バイナリーサーチツリーを論ずる。検索の順番はユーザの希望によりいろいろなつけ方の可能性があるが，まずは数字の大小順とか辞書的並び順とかを意識して説明を読んでほしい。

二分木の基本要素は，一つのキーと二つのポインタからなる**ノード** (node) である。**図** 11.10 に見られる横に三つのメモリーが連続して書かれたものがノードである。すべてのノードはルートポインタの元に構造化されていく。最初の `key0` を収容する際は，スタック領域に三つの連続したメモリーを確保する。図ではこれが横向きに並べて描かれている。この最初に作られるのがルートノードである。実際にポインタ，キー，ポインタと並べるかどうかは，あらかじめ決めておきさえすれば，自由

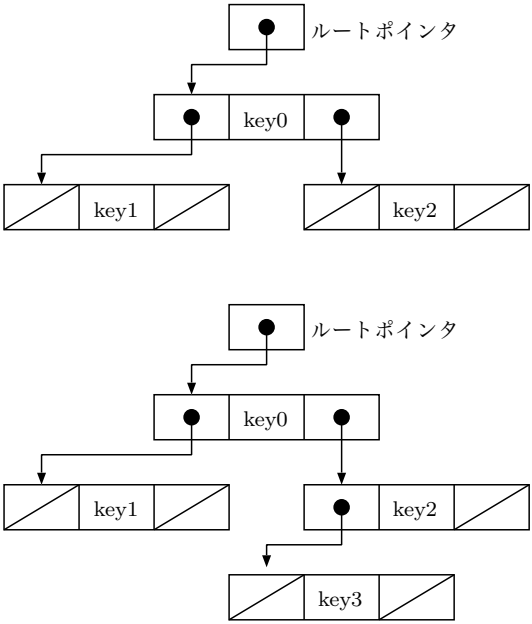


図 11.10　二分木の例（key1<key0<key3<key2 とする）

である。メモリーを確保した段階では，二つのポインタには Null を入れておき，ポインタがまだどこも指していないことを示しておく。さらに，ルートポインタにこのノードのアドレスを代入する。これをしばしば (Null key0 Null)，あるいは Null を省略し (key0) と書く。

二つ目の key1 を格納するときは次のようにする。まず，スタック領域に三つの連続したメモリーを確保し，キーの部分に key1 を格納し，二つのポインタには Null を格納する。もし，key1 が key0 より小さな（順番で前）場合には，この新しいノードのアドレスを key0 のノードの左のポインタに入れる。その様子を図では矢印と黒丸で示した。また括弧を使って記述すると ((Null key1 Null) key0 Null)，あるいは Null を

省略して（(key1) key0）となる。なお，key1 のほうが大きい場合には右に格納することになる。

以下同様に，新しいキーが提示されると，まずノードのメモリーを確保し，そこにキーと Null を代入し，続いてルートポインタから順にキーの大小関係を調べていくことになる。例えば，**図** 11.10 上図のような二分木ができていたとする。括弧式で示すと（(key1) key0 (key2)）。この場合には key1<key0<key2 の順番が成立していることになる。ここに新たに key0 と key2 の間に位置すべき key3 を加えることを考える。ルートポインタからたどり最初の key0 と比較すると，それより大きいので右のブランチ（枝）をたどることになる。続いて次の key2 と比較すると，これよりは小さいので，左をたどることになる。しかし，左には Null ポインタが入っているので，これ以上はたどれない。そこで，key2 の左ポインタへ key3 のノードアドレスを代入することになる。((key1) key0 ((key3) key2)) である。二分木の特長は，キーをソートしながら構造を構成することが可能なことである。後からキーを追加しても，順番がきちんと保たれた構造のキーの集合が得られるのである。

この構造からわかるように，二分木のノード間には親（一つ上のノード），子（自分の直下のノードで 0，1，2 個）といった人間の親族のような関係がある。また，木の構造になぞらえて，木を上下逆にした名称が使われる。一番上，つまり根（ルート）にあるのがルートノードである。また各節（ノード）で最大二つに分かれるが，その分かれた先をブランチ（枝）という。各ブランチごとに，これより先にはたどれないノードをリーフ（葉）という。

図 11.10 の下の図から推察できるように，ルートノードを 1 段目として、3 段目には 4 個の情報を格納することができる。1 個の二分木で 1 個の情報，二分木を一階層増やすごとに格納領域は一段上の 2 倍だけ増え

る。したがって，N 段の二分木で最大 $1+2+2^2+\cdots+2^{N-1}=2^N-1$ 個のキーを格納することができる。

キーの削除はあまり容易ではない。それが葉であれば，そのノードを削除し，親ノードにあるポインタを Null に変更するだけでよい。しかし，葉でない場合，例えば**図** 11.10 下図から key0 を削除することを考えると，ポインタの付け替えが結構大変であることがわかる。このような場合には key0 の部分にキーが無いということを示す（例えば Null を入れる）だけにして，二分木の再構築は行わないという方法もある。また，下の方のノードを持ち上げて，きちんとした二分木にする方法もあるが，詳細は専門書へ譲る。

検索は簡単である。あるキーが存在するかどうかを調べるにはルートポインタの指すアドレスのキーと比較する。一致すれば，そこで検索は停止であるが，大きすぎるのなら右のブランチへ移動する。小さすぎるのなら左のブランチへ移動する。以下，同じ作業を繰り返せばよい。この方法であると，N 段の二分木では，最大 N 回の比較だけで終了することになる。

同じキーの集合でも，キーの提示の仕方により，いくつもの異なる二分木が構成される。キーがたまたま小さいものから順に与えられると，新しいノードは次々にそれぞれの親ノードの右にリンクされていく。このため N 段の二分木となっても，たった N のキーしか収容できないのである。どの階層のノードでも，その下の左右のノード数がほぼ等しいときには**平衡** (balance) のよくとれた二分木，そうでない場合は平衡のとれていない二分木と呼ばれる。平衡のとれていない二分木は，検索にも時間がかかるので，平衡のとれた二分木の方が有利である。キーの与え順で異なる二分木が作成されるということは，不平衡な二分木でもポインタの差し替えにより，検索効率のよい平衡木に変更可能ということにな

る。そのアルゴリズムはやや面倒なので，専門書に譲ることとする。

実装に当たり，すべてのノードはスタック領域に置かれることが多いが，データを減らしていく場合には，いろいろなノードが使われなくなるため，スタック領域は虫食い状態となる。こうした切れ切れのメモリー領域の管理は大変であるため，すべての作業が終わったところあるいは定期的に，ノードが使いまくったすべての領域を一気に整理整頓する。ルートから順にたどっていき，使われていない領域をきちんと把握してから，ノードの配置を変えて，未使用領域を集め，メモリーを解放する。こうした作業を**ガーベッジ コレクション** (garbage collection, GC)[4] と呼ぶ。

二本木をデータベースとして利用する場合には，キー以外に値も収容することを考えなければならない。しかし，それは容易である。**図** 11.10 に示したキーの入っている箱を拡大して，キーと複数の値を一緒に格納すればよい。つまり，キーの箱をキーだけの容量ではなく，値を含めた容量を持つ箱にすればよいのである。もちろん，検索には，そのキー部分だけを利用する。また，値が不定長の場合には，値を別のメモリー部分に確保し，キーの箱には，その不定長の値へのポインタを置けばよい。

二分木と同じような構造ではあるが，二個以上のブランチに分岐するものを**マルチウェイツリー** (multiway tree) という。特に，**四分木** (quadtree) は，二次元の情報，例えば画像処理などでよく使われる。

11.4.2 ドット対

データベースとして積極的に使われるわけではないが，二分木と似た構造を有するドット対という概念をここで紹介しておこう。ドット対は，LISP に代表される関数型プログラム言語のプログラムの記述で積極的に

4) ゴミ収集の意味

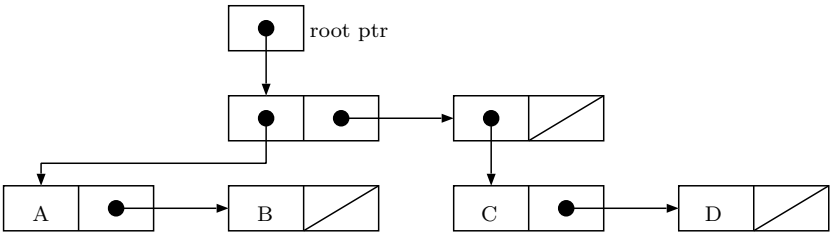


図 11.11　リスト（(A B) (C D)）のドット対での実現

使われる構造である。

図 11.10 に示した各ノードが二つの箱で構成されているような場合のノードを**ドット対**(dot pair) と呼ぶ。二分木は 1 個のキー用の箱と 2 個のポインタ用の 3 個の箱で構成されていたが，ドット対は 2 個の箱で構成される。ただ，この箱にはデータ（この構造では，二分木と異なり検索のキーを入れることが少ないため，データと記す）またはポインタのいずれが入ってもよい。ポインタを入れることにより，一つのルートポインタのもとに，多くのデータを格納することが可能となる。

ドット対が前にデータ，後にポインタという構成の場合には，前節の線形リストの要素であるノードと同じ形をしている。したがって，線形リストと同じ一次元的構造は容易に作ることができる。さらに，データ部分にポインタを入れることも許されているため，線形リストの要素に別の線形リストを入れることも可能である。つまり入れ子のリスト構造が容易に作成できる。これを単に**リスト** (list) と呼ぶ。こうした入れ子構造が作れるのは，各ノードがドット対だからなのである。

LISP はこの一般化されたリストを使ってプログラムを記述する。ドット対をプログラム中にあらわに利用することはほとんどないが, どうしても表現したい場合には次のようにする。例えばドット対の二つの要素として A,

B を持つ場合 (A.B) のようにドットを挟んで記載するのである。したがって，リスト (E F) はドット対で表現すると (E.(F.nil)) となる。ドット対をよく使う LISP では Null の代わりに nil というため，本節では nil とした。また，リスト ((A B) (C D)) は上記 E と F を (A.(B.nil)) と (C.(D.nil)) で置き換えればよいから，((A.(B.nil)).((C.(D.nil)).nil)) となる。後者をノードで記載すると**図** 11.11 のような構造となる。括弧式で書かれたものとよく比較して構造を理解して欲しい。

LISP のプログラムは括弧を多用するというやや特異な構造をしているため，親密感が得られにくいが，人工知能の分野の発展に寄与したという意味で，特記する意味があろう。また，同系の言語として Prolog[5] というものがあるが，これも人工知能，特に連想データベースに寄与した。

