ビットから始める

プログラミング

株式会社アイティネット
高橋哲夫

目 次

はじめに

わたしたちは長年、知識を表現する手段として言語や図表、数式、楽譜、絵画、造形物、写真などを用いてきたが、１９６０年代になってプログラムという手段が加わった。プログラムはそれまで表現できなかった世界を描けるようにした。プログラムで表現された知識はコンピュータ上で動作する動的なものである。ここに大きな可能性が生まれた。プログラムは単純なものから巧妙なものまでいくらでもある。数値計算、コンピュータ科学、企業システム、自治体サービス、音楽の演奏、病理診断、機械の制御、書物読み聞かせ、各種シミュレーションなどをはじめとして数えきれないほどある。実際、どのようなプログラムがあるのか知りたければ、コンピュータがどこで使われているか見るのがよい。

家庭内で使われているテレビ、ビデオ、洗濯機、冷蔵庫、給湯器、エアコン、炊飯器などはコンピュータで制御されている。

駅に行くと、ここにもコンピュータを搭載した自動券売機・自動改札機がある。電車の運行システムはコンピュータで制御されている。

コンビニに行けば POS 端末、銀行の ATM、チケット販売機、コピー機、エアコンなど、店舗の機器は殆どコンピュータが搭載されている。

銀行にはオンライン・システムのコンピュータ搭載の ATM 端末が並んでいる。窓口にも専用の端末がある。

病院に行くと、受付は診察カードを機械に読ませて行い、診察室では医師が電子カルテの画面に診察結果を入力、別室で撮った CT スキャンやレントゲン写真は医師の前の画面に表示、支払いは精算機で行なう。すべてはコンピュータで処理されている。

自動車はまるで走るコンピュータである。ドアの施錠・開錠からエンジン制御、安全運転、エコ運転などはコンピュータで制御している。

企業は人事・給与、会計・財務、広報などにコンピュータを使用している。さらに、企業固有の事業用として、製造業なら生産管理に、証券会社ならば証券の売買に、流通業なら物品の配車・人員配置などにコンピュータを用いている。その他の業種でも、それぞれの目的に合わせてコンピュータが利用されている。

自治体では、住民サービス（住民票、戸籍、印鑑登録、etc.）、地方税・固定資産税、健康保険・年金保険などの管理にコンピュータを用いている。

上に挙げたコンピュータ利用は個人や企業内に閉じている。これに対して、e-mail やホームページ、ブログ、フェイスブック、ツイッターなどはインターネットに接続した外部のコンピュータのサービスを利用している。

インターネットに接続するコンピュータ群は 1 つの巨大なコンピュータを構成していると言って良いでだろう。

インターネットにはパソコンやタブレット、スマートフォンだけでなく、家電製品や自動車、センサーなどを接続し、新しい付加価値を生み出している。それは IoT（Internet of Things、もののインターネット）と呼ばれているインターネットの利用法である。

インターネットの基幹装置であるルータや DNS（Domain Name System、ドメイン・ネーム・システム）などの通信機器はコンピュータを用いている。

天気予報はわたしたちの生活に不可欠だが、自然に向き合っている農業・漁業、航空機の運航などでは、精度の高い天気予報は欠かせない。天気予報は気象衛星のデータ、気象レーダ、地上気象観測などのデータを使用してシミュレーションを行い、各地の予報を出している。このシミュレーションは膨大な計算を必要とするため、スパコン（Super Computer：スーパー・コンピュータ）を使用している。

膨大な計算が必要なのは天気予報に限らない。安全な自動車の設計のために、自動車が衝突したときの状態を調べる必要がある。実際に自動車を衝突させる実験を行うと時間や費用がかさむ。代わりにスパコンでシミュレーションすることにより費用と時間を大幅に減らしている。

航空機の設計に必要な風洞実験もスパコンでシミュレーションして費用と時間を大幅に減らしている。

この他にも創薬、つまり新しい薬の開発において、大量の化学物質の組み合わせの中から期待できる候補を絞り込むために膨大な計算を必要としている。ここでもスパコンが使われている。

このようにコンピュータの利用形態は多岐にわたりさまざまである。そのすべてにおいてそれぞれに適したプログラムが使われているのである。

プログラムはこのように多くの領域で使われているが、これらのプログラムが動作するすべてのコンピュータは同じ原理で動作している。そしてプログラミングの基本的考え方は同じである。

本書はプログラミングの基本的考え方について紹介する。

はじめに、コンピュータのしくみについて紹介する。プログラミングの理解にはコンピュータの知識が必要である。詳細な知識でなくても、コンピュータの原理程度の知識は欠かせない。

次にコンピュータで扱うデータ表現について紹介する。代表的なデータは文字列データと数値データである。さらにこれらのデータを組み合わせたデータ構造の典型的な例である配列、キュー、スタックなどについて紹介する。プログラミングでは、情報を集めてその性質をつまびらかにすることが大切である。この作業を通してデータ構造が明らかになると、それをどのように処理するかが見えてくる。

さらにデータの処理方法について紹介する。データ処理は、最終的にはコンピュータが用意したマシン命令で表現するが、マシン命令のレベルは人にとって分かりにくいの

で、プログラミング言語を使うのが一般的である。本書では、広く使われているプログラム表現に準じた方法を用いて、データ処理について紹介する。

本書はプログラミングとは何か知りたいと思っている人たちを対象にしている。比喩を用いてなんとなく分かるというのでなく、厳密さを失わず、論理の展開をはしょることなく、ビットの説明からはじめ、最後にはプログラムをどのようにまとめるかまで分かるように努めた。

本書がプログラミングの理解にいささかでも役立つことができれば幸いである。

第１章　コンピュータのしくみ

わたしたちの身の回りで使われているコンピュータはフォン・ノイマン型と呼ばれる方式で動作している。

この型のコンピュータは次の特徴をもっている。

- ２進法
- アドレス付メモリ
- メモリ内蔵プログラム
- プロセッサ
- 逐次処理
- インプット/アウトプット

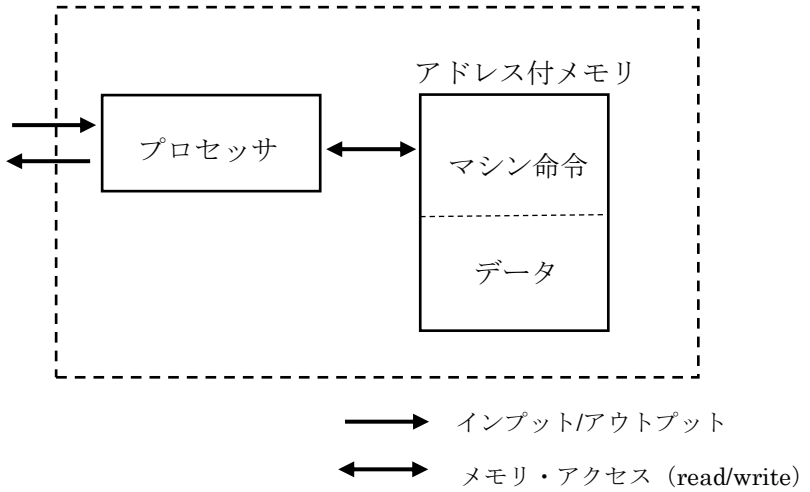


図 1-1　フォン・ノイマン型コンピュータのスケッチ

図 1-1 はフォン・ノイマン型コンピュータのしくみのスケッチである。内部にはプロセッサとアドレス付メモリがある。アドレス付メモリ上にはプログラムが格納される。プログラムは２進法で表現されたマシン命令とデータとをもち、プロセッサがメモリ上のプログラムのマシン命令を逐次読みだし、マシン命令の指示に従った処理を遂行する。プロセッサはプログラムのマシン命令に従い、外部の装置（例えば、ディスプレイ、通信機器、SSD メモリ、USB メモリ、･･･）からデータを読み込み（インプット）、あるいは書き込む（アウトプット）。

プログラムはアドレス付メモリに格納されることにより、アドレスを指定すれば、そのアドレスにあるマシン命令から実行することができ、また、指定されたアドレスのデータをアクセス（読み/書き）することができる。

フォン・ノイマン型コンピュータはこのような特徴をもっているが、以降では ２ 進法、アドレス付メモリ、プロセッサの３つについてもう少し詳しく述べよう。

注意　フォン・ノイマンは数学者で、物理学をはじめ多くの分野で業績を残したハンガリー出身の天才である。コンピュータは彼一人の発明ではないが、彼が現代コンピュータの原型となる EDVAC

の開発に参加し、内部レポートを彼の名前で執筆したことから、EDVAC タイプのコンピュータはフォン・ノイマン型コンピュータと呼ばれるようになった。

１．１　２進法

コンピュータで扱う情報はすべて 2 進法で表現する。

情報の最小の表現手段をビット（bit：binary digit の略）と呼んでいる。
ビットは On あるいは Off の状態を表わす。On を数の 1 に、Off を数の 0 に対応づける。

1 つのビットでは 2 通りの表現ができる。0 か 1 である。

2 つのビットの列の 2 進法表記は、

00　01　10　11

の通りである。4 通りの表現ができる。

ここで、2 進法表記を 10 進法表記へ変換する方法を与えておこう。
連続するビットの列を $b_{n-1}b_{n-2}\cdots b_0$　（b_i=0 or 1 , i=0,1,…,n-1）とし、右端のビットを 1 桁目、その左のビットを 2 桁目、…と考えると、ビット列 $b_{n-1}b_{n-2}\cdots b_0$ は次式のように表わすことができる。

$$b_{n-1}b_{n-2}\cdots b_0 = b_{n-1}\times 2^{n-1} + b_{n-2}\times 2^{n-2} + \cdots + b_0\times 2^0$$

上式の右辺を計算した結果は、左辺のビット列の 2 進法表記に対する 10 進法表記となっている。

この方法に従って、2 つのビットの列を 10 進法に直してみると、

$$00 = 0\times 2^1 + 0\times 2^0 = 0$$
$$01 = 0\times 2^1 + 1\times 2^0 = 1$$
$$10 = 1\times 2^1 + 0\times 2^0 = 2$$
$$11 = 1\times 2^1 + 1\times 2^0 = 3$$

となる。

2 つのビットの列の 2 進法表記と 10 進法表記を並べて書くと次の通りである。

2 進法表記	10 進法表記
00	0
01	1
10	2
11	3

3 つのビットの列についても同様の計算を行い、2 進法表記と 10 進法表記を示すと次の通りである。

2 進法表記	10 進法表記
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

3 つのビットの列では 8 通りの表現ができる。

4 つのビットの列についても上と同様の計算を行い、2 進法表記と 10 進法表記を並べて書くと次の通りである。

2 進法表記	10 進法表記
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

4 つのビットの列では 16 通りの表現ができる。

上表の 10 進法表記で 10 以上の値：10、11、12、13、14、15 に対して、文字 A、B、C、D、E、F を対応させたものを 16 進法表記という。上表に 16 進法表記を加えると次の通りである。

2 進法表記	10 進法表記	16 進法表記
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

4 つのビットの列は 10 進法表記や 16 進法表記で置き換えることができる。

長いビット列は 4 つのビットの列毎に刻んで、16 進法表記で表わすと長い表記を短縮できる。

例

2 進法表記	16 進法表記
00001110	0E
11111111	FF

ここまで、1 つのビット、2 つのビットの列、3 つのビットの列、4 つのビットの列について見てきたが、最後に 8 つのビットの列について見てみよう。

8 つのビットの列を半分に分け、それぞれを 16 進法表記で表わすことにする。

x_1x_2　　（x_1、x_2 はそれぞれ 0～F の 16 進法表記）

x_1、x_2 はそれぞれ 16 通りの表現をもつので、x_1x_2 で表現できる数は 16 通りのものが 16 通りあると考えることができる。つまり、16×16＝256 である。
このことから、8 つのビットの列では 256 通りの表現ができることが分かる。