LISP では，例えば，3 に 4 を掛けることを (MUL 3 4) などと記載することになる。このように，LISP におけるプログラムは (関数名 要素 ...) という形を何行か続けたものとなっている。また，ADD を加算を意味することとすると，(ADD 2 (MUL 3 4) 1) などと，要素の中に別の関数を入れることもできる。この場合には，内側の括弧は 12 を返し，外側の括弧は 2 と 12 と 1 を加えた 15 を返すことになる。

LISP では，リストの頭に関数名があると，そのリストの残りを引数だと思って処理し，その結果をリストの値として置き換える。したがって，(ADD 2 (MUL 3 4) 1) は (ADD 2 12 1) になり，最終的に 15 が戻ってくるのである。もちろん，リストを単なるデータに見せたり，逆に関数名の入った単なるデータを実行すべき関数のように見せる関数も定義されており，ちょっとした蓄積プログラム方式のような動作も可能である。さらに，新たな関数をより簡単な基本的な関数を用いて定義する方法も

5) Prolog は括弧式のみで完結した言語ではない。

用意されているため，どんな処理も可能なのである。もちろん，本当に基本的な関数は，機械語により書かれていて，それにより実行される。

最後に Perl などのプログラム言語でデータベースを扱う際に使われている**連想記憶** (associative memory) と呼ばれる構造を紹介しておこう。これは，リストの各要素を 2 要素ごとに，前をキー，後ろを値としたものである。例えば，

(りんご 果物 キャンディ 菓子 鯵 魚 米 穀類 梨 果物)

といった構造である。データベースとしても十分利用でき，また，配列としても直接各構成要素にアクセスできるため，便利な構造である。一つのキーに対応する値が複数ある場合，例えば学籍データベースでは，学生番号をキーとして，学生の氏名，性別，...といくつかの属性が対応している。このようなデータベースは，前者の表現では，以下のように，値のほうをリストにすればよい。

(0001（山野太郎 男 ...） 0002（谷川花子 女 ...） ...)

ちなみに，LISP とは異なり，Perl などではこうしたデータベースはハッシュ関数を利用した配列として実装されているため，**ハッシュ** (hash) ともいう。

11.5　関係データベース

現在データベースというと，ほとんどが**関係データベース** (relational database, RDB) である。例えば，学務データベースでいうと，成績データ，科目データ，学生の学籍データ，などの集合からなる。このそれぞれを**テーブル** (table) という。

図 11.12 に示すように，すべてのテーブルには，それぞれのキー（ID ともいう）から始まるデータが並んでいる。成績テーブル（すべての科目のすべての学生の成績を各 1 行ごとに収納したもの）には，成績と共

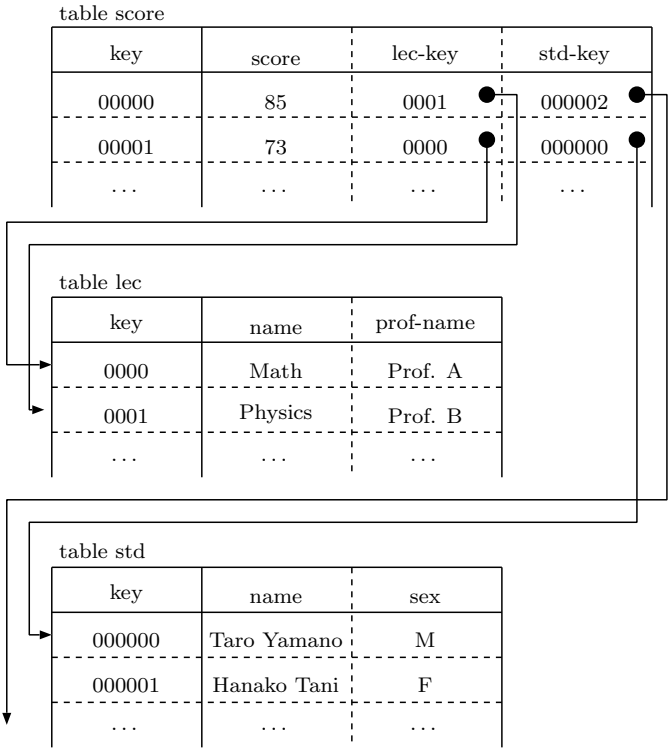


図 11.12　関係データベース

に，どの科目でどの学生の成績であるかが記載されている。その科目の詳細は，科目テーブルに記入されており，成績テーブルにはその科目テーブルとのリンクをとるための科目キーのみが記載される。学生についても同様であり，成績テーブルには，学生の詳細のデータが収納されている学生テーブル上の学生キーが記載される。

このように，キーを利用しながら，いくつかのデータベース（テーブル）間に関係を持たせていくのが関係データベースの基本である。これ

を一つの成績データベースでまとめようとすると，例えば科目の担当教員名を何箇所にも打ち込まねばならないといったような重複打ち込みの無駄が発生するのである。

こういった関係データベースへのアクセスには，現在，SQL と呼ばれる問い合わせ言語が標準となっている。検索だけでなく，関係データベースの構築，テーブルの作成，データの入力など，関係データベースの操作すべてが SQL によって可能である。詳細は Web などで調べてほしいが，例えば，ある特定の教員の採点した結果すべてを見たければ

```
SELECT lec.prof-name, lec.name, score
  FROM score
    INNER JOIN lec ON score.lec-key=lec.key
    INNER JOIN std ON score.std-key=std.key
  WHERE lec.prof-name='Prof. B'
/*
lec.prof-name, lec.name, score を選択表示
  テーブル score から
    score.lec-key を lec.key と同じにして lec と組み合わせて
    score.std-key を std.key と同じにして std と組み合わせて
  lec.prof-name='Prof. B' の条件で
*/
```

という SQL をデータベースに投げれば，答えが返ってくる。何をいっているのかわからないかも知れないが，まず「.」は複数あるテーブルから特定のテーブルを選ぶのに使い，テーブル . 属性（キーも含む）のように記載する。その他の詳細については，下半分に記載したコメントと見比べてほしい。さらに，My SQL や PostgreSQL といった無償のデータベースがあるので，これらで体験することも可能である。

なお，SQL はある程度わかりやすい言語ではあるが，やはりそのつど学習しないと簡単には思い出せない。このため，多くのデータベースを利用した応用プログラムでは，この前に Web アプリケーション（第 13 章で述べる）を置き，そこの窓に情報を打ち込むことにより，ユーザにとって優しいユーザインタフェースを確保したものが多い。

演習問題 11

問題 11.1　次の用語を理解したかどうか確認せよ。

1）ハッシュ関数
2）スタック，キュー
3）線形リスト，ツリー
4）二分木，ドット対
5）関係データベース

問題 11.2　キューを構成する際，一番最初に入ったデータを終端側，また最新データを先端側に配置するような線形リストを使うことは可能かどうか検討せよ。

問題 11.3　両端キューを構成する際，各ノードはなぜ二つのポインタを持つ必要があるかを考察せよ。

問題 11.4　key3 と key2 の間の順番となる key4 のノードを入れてみよう。

問題 11.5　検索可能なデータベースを実現する方法として，ハッシュ関数を利用した配列と二分木の二つについて説明したが，それぞれの得失を述べよ。

問題 11.6　(ADD 2 (MUL 3 4) 1) をドット対で表現するとどのように書けるだろうか。

12 応用ソフトウェア

放送大学 岡部 洋一

《目標&ポイント》ソフトウェアの一例として，ワープロのようなテキストを扱うプログラム，表計算ソフトウェア，プレゼンテーションソフトウェアなどの動作原理，プログラミングを概観する。

《キーワード》応用プログラミング，テキストエディタ，ワードプロセッサー，表計算ソフトウェア，プレゼンテーションソフトウェア

12.1 文字列とエディタ

コンピュータの世界で最初に生まれたデータは，**テキストデータ** (text data) である。まず，ある一定のルールで，**文字コード** (character code) というものが制定された。例えば英数文字の ASCII とか，それからはみ出すかなや漢字などの文字に対応する UTF8 などで，それぞれの文字に対し 2 進数が割り振られる。また，それらをコンピュータの表示装置に示す際の，文字の形も同時に制定された。こうして，しかるべき順に文字コードを並べたデータを用意すると，それを文字の並びとして表示装置に表すことができる。文字の並びを**文字列** (string) という。また文字列の対応する文字コードの 1 次元的並びをテキストデータと呼ぶ。文字コードの中には，空白や改行記号も定義されているから，文章を表示することが可能となる。もちろん，平板な文字の並びであり，題目を大きく見せたりすることはできない。

こうした文字の並び，つまり文字列は，各文字に対応する文字コード，

つまり 2 進数を記憶装置に入れることにより作成可能であるが，人間にとっては極めて使いづらいやり方である。人間にとっては，文字コードではなく，文字の形を見ながらのほうが，ずっと作業効率が高い。したがって，文字コードに対応する文字をディスプレーに表示し，文字レベルで文章を作成したり，編集したりできるとよい。こうした文字レベルで文字列を編集できるソフトウェアを一般に**テキストエディタ** (text editor) と呼ぶ。

12.1.1 ラインエディタ

最初に登場したのは，**ラインエディタ** (line editor) と呼ばれるものである。行単位で編集作業を行う。「1 行目に次の文字列を入れます」，「5 行目を削除します」，「3 行目を削除して次の文字列を入れます」，「7 行目を表示してください」といった命令により，文章を作成，編集していくのである。現在のテキストエディタに比べると極めて使いづらいものであった。

ラインエディタは，ちょっとプログラミングに慣れた人ならば自分でプログラム作成することも可能であろう。英数字の文字コードは 7bit であり，1B=1byte（ 8bit）の整数倍である 1 アドレスのメモリーとの相性がよい。これに対し，日本文字コードは 2〜3B であり，英数字と混ざると，アドレスの計算がやや面倒である。また，日本語の入力にはかな漢字変換なども行わなければならない。そこで英文に限って話をする。

まず，行ごとに対応して連続したメモリーを確保する。また，複数の行を管理するには，各行の先頭アドレスへのポインタを並べた線形リストを利用する。編集の際には行を前進させるだけでなく戻ることもあるので，双方向連結リストである両端キューを利用する。各行の文字数を制御するために，各ノードに文字数データを加えるか，各行の文字列の

最後に NULL を置くのがよい。

編集対象行を指定するには，線形リストを前から順に辿っていく。その行を削除するには，その対象ノードをジャンプするように線形リストのポインタを付け直し，対象行および対象ノードのメモリーを解放する。逆に行を追加するには，新しい行をメモリー上に確保し，その行を指すポインタを格納したノードを作成し，そのノードを線形リストへ挿入する。行の訂正は，行の削除と挿入で対応する。あとは，メモリー上の行の情報をディスプレーに表示する機能であるが，その際，文字の形（フォント）を作って，表示装置上の所定の場所へ表示する必要がある。それは OS が対応してくれる。

これ以外に，ファイルへのセーブ機能や，ファイルからの読み込み機能が必要である。この際，ファイル上のテキストデータは行と行の間が改行記号になっているのに対し，メモリー上では行単位となっているため，簡単な変換作業が必要である。

12.1.2　WYSIWYG

ラインエディタはあくまでも行単位の削除，入力，出力しかできない。現在のワープロのように作成した文章が常に見えている状況で，任意の場所で任意の数の文字数を削除，入力できる機能がほしくなるのが，当然の方向であろう。そこで，この機能を実現したのが，現在主流になっている WYSIWYG (what you see is what you get) というニックネームを持った**テキストエディタ** (text editor) なのである。文字通り「見えているものがそのまま手に入る」というエディタであり，現在**テキストエディタ** (text editor) といえばこれを指すといってよい。

まず，行内のコラム単位で訂正できる機能をどう実現するかであるが，対象行だけ線形リストに入れ直して作業する方法もあるが，それほど大

容量でもないので配列とし，文字削除や挿入に対しては配列内コピーで対応することも可能である。

文章を表示する際，文章の位置は基本的に変らず，カーソルがいろいろな位置に移動することで作業が進行していくエディタモードと呼ばれる形式がとられる。また，編集が終了し，テキストデータの入ったファイルが完成すると，再びコマンドモードに戻れるようなしかけも必要となる。

エディタモードでは，メモリー上の指定した行から，テキストデータを画面全体に表示する機能が必要であるが，これは現在の OS のでは一つの機能として用意されている。また，編集者の着目している場所を示すカーソルを，テキストの任意の位置に移動する機構も用意されている。具体的には，指定されたメモリーの内容に対応したフォントを VRAM にコピーしていくことになる。

カーソルの移動については，編集者の意志によって自由な位置に移動できることが必要である。しばしば利用されるのはキーボードの矢印キーや，Ctrl と通常のキーのいずれかとを組み合わせたもの，また最近はマウスによって移動可能となっている。したがって，マウスの現在位置を知る方法も OS に依存することになる。逆にエディタ側からマウスを動かす機能も必要となる。

ファイルとのデータの移動はラインエディタと同様である。テキストエディタになると，印刷機能も用意されていることが多いが，それについては割愛する。

12.2　ワープロ

WYSIWYG でも，人間にとってかなりの親和性のある文章が作成可能であるが，文字修飾や書籍のような章構えやページ配分といった組版などのデザインはできない。こういったデザインもできるようにしたものが，p.97 で述べた現在の**ワードプロセッサ** (word processor) である。

デザインの入った文書の保存にはいろいろな方法があるが，大きく分けてワープロソフトウェアに大きく依存するバイナリー形式と，デザイン情報をタグと呼ばれる特別な文字列を挿入することにより残す方法である。もちろん，タグは一種類ではなく，デザインの種類だけある。前者の代表は，やや古い MS-Word である。後者の代表は，TEX や新しい MS-Word[1] である。

いずれも，テキストを画面に表示する際，デザイン情報を見ながら必要な形や位置に出力することになる。印刷機へ出力する際も同様である。したがって，画面と印刷出力の間には，デザイン情報の解釈のずれにより，若干のずれが生じることがある。

ここで，**文字コード** (character code) について，改めて説明しよう。かつて，コンピュータにより表示できる文字は英語のアルファベットと若干の記号だけであった。現在はあらゆる国のあらゆる文字が表示できるといっても過言ではない。アルファベットと必要最小限の記号を表示するだけで，7bit（アルファベット大小文字で 26×2，数字 10，その他数文字として，2^6=64 を越える）を必要とする。米国規格の ASCII はこれに対応する。さらに，7bit×2 の 14bit を使って当初，漢字 6 千字余を収納したのが JIS コードある。なお，14bit では 16,384（$=2^{14}$）個の文字が収容できる。JIS コード表は ASCII 表と重なるため，表を切り換え

1）Word2007 以後では，後に p.157 で説明する XML という方式を利用している。

OS	文字コード	改行コード
Windows	SJIS, UTF-8	CR+LF
Mac	SJIS, UTF-8	CR
Unix	EUC, UTF-8	LF

図 12.1　OS と日本語の文字コード

るために，ASCII 表のアルファベットなどに使われていない 2 文字分を使って JIS コード表へ Shift-In で入ったり，Shift-Out で出たりする必要がある。

シフトを避けるために，文字コードを 2byte（=8bit）とし，最初の 1byte は ASCII で使われていないコードを使うようにしたものが，SJIS（Shift-JIS）コードである。このコードは永らく，日本語対応の Windows や Mac で使われてきたものである。また，同様な 2byte を使うが，別の割り当てをした**日本語** EUC (Japanese EUC) というコードもある。これは同じコードを別の言語に割り当てることにより，国際的にも利用できるもので，Unix で使われてきた。

当初，各国ごとにそれぞれ独自の文字コードの割り当てを行ってきたが，現在は，世界にあるすべての文字を入れたコード表が作られるようになり，その一つが UTF-8 である。これにより，日本語の中にアラビア語を入れるなどの文書も可能となってきている。

改めて**図** 12.1 に OS と利用されてきた文字コードを示す。この他，IBM の OS で使われた EBCDIC など，いくつかの種類がある。いずれも「ひらがな」,「カタカナ」,「漢字」を中心とする文字一つ一つに対応するコードを制定する必要性がある。幸いにして，UTF 系を除いて文字の割り当ての順番は大きく変わっておらず，各文字種ごとに，先頭のコードがず

れているだけであるため，コード間の変換は容易である。ただ，いずれのコードを利用しているかをしっかりと理解していないといわゆる「文字化け」が発生する。

12.3　表計算ソフトウェア

会計処理などで行われる合計などの計算を，プログラムを書くことなく，より直感的に簡便に行おうということで，数値を表形式に並べておき，それを通常の表計算のように縦横に合計したりする目的で作成されたのが**表計算ソフトウェア** (calculation software) である。これが徐々に進化し，平均値や標準偏差の計算や各種の関数計算，さらに条件式など，ほとんどあらゆる計算ができるようになった。さらに，マクロと呼ばれる簡易プログラムを起動できるようになり，純粋のプログラムとの差はほとんど無くなりつつある。しかし，利便性と引き換えに計算速度はかなり劣る。

表を構成する各セルには数値や文字列といったデータ，または「どこからどこまでの和をとる」といった式（命令）が記載されている。通常，頭に特殊な記号のついたものが命令である。これらは巨大な仮想的な 2 次元配列に記載される。例えば，あるセル（表を構成する各箱）に「どこからどこまでの合計をとる」といった式が入っていると，それを解釈して，必要なセルの合計を計算し，そのセルにはその合計値を表示する。セルをクリックすると，その内容である元のデータや式が表示されるので，その表示窓で値や式を変更することが可能である。したがって，表示用の値とは独立に，元の式があれば，それも記憶しておく必要がある。その際のメモリーの確保のしかたにはいろいろあるので，考えてみてほしい。

以下，どんどんユーザの要望が高くなり，現在のように，覚えきれないほどの式や機能が搭載されるようになったのである。

12.4 プレゼンテーションソフトウェア

プレゼンテーションソフトウェア (presentation software) の実装は，ワードプロセッサと基本的にはほとんど変わらない。題目，大項目，中項目，小項目などからなるテキストデータにそれぞれのタグを付けたものをデータとする。それを，編集中から最終画面に表示されるような形で表示する機能があれば，ほぼ完成である。あとは，背景の設定と，題目などをどの位置に出すかといったレイアウトの設定さえできれば完成である。

12.5 種々の応用ソフトウェア

世の中には，種々の応用ソフトウェアが満ち溢れているが，いずれもユーザと，何らかのコミュニケーションが必要である。そのほとんどがキーボード，マウス，ディスプレーそれもウィンドウシステムを前提としている。特にウィンドウシステムとなると，出力はきちんとウィンドウ内に出さなければならない。そこで，こうした複雑な作業は OS の拡張された機能である GUI ソフトウェアにゆだねることとなる。大部分の応用ソフトウェアは，GUI に装備された OS サブルーチンを呼び出し，キーボードやマウスからデータを入力し，それを処理し，現在利用中のウィンドウに対し，データの出力を行うことで実現されているものが多いが，入出力が OS に任せることが可能であるため，中心の処理部に傾注して開発すればよい。しかし，それでも入出力部分のサブルーチンの呼び出しの記述は，相当の負担となる。

演習問題 12

問題 12.1　文書処理ソフトウェア，表計算ソフトウェア，プレゼンテーションソフトウェアの処理をあらましを理解したかどうか確認せよ。

問題 12.2　全角の「あ」が UTF-8，SJIS，EUC，JIS ではどんなコード（16 進数）となっているかを，Web などを使って調べてみよう。

問題 12.3　ラインエディタを作成することを想定し，メモリー管理をどのようにしたらよいかを考えてみよう。ラインエディタでは行単位のメモリー確保がよいと思われる。しかも，行が入ったり，行が消滅したりする。行の一部を変更する場合には，その行を消し，修正後の行を新しい行として追加すればよい。こうした，出入りの多いメモリー管理には，二分木の key の代わりに，行番号と自由な長さを有する文字列へのポインタを使ったものを利用するとよいであろう。

13 Web

放送大学 岡部 洋一

《目標＆ポイント》世界中に存在するデータへのアクセスを容易にした Web のしくみについて概観する。また，近年，多くのプログラムが Web 上で実施できるようになったが，そのしくみについても解説する。

《キーワード》Web，ハイパーテキスト，HTML，Web アプリケーション，XML

13.1 Web の概念とそれを支える技術

かつて，インタネットの主な役割は，電子メール，電子メールを基本とした掲示板などであった。それがインタネットの能力が上がるにつれ，研究データなどを中心にしたファイルの移動などにも使われるようになってきた。ファイルの移動といっても，特定個人から特定個人への移動である。やがて，それが同じ研究グループへの複数の研究者間の移動になり，さらに，一般公開へ発展していくことになる。受け渡したいファイルは，当初はテキストであったが，そのうち写真などの画像ファイルも対象となる。さらに，画像の説明を画像そのものと連続的な概念で送付できないかということになる。こうして世界中のファイルへのアクセスを可能とし，かつそのファイルの内容を気軽に見えるようにしたものが，WWW (world wide web) である。そのまま翻訳すれば世界に拡がった蜘蛛の巣であり，単に Web とも略される。