8 つのビットの列を一括りにしたものをバイト（byte）と呼ぶ。

このようにして、On(1)、Off(0)という情報しか持たないビットをつなげることにより、情報の表現の幅が、2 通り、4 通り、8 通り、16 通り、256 通りと拡がってきた。

1 つのバイトでは 256 通りの表現ができるが、これだけあれば、26 文字の英大文字、26 文字の英小文字に 10 文字の数字、さらにいろいろな記号をそれぞれ各バイトに対応させてもまだおつりがくる。また、0 から 255 までの数値なら 1 つのバイトで表現できる。

2 つのバイトの列を使うと、日本語などの文字や桁数の多い数値が表現できる。さらに、4 つのバイトの列を使うと、もっと大きな数値が表現できる。

2 進法を使って、わたしたちが日常使用している文字データや数値データだけでなくマシン命令をバイトの列に対応づけてコンピュータで扱えるようにするのである。

本書では、2 進法表記、16 進法表記について、必要ならそれぞれ左端に 0b、0x をつけて表わすことにする。10 進法表記に対しては何もつけない。

例	2 進法表記	`0b10001000`
	16 進法表記	`0x88`
	10 進法表記	`136`

上例のように、`0b` や `0x` をつける理由は、もしつけないと、`10001000` は 10 進の `10,001,000`、つまり 1 千万 1 千なのか 2 進数なのか、`88` は 10 進数なのか 16 進数なのか判別できないためである。

１．２　アドレス付メモリ

コンピュータはプロセッサがアクセス（読み/書き）できるアドレス付メモリ（図 1-2 参照）をもっている。メモリは連続するバイトの列として構成する。

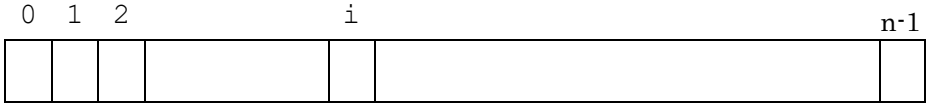


図 1-2　アドレス付メモリ

各バイトは左端のバイト位置を 0 とし、右へ 1 ずつ増加していくアドレスが付与される。

プロセッサがメモリ内の特定のバイトをアクセスするには必ずそのバイトのアドレスを使用する。これをメモリ・アドレスという。

メモリ・アドレスを使用すれば、特定のバイトを直接アクセスできる。

このようなしくみをもつメモリをアドレス付メモリという。

「１．１　２進法」の最後で、「2 進法を使って、わたしたちが日常使用している文字データや数値データだけでなくマシン命令をバイトの列に対応づけてコンピュータで扱えるようにするのである」と述べたが、これらはバイト表現なので、アドレス付メモリに配置できる。ここで、配置とはデータのバイト列をメモリ上のバイト列に写すことを意味する。

メモリ・アドレスもまたバイト列で表現してメモリに配置できる。

アドレス付メモリにはプログラムを構成するマシン命令の系列とデータの系列を配置する。次に説明するプロセッサはメモリ上のプログラムを読み込んで処理する。

１．３　プロセッサ

プロセッサは次の機構を備えている。

- 制御機構
- 処理機構
- 汎用レジスタ群
- イベント処理機構

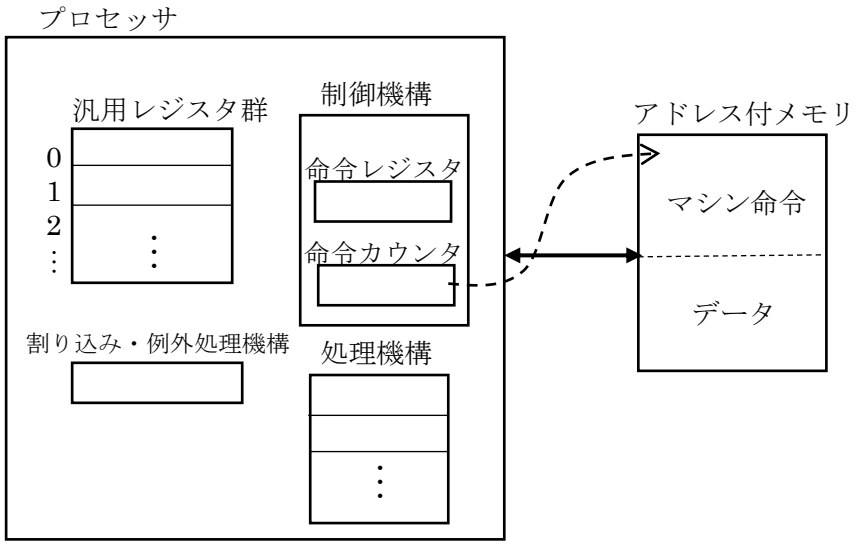


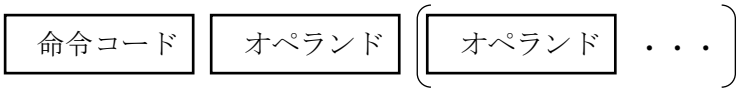
図 1-3　プロセッサの概要

図 1-3 の各機構の働きは次の通りである。

（ａ）制御機構

アドレス付メモリ上のマシン命令はメモリ上では動作しない。メモリ上に情報（マシン命令とデータ）があるだけである。
制御機構はこのマシン命令を処理できるようにするために、命令レジスタに読み込み、マシン命令内の命令コードに従った処理機構を作動させる。

マシン命令は次の形式をしている。



命令コードは、分岐命令、データ処理命令、ロード/ストア命令、ステータス・レジスタへのアクセス命令などがある。
オペランドは命令コードの処理対象となる汎用レジスタ、アドレス指定、定数などである。

オペランドにいくつアドレス指定ができるかで、コンピュータには次の３つのタイプがある。

-- RISC: Reduced Instruction Set Computer
  - ロード/ストア命令ではオペランドにアドレス指定を許すが、その他の命令のオペランドにアドレス指定を許さない
  - 演算はレジスターレジスタ間で行なう
  - レジスタとメモリとのやり取りはロード/ストア命令で行なう
-- CISC: Complex Instruction Set Computer
  - オペランドにアドレス指定しか許さない
  - 演算はメモリーメモリ間で行なう
-- RISC と CISC の中間
  - オペランドにアドレス指定を１つだけ許す
  - 演算はレジスターメモリ間、メモリーメモリ間で行なう

最近のコンピュータは性能が出しやすい RISC を採用する傾向があるので、本書は RISC を前提にした。

命令カウンタは次に実行するマシン命令のメモリ・アドレスを保持するレジスタである。

命令カウンタが指すメモリ・アドレスのマシン命令が命令レジスタに読み込まれると、命令カウンタには現在読み込んだマシン命令のサイズ（バイト）が加算される。その結果、命令カウンタは次に実行すべきマシン命令を指している。このように命令カウンタを使用して、メモリ上のマシン命令は逐次処理される。分岐命令やサブルーチン呼び出しの場合、異なる方法で命令カウンタの内容が変更されるが、それらについては処理機構で述べる。

（ｂ）汎用レジスタ群

演算（加減算演算、論理演算、比較演算など）はメモリ上のデータをレジスタに読み込んで処理する。データのアドレスやマシン命令のアドレスもまたレジスタに読み込んで処理する。さらに、比較演算の結果もレジスタに格納する。

このようにレジスタは多用途で用いられるので、汎用レジスタと呼んでいる。

汎用レジスタは複数個あり、0、1、2、…のように番号で識別される。マシン命令のオペランドでは、この番号を用いて使用するレジスタを指定する。

（ｃ）処理機構

処理機構は命令コードに対応する処理を遂行する。

実際の命令コードは複雑だが、以下では簡略化したいくつかの命令コードを紹介する。

（ｉ） ロード/ストア

LD（ロード） LA（ロード・アドレス） ST（ストア）

- ロード命令はメモリ上のデータを汎用レジスタにロードする。
- ロード・アドレス命令はメモリ・アドレスを汎用レジスタにロードする。

・ ストア命令は汎用レジスタの内容をメモリ上にストアする。

（ii） 加減乗除算

ADD（加算） SUB（減算） MUL（乗算） DIV（除算）

・ 2 つのオペランドの汎用レジスタ間で加減乗除算を行い、結果を 1 つのオペランドの汎用レジスタに入れる。
・ 命令コードはオペランドのデータの種類（2 進固定小数点、2 進浮動小数点）に合わせて用意する。
・ 2 進固定小数点および 2 進浮動小数点の 2 進法表現は加減乗除算の演算法を決定するが、詳しくは「第 3 章 データの表現」を参照。

（iii） 論理演算

AND（論理積） OR（論理和） NOT（否定）

XOR（排他的論理和） SHIFT（シフト）

・ 2 つのオペランドの汎用レジスタ間で論理積、論理和、否定、排他的論理和を行い、結果を 1 つのオペランドの汎用レジスタに入れる。
・ シフトは汎用レジスタの内容をビット単位で左シフトあるいは右シフトする。

（iv） 比較演算

COMP（比較）

・ 2 つのオペランドの汎用レジスタ間で比較を行い、結果をステータス・レジスタに入れる。
・ 文字列データの場合、汎用レジスタにはメモリ上の文字列データのアドレスを格納し、比較は汎用レジスタが指すメモリ上のデータに対して行う。
・ 命令コードはオペランドのデータの種類（2 進固定小数点、2 進浮動小数点、文字列データ）別に用意する。

（v） 分岐

B（無条件分岐） BC（条件分岐） BL（サブルーチン分岐）

BR（サブルーチンからの戻り）

・ “無条件分岐”はオペランドに指定したメモリ・アドレスの指すマシン命令に分岐する。言い換えれば、命令カウンタの内容がこのマシン命令のアドレスになる。
・ “条件分岐”はステータス・レジスタの値に従って、次の 6 通りの分岐*を行う。

＊分岐先はオペランドに指定したメモリ・アドレスの指すマシン命令である。

BCEQ	比較結果が等しければ分岐
BCNE	比較結果が等しくなければ分岐
BCLT	比較結果がより小さければ分岐
BCLE	比較結果がより小さいか等しければ分岐
BCGE	比較結果がより大きいか等しければ分岐
BCGT	比較結果がより大きければ分岐

- “サブルーチン分岐”はオペランドに指定した汎用レジスタ（これをリンク・レジスタと称し、固定番号を使用）にこの分岐マシン命令の次のマシン命令のアドレスを格納し、オペランドに指定したメモリ・アドレスの指すマシン命令に分岐する。汎用レジスタの内容はサブルーチンから戻るために使われる。
- “サブルーチンからの戻り”はサブルーチン側で用いる。オペランドに指定した汎用レジスタにリンク・レジスタの内容を設定して実行することにより“サブルーチン分岐”マシン命令の次のマシン命令に分岐する。

（d）割り込み・例外処理機構

コンピュータは同時に複数の処理を遂行する。あるプログラムの動作中に、ディスプレイを通して利用者と対話し、ネットワークにつながった他のコンピュータと交信し、入出力装置（ハード・ディスクなど）からのデータ読み込みや書き込みなどをほぼ同時に行っている。OS（オペレーティング・システム）はこうした処理が適切に動作するように制御している。そのためには、コンピュータはディスプレイや入出力装置、他のコンピュータとの間のメッセージを OS に渡す必要がある。

このようなコンピュータのメッセージ渡しは割り込みと呼ばれる。割り込みは入出力装置の正常な動作時だけでなく、装置の動作不良によっても引き起こされる。

コンピュータは例外処理と呼ぶ機能を備えている。プログラムに誤りがあり、例えば、ゼロによる除算、あるいはマシン命令が存在しないアドレスを命令カウンタに設定した場合、コンピュータは OS に例外を通知する。

コンピュータは定常的な動作のみならず、上記のような不規則な事象に対しても対処する。

１．４　しくみの普遍性

これまで述べたことは、フォン・ノイマン型コンピュータの概念的なしくみである。ほぼすべてのコンピュータはこのしくみで出来ている。大きなコンピュータも小さなコンピュータも、また速いコンピュータも遅いコンピュータもこのしくみで動作している。コンピュータの心臓部として使われている CPU チップはメーカが異なっていてもやはりこのしくみを採用している。このようにこのしくみは普遍性をもっているのである。

本書で述べるプログラミングの説明はフォン・ノイマン型コンピュータのしくみを前提としている。従って、フォン・ノイマン型コンピュータである限り、その上で動作するすべてのプログラムに対して、本書で述べる考え方が矛盾なく適用できる。

第２章　データの表現

本章では、コンピュータで扱う文字列や数値の 2 進法表現について、また、文字列や数値を組合せたデータ構造について紹介する。

これらの説明にはプログラムという用語が登場するので、予めその意味を明らかにしておこう。

プログラミングとは、何をすることなのか。「1 から 1000 までの間にあるすべての素数を求めよ」という問題で考えてみよう。

素数とは 1 か自分自身でしか割り切れない１より大きい自然数のことである。

この素数の定義から、2、3 は素数である。

4 は 2 で割り切れるので素数ではない。

ここまでは直ぐに分かるが、それに続く自然数 1 つひとつについて素数か否か調べることにする。

偶数は必ず素数 2 で割り切れるので、素数ではない。そのため、偶数は調査対象から外す。

調べる対象の数を候補と呼ぶことにする。

3 に続く奇数を 1 つ求めそれを候補とする。候補を 3 から始まる素数の配列（素数配列と呼ぶ）のすべての要素で割って余りに 0 になるものがあるならば、この候補は素数ではないので、現在の候補に 2 を加算して次の候補として調査を繰り返すが、この候補が 1000 以上ならば調査は終わりである。

候補を素数配列のすべての要素で割った余りに 0 になるものがなければ、候補は素数であり、この候補を素数配列に加える。続いて現在の候補に 2 を加算して次の候補として上の調査を繰り返す。

調査が終了したとき、2 と素数配列のすべての要素が 1000 までの素数である。

このような 1000 までの素数を求める手順のことをアルゴリズムという。

このアルゴリズムをマシン命令やプログラミング言語で表現したものをプログラムという。アルゴリズムを見つけ出し、プログラムを作成することをプログラミングという。

1000 までの素数を求めるプログラムは「３．６　繰り返し」の「（２）for-end」の例に示しておいた。

上記の 1000 までの素数を求めるアルゴリズムは日本語で表現しているが、これをプログラミング言語で表現したものや、その他の特定の形式言語で表現したものも計算手順であることに変わりがないのでアルゴリズムと呼ぶことができる。

上の例では、素数の“候補”や“素数配列”というデータが使われているが、これらに対するコンピュータにおける 2 進法表現を与えておかなければプログラムを作成することはできない。

上例の“候補”は数値であり、“素数配列”は数値の配列（データ構造の 1 つ）である。

本章では、文字列および数値に対する 2 進法表現と、これらを組み合せたデータ構造について紹介する。文字列および数値をまとめてスカラー・データと呼ぶ。

２．１　文字列

文字は人にとってグラフィカルなその表示形が意味をもつ。しかし、コンピュータには、表示形に対して人が捉えるような意味など存在しない。コンピュータには文字の表示形に対応する 2 進法表現（バイト）が意味をもつ。文字の表示形と 2 進法表現が同じものであると定義することにより、人とコンピュータがつながりをもつことができる。

なお、文字の 2 進法表現については「１．１　2 進法」の説明で少し触れたが、詳細は「２．２　文字コード」で紹介する。

文字列は複数の文字を並べたものであり、図 2-1 のようにメモリに配置される。

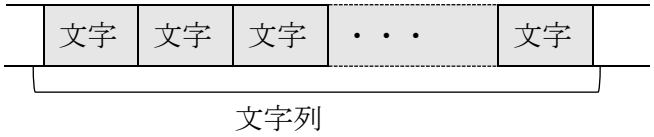


図 2-1　文字列のメモリ配置

複数の文字列をつないで並べたものを文字列の系という。

文字列の系は、

　　文字列 文字列 文字列　…

のように表わす。

文字列と文字列をつなぐ文字は存在しなくても“空白”文字でもよい。

例　次の例文の「」内は文字列の系である

日本語　　「リーマン予想は数学最大の難問として有名なものですが、1859 年にリーマンが提出してから、150 年以上解けていません。」

英　語　　「In American universities a course covering roughly the material in this book is ordinary given in the first graduate year.」

プログラム　「pv = 0
for i (0, n+1, 1)
　x = 1
　pv += A[i]*x$_0$**i
end」

- 日本語では文字列をつなぐ文字はない。日本語の文字列の系をコンピュータで処理するには形態素解析*による単語分けが必要である。
  ＊形態素解析とは辞書を使って文字列の系を品詞に分ける技術である。
- 英語やプログラムでは文字列は英数字列と= ( ) ; : , . + [ ] * などの特殊文字からなり、つなぐ文字はないか、“空白”であり、文字列はコンピュータで容易に抽出することができる。

文字列や文字列の系の場合も、文字の場合と同じように、人にとってはその表示形が意味をもち、コンピュータにはその 2 進法表現（各文字の 2 進法表現がつながったもの）が意味をもつ。

プログラミング言語には多くの種類がある。そのいくつかを挙げると、

C、C++、Fortran、COBOL、JAVA、PHP、JavaScript、Perl、Python、･･･

などである。これらはよく使われている言語で、用途は異なるが、どのプログラムも文字列の系になっている。

これまで述べたように、複数の文字をつないで文字列をつくり、複数の文字列をつないで文字列の系をつくり、文字列の系を使用してプログラムが表現されていることが分かった。

２．２　文字コード

２．１では、文字が人とコンピュータとの橋渡しの役割を担っていること、そして人の考えをコンピュータ上で表現する手段になっていることが理解できたのではないだろうか。そこで重要となるのが文字コードである。

文字コードとは、コンピュータで扱われる文字の２進法表現のことである。

コンピュータが発展していく過程で文字のいろいろな 2 進法表現が使用されてきた。コンピュータがインターネットにつながり、コンピュータ間での情報交換があたりまえになると、世界共通の 2 進法表現を用いる必要性が生じ、今日では統一規格として Unicode、utf-8、utf-16 が使われるようになった。

Unicode は英語、スペイン語、ギリシャ語、日本語、中国語など、世界中の言語の文字のすべてを集めた文字集合である。この文字集合の中の文字は順序付けられて並んでいる。そのため、順序集合と呼んでも良い。

この文字集合の一番初めにある文字は英語で使われている文字群である。
日本語の文字群は欧米系の文字に続いて位置付けられている。

すべての文字は先頭から順番に位置付けされている。
Unicode は 0x0000 から 0x10FFFF までの文字位置（最大 21 ビット）で指定される文字である。

文字位置順に見ると次表の通りである。文字位置は 2 バイトの 16 進表記である。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000																
0010																
0020	空白	!	"	#	$	%	&	´	(	)	*	+	,	−	.	／
0030	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	P	Q	R	S	T	U	V	W	X	Y	Z	[	＼	]	^	_
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	p	q	r	s	t	u	v	w	x	y	z	{	|	}	～	

この領域は制御コードとして使用（掲載省略）

文字位置順に 0 番地から 16 文字ずつ区切って、1 行ずつ縦に並べたのが上の表である。

「a」という文字は 0060 行の 1 列目にあるので、「a」の文字位置は 0061 である。

日本語のひらがなが置かれている辺りの表は次の通りである。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
3040		ぁ	あ	ぃ	い	ぅ	う	ぇ	え	ぉ	お	か	が	き	ぎ	く
3050	ぐ	け	げ	こ	ご	さ	ざ	し	じ	す	ず	せ	ぜ	そ	ぞ	た
3060	だ	ち	ぢ	っ	つ	づ	て	で	と	ど	な	に	ぬ	ね	の	は
3070	ば	ぱ	ひ	び	ぴ	ふ	ぶ	ぷ	へ	べ	ぺ	ほ	ぼ	ぽ	ま	み
3080	む	め	も	ゃ	や	ゅ	ゆ	ょ	よ	ら	り	る	れ	ろ	ゎ	わ
3090	ゐ	ゑ	を	ん	ゔ	ゕ	ゖ			゙	゚	゛	゜	ゝ	ゞ	ゟ

各文字の文字位置のことを Unicode の規約ではコードポイントと呼んでいる。

上の表ではコードポイントを 16 ビットで表現しているので Unicode の文字の数の上限は 65536 と考えるかもしれないが、実際は Unicode で表現できる文字の数の上限は 0x10FFFF である。コードポイントを表現するビット数を増やせば良いのである。実際、すべての漢字や世界中の文字を表現するには、65536 では足りない。

コードポイントをメモリ上の 2 進法表現に対応づける方法（エンコーディングという）にはいくつかある。ここでは utf-16 と utf-8 の 2 つを紹介する。

（1）utf-16

utf-16 は Unicode の文字のコードポイントを使い、次のように表わす。

① コードポイントが 0x0000-0xFFFF の範囲の文字の場合
コードポイントをメモリ上の 16 ビットの 2 進法表現とする。

例 文字列 「Alpha」 の utf-16 による 2 進法表現は、
0041 006B 0070 0068 0061
である。
文字列 「あるふぁ」 の utf-16 による 2 進法表現は、
3041 308B 3075 3041

である。

② コードポイントが 0x010000-0x10FFFF の範囲の文字の場合

文字が割り当てられていないコードポイントの２つの領域：

0xD800-0xDBFF（ハイサロゲートという）

0xDC00-0xDFFF（ローサロゲートという）

を用いる。

範囲 0x010000-0x10FFFF の 1 つのコードポイントを X としたとき、

H = (X-0x10000) / 0x400 + 0xD800

L = (X-0x10000)mod(0x400) + 0xDC00

を求め、H（16 ビット）と L（16 ビット）のペア HL をメモリ上の 32 ビットの 2 進法表現とする。

上の式はハイサロゲートとローサロゲートの領域に、0x010000-0x10FFFF の範囲のコードポイントをマッピングしているのである。

（２） utf-8

Unicode 文字集合の先頭にある文字群は英語やプログラムで使われる ASCII（American Standard Code for Information Interchange,アスキー）の 0x00-0x7F と同じである。utf-16 ではこれらの文字を表わすのに必ず 0x00 を伴う。例えば、文字「a」の 2 進法表現は 0x0061 である。この 0x00 はメモリの無駄遣いとなるのである。

この問題を解消するために utf-8 の考え方が生まれた。utf-8 では ASCII 対応の文字を 1 バイトで表わし、他の文字をマルチバイトで表わす。

1 バイト目は ASCII と同じか、マルチバイトの開始バイトである。次表は 1 バイト目と 2 バイト目以降のバイトの値の範囲を示している。

1 バイト目	0x00-0x7F	ASCII と同じ文字
	0xC0-0xDF	2 バイト文字の開始バイト
	0xE0-0xEF	3 バイト文字の開始バイト
	0xF0-0xF7	4 バイト文字の開始バイト
	0xF8-0xFB	5 バイト文字の開始バイト
	0xFC-0xFD	6 バイト文字の開始バイト
2 バイト目以降	0x80-0xBF	

日本語の文字は、utf-8 では 3 バイトになるので、utf-8 で ASCII と日本語文字とを同時に扱う場合は別にして、メモリを節約するために 2 バイトで済む utf-16 を併用した方が良い。

utf-8 による英字「a」の 2 進法表現は 0x61 である。utf-8 によるひらがな「あ」の 2 進法表現は 0xE38182（3 バイト）である。utf-8 のすべての文字の 2 進法表現については utf-8 コード表（公開されている）を参照されたい。

これまで述べたように、Unicode、utf-8、utf-16 により、世界中の文字が 1 つの文字コードとして統一的に扱うことが可能となった。

今日、多くのソフトウェアは Unicode、utf-8、utf-16 を採用しているので、ソフトウェア間の情報交換は容易になった。

２．３　数値

数値は、ものや現象、活動を測るために重要な役割を果たしている。数値が使われる身近な例として、

・長さ　・距離　・縦／横／高さ　・重さ　・量
・面積　・時間　・速さ　・温度　・湿度
・気圧　・個数　・音の高さ／大きさ・金額　・割合
・確率　・etc.