Web を支える技術として，組版プログラムとでもいうべきものがある。組版プログラムでは章立てや文字の修飾といったことが必要であるが，そ

の方法としてタグを付けておき，文書表示ソフトウェア側ではそのタグを利用して，美しい印刷のような形のイメージを作るという**タグ** (tag) による制御方式がある。科学者の世界では，TEX（テフ）が良く知られているが，出版業界では SGML (standard generalized markup language) と呼ばれる形式[1)]が注目された。これは，ある文字列が表題ならば<title>...</title>のように記載する手法である。つまりかぎ括弧で包まれた部分がタグであり，タグとタグで囲まれた部分を修飾しているのである。表示ソフトウェアでは，このタグを発見すると，表題のような体裁で画面上に表示するのである。タグも文字列であり，ワープロ文書全体も文字列，つまりテキストデータであるため，印刷イメージより格段に，保存，修正などの保守，移動に優れているのが特長である。SGML は米軍が規定集を編纂する際，利用したこともあり，急速に拡がった。この技術は現在，電子ブックで使われている EPUB (electronic publication) に発展してきている。

Web ではこの形式に似せた HTML (hypertext markup language) と呼ばれるものが採用された。これは，SGML に極めて近いが，それにすぐ後に述べるハイパーテキストという概念を追加したものである。この方式により，遠距離でも比較的美しいワープロ文書が転送できるようになったのである。**ハイパーテキスト** (hypertext) とは，テキストを越える，つまり他の文書をも参照できる機能である。Web で優れているのは，これが外部の第三者のサーバに置かれた文書をも参照できることで，その文書に関する単語なり文章をクリックすると，その文書の置かれたサーバから，当該文書を持ってきて表示するという機能である。なお，他の文書（自分の文書も含め）への繋がりを**リンク** (link) という。したがって，他の文書へのリンクとか，他の文書へリンクを張るといった使い方がされる。

1）markup とは英語では組版指定を指す。

Web の書き方について，若干説明をしておこう。英語以外の言語で表示する際は，Web ブラウザにどのコードを使用しているかを伝える必要がある。<meta .../>なるタグを使い，charset として Shift_JIS（SJIS），UTF-8（UTF8），EUC-JP（日本語 EUC），ISO-2022-JP（JIS）などを指定すればよい。

次に主なタグを示す。これらタグの詳細，あるいはこれ以外の多数のタグについては，「html タグ」などで Web 検索してみよう。

構造:

```
<html>
  <head>ヘッダ</head>
  <body>本文</body>
</html>
```

ヘッダ:

```
<meta .../>
<title>題目</title>
```

組版:

```
<h1>章</h1>, ..., <h6>小々節</h6>,
<div align=left (または center, right)>文</div>
<p>段落</p>
改行<br/>
<pre>そのまま</pre>
<blockquote>引用文</blockquote>
横線<hr/>
```

文字飾り:

```
<font (name=... size=... color=...)>フォント</font>
<i>イタ</i>
```

```
<b>ボールド</b>
<u>下線</u>
<strike>打消</strike>,
<tt>等幅</tt>
<sub>下付</sub>
<sup>上付</sup>
```

箇条書き:

順番なしリスト:

```
<ul>
  <li>項</li>
  ...
</ul>
```

順番付きリスト:

```
<ol>
  <li>項</li>
  ...
</ol>
```

説明付きリスト:

```
<dl>
  <dt>定義語</dt><dd>内容</dd>
  ...
</dl>
```

表:

```
<table (align=... (border=...))>
  <tr><th>...</th><th>...</th>...</tr>
  ...
```

(<caption (align=...)>...</caption>)

</table>

リンク:

説明

URL リンク： "http://www.moge.org/okabe/index.html"の形

同じサーバ内の絶対リンク： "/okabe/index.html"の形

同じサーバ内の相対リンク： "example/html/tag.html"の形

メール先： "mailto: mail アドレス?subject=..."

同一ページ内のアンカへのリンク： "#anchor"の形

アンカ： <a name=anchor/>の形

画像:

<img src="リンク"
(alt=... width=... height=... align=...)/>

フォーム:

後述

特殊文字:

‘<’: &lt, ‘>’: &gt, ‘&’: &amp, ‘"’: &quot, ‘©’: &copy

ファイルの置かれているところを一意に示す住所に対応するものがURL (uniform resource locator) と呼ばれるものである。http://から始まり，その後ろにサーバ名，以後スラッシュで区切って，各サーバの指定された場所からディレクトリーをたどっていって，ファイルにたどり着くまでのパスを記載したものである。

これらの機能により，ユーザは世界中の情報に容易にアクセスできるようになり，また，その内容を簡単に見ることができるようになったのである。ちなみに，タグ付きテキストを組版されたような形で見るためには，専用の Web **ブラウザ** (Web browser) と呼ばれるソフトウェアが必要

である。古くは Mozila，Netscape などが使われたが，現在は Explorer，Safari，Chrome，Firefox などが使われている。

13.2　検索エンジン

昔は，いろいろな Web サイトを見るには，それぞれの URL を互いに知らせあって，そのサイトを見にいくという方法しかなかった。やがて，組織単位に，組織内へのリンク先をハイパーテキストとしてまとめたものが作成されるようになった。このページのことを**トップページ** (top page) や**ホームページ** (home page)[2] などと呼ぶ。やがて，ある概念に関連するサイトへのリンク先をまとめたページが作られるようになった。例えば，ある特定の研究領域に関するページを探し出して，そこへのリンク先をまとめたページなどである。当初は，メールによる情報交換などにより集めた情報を手動で Web にまとめて作成された。

やがて，これを手動ではなく，自動的に行うようになった。もちろん，検索範囲は広いほうがよく，可能な限り拡がっていき，最後には全世界のすべての公開されたページを探すようになっていったのである。検索システムは，世界中の公開された Web サイトから情報を収集する必要があるため，現在は人手にはよらず，各サイトを自動的に見にいくようなロボット[3] と呼ばれるしかけにより達成されている。これを行うサーバやそこに搭載されている検索ソフトウェアを，**検索エンジン** (search engine) と呼ぶ。

これを特定の単語だけではなく，あらゆる単語や文章に対して対応で

2) 日本語では Web ページのことをホームページと呼ぶが，それは誤用であり，ホームページは Web ページの中で中核となる特別のページを指す用語である。
3) ロボットといっても，何かが各々のサイトへ行くわけではなく，ページを見に行くだけである。

きるようにしたシステムが，現在広く利用されている Google や Yahoo などの検索エンジンである。実際には全世界の各 Web サーバに置かれた文書を分析して単語に分け，その単語の一覧を作成し，その単語にそれぞれの単語の置かれているリンク先である URL を付けたものである。興味のある単語をクリックすると，その単語の含まれているファイルへ飛んでいくことになる。実際には単語だけではわかりづらいため，前後の文章も表示されるようにしてあったり，単語を組み合わせた文章でも検索可能にしたものが多い。そのため，世界中から公開された情報を収集している。

このように獲得された URL のリストをどのような順番で提示するかによって，検索ページの便利さが大きく変わる。Yahoo は長い間，主として人手によって順番付けしてきた。これに対し，Google は，対象としている URL を参照しているリンク元の数の多い順に並べるという自動的な方法でページを作成してきている。つまり，多数の人が見ているページを優先するという方法である。誰でも Web ページ作成が容易になった現在，検索範囲は大幅に拡がり，自動的な検索エンジンの優位性が確実になり，現在は有名な検索エンジンはほとんどすべて自動化されている。こうしたメガデータとも呼ばれる大量の情報を利用すると，ビジネスも含め，場合によっては政治にも利用できるなど，いろいろな可能性がでてくる。よく「情報を制するものは世界を制す」などといわれるが，軍事目的などには利用されたくないものである。

13.3 Web アプリケーション

Web がこれほど普及したのは，ブラウザが比較的軽いソフトウェアであり，それ故，無償で提供されたものが多かったからである。それに加

え，Web が単なるファイル転送で留まらず，プログラムを実行する機能を有していたからである。Web ブラウザで，URL にしたがってサイトを見にいくと，そこから基本的にはタグを含めた文字列が送られてきて，その文字列にしたがって修飾された組版されたページが表示される。先方のサイトが実在するファイルの文字列を送ろうが，そのサイトに置かれたプログラムの掃き出す文字列を送ろうが，手前のブラウザはどちらも同じように処理できるはずである。もちろん，手前のブラウザは，相手がファイルなのかプログラムなのかを知ることはできるが，ブラウザの行う作業はほとんど同じなので，このような機能は容易に実現できることは理解できよう。

Web で実行されるプログラムを Web **アプリケーション** (Web application) と呼ぶ。アクセスするつど，「あなたは...番目の訪問者です」とか，訪問時刻が表示されるサイトなどがあるが，これらが固定テキストの送付だけでは実現できない機能であることは容易に想像できよう。Web アプリケーションは，原理的にはどんな言語でも作成できるが，現在の主流はテキスト処理に強い Java，PHP，Perl，Python，Ruby などである。

アンケート調査や，お金の払い込みのような，こちらから情報を送ると，その内容に応じた返事の得られる Web ページがある。これには，Web への入力を受け入れてプログラムへ転送するしかけが必要となってくる。このしかけは form と呼ばれるタグで実現できる。この**フォーム** (form) 内に，テキスト，選択結果などを読み込むタグを用意すると，例えばテキストを書き込むための窓などが用意される。そこへ書き込んだデータは，このページから Web アプリケーションプログラムへ送付されるのである。Web アプリケーションプログラムでは，これらのデータを利用して，HTML と同じ形式の文字列を作成し，出力するのである。この結果，利用者側のブラウザは，あたかも通常のテキストファイルと同じようなワー

プロ文書を表示することができることとなる。参考のため，form の書き方を記載しておこう。例のごとく，詳細については，必要に応じ「html form」などで検索してみよう。

構文:

```
<form action="URL" method=post (or get)>...</form>
```

input:

```
<input name=... value=... type=text/>
```

type には他に password，check，radio，hidden，submit，reset などがある。

選択:

```
<select name=... (multiple)>
  <option value=... (selected)>...</option>
  ...
</select>
```

長文:

```
<textarea name=... rows=... cols=...>...</textarea>
```

Web アプリケーションはしばしばデータベースと組み合わせて利用される。例えば，フライト座席予約システムでは，各フライトごとの情報以外に座席の空き状況をデータベースとして保有している。ユーザが，座席予約システムのページからフォームを経由して，ある特定な日の特定なフライトの空き状況を問い合わせると，その情報は Web アプリケーションプログラムへ送られる。Web アプリケーションプログラムでは，その情報を利用して，空席データベースへ問い合わせを行い，空いているかどうかをチェックする。その結果を利用して HTML 形式のテキストに仕上げ，あたかも Web ページの一つのような形で出力するのである。そしてユーザは空席情報を得るというしかけになっている。