などがある。実際はここに書きつくせないほどたくさんの対象が数値で測られている。

数値をコンピュータで扱うために数字を使うが、数字は文字であり、たし算やひき算を行うコンピュータのマシン命令の処理対象にはできない。

例えば、富士山の高さ 3776mの utf-8 コードの 10 進数字列 3776 をコンピュータに入力すると、次のような 10 進数字の 2 進法表現の並びになっている。

0x33　0x37　0x37　0x36

これらの 10 進数字列（文字列）をコンピュータの処理対象にするには、数値に変換する必要がある。その方法として次のやり方がある。

① 数字の数値部分だけを取り出す。

3 → 0x33 (0b00110011) → 0x3 (0b0011)
7 → 0x37 (0b00110111) → 0x7 (0b0111)
7 → 0x37 (0b00110111) → 0x7 (0b0111)
6 → 0x36 (0b00110110) → 0x6 (0b0110)

この操作の結果、2 進法表記の 10 進数が得られた。

3776 → 0b0011　0b0111　0b0111　0b0111　0b0110

② ①の結果の 10 進数を次のようにして 2 進法表現に変換する。

$0b0011\times(0b1010)^3+0b0111\times(0b1010)^2+0b0111\times0b1010+0b0110$
$= 0b111011000000$

上記の②は、2 進数演算を用いたが、10 進数演算を用いた変換については本節の（１）の「（ｂ）10 進数から 2 進数への変換」を参照されたい。

数値のデータ表現の説明に入る前に、数学で使われている数について説明しておく必要がある。というのは、コンピュータでは数学で使われる数のほんの 1 部しか扱えないことが分かるからである。

数学で使われる数（実数）は次の 4 種である。

自然数　：　1，2，3，･･･　など。ものの順位またはものの個数を示すために用いられる。
整数　：　0，±1，±2，±3，･･･など。自然数は正の整数である。
有理数　：　0 および a / b。ただし、a、b は自然数。
無理数　：　有理数以外の実数。

例

$\sqrt{2}$ = 1.4142135･･･
e = 2.718281828･･･
π = 3.1415926535･･･

有理数を 2 進法で表わせば、分母が 2 のベキになるものの他は循環 2 進数になる。

例

$\frac{5}{8}$ = 0.101

$\frac{2}{3}$ = 0.101010･･･

無理数を 10 進法で表わすならば、小数以下が無限に続くが循環することはない。

このことから整数を除く実数の 2 進数は殆どの場合小数以下が無限に続くこと分かる。

コンピュータが無限の広がりのあるメモリをもつとしても、無理数の 1 つを記憶させると、それだけでメモリは一杯になってしまう。

そこで、$\sqrt{2}$や e、π などの無理数をコンピュータで扱う場合、計算の精度に合わせて必要な有限桁の有理数を対応させて扱うことにする。

例

$\sqrt{2}$ → 1.4142
e → 2.7183
π → 3.14156

循環小数や小数以下が長い有理数をコンピュータで扱うとき、必要とされる精度の桁数に合わせた有理数に対応させる。

10 進小数を 2 進小数に変換したとき、殆どの場合、小数点以下が無限桁になるので、必要とされる精度の桁数に合わせた有理数に対応させる。

このように、無限桁あるいは桁数の大きい小数を、適切な有限桁に抑えることにより、真の値との差異が生じることになる。この差異を誤差と呼んでいる。

数学で使われる整数を除く実数は、コンピュータではそのまま扱えず、有限桁の有理数で近似する必要があることが分かった。

科学の世界では、絶対値が非常に大きな数値や非常に小さな数値を扱う場合がある。
たとえば、

1234567890000000000

0.0000000000123456789

のような数値である。
この 2 つとも、00･･･0 の部分は冗長である。
これらはそれぞれ冗長さを避けて次のように表わすのが一般的である。

1.23456789×10^{18}

1.23456789×10^{-11}

このような表わし方を指数表示という。指数表示はあとで述べる 2 進浮動小数点数で使うのであらかじめ紹介しておいた。

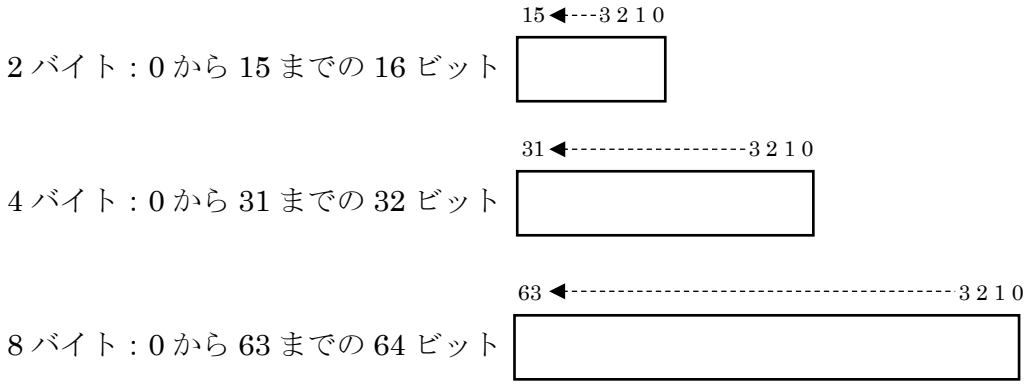
それでは、コンピュータが処理する数値のデータ表現について話を進めよう。
数値のデータ表現には次の 3 つがある。

2 進固定小数点数
2 進浮動小数点数
10 進固定小数点数

これらの数値のデータ表現はそれぞれ、コンピュータの 2 進固定小数点用のマシン命令、2 進浮動小数点用のマシン命令、10 進固定小数点用のマシン命令で扱われる。

（1）2 進固定小数点数

2 進固定小数点数は、連続する複数バイト（一般には 2 バイト、4 バイト、8 バイト）を連続するビット列と見なし、数値を表現する。



整数をこのビット列で表現する場合、小数点の位置は右端のビットの右にあるものとする。

2 進固定小数点数のビット列表現を理解するために、2 の補数、10 進整数から 2 進整数への変換、10 進小数から 2 進小数への変換、小数を含む 2 進固定小数点数のビット列について述べる。

（a）2 の補数

2 進固定小数点数には明示的な符号ビットはないが、左端のビットが 0 のとき数値は正、1 のとき数値は負である。

この奇妙な言い回しは 2 進固定小数点数の演算で用いられる 2 の補数の考え方から来ている。

2 進法表現のビット列のビット数を、一定の大きさ n としたき、2^n に対して、n ビットで表わされる 2 つの数 a と b が $a+b=2^n$ となるとき、a と b は互いに補数であるという。

補数について、理解を容易にするため、8 ビットのビット列を用いて説明する。

8 ビットのビット列で表わされる数は、10 進数表記の 0 から 255 までの数であるが、補数の定義により、10 進数表記の 0 から 127 までの数と、-1 から-128 までの数に分けることができる。

0 から 127 までの数は 8 ビットのうち、左端ビットを 0 とする残り 7 ビットで表わされる数である。これらの数の 2^8 に対する補数を求めると、左端ビットが 1 となる数が得られる。ただし、0b00000000 の補数は 0b100000000（2^8）、0b10000000 の補数は 0b10000000 とする。表 2-1 はこれらの補数関係を示している。

表 2 - 1　補数関係

2 進数表記	10 進数表記	2 進数表記	10 進数表記
0b00000000	0		
0b00000001	1	0b11111111	-1
0b00000010	2	0b11111110	-2
…	…	…	…
0b01111111	127	0b10000001	-127
		0b10000000	-128

実際に 10 進数表記の 1 と-1 の 2 進数表記の加算を行ってみると、

$$0b00000001 \quad + \quad 0b11111111 \quad = \quad 0b100000000 \quad = \quad 2^8$$

となる。結果は 9 ビットになるが、オーバーフローした分は無視し、8 ビットだけに注目すると 0b00000000 である。これは 10 進数表記の 0 である。

このように、補数を使えば、符号ビットをもたなくても正負の値を考慮した計算が可能となる。

16 ビットのビット列を使用した場合も同様である。
たとえば、10 進数表記の 2 と-2 に対する 2 進数表記は表 2-2 の通りである。

表 2-2　16 ビットの補数表示

10 進数表記	8 ビットの 2 進数表記	16 ビットの 2 進数表記
2	0b00000010	0b0000000000000010
-2	0b11111110	0b1111111111111110

2 進固定小数点数のバイト数にかかわらず、符号ビットと見なされるビットが左側に連続して並ぶ。

最後に、補数の求め方を紹介する。

補数の定義は「2 進法表現のビット列のビット数を、一定の大きさ n としたき、2^n に対して、n ビットで表わされる 2 つの数 a と b が a＋b＝2^n となるとき、a と b は互いに補数である」であった。

この定義に従うと、8 ビットのビット列の数 0b00000101（10 進数の 5）の補数Ｘは次の計算で求めることができる。

0b00000101 + X = 2^8 = 0b100000000

X = 0b100000000 - 0b00000101

```
    100000000
-)  00000101
    --------
    11111011
```

従って、X は 0b11111011 である。

もっと簡単な補数の求め方は、元の 2 進法表現の各ビットを反転して（0→1、1→0）、その結果に 1 を足すという方法である。

このやり方で 0b00000101（10 進数表記の 5）の補数を求めると次の通りである。

0b00000101 ← この各ビットを反転
0b11111010 ← これに 0b1 を加算
0b11111011 ← 求める補数

（b）10 進数から 2 進数への変換

わたしたちは日頃、数値を表わすために 10 進数を使用している。2 進固定小数点数の値についても 10 進数を用いることが多い。そのため、10 進数から 2 進数への変換が必要となる。

10 進数が整数か、小数かで変換方法は異なる。

10 進整数から 2 進整数への変換

10 進整数 N が与えられたとき、数の値は進法に依存しないので N の 2 進法表記を、

$$N = b_{n-1}b_{n-2}\cdots b_1b_0 \quad bi = 0 \text{ or } 1 \quad (i=0, 1, \cdots, n-1)$$

とする。

この各 bi （i=0, 1, … , n-1）を求めることができれば、10 進整数 N の 2 進整数への変換ができたことになる。

N は、

$$b_{n-1}\times 2^{n-1} + b_{n-2}\times 2^{n-2} + \cdots + b_0\times 2^0$$

で与えられるので、この式に 2^{-1} をかける（つまり 2 で割る）と、

$$\underbrace{b_{n-1}\times 2^{n-2} + b_{n-2}\times 2n^{n-3} + \cdots b_1\times 2^0}_{\text{商}} + \underbrace{b_0}_{\text{余り}}$$

となる。

この商に 2^{-1} をかけると、

$$\underbrace{b_{n-1}\times 2^{n-3} + b_{n-2}\times 2n^{n-4} + \cdots b_2\times 2^0}_{\text{商}} + \underbrace{b_1}_{\text{余り}}$$

となる。

このようにして求められる商に対して、商が 0 になるまで繰り返すと、

余り：b_{n-1}、b_{n-2}、… 、b_1、b_0

が求められる。

これらの余りを右から左へ並べると、10 進整数 N の 2 進法表記となる。

例　10 進整数 50 の 2 進整数を求めよ。

		余り	
2）50	…	0	← 右端
2）25	…	1	
2）12	…	0	
2）6	…	0	
2）3	…	1	
2）1	…	1	← 左端
0			

整数 50 の 2 進整数は 0b110010 である。

10 進小数から 2 進小数への変換

10 進小数 N'が与えられたとき、N'の 2 進法表記を、

$$N' = b_{-1}b_{-2}\cdots b_{-n}\cdots \quad b_{-i} = 0 \text{ or } 1 \quad (i=1, 2, \cdots, n, \cdots)$$

とする。

この各 b_{-i} （i=1, 2, … , n, …）を求めることができれば、10 進小数 N'の 2 進小数への変換ができたことになる。

このN'の 10 進数は、

$$b_{-1}\times 2^{-1} + b_{-2}\times 2^{-2} + \cdots + b_{-n}\times 2^{-n} + \cdots$$

で与えられるので、この式に 2 をかけると、

$$\underbrace{b_{-1}}_{\text{整数部}} + \underbrace{b_{-2}\times 2^{-1} + b_{3}\times 2^{-2} + \cdots + b_{-n}\times 2^{-n+1} + \cdots}_{\text{小数部}}$$

となる。

この小数部に 2 をかけると、

$$\underbrace{b_{-2}}_{\text{整数部}} + \underbrace{b_{-3}\times 2^{-1} + b_{-4}\times 2^{-2} + \cdots + b_{-n}\times 2^{-n+2} + \cdots}_{\text{小数部}}$$

となる。

この操作を小数部が 0 になるか、2 進固定小数点数の小数部に与えられたビット列の巾になるまで繰り返すと、整数部の並び、

$$b_{-1}, b_{-2}, \cdots, b_{-n}$$

が得られる。

これらの整数部を左から右へ並べると、10 進小数 N'の 2 進法表記となる。

注意　上記の 2 のかけ算において、桁上がりすると整数部は 1 になり、桁上がりしないと整数部は変化しない。つまり 0 であることに注意。

例　10 進小数 0.43 の 2 進小数を求めよ。

	整数部	小数部		
	0 .	4 3		
	×	2		
左端 →	0 .	8 6		
	×	2		
	1 .	7 2	0.25	
	×	2		
	1 .	4 4	0.125	
	×	2		
右端 →	0 .	8 8		0.375 (4 ビットの合計)
	×	2		
	1 .	7 6	0.03125	
	×	2		
	1 .	5 2	0.015625	
	×	2		
	1 .	0 4	0.0078125	
	×	2		
	0 .	0 8		0.4296875 (8 ビットの合計)

2 進小数点数の小数部に与えられたビット列の巾を 4 とすると、

0 . 4 3 ≒ 0b0110

である。

ビット列の巾を 8 とすると、

0 . 4 3 ≒ 0b0110111

である。

この例で、0.43 の 4 ビットの 2 進小数を 10 進数にすると 0.375 で、8 ビットでは 0.4296875 である。4 ビットでは誤差が大きすぎることが分かる。

小数を含む 2 進固定小数点数のビット列

この場合も、理解を容易にするため、8 ビットのビット列を用いて説明する。
いま、小数点は左から 4 つ目のビットの右にあるものとする。

これを、

$b_3b_2b_1b_0.b_{-1}b_{-2}b_{-3}b_{-4}$　　$b_i = 0 \text{ or } 1 \quad (i=3,2,1,0,-1,-2,-3,-4)$

とすると、この数の 10 進数は、

$$\underbrace{b_3\times2^3 + b_2\times2^2 + b_1\times2^1+b_0\times2^0}_{\text{整数部}} + \underbrace{b_{-1}\times2^{-1}+b_{-2}\times2^{-2}+b_{-3}\times2^{-3} + b_{-4}2^{-4}}_{\text{小数部}}$$

で与えられる。

整数部については、先の 10 進整数から 2 進整数への変換を、小数部について 10 進小数から 2 進小数への変換を用いて 2 進数表記を求める。

表 2-3 はこのようにして求めた 2 進数表記と 10 進数表記を併記している。

表 2-3　$b_3b_2b_1b_0.b_{-1}b_{-2}b_{-3}b_{-4}$ の 2 進数表記と 10 進数表記

2 進数表記	10 進数表記	2 進数表記	10 進数表記
0000.0000	0.0		
0000.0001	0.0625	1111.1111	-0.625
0000.0010	0.125	1111.1110	-0.125
0000.0011	0.1875	1111.1101	-0.1875
0000.0100	0.25	1111.1100	-0.25
0000.0101	0.3125	1111.1011	-0.3125
0000.0110	0.375	1111.1010	-0.375
0000.0111	0.4375	1111.1001	-0.4375
0000.1000	0.5	1111.1000	-0.5
0000.1001	0.5625	1111.0111	-0.5625
0000.1010	0.625	1111.0110	-0.625
0000.1011	0.6875	1111.0101	-0.6875
0000.1100	0.75	1111.0100	-0.75
0000.1101	0.8125	1111.0011	-0.8125
0000.1110	0.875	1111.0010	-0.875
0000.1111	0.9375	1111.0001	-0.9375
0001.0000	1.0	1111.0000	-1.0
…	…	…	…
0001.1111	1.9375	1110.0001	-1.9375
…	…	…	…
0111.0000	7.0	1001.0000	-7.0
…	…	…	…
0111.1111	7.9375	1000.0001	-7.9375
		1000.0000	-8.0

例　10 進数 -2.375 を 8 ビットの 2 進固定小数点数としたビット列を示せ。ここに、小数点は左から 4 つ目のビットの右にあるものとする。

まず、2.375 の 2 進数表記を求める。
整数部 2 の 2 進数表記は 0010 である。
小数部 0.375 の 2 進数表記は 0110 である。
従って、2.375 の 2 進数表記は 0010.0110 である。

次に、先に述べた補数の簡単な求め方を使って、この 2 進数表記の補数を求める。

```
0010.0110  ←  この各ビットを反転
1101.1001  ←  これに 0b1 を加算
1101.1010  ←  求める補数
```

こたえ　1101.1010

小数点が左から 4 つ目のビットの右にあるという約束で計算すれば、小数を含む 2 進固定小数点数のたし算やひき算、かけ算やわり算ができる。10 進数で行なっている計算と変わりはない。

（２）２進浮動小数点数

絶対値が非常に大きな数値や、非常に小さな数値を扱う場合、指数表示を使うのが一般的であると先に述べた。

たとえば、

1234567890000000000

0.0000000000123456789

に対して、それぞれの指数表示を、

1.23456789×10^{18}

1.23456789×10^{-11}

と書いた。

指数表示では、小数点の位置を固定して、もとの数から小数点のずれ分を指数で調整していることが分かる。

２進浮動小数点数の数値表現はこの考え方を反映している。

２進浮動小数点数は２進数なので、上の 10 進数による指数表示を２進数による指数表示に変換しよう。

10 進数の 1234567890000000000 は、２進数では、

1000100100010000100001111010001110110100011011011010000000000

である。

これを２進数の指数表示にすると、

$1.0001001000100001000011110100011101101000110110110100000000000\times2^{60}$ （☆1）である。

1.xxx…×2^{xx} のように、整数部が１の 1.xxx…という表現にすることを正規化と呼んでいる。

もう一つの小さな数値、10 進数の 0.0000000000123456789 は、２進数では循環小数になっているので、100 桁まで求めると、

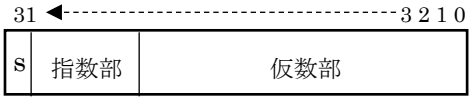
0.00000000000000000000000000000000000011011001001011111111111101011000101011010000110001000100101011010

である。

これを 2 進数の正規化された指数表示にすると、
1.10110010010111111111110101100010101101000011000100010010101101
0×2^{-37}（☆2）である。

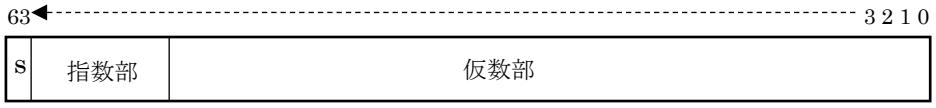
2 進浮動小数点数は、連続する複数バイト（一般には 4 バイト、8 バイト）を連続するビット列と見なして数値を表現するが、上に述べた 2 進数の正規化された指数表示を反映した形式を採っている。ここでは、広く採用されている標準規格 IEEE754 の形式を紹介する。

4 バイト（単精度）：0 から 31 までの 32 ビット



符号（S） 1 ビット
指数部 8 ビット
仮数部 23 ビット

8 バイト（倍精度）：0 から 63 までの 64 ビット



符号（S） 1 ビット
指数部 11 ビット
仮数部 52 ビット

S は 2 進浮動小数点数全体の符号である。

S = 0 : 正
S = 1 : 負

この形式の仮数部には、（☆1）と（☆2）の 1.xxx…の xxx…の部分を左から順に格納する。仮数部のビット数に満たなければ 0 で補い、超えれば切り捨てる。

実際の小数は 0.1xxx…であるが、1 は常に存在するため、この形式には含めない。実質の精度は、単精度では 24 ビット、倍精度では 53 ビットを実現していることになる。

指数部には 2 のベキ数である 60 や-37 を格納すれば出来上がりとしたいところだが、指数部に負の値が入り、2 進浮動小数点数同士の比較が複雑になってしまう。

そこで導入されたのがバイアスという考え方である。指数部が常に正となるように調整して格納するのである。

単精度ではバイアスとして 127 をとり、60 や-37 に 127 を加えた 187 や 90 を指数部に格納する。

単精度で表現できる指数の 2 のベキ数の範囲は-126～127 である。従って、指数部の値は 1～254 である。

単精度でのバイアス 127、指数の 2 のベキ数、指数部の値は次のようにして決まる。

① 単精度の表現形式の指数部は 8 ビットであり、表現できる数値は 0～256 である。ただし、0 と 255 は特殊な意味をもたせるため 1～254 を使う。

② 1～254 の中間の値である 127 を指数の 2 のベキ数の 0 に対応させ、127 より大きい値を正に、127 より小さい値を負に対応させる。次表を参照。

指数の 2 のベキ数		指数部の値
正の数		
127	----------	254
…		…
1	----------	128
ゼロ		
0	----------	127
負の数		
-1	----------	126
…		…
-126	----------	1

このようにして、指数部は「正の数」 > 0 > 「負の数」という関係を維持し、2 進浮動小数点数同士の大小比較は単純化される。

2 進浮動小数点数を解釈するとき（コンピュータのマシン命令が実行するとき）、指数部の値からバイアスを減算して、実際の指数を求める。

倍精度の場合はバイアスとして 1023 をとり、単精度と同様に表現する。

（３）10 進固定小数点数

お金の計算に、2 進固定小数点数や 2 進浮動小数点数を使うと不都合な結果を招くことがある。

たとえば、年利 2.3%で 1 億円の借入の金利計算を行うとする。

この計算を 2 進数で行なうと、次のデータを使用することになる。

	10 進数	2 進数
預金残高	100000000	101111101011110000100000000
年利	0.023	0.000001011110001101010011111101…