同様なデータベースを利用した Web アプリケーションは経理，人事，学務といった事務組織のあらゆる分野で利用されている。昔は，こういうシステムはそれぞれの対応ソフトウェアで運用されてきた。それでもよいのではないか，なぜ Web なのだという疑問が湧くかと思う。Web は，ユーザ側にそれぞれ別々のソフトウェアを用意しなくてよいというのが利点である。先方から送られてきたタグ付き文字列を解釈して組版する機能だけを有すればよいので，ある意味単一機能なのである。つまり，ユーザ側の負担を軽くし，細い作業はすべてサーバ側に置いたシステムが行う。その結果，ユーザ側の端末ごとに，それぞれ利用分野により異なるソフトウェアをインストールする必要もない。ただ一つ，Web ブラウザをインストールさえしておけば，非常に広い利用分野に簡単に対応できるのである。

一方，開発側にも大きなメリットがある。開発側はユーザインタフェースにあまり気を使わなくてもよいのである。Web を使わない場合，入出力は OS の GUI に任せることができるが，それでも GUI の OS サブルーチンの呼び出しはかなりの負担となる。それをほとんどテキストの入出力だけで処理できるのである。これらの要因が，Web を中心とするソフトウェアが急速に拡がった最大の理由である。

13.4　XML

XML (extensible markup language) は Web とは直接関係しない概念であるが，SGML や HTML の進化したものなので本章で説明する。SGML や HTML では利用できるタグに制限があり，それぞれ，送り側と受取り側には，このタグはこういう意味といった約束があった。しかし，タグを自由に設定できないかという希望もある。こうした要望に応え，自由に設定した

タグの使える XML という形式が作られた。自由といっても，タグは必ずかぎ括弧で包むことになっている。一方，勝手にタグが定義されるため，送り側は，それぞれのタグが何を意味するかを記載したファイルを同時に送付する必要がある。タグ付きテキストとタグの定義ファイルが組になって初めて意味をなすのである。

例えば，<title>...</title>は SGML や HTML では表題を表すと定義されている。しかし，XML ではこれを<題目>...</題目>と記載することも許されている。その代わり，<題目>や</題目>がどのような意味を持つかをブラウザに伝える別のファイルを用意する必要がある。ブラウザはそれを読んで，XML ファイルの内容を理解して表示することになる。

演習問題 13

問題 13.1 Web アプリケーションとはどのような概念か確認せよ。

問題 13.2 Web で HTML の書き方を調べ，次のような出力を行う Web ページを作成せよ。

表題 Greeting

章題 Welcome

本文 Hello!

なお，表題タグ名は本章で説明したように title であるが，章題，本文のタグ名は h1，p である。

14 ソフトウェア工学

放送大学 岡部 洋一

《目標&ポイント》巨大なソフトウェアの開発に当たっては，一人で書くソフトウェアとは異なる智慧が必要である。どのように連携をとるかなどについて解説する。

《キーワード》ソフトウェア工学，構造化分析，オブジェクト指向分析，設計，テスト，バグ，プロジェクトマネジメント

14.1 ソフトウェア開発

かつて，ソフトウェアの開発は一人でなされた。一人がすべてのプログラムを書き上げるのが普通であった。しかし，ソフトウェアの重要性は上がる一方であり，現在，ソフトウェアを利用する機器では，その大部分の経費がソフトウェアの開発経費となっている。プログラムはどんどん機能を上げ，丁寧なユーザインタフェースが要求され，複雑なデータを扱うようになり，ということで，どんどん巨大になってきているのである。作るべきソフトウェアの種類も本数も膨大となってきており，昔は一人でせいぜい数十時間で開発できたソフトウェアも数千時間かかるのも珍しくなくなり，それに合わせ，多人数で開発されるようになった。現在の OS の開発などでは数百人が絡むことも珍しくない。

一人の人間ならば，そのプログラム全体の構成に関する考え方も統一されており，トラブルが発生したときのデバッグの速度も比較的速く，いわば一人のピアニストが弾く曲のようなものである。ところが，多人

数が関わるようになると，その意思の疎通が難しくなる。また，ソフトウェアは，現在，金融，軍事，巨大プロジェクトなどにも深く浸透しており，バグが巨大な損失に繋がる場合も珍しくなくなっている。例えば，銀行システムのトラブル，宇宙開発でのトラブルのかなりの部分がソフトウェアのバグに起因している。

このように巨大化した商品を作成するには，それなりの知識や経験が必要となる。このため，ソフトウェアを製品として作る技術，つまり**ソフトウェア工学** (software engineering) が発展した。しかし，ハードウェア製作のような産業革命の時代から築かれた技術とは異なり，まだ歴史は浅い。ハードウェア製作技術から学ぶことは多くても，ソフトウェアに固有な問題も多々あり，まさに発展途上の技術なのである。

ハードウェア製品の開発と同様，ソフトウェア製品の開発もいくつかのプロセスに分離することができる。どんなソフトウェア会社でも最低，システム提案 → 受注 → 外部設計 → 内部設計 → プログラミング → テスト → 運用保守といったプロセスの流れが存在する。外部設計は仕様決定，また，内部設計は製品企画とも呼ばれる。いずれも設計であるが，相手が外部の顧客であるか内部のプログラマであるかの差である。最後のプロセスを運用保守といったが，ハードウェア製品では通常出荷とか製品化と呼ばれている。しかし，ソフトウェアの場合には，出荷後でもバグ発生や仕様変更が限りなく発生するため，あえて運用保守という言葉を用いた。この流れを**落水型** (waterfall model) という。上流から順番に固めていこうという方式である。また，各プロセスを日程に合わせてバー状にした図を**ガントチャート** (Gantt chart) という。

まずシステム提案であるが，顧客に対し，提案するシステムの概要を示すことが最大の眼目となる。新しいシステムを導入すると，顧客の今までやっていた仕事のどこがどのように変わるのか，メリットは何かを示

す。また，開発期間の提示も重要である。一番の強調すべき点は，サービスをあまり落とすことなく，むしろ上げながら，どのくらいの経費削減が可能かを示すのである。例えば，ユーザがあるソフトウェアを導入した際，年間 10 人の人員を減らすことが可能だったとしよう。一方，ソフトウェアには寿命がある。OS が対応しなくなったり，より効率の高い技術が導入されるかも知れず，ソフトウェアの価値は年々相対的に下がっていく。最大でも 5 年が使用限度であろう。一人当たりの人件費がおよそ 1 千万円として，ソフトウェアの導入により減らすことのできる人件費はおよそ 5 億円となる。これより高いソフトウェアは導入しても意味がないことになる。

顧客が満足すると，受注ということになる。それに合わせ開発計画を作成する。開発のプロセス予定をガントチャートなどで決定する。経費問題は開発側から見ても極めて重要である。5 億円で運用も含め作成しなければならないことになる。開発期間を半年として，10 億/年の経費であるので，人件費を同じく 1 千万円として，総務，営業，設計，テスト，運用保守も含め，100 名で当たらなければならない。これ以上のリソースが必要な場合には，開発できないことになる。必要なコード数，人員の大まかな割り当てや体制などを決定する。なお，プログラマはソースコード 1 行当たりおよそ 500 円が必要といわれていることも知っておくとよいであろう。年におよそ 250 日働くから，日給 4 万円なので，1 日 80 行，1 時間 10 行ということになる。一見，楽な仕事に見えるが，単純作業ではなく知的な作業なので，プログラムの構想を練ることも，打ち合わせや会議の時間も含め，さらに，プログラム終了後の文書書き，自分で行うテストも含んだ行数ということである。

以後，外部設計，内部設計，プログラミング，テスト，運用保守と続いていくことなる。

ソフトウェア製品には出荷後もバグという概念が付きまとう。これがハードウェア製品とは大きく異なる点である。ハードウェア製品でもリコールに見られるように，出荷後の製品トラブルがないわけではないが，その頻度がかなり高いのである。ソフトウェア製品の場合，製品開発中にも無視できない大きな要素となっている。バグをいかに最小限に減らすか，バグが発見されたとき，いかに素早く修正できるのか，開発途中の仕様変更にいかに素早く対処できるのか，また，変更記録をどうやって残すのかといった点が，ハードウェア開発に比べ，大きく異なる点である。

これには，ソフトウェアが一種のサービス産業に近いこともあろう。ハードウェア製品を使うユーザは，よほどのことがない限り，製品の変更を考えない。多少の使いづらさがあっても，諦めて受け入れるか，他の製品を買換えるということで対応する。しかし，ソフトウェアの場合には，使いやすさというのがその製品の決定的価値を決める。このため，仕様変更も起こりやすいのである。また，ハードウェア製品のようにマスプロではないことも大きな差を生んでいる。OS や事務系ソフトウェアのように，大量に出回るソフトウェアがあるではないかと思うかも知れないが，あれはコピーで大量化できるのであり，作成という意味では一品生産なのである。つまり，ソフトウェア製作は，一人の行う作業の単純作業の繰り返しではなく，一回限りの複雑作業であるため，バグの発生確率が高く，かつ，発見が困難という短所に繋がっているのである。したがって，製品化してから運用へ入ってからの保守も必要である。例えば，製品開発が終了して運用のフェーズになってからトラブルが発生したが，開発者はすでに入れ替わり，誰もプログラムのソースコードが読めないので対応ができないなどということは，本来あってはならないのである。なお，落水型は徐々に修正が加わってはいるが，やはり設計

の基本的考え方であることは変わっていない。それを前提とすると，ソフトウェア開発といえども最終的には，情報伝達，意志決定，修正確認，テスティング，文書化といったメカニズムが最重要となってくることがわかる。

落水型は，当初は，各プロセス間のフィードバックの少ない手法と見られ，そのため提案されてからいくつかの対立的プロセスの流し方が提案されてきた。例えば，落水型を何度か繰り返す**スパイラルモデル** (spiral model) などである。しかし，これはひたすら作業を直線的に進めるのではなく，下から上へのフィードバックやプロセスの戻りも必要なことを言い替えているとも理解できる。その意味で，プロセスは落水型を基本とし，必要に応じ，臆することなく上部へフィードバックをかけ，見直しをすればよいのである。