0.023 の 2 進法表現は循環小数となるため、下位の桁を切り捨てる。そのため、正確な年利を表わすことができない。

2 進法でどの程度の誤差を生じるのか、次の Python プログラムで確認してみよう。

```
sum = 0.0
i = 0
while (i < 100000000):
    sum += 0. 023
    i += 1
print(sum)
```

このプログラムは 2 進浮動小数点数 sum に 0.023 を 1 億回加算している。その結果の sum の値は、

2300000.003225438

であり、真の値 2300000 との間に、

0.003225438

の誤差が生じていることが分かる。

この程度の誤差なら大したことはないではないかと考えてはならない。金利や金額が大きくなれば誤差は無視できなくなるのである。

正確な計算を行うため、10 進固定小数点数を導入する。

10 進固定小数点数では、各桁の 10 進数は 4 ビットからなる 2 進数を使って表 2-4 のように表現する。この表現法を 2 進化 10 進数（BCD：Binary Coded Decimal）という。BCD コードともいう。

表 2-4　BCD コード

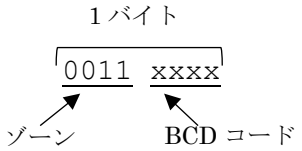
10 進数	BCD コード
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

10 進固定小数点数は、一般に次の 2 つの形式を採用している。

- アンパック形式
- パック形式

（a）アンパック形式

アンパック形式は 10 進数の各桁の BCD コードを 1 バイトに対応させる。



ここに、バイトの右 4 ビットに BCD コードを配置し、左 4 ビットに（ゾーンという）には 0b0011 を配置する。

このようなゾーンつき BCD コードのバイトを使って 10 進数 1234 を、

0011	0001	0011	0010	0011	0011	1100	0100
	1		2		3	+	4

のように表わす。右端のバイトのゾーンには符号を置く。

符号は、

```
+  :  1100
-  :  1101
```

である。

ゾーンの 0b0011 や符号の 0b1100、0b1101 は実際のコンピュータでは異なることがあるので注意されたい。

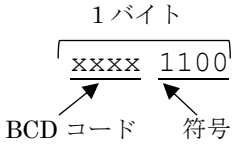
アンパック形式の 10 進固定小数点数の桁数は任意である。

コンピュータが utf-8 を採用しているなら、符号を 0b0011 に書き換えると、この 10 進固定小数点数は utf-8 の文字列、すなわち 10 進数字列となる。

（b）パック形式

アンパック形式はゾーン部分が冗長である。数値を文字列に変換する必要がないなら、パック形式を用いる。

パック形式ではゾーン部分を使用せずに、2 つの BCD コードを 1 バイトに収める。この場合、右端バイトは、



のように、左 4 ビットに BCD コードを、右 4 ビットに符号を配置する。
たとえば、10 進数 1234 はパック形式で次のように表わされる。

0000	0001	0010	0011	0100	1100
0	1	2	3	4	+

パック形式の 10 進固定小数点数の桁数は任意である。

アンパック形式、パック形式共に小数点の位置は、わたしたちが小数点を含む 10 進数の四則演算を行うときと同様に、任意の位置にあるものと想定できる。

10 進固定小数点数を計算するには、コンピュータが対応するマシン命令を備えているならば、コンピュータで行なう。コンピュータがマシン命令を用意していないならば、ソフトウェアでシミュレーションするという方法もある。

２．４　データ構造

ここまで、文字列、数値というスカラー・データを見てきたが、スカラー・データだけを使ったプログラムで表現できる世界は限られている。複数のスカラー・データを組み合わせて一つにまとめると、プログラムで表現できる世界が広がるのである。

この組み合わせによるデータの表現方法のことをデータ構造という。

データ構造にはプログラムの特定の動作を伴うものがあるが、まだプログラムの表現方法を学んでいないので、ことばや図を使って説明する。

データ構造の代表的なものとして、配列、構造体、待ち行列、スタック、連結リスト、辞書を取り上げる。

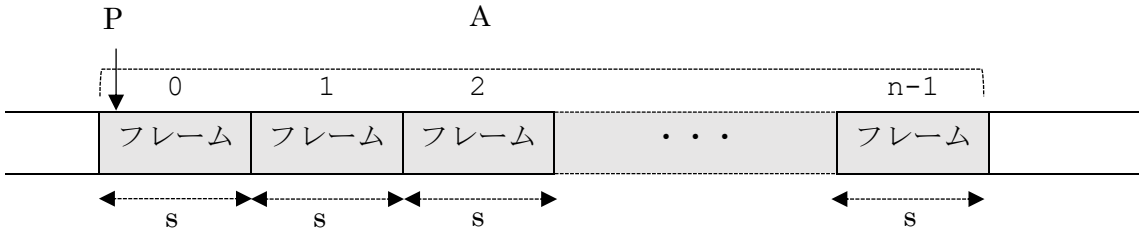
データの構造化を可能とする重要な手段はメモリ・アドレスである。これについては、「（１）配列とポインタ」で詳しく述べ、あとにつづくデータ構造の説明では配列やポインタを使用する。

データ構造はメモリ上のバイト列に注目するのでスカラー・データにはこだわらない。そこで、メモリ上の任意サイズのバイト列をフレームと呼ぶことにする。

（１）配列とポインタ

複数のフレームの並びを配列という。

サイズが等しい n 個のフレームが図 2-2 のように隣接して配置されているとする。



s　：　フレームのサイズ

P　：　先頭フレームの左端バイトのアドレス

図 2-2　配列のフレーム

このフレームの並びは配列である。

このとき、0 番目から数えた i 番目（0≦i≦n-1）のフレームの左端バイトのアドレスは、

P + s×i

で求められる。

この配列に名前を与え、それを A とし、配列 A の i 番目の要素を A［i］と表わすことにすると、配列 A の i 番目の要素は A［i］でアクセスできる。A［i］をアクセスするとき、A[i]に対応する P + s×i を使うのである。

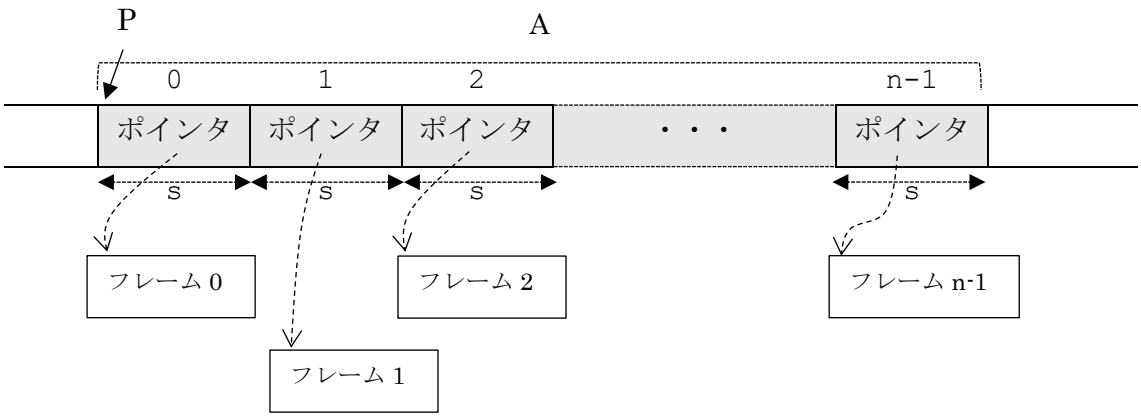
配列の要素を特定するために、メモリ・アドレスではなく要素番号が使われることに注意されたい。

配列の要素のアクセスのようにアドレス情報を隠すのでなく、明示的にアドレス情報を使用することもある。あるアドレス・データ（メモリ・アドレス自身をデータとして扱う）の名前を P とするとき、P の値のアドレスにあるフレーム A を、

P ー> A

と表わす。アドレス・データのことをポインタ・データあるいは単にポインタという。

ポインタ・データを導入したところで、サイズが異なる複数のフレームの配列を考えよう。この場合も、配列の要素を特定するために要素番号を使いたい。それには、各フレームをメモリ上のどこかに置き、そのアドレスを値に持つポインタ・データを隣接するように配置すれば（図 2-3 参照）、すべてのポインタ・データのサイズは等しいので、各要素を要素番号でアクセスできる。



A ： 配列の名前

s ： ポインタのサイズ
P ： 先頭のポインタの左端バイトのアドレス

図 2-3 要素サイズの異なる配列

配列 A の i 番目の要素を A［i］としたとき、この要素のアクセスは、

（P + s×i） ―> A´

で行なう。

ここに、

- （P + s×i）はアドレス：P+s×i にあるポインタ・データ
- A´はポインタ・データが示すアドレスにあるフレーム

である。

このように、ポインタは重要な役割を果たしている。

ポインタは連結リストや辞書でも効果的に用いられている。一方、ポインタはメモリ上の任意のアドレスを値に持つことができるので、思わぬところでデータ破壊などのトラブルを引き起こすことがある。最近のプログラミング言語はポインタ・データを抑制的に扱うように工夫している。

（２）構造体

構造体について、学校の学籍簿を例にして説明しよう。
学校では、学生１人ひとりについて、次のような情報をもっているものとする。

- 学籍番号
- 氏名
- ふりがな
- 生年月日
- 住所

これらの情報を一括りにしたものが構造体である。関係のある情報をまとめることにより、新たな意味のある情報となる。この例では、“学籍簿”は構造体の名前になる。

構造体をメモリ上に配置する場合、各要素はスカラー・データ、ポインタ・データ、配列、他の構造体が許される。

構造体“学籍簿”のメモリ上の配置は図 2-4 の通りである。

学籍簿

	学籍番号	氏名	ふりがな	生年月日	住所	

※　各要素はバイト列である

図 2-4　構造体の例

構造体はメモリだけでなく記憶装置上でも利用される。たとえば、次のような使い方がある。

① リレーショナル・データベースの登録の単位は構造体である。
② ディスク装置に記憶させる単位となるレコードは構造体である。
③ スプレッド・シートの各行は構造体である。
④ XML、JSON、CSV などを使用して、構造体を文字列で表現することができる。

（３）待ち行列（Queue、キュー）

待ち行列の身近な例は、バス停、タクシー乗場、空港の手荷物検査場、評判のラーメン店などにできる行列である。

コンピュータの処理においても同様の待ち行列がある。

たとえば、プリンタは処理速度が遅い装置であるため、複数のプログラムから印刷要求があるとき、他のプログラムからの印刷処理の終了を待っていては、コンピュータの処理能力が生かせない。そこで印刷データはスプーラと呼ばれる待ち行列に保存し、プログラムはその他の処理を続行する。

スプーラに保存された印刷データはプログラムの実行と切り離されて順番にプリンタに印刷される。

このような待ち行列は配列を使って表現する。

例として、図 2-5 に示すような配列を考えよう。

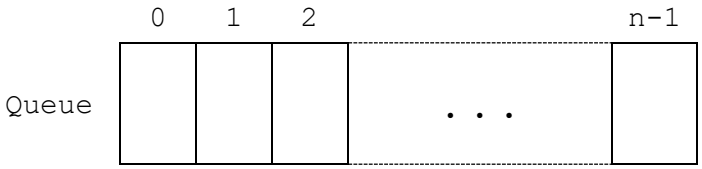
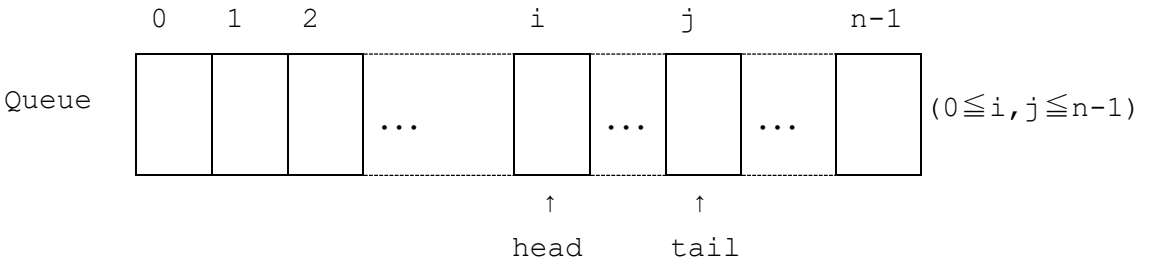


図 2-5　キュー

Queue は待ち行列（配列）の名前で、最大 n 個までのデータが格納できる。

タクシー乗場で目撃される待ち行列は先頭の何人かが乗りこめば待ち行列の後ろの人が前に移動するが、コンピュータではデータの移動はコンピュータ・パワーの無駄遣いになる。そこで、待ち行列に登録されている先頭と末尾の要素番号を変化させることにする。

先頭と末尾の番号をそれぞれデータ head、tail（共に 2 進固定小数点数）に持たせることにする。



Queue にデータを格納することをエンキュー、Queue からデータをとり出すことをデキューという。エンキュー、デキューが実施される度にそれぞれ tail、head は 1 だけ加算される。head も tail も加算される一方であるから、Queue の最後の要素の番号 n-1 を超えてしまう。

そこで、Queue の n-1 番目の次の要素の番号を 0 にして、Queue が環状になるようにする。

これは head、tail にそれぞれ+1 する度に、n で割った余りを head、tail にセットすれば実現できる。この計算を次のような式で表わす。

```
head  ←  head mod n
tail  ←  tail mod n
```

Queue に対するエンキューの回数がデキューの回数よりも多いとき Queue に格納されるデータの個数が Queue の要素数 n を超えてしまうことがある。これをオーバーフローというが、オーバーフローをチェックしてエンキューを抑止しなければ前の登録データが破壊されてしまう。

オーバーフローのチェックは次のように行う。

エンキューで tail が+1 され、その tail の値が head の値と等しくなったとき、オーバーフローしたことが分かる。しかし、tail と head の値は、Queue に格納されているデータがない“空”の場合でも等しくなる。そこで、tail と head の値が等しいとき Queue は“空”であるとし、オーバーフローの場合は、+1 された tail の値が head の 1 つ手前の要素の番号と等しいときとすることによってこの問題を回避する。

エンキュー、デキューの操作を整理すると次の通りである。

初期状態では head、tail の値は共に 0 にしておく。

エンキュー（enqueue）

1　tail に 1 を加算
2　tail　←　tail mod n
3　(tail+1) mod n の値が head に等しいとき（オーバーフロー）、適切なメッセージを出力して処理を終了。等しくなければ 4 へ
4　Queue［tail］にデータを格納

デキュー（dequeue）

1　　head の値が tail と等しいとき（Queue は“空”、アンダーフロー）、適切なメッセージを出力して処理を終了。等しくなければ 2 へ
2　　Queue［head］のデータを取り出す
3　　head に 1 を加算
4　　head ← head mod n

（4）スタック（Stack）

スタックとは積み重ねることである。身近な例は、本屋の本の平積みである。店では売れ筋の本を平積みにし、客は上から順に取っていく。売れるとさらに上に本を積み重ねる。この平積みがスタックである。

コンピュータ処理の場合、最初に入れたものが最後に取り出されるというスタックのしくみが役立つのである。

スタックは、待ち行列と同じように、n 個の要素からなる配列を使う。配列の名前を Stack とする。

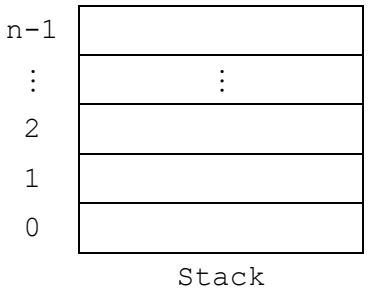


図 2-6　スタック

図 2-6 は配列を縦に描いて、積み重ねる感じを表わしている。底には 0 番目の要素があり、最上部には n-1 番目の要素がある。Stack にデータを格納するときは、底から順に積み上げ、取り出しは積み上がった最上部から底に向けて順に行う。

Stack にデータを格納することをプッシュ（push）といい、Stack からデータを取り出すことをポップ（pop）という。

スタックに格納されているデータの最上部の要素の 1 つ上の要素の要素番号を top（2 進固定小数点数）がもつものとする。このようなデータをスタック・インデックスという。Stack が“空”のとき、スタック・インデックス top の値は 0 である。

プッシュによりデータが Stack に積み上げられ、top は 1 だけ加算される。ポップにより Stack の積み上げられた最上部のデータが取り出され、top は 1 だけ減算される。

Stack の要素数は n なので、それを超える積み上げはできない。top の値が n のときにプッシュがあると Stack はオーバーフローする。

また、top の値が 0 のときにはポップによるデータの取り出しができない、つまりアンダーフローする。

プッシュ、ポップの操作を整理すると次の通りである。
初期状態では Stack は“空”なので、top の値は 0 にしておく。

プッシュ（push）

1　top の値が n のとき（オーバーフロー）、適切なメッセージを出力して処理を終了。n でなければ 2 へ
2　Stack［top］にデータを格納
3　top に 1 を加算

ポップ（pop）

1　top の値が 0 のとき（アンダーフロー）、適切なメッセージを出力して処理を終了。0 でなければ 2 へ
2　top から 1 を減算
3　Stack［top］のデータを取り出す

スタックは単純なしくみだが、コンピュータ処理で重要な効果を発揮するのである。その例を２つ示そう。ひとつは逆ポーランド記法であり、もうひとつは関数呼出しである。

例１　逆ポーランド記法

逆ポーランド記法とは次の算術式、

（2 ＋ 5）×（3 ＋ 7）

を

2 5 ＋ 3 7 ＋ ×

と表記し、左から右へ読み取りながら計算を進め、後戻りすることなく計算ができる記法である。

『2 に 5 をたした（+）ものと、3 に 7 をたした（+）ものとをかける（×）』という具合である。

ポーランド人のヤン・ウカシェヴィッチが考案したポーランド記法では上の算術式は、

× ＋ 2 5 ＋ 3 7

と表記したが、演算子 ＋ ， － ， × が左側でなく逆の右側にくるという意味で、

2 5 ＋ 3 7 ＋ ×

を逆ポーランド記法という。

逆ポーランド記法の優れている点は、左から右へ読み取りながら後戻りせずに計算できることである。これはコンピュータの逐次処理に適している。つまり、行ったり来たりする計算の順を逐次処理に置き換えているのである。

算術式を逆ポーランド記法に変換するにはどうすれば良いだろうか。変換の準備からはじめよう。

【変換の準備】

（a）文字列操作

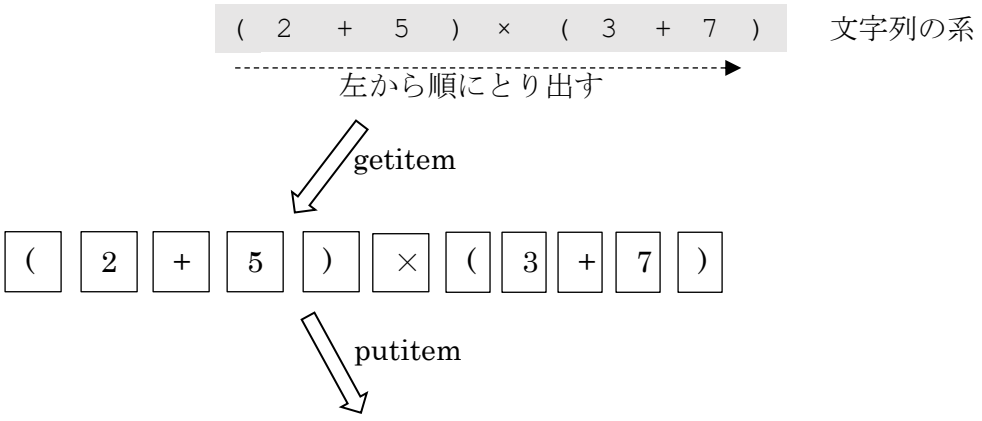
算術式は文字列の系で与えられているとする。

文字列の系とは、

文字列 文字列 文字列 …

のような文字列の並びである。ここで、文字列は英数字列、あるいは ( ) ＋ － × ÷ などの特殊文字であり、複数の文字列をつなぐ文字はないか、“空白”であるとする。このような文字列の系を左から順にたどり文字列をとり出す操作に getitem という名前を与える。1 回の適用で 1 つの文字列をとり出す。

たとえば、算術式「(2 + 5) × (3 + 7)」は文字列の系であり、操作 getitem を繰り返すと図 2-7 のように 11 個の文字列（ここでは 1 字の文字列）が得られる。



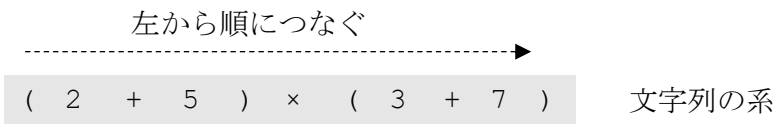


図 2-7　文字列の入力と出力

逆に 11 個の文字列を左から順につないで文字列の系をつくる操作を putitem という。1 回の適用で 1 つの文字列を文字列の系に追加する。

（b）演算子の優先順位

優先順位とは、複数の演算子の間の評価の順位のことである。

（a）に示した ( ) + − × / などの特殊文字それぞれを演算子とし、これらの間に次のような優先順位を与える。

- (は) より優先順位が高い
- (は+, −, ×, / より優先順位が高い
- +, −, ×, / は) より優先順位が高い
- ×, / は +, − より優先順位が高い
- + と − の優先順位は等しい
- ×と / の優先順位は等しい

このような演算子の中から２つの演算子 1 と演算子 2 をとり出し、演算子 1 が演算子 2 より優先順位が高いか等しいなら、演算子 1 を優先して評価することにする。評価するとは、演算子に関係した動作を実行することである。