14.2　外部設計

外部設計とは，顧客側の希望を聞いて開発ができるようにするプロセス，つまり顧客側と開発側とのマッチングをとりながら仕様決定する作業のことである。したがって，顧客側から何をしたいかという目的を 聞くだけでなく，ユーザインタフェースをどうしたらよいか，導入，運用保守体勢についても相談を行う。一方，開発側から見ると，どういったシステムを作るか，さらにどういったサブシステムを作るか，サブシステム間のインタフェースはどうするのか，必要なデータベースのテーブルなどを決定できるだけのかなり具体的な情報を入手できないといけない。このため，何週間かの双方のやりとりが必要となる。もちろん，内部設計ができないような仕様を作ってはいけない。設計側がどのような技術を持っているかにより，外部設計のしかたは微妙に変わるのである。

ほとんどのシステムがデータベースを中心とし，それを窓やWeb経由で扱うものが多いので，本章ではある程度，それを前提として記述する。その例として，学生の学籍簿管理，科目管理，成績管理などを行う学務システムを考えよう。

ユーザインタフェース仕様では，顧客側でこのシステムを使うユーザがどのような画面を見ることになるのかを図で示す。例えば，学生の見られる画面をどうするか，そこにどのようなボタンを配置するか，さらに例えば登録科目一覧のボタンを押すと，どんな画面が表示されるかなどの図である。

サブシステム仕様とは，画面のどのボタンが具体的に何をするのかを，多くの場合，表で示す。登録科目一覧とは何をするボタンであるかなどを，各表示画面ごとにきちんと記述する。

データテーブル仕様とは，このシステムにとってどんな関係データベースが必要かを，大まかに示す。例えば，学生個人ごとの情報の入ったテーブル，科目情報の入ったテーブル，成績情報の入ったテーブル，場合によると教員情報の入ったテーブルも必要であろう。

ネットワーク仕様とは，学生の使える端末，教員の使える端末，職員の使える端末があるが，これらとシステムサーバとがスター状に並んでいくことなどを示す図である。例えば，学生端末とサーバとの間に，学生端末を管理するコンピュータが入るとすると，そういったものも図中に記載される。

移行・導入仕様とは，移行（旧システムがある場合）や導入に，どの場所でどのくらいの日数が必要かを記載する。端末室が使えなくなる期間，サーバ室のどのくらいがどのくらいの期間使えなくなるかなどを記載する。

運用・保守とは，バグが出たときに，どのような体制で対応するのか，

保守要員を置くのか，置かないとするとどうするのかなどの取り決めを記載する。

14.3　内部設計

外部設計により，粗い設計が達成されると，次は内部設計である。内部設計から顧客は関与しなくなる。内部設計では，分析と設計が必ずセットになるが，いくつかの手法が存在する。例えば，構造化分析というのがあり，これとセットで構造化設計という概念がある。

まず，この**構造化分析** (structural analysis) であるが，これは構造化プログラミングと同じようにユーザが行う作業をフローチャート化していこうというものである。最初にこのシステムに入ったら，ユーザはどんな作業をしたいのだろうかを顧客から聞きだす。例えば学務システムであれば，学籍管理，科目管理，成績管理をしたいとすると，最初にこれら三つに分岐する。続いて成績管理の場合には，採点結果記入，採点集計，採点確定，成績表示，合格決定といった作業に分岐する。というような作業を洗い出していき，作業のフローチャートを作成していく。

通常，特に意識しないと，作業分析を重点的に意識した**プロセス指向** (process oriented approach) あるいは**機能指向** (function oriented approach) と呼ばれる分析となる。自然の流れではあるが，機能優先であるため，分析の結果，利用すべきデータに統一性を欠くという問題が発生しやすい。例えば成績集計で必要とされる科目データの属性と科目管理で必要とされる科目データの属性は必ずしも一致せず，それぞれが勝手に決めてしまうと，あとから設計のフェーズで混乱が発生する。

このため，利用するデータに着目して分析する**データ指向** (data oriented approach) という概念が生まれた。つまり，顧客の必要とする要求を満た

すにはどのようなデータを必要とするかに着目するのである。それにより，全システムにわたるデータの整合性がとれるようになる。例えば，学務システムであれば，学生に関するテーブル，科目に関するテーブル，教員に関するテーブルが重要であることはただちにわかる。ここでテーブルとは，関係データベースのテーブルのことである。成績については，これらを参照するように作るべきであろうから，一科目一人ごとの成績をレコード（表計算ソフトであれば行に対応）とするのがよいであろうということになる。それを意識しつつ，顧客の要求する作業をフローチャート化していくのである。このデータ指向の分析により，構造化分析は大きく進展したのである。

構造化分析を元に**構造化設計** (structural designing) を行っていくことになる。まず，扱うべきデータベース，テーブル設計を行う。また，いくつかの適切なモジュールを作成する必要があり，モジュール設計が決定されることになる。もともと，構造化分析は構造化プログラミングを意識した手法なので，これをより詳細化してサブルーチンのような集合に直していくことで，構造化設計が達成される。さらに，通信プロトコル設計といって，クラス間やモジュール間の情報をやり取りする際の取り決めであるインタフェースを決定する。

データ指向をさらに進めたものとして，最近は，**オブジェクト指向分析** (object-oriented analysis) という手法が採用されるようになってきている。これも名前からわかるように，オブジェクト指向言語に合わせた形で分析を進めていくというものである。クラスとしては，同じような画面で作業できるものを一体化する。例えばある科目に対して教員が行う成績登録，成績確認は，いずれもその科目の履修学生の総覧に成績欄のついたほぼ同じような画面で実施できるため，これらをメソッドとしたクラスを作成するのである。このクラスのインスタンス変数は科目名（お

よび同キー）である。そのインスタンス変数を決定するメソッドに加え，成績を登録するメソッド，成績を示すメソッドによりクラスを構成する。このように，データと機能をまとめてブロック化することにより，設計時のブロック化を促進しようという概念である。もちろん，クラスを使うメインルーチンでは，従来通り構造化分析の手法がとられるが，データの存在するクラスのメソッドの導入により，データを直接マネジするルーチンまではブロック化されたことになり，メインルーチンもかなりわかりやすい構成になり得る。

オブジェクト指向分析をより具体化して設計することを**オブジェクト指向設計** (object-oriented designing) と呼ぶ。この場合もまず，扱うべきデータベース，テーブル設計を行う。続いてクラスを決定する。これにより内部データ・クラス設計が決定される。さらに，クラス間の情報をやり取りする際の取り決めである通信プロトコル設計を決定する。オブジェクト指向の問題点は，ソフトウェア業界の学習のスピードが遅く，未だにオブジェクト指向言語を使わないで，昔ながらの C などに頼っているため，概念が十分に浸透していない点にある。日本のソフトウェアの伸びが今一つなのは，この辺に要因があるかも知れない。しかし，現在のソフトウェア技術では，もっとも急速に取り入れてほしい技術である。

内部設計はプログラミングやテストという実行直前の設計であるため，プログラミングの際のソースコード作成の環境の整備やプログラマの守るべき各種の取り決めも必要である。後者は，クラスやモジュール，変数名などの付け方，コメントの入れ方など，最低限，他人にソースをチェックしてもらっても困らない程度のルールを決める。さらに，テストの際の環境の整備や，相互にクロスチェックによるソースコードレビューの方法も決定する。また最後に行われるプログラムの単体テスト，プログラムを組み合わせたときの結合テスト，システム全体の総合テストなど

のやり方やバグが発見されたときの情報の流し方や記録のとり方なども決定する。

一見，多岐にわたる大変な作業に見えるが，過去の経験がもっとも生きるところであり，過去の蓄積があれば，このいくつかはそのまま流用したり，わずかな修正で事足りる場合が多い。下流側の作業の結果，上流側の設計を変える必要が生じることもたびたびある。その場合には，上流へのフィードバックをかけることになる。こうして内部設計が完成する。

14.4 プログラミングとテスト

プログラミングとテストは，もっとも中核となるある程度独立した作業プロセスである。**プログラミング** (programming) とはまさにソースコード作成であるが，集団で開発している以上，コーディングに関する取り決めは遵守しなければならない。これを怠ると以後の作業で必ずトラブルを起こす。極めて小さなシステム開発の場合には，個人ですべてのソースプログラムを作成する場合もあるかも知れないが，その場合にも，読みやすいコードを作成しないと，後にバグが発生した場合や修正を要求された場合に，相当苦労することとなるので，最初から読みやすいように努力すべきである。

プログラムが完成するとコンパイルをし，コンパイルエラーがなくなるまで，プログラムを修正することとなる。その後に，他人にプログラムをチェックしてもらうクロスチェックによる**ソースコードレビュー** (source code review) が行われる。この段階で間違いが見つかると，再び，修正，コンパイルを行うこととなる。

テスト (test) は単体テスト，結合テスト，総合テストと行われる。**単体**

テスト (unit test) とは，各クラスやモジュールごとのテストである。モジュールの中の制御構造を意識したテストを行う。単体だけでテストをするには，これが利用する他のモジュールが必要である。しかし，それらを一気に動かすのは，あちこちにバグがあり，どこがどうか，わからなくなってしまう。そこで，極めて簡単な，例えば呼び出しがあったことがわかるだけの `print` 文 1 行からなるようなプログラムを用意する。こうした呼び出しの対象となる代替プログラムを**スタブ** (stub) という。単体テストの対象となるモジュールを呼び出すプログラムも，完成品に近いものを使うのは危険である。完成品に近いものは，複数のモジュールを呼び出すからである。そこでやはり，このモジュールをだけを駆動するような上位のプログラムが必要である。こうした呼び出し側のプログラムを**ドライバ** (driver) という。さらに，テストごとにテストデータが必要なことはいうまでもない。

同様にして，すべての生きたモジュールを繋ぎ合わせて行う**結合テスト** (join test) を行い，全体を組み合わせても問題がないかをテストする。結合テストは基本的には，外部仕様に書かれた動作が正しく行われるかを確認する作業である。最後に**総合テスト** (integration test) を行う。これは，できたシステムの速度といった性能や，何かの理由により，システムがダウンした場合に，被害が最小限に納まっているか，さらにきちんと回復できるか，ボタンを連打するとか，想定外のキーボードを打たれても正しく動作するかなどの，やや極限的な状況に対応できるかのテストである。結合テストと総合テストでは各モジュールはブラックボックスとして扱い，内部構造には立ち入らない。

テストにより，バグが発生するが，バグ管理は極めて重要である。まず，日々発生するバグの件数を記録していく。日々の発生バグを図表化したものは，日々の変動が大きく，あまり見やすいものではない。通常

はこれを累積した**バグ曲線** (bug curve) により管理する。バグは徐々に減っていき，その結果，累積数は日数に対して徐々に飽和的になるはずであるが，こうならない場合が問題である。バグが容易に無くならないということは，テストのしかたに問題があるか，テストの結果行われるプログラムの修正がうまく行っていないことを示す。急いでテスト方式を見直すとか，プログラマにエラーを収束する能力があるかなどをチェックする必要がある。

14.5 プロジェクトマネジメント

十人前後未満のメンバによるソフトウェア開発の場合には，いかに仕事をうまく分割できるか，またいかにグループ内の情報の流れをよくするかが，ポイントとなる。そこで重要なのはモジュール化と，モジュール間のインタフェースをいかにわかりやすくするかという点に集約される。オブジェクト指向言語はその意味で，これらの状況を満たしやすい環境をなかば自動的に与えてくれる。さらに，製品出荷後の手直しも多いことから，プログラムの読みやすさや周辺の文書化といった面の要因も極めて重要である。

さらに何十人，何百人もの人間が関わるようになると，プロジェクトマネジャが必要になり，プロジェクトマネジメントも系統的に行わなければならなくなる。プロジェクトマネジメントについては Web 検索することで詳細を知ることができる。特に幅広い製造技術に関するプロジェクトマネジメントの知識体系を記述した PMBOK (project management body of knowledge) ガイドと呼ばれるものが，ソフトウェア工学においてもよい指針となっている。詳細は Web を参考にしてほしい。

物を扱うハードウェアの世界では，特に日本の大学において，その科学

を扱う理学系と，その技術を扱う工学系が整備され，それが日本における物作りの文化を支えてきた。しかし，ソフトウェアに関しては，プログラミングを中心とする情報科学に偏っており，技術を扱うソフトウェア工学の分野の整備はかなり遅れている。この分野を整備することが，日本のソフトウェア産業の成長に大いに役立つのではないかと信じている。

14.6　おわりに

以上，全 14 章を使って，ソフトウェアの概要を示した。現在，ソフトウェアといえば高水準プログラム言語を用いて作成されるが，本書はプログラム言語教育が目的ではないため，なるべく特定のプログラムに依存しないように記述した。そのため，どうしても言葉不足の感が拭えなかった。言語を知らない人は，速修でもよいから，ぜひ何らかの言語を学び，改めて読んでもらうことで，深い理解が得られるのではないかと思う。

また，各章の記述は，著者自身のこれまでの経験と，それにより蓄積した知識により記載した。ただし，本章のソフトウェア工学については，著者の直接の経験は乏しく，ソフトウェア発注者としての経験，Web からの知識しかない。このため，鶴保，駒谷「ずっと受けたかったソフトウェアエンジニアリングの授業 1，2」翔泳社（2006）を参考にした。

演習問題 14

問題 14.1　次の用語を理解したかどうか確認せよ。

1）外部設計，内部設計

2） 構造化分析/設計
3） プロセス指向，機能指向，データ指向
4） オブジェクト指向分析/設計
5） バグ曲線
6） プロジェクトマネジメント

問題 14.2 p.136 の**図** 11.12 に示した学務データベースで，学生の行う科目登録/確認に関するクラスを設計してみよう。科目登録と登録された科目の確認はほぼ同じような科目名（必要があれば教員名も）の総一覧にチェック欄のある画面で行うこととし，登録は空欄であるチェック欄にチェックを入れることで行い，確認はすでに登録されている科目のチェック欄にチェックを示すことで行うこととする。同じ画面を利用することから，これらを一つのクラスとすることで，このクラスを設計してみよう。設計するといっても，必要なインスタンス変数とメソッドを羅列するだけでよい。

科目が登録されると，成績（score）テーブルに新しいレコード（この時点では成績欄は空欄）が追加されることとする。データベースはテーブルを組合せたものも含め，いろいろな条件で検索可能とする。また，どのテーブルに新しいレコードを追加/削除することも容易であるとする。なお，ほぼ同じ画面を使って，成績確認も可能であるので，同じクラスに含めるとよい。その場合，チェック欄に成績が示されることとする。

演習問題解答

第 1 章

1.1　本文を見よ。以後，用語の理解については解答は示さない。

第 2 章

2.2　**図** 15.1。`3bit` の範囲で正しい解とならないものには × を付した。

2.3　**図** 15.2。数値は前問の解答と一致しているが，正しい解の得られる範囲が異なっていることに注意。

111	111	×000	×001	×010	×011	×100	×101	×110
110	110	111	×000	×001	×010	×011	×100	×101
101	101	110	111	×000	×001	×010	×011	×100
100	100	101	110	111	×000	×001	×010	×011
011	011	100	101	110	111	×000	×001	×010
010	010	011	100	101	110	111	×000	×001
001	001	010	011	100	101	110	111	×000
000	000	001	010	011	100	101	110	111
	000	001	010	011	100	101	110	111

図 15.1　ビット幅 3bit の符号なし整数の和

111	111	000	001	010	×011	100	101	110
110	110	111	000	001	×010	×011	100	101
101	101	110	111	000	×001	×010	×011	100
100	100	101	110	111	×000	×001	×010	×011
011	011	×100	×101	×110	111	000	001	010
010	010	011	×100	×101	110	111	000	001
001	001	010	011	×100	101	110	111	000
000	000	001	010	011	100	101	110	111
	000	001	010	011	100	101	110	111

図 15.2　ビット幅 3bit の符号あり整数の和

第 3 章

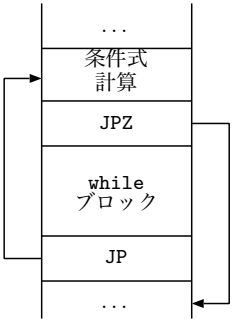
3.2　一例をあげる。

```
        LD 0, Data0;   // 被減数をロード
        LD 1, Data1;   // 減数をロード
        SUB 0, 1, 0;   // 減算
        ST 0, Data2;   // 結果をストア
        HLT;           // 停止
Data0:  0x0005;        // 被減数
Data1:  0x0004;        // 減数
Data2:  0x0000;        // 結果
```

3.3　第 4 章**図** 4.1。

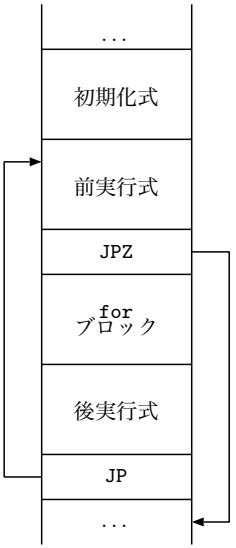
第 4 章

4.1 while 文の構造とアセンブラプログラム。



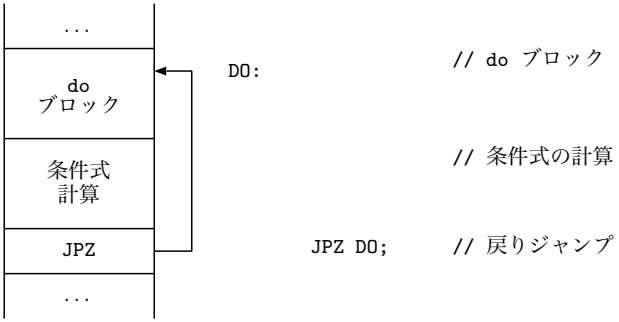

```
COND:                   // 条件式計算
        JPZ ENDWHILE;   // cond==F なら終了
                        // while ブロック
        JP COND;        // 戻りジャンプ
ENDWHILE:               // while-do 構造の終了
```

4.2 for 文の構造とアセンブラプログラム。




```
                        // 初期化式
PRE:                    // 条件式の計算
        JPZ ENDFOR;     // cond==0 なら終了
                        // for ブロック
                        // 後実行式
        JP PRE;         // 戻りジャンプ
ENDFOR:                 // for 構造の終了
```

4.3 do 文の構造とアセンブラプログラム。



4.4 then ブロックの途中に if-goto 文の入ったプログラムを構造化。

```
...
flag2=0;
if (条件 1) {
    ブロック 1
    flag2=条件 2;
    if (!flag2) {
        ブロック 2
    }
}
if (!flag2) {
    ブロック 3
}
...
```

4.5 if-goto 文から do 文へジャンプするプログラムを構造化。

```
...
flag2=0;
do {
    do {
        if (!flag2) {
            ブロック 1
        }
        ブロック 2
        flag2=0;
    } while (条件 1);
    ブロック 3
    flag2=条件 2;
} while (条件 2);
...
```

第 5 章

5.2 このサブルーチンを呼び出し中に，もう一度同じサブルーチンを呼び出すと，最初の呼び出しの戻りアドレスが消されてしまい，2 度目の呼び出しの戻りアドレスに戻ってしまう。しかし，こうしたことは，再帰呼び出しの際にしか起きないので，結論としては，再帰呼び出し厳禁となる。

5.3 前問と同じようなことが発生する。この場合，2 度目の呼び出しの際，戻りアドレスはきちんとスタックされるが，サブルーチン終了の際に，スタックの解放が行われ，この際，1 度目の呼び出しの戻りアドレスが消失してしまう。ただし，戻りアドレスが一つでもスタックに残っている場合には，スタックを解放しないというプログ

ラミングを行えば，この問題は回避される。

第 6 章

6.1 for 文を N 回，回すので，計算時間はおよそ N^1 に比例する。つまりオーダ 1 である。

6.2 毎回の n では n^2 という計算をしているが，どの n でもほぼ同じ時間の計算ですむ。for 文を N 回，回すので，計算時間はおよそ N^1 に比例する。つまりオーダ 1 である。このように，計算時間のオーダは計算式のオーダとは無関係である。

6.3 i×j の計算とその出力は i，j によらず，ほぼ同じ時間である。しかし for 文は i の変化と j の変化に対応して二重ループになり，N^2 回，回すので，計算時間はおよそ N^2 に比例する。つまりオーダ 2 である。i と j を入れ換えた計算は不要であることに気付いた人もいよう。その場合のループ回数は $N(N+1)/2=N^2/2+N/2$ であるが，これは N の最大べき数である 2 に着目し，やはりオーダ 2 である。しかし，総当たりに比べ，計算時間は約半分になるため，無視できない工夫である。

6.4 最初の訪問地の候補数は N，次は N-1，...であるので，すべての可能性のあるコースの数は N! である。$N! \approx N^{N \log N} \approx N^N$ であるので，オーダは N である。

6.5 データの個数を N 個とすると，$(\sum(x[i]-ave)^2)/N$ では引き算を N 回しなければならないが，$(\sum x[i]^2)/N-ave^2$ だと 1 回ですむ。なお，乗除の計算は前者は N+1 回であるのに対し，後者は N+2 回で 1 回多いが，N が多いとトータルで明らかに後者の方が勝る。

第 7 章

7.2 まず，(ADD 2 (MUL 3 4) 1) の一番下位の構成要素を見てみると，ADD と MUL は symbol，2，3，4，1 は number であり，いずれも atom であり，expression である。list は expression の 0 個以上の集合を括弧で包んだものであるので，(MUL 3 4) は list であり，かつ expression である。したがって ADD 2 (MULTI 3 4) 1 という expression の集合を括弧で包んだ (ADD 2 (MUL 3 4) 1) は list であり，文法上は確かに LISP のプログラムであることがわかる。LISP の文法はこのように単純であるが，他の言語はもう少し複雑である。

第 8 章

8.2 例えばクラス Diesel は次のようになる。

```
# クラス DieselEngine の定義
class DieselEngine < Engine
  def start()       # エンジンの起動
    puts("#{@disp} cc の D エンジン起動")
  end
  def stop()        # エンジンの停止
    puts("D エンジン停止")
  end
end
```

またクラス Truck は次のようになる。

```
# クラス Truck の定義
class Sedan < Auto
  def initialize(disp="")
    @aEngine = DieselEngine.new(disp)
  end
end
```

Truck のインスタンスをスタート，ストップすると次のようになる。

```
# プログラム
aTruck = Truck.new(2000) # インスタンス aTruck 生成
aTruck.start()       # 出力: ハンドブレーキを緩める
                     #       2000 cc の D エンジン起動
aTruck.stop()        # 出力: ハンドブレーキを締める
                     #       D エンジン停止
```

第 9 章

9.1 テキストファイルとは ASCII コードのような文字コードと呼ばれる 0/1 の集合で構成されている。数字・アルファベットの場合には 7bit，ちょっと余裕を持っても 1B(8bit)，日本語の場合には 2B から 3B ぐらいで構成されている。これをディスプレーに表示するには，一文字およそ 100 から数 100pixel 必要なので，それだけのビッ

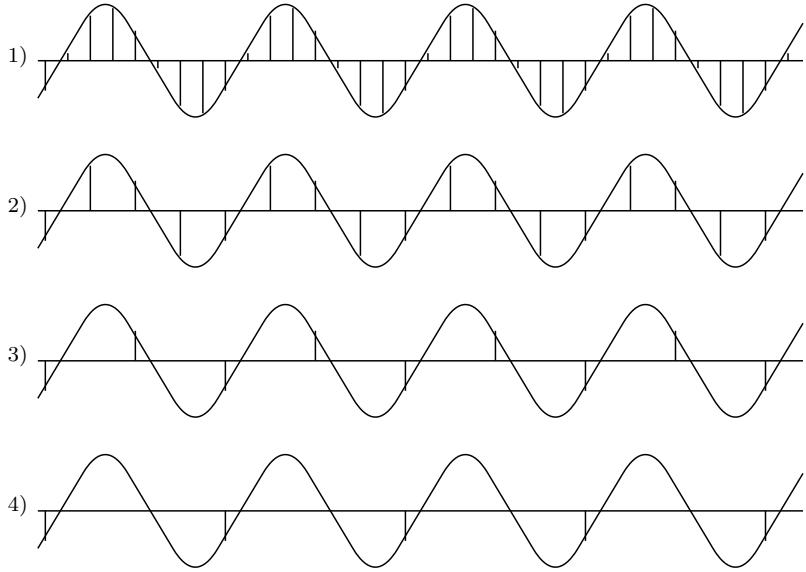


図 15.3　正弦波を各種周波数でサンプリングした結果 (縦棒に注意)。

トが必要ということになる。各コードからディスプレー上の文字を作成する際には，各文字形のビットマップを利用する。

つまり，ディスプレー上に表示されたテキストを，ファイル化する際には，コードを利用するということで，明らかにデータ圧縮されているのである。では，テキストを作成する側とそれを見る側で共有しているルールは何かというと，文字コードに対する文字形のビットマップなのである。

9.2　**図** 15.3。

9.3　x 方向の空間周波数が低く，y 方向の空間周波数が高い場合，画面の明るさは x 方向に激しく変化し，y 方向にはあまり変化しない。

第 10 章

10.2 スタック領域は番地の高いほうから低いほうへ利用するものとしておこう。スタックの管理のためには，スタックの底と現在積まれているスタックトップのアドレスを管理するだけでよい。つまり，この二つの値を記憶していればよい。OS に要求があるのは，push 命令と pop 命令だけである。push 命令が来た場合には，現在のスタックトップの上 (番地の低いほう) に push したい値をコピーし，スタックトップの値を 1 減ずればよい。逆に pop 命令が来た場合には，スタックトップの値を 1 増せばよい。ただし，スタックトップがすでにスタックの底の値の場合には，スタックがすでに空ということなので，これ以上，pop はできないので，error であることを示して，終了する。ユーザが厳密なスタックの push，pop をしていれば，このようなことは起きないはずであるが，人間はミスを起こしやすいので，エラー通知は重要である。

10.3 窓のインスタンス変数としては，最低限，位置とサイズであろう。対角線両端の座標を記憶してもよい。メソッドは，窓の生成，消滅，位置とサイズを返すこと，ユーザが与えたテキストを表示すること，図形を表示することである。さらに，表示内容の消去も必要かも知れない。

第 11 章

11.2 デキューの際に問題が発生する。末尾のノードを一つ外す際，末尾ノードのアドレスを見張っている末尾ポインタ変数の値を，一つ手前のノードのアドレスに変更しなければならない。その際，末尾

ノード側から，一つ手前のノードのアドレスを知る方法はない。どうしても知ろうとすると，先頭から順に線形リストをたどっていかなければならない。リストが長くなると，その探索時間は無視できなくなるため，よい方法とは言えない。

11.3 ヒントは前問の解答にある。末尾のノードを削除する際，一つ前のノードのアドレスを知る必要があるが，末尾ポインタの内容を見ても，それはわからない。そこで，末尾ノードには一つ前のノードのアドレスを収容しておく必要がある。末尾ノードを削除すると，一つ前のノードが末尾ノードとなるが，そこにも一つ前のノードのアドレスを収容しておく必要がある。これより，すべてのノードに前後の隣接ノードアドレスを収容する二つのポインタが必要となる。

11.4 **図** 11.10 下図で，まず key4 と左右 Null からなるノードを新規に作成する。続いてルートポインタから順にキーの大小を探っていくと，key0 より大きいので右下へ，key2 より小さいので左下へ，key3 より大きいので，右下へ行こうとするが，そこが Null なので，この Null のところへ key4 へのポインタを代入する。

11.5 ハッシュ関数を利用した配列では，データの追加や削除さらに検索もかなり簡単である。しかしおよそのデータ量がわからないと，ハッシュ表の大きさが決められない。二分木はこれと逆で，追加，削除，検索には若干の負担がかかるが，データ量に対してはかなりの融通性がある。

11.6 まず与式で積の括弧部分を A とでも書くと，(ADD 2 A 1)。これをドット表現すると (ADD.(2.(A.(1.nil))))。続いて (MUL 3 4) はドット表現では (MUL.(3.(4.nil)))。これを A の部分に代入すると (ADD.(2.((MUL.(3.(4.nil))).(1.nil)))) が得られる。括弧が多いので数を間違えないように。

第 12 章

12.2 「あ」の UTF-8, SJIS, EUC, JIS での 16 進表現コードは 0xE38182, 0x82A0, 0xA4A2, 0x2422 と，すべて異なる。

12.3 あらかじめ一行の最大文字数を決めておき，行数分のメモリーを例えばヒープ領域などに確保する。入力されたテキストを，このメモリー上に，行単位で収容していく。行の終端を示す特別なコードを用意し，各行の文字数が最大文字数に達しない場合には，それを置くことにする。このメモリー全体をテキストメモリーと名付けておこう。この各行の先頭アドレスをデータとする二分木を用意する。二分木に示された順に行データを見ていくと，元のテキストが再現することになる。

各行の変更の際は，別の一行分の作業用メモリーにコピーし，文字挿入や文字抹消はその作業用メモリー上で行う。修正作業が終了したら，作業用メモリーの内容を，元の行データにコピーする。行削除，あるいは修正の結果，その行の長さが 0 になってしまった際には，二分木のノードを削除する。新しい行を追加する際には，テキストメモリーを一行分増加し，二分木の適切な位置にその行頭を指す新たなノードを生成する。

最後に全テキストをセーブするには，二分木を順にたどりながら，ファイルに書き出していけばよい。これがラインエディタの基本であるが，さらに詳細な設計については，読者の創造性に任せる。

第 13 章

13.2 書き方にはかなりのバリエーションがあるが，一例を示す。

```
<html>
 <head>
  <meta http-equiv="Content-Type"
   content="text/html; charset=Shift_JIS">
  <title>Greeting</title>
 </head>
 <body>
  <h1>Welcome</h1>
   <p>Hello!</p>
 </body>
</html>
```

作成したファイル名は hello.html または hello.htm とし，パソコンの Web ブラウザの「ファイルを開く」で開くと正しいかどうかが確認できる。

第 14 章

14.2 登録にせよ確認にせよ，登録/確認画面に入る前に登録者は決まっているはずであるので，インスタンス変数として登録者の名前やキーは必要不可欠である。メソッドとして登録者の名前（同時にキー）の設定/読み出しは最低限必要であろう。その他必要なメソッドは登録，確認，成績表示である。登録の際は登録画面を表示し，チェックが入ればその位置からどの科目が選ばれたかを判定し，必要情報を入れたレコードを成績テーブルに追加する。確認は，成績テーブルから登録者のキーの入ったレコードがあれば，科目一覧表にチェックを示す。成績の場合には，科目一覧表に成績を示せばよい。

索引

●配列は五十音順。

●D

●E

●F

●G

●H

●I

●J

●S

●T

●か

●さ

著者紹介

岡部　洋一（おかべ・よういち）

1943 年	東京都に生まれる
1967 年	東京大学工学部卒業
1972 年	東京大学大学院工学系研究科 (工学博士)
1972 年より	東京大学講師，助教授，教授
2006 年より	放送大学教授，副学長，理事
2011 年より	放送大学長
専攻	電子工学・情報工学
主な著書	絵でわかる半導体と IC(編著，日本実業出版社)
	素人の書いた複式簿記 (オーム社)
	電磁気学の意味と考え方 (講談社)
	コンピュータのしくみ (放送大学教育振興会)
	リーマン幾何学と相対性理論 (プレアデス出版)

放送大学教材　1570099–1–1411（テレビ）

ソフトウェアのしくみ

発　行　　2014 年 3 月 20 日　第 1 刷
著　者　　岡部洋一
発行所　　一般財団法人　放送大学教育振興会
〒 105-0001　東京都港区虎ノ門 1–14–1　郵政福祉琴平ビル
電話　03–3502–2750

Printed in Japan ISBN978–4–595–31499–5 C1355