（c）演算子スタック

逆ポーランド記法変換では演算子に対する優先順位を評価するためにスタックを使用する。

これで、算術式を逆ポーランド記法に変換するための準備ができた。

【逆ポーランド記法への変換】

1　top を 0 にする

2　算術式の文字列の系から、getitem を適用して得た文字列を作業用メモリ w に格納する。文字列の系が尽きたときは文字のセミコロン（;）をwに格納する

※文字 ; は演算子と見なし、他のすべての演算子が ; よりも優先順位が高いものとする

3　(1)　w の内容が演算子なら w を演算子スタックにプッシュ（push）し、演算子でないなら w を出力の文字列の系につなぐ（putitem）

(2)　top ≧ 2 の場合

(a)　Stack［top−2］が Stack［top−1］より優先順位が高いか等しいとき、次を行う

① Stack[top-2]が (ならば 2 の処理へ

② Stack［top−2］の内容を出力の文字列の系につなぐ（putitem）

③ Stack［top−1］が ; ならば、

- Stack［top−1］を Stack［top−2］に移動
- pop を実行
- top が 0 なら処理を終了。そうでなければ (a) の処理へ

④ Stack［top−1］が ） ならば、

- pop を実行
- pop を実行
- pop を実行
- 2 の処理へ

⑤ Stack［top−1］が ; や ） でなければ、

- Stack［top−1］を Stack［top−2］に移動
- pop を実行
- 2 の処理へ

(b)　(a) でないとき、2 の処理へ

(3)　top ＜ 2 の場合、2 の処理へ

例 2　関数呼出し

プログラミン言語の関数は定められた機能を遂行するプログラムのことで、必要な個所で利用できるしくみを伴う。

これまでの説明で使われた操作の push、pop、getitem、putitem などは関数といえるものである。

関数についての説明は後述するが（「３．７　関数」参照）、ここでは簡単な紹介に留めたい。

関数には名前がつけられ（push、pop など）、関数の機能を利用するために、その名前を使う。このことを関数の呼出しという。

関数 a、b、c があり、関数 a の中から関数 b を、関数 b の中から関数 c を呼出しているとする。図 2-8 の上図はこの呼出し関係を示している。

関数 b が呼び出されている間、関数 a はまだ終了していないので、実行中の状態を保持しておく必要がある。関数 b と c との間でも同様である。

関数の実行中の状態の保持に使われるのがコール・スタックである（図 2-8 の下図参照）。

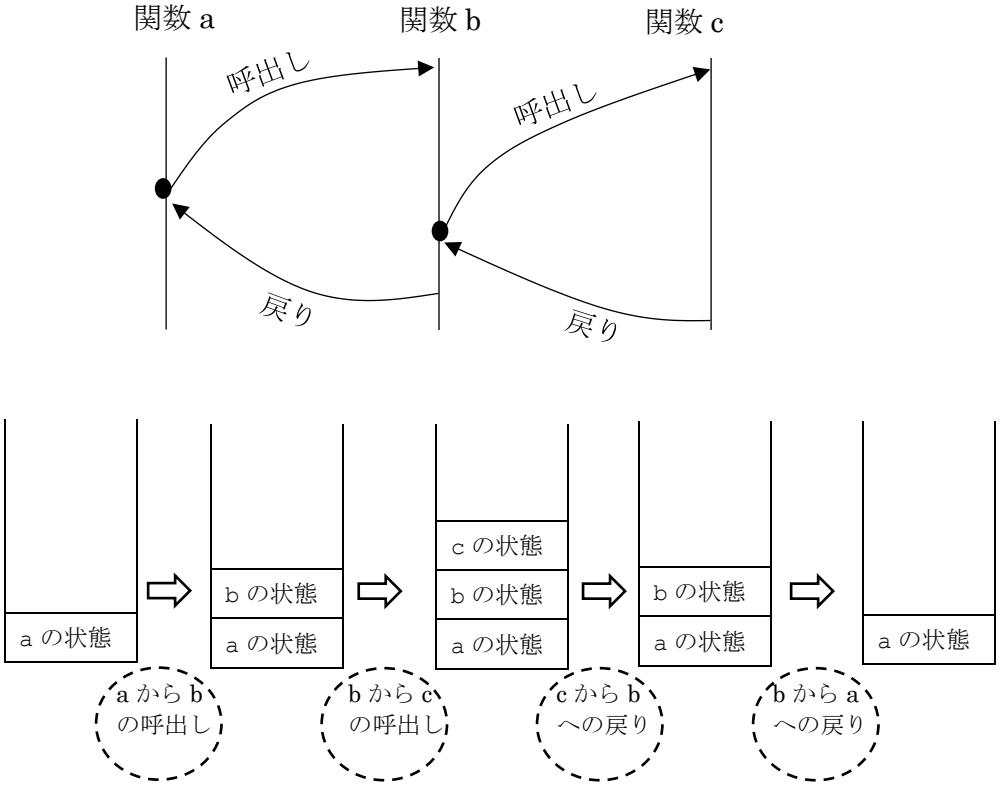


図 2-8　コール・スタックの遷移

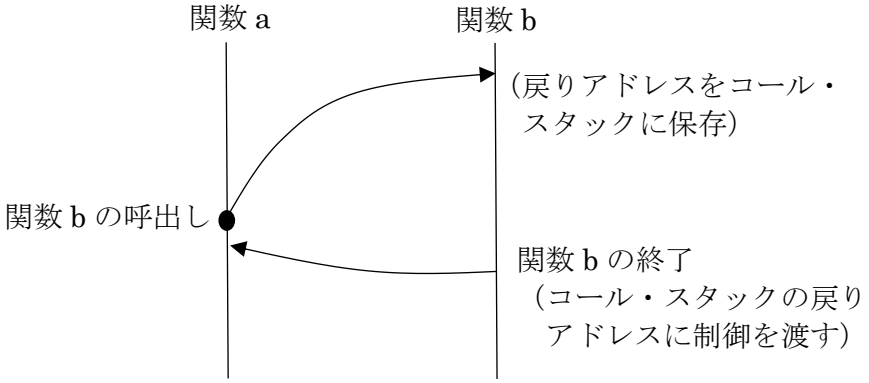
コール・スタックの最上位の要素は現在実行中の関数の状態を保持している。

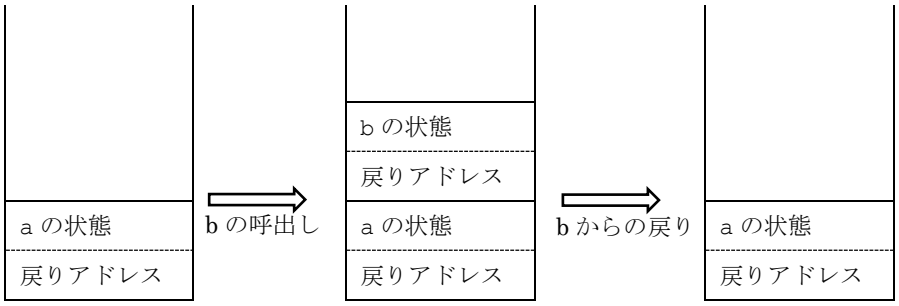
コール・スタックには、

- 関数内部で使用するデータ
- 戻りアドレス

を保存する。

戻りアドレスとは、関数呼出しで使われるマシン命令の次のマシン命令のアドレス（「１．３　プロセッサ」に記載のサブルーチン分岐を参照）であり、リンク・レジスタに格納して関数に渡される。呼び出された関数側ではリンク・レジスタの内容をコール・スタックに保存し、関数からの戻りの際にこの戻りアドレスの指すマシン命令に制御を移すために用いる。図 2-9 参照。





コール・スタックの遷移

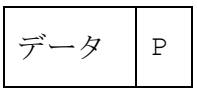
図 2-9　戻りアドレスの扱い

コール・スタックは関数呼出しが正しく動作するための必須のしかけになっている。

（５）連結リスト

メモリ上のデータを管理する方法として配列をベースにしたキューとスタックを見てきたが、データの追加、削除を自由に行いたいとき、配列は適していない。ここで紹介するポインタをベースにした連結リストは、そのような目的に適したデータ構造である。

任意のデータとポインタとの組は構造体である。この構造体をセル（図 2-10 参照）という。



P：ポインタ・データ

図 2-10　セル

セルのポインタ･データの部分をポインタ部、それ以外をデータ部と呼ぶことにする。あるセルを指すポインタ Q があるとき、Q が指すセルのポインタ部は、

Q ー> （セルのポインタ部）

と表わすことにする。

メモリ上の適当な場所に n 個のセルが存在し、それぞれの場所のアドレスはポインタ・データ P_1、P_2、…、P_n に格納されているとする。

このとき、各セルのポインタを図 2-11 のように設定しておくと、この図は連結リストというデータ構造を表わしている。

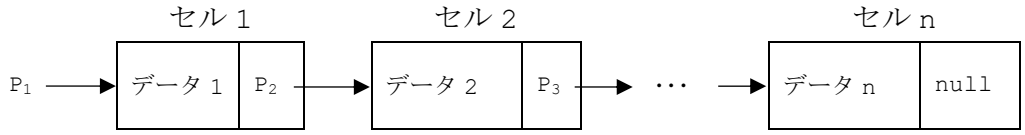


図 2-11　連結リスト

連結リストの最初のセル 1 のアドレスはポインタ P_1 にあり、セル 1 のポインタ部を P_2、セル 2 のポインタ部を P_3、･･･とする。最後のセル n のポインタ部には null（ヌ

ルと読む）が設定されているが、null は何も指さないポインタを意味する。何も指さないポインタなので、null の 2 進法表現はポインタとして無意味であればどんな値でもかまわない。一般的には null に 0 を当てはめている。

連結リストに新しいデータを登録（挿入）、削除する方法や連結リストから目的のセルを見つけ出す方法を考えて見よう。

（a）連結リストに新しいセルを挿入

（a1）リストの間 or 最後に挿入

連結リストの 1 つのセルを指すポインタ P が分かっていて、P の指すセルとその次のセルの間にポインタ Q が指すセル N を挿入するには次の操作を行う。

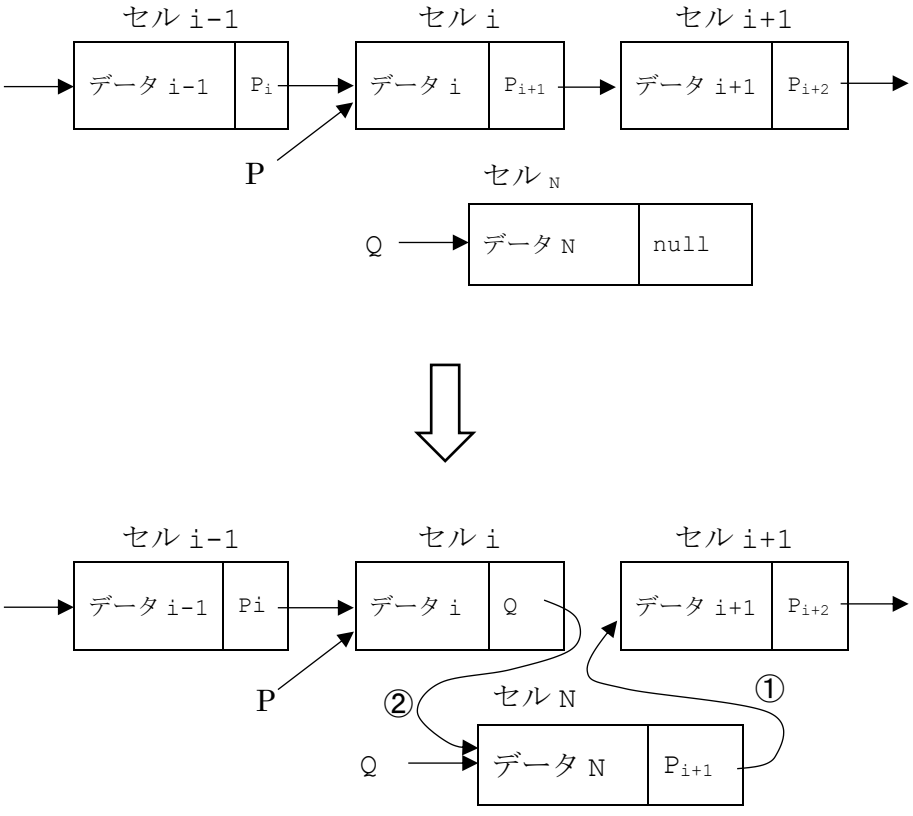


図 2-12　セルの挿入（中間 or 最後）

挿入は次の 2 つのステップで行なう。

① Q －> （セルのポインタ部） に P －> （セルのポインタ部）を格納する
② P －> （セルのポインタ部） に Q を格納する

注意　P が指すセルが連結リストの最後のセルの場合、
　　P －> （セルのポインタ部）は null である。

（a2）リストの先頭に挿入

連結リストの先頭のセルを指すポインタを H とする。連結リストにセルが存在しない場合、H の値は null であるとする。

H が null でない場合、連結リストの先頭に新しいセルを挿入するには図 2-13 に示す操作を行う。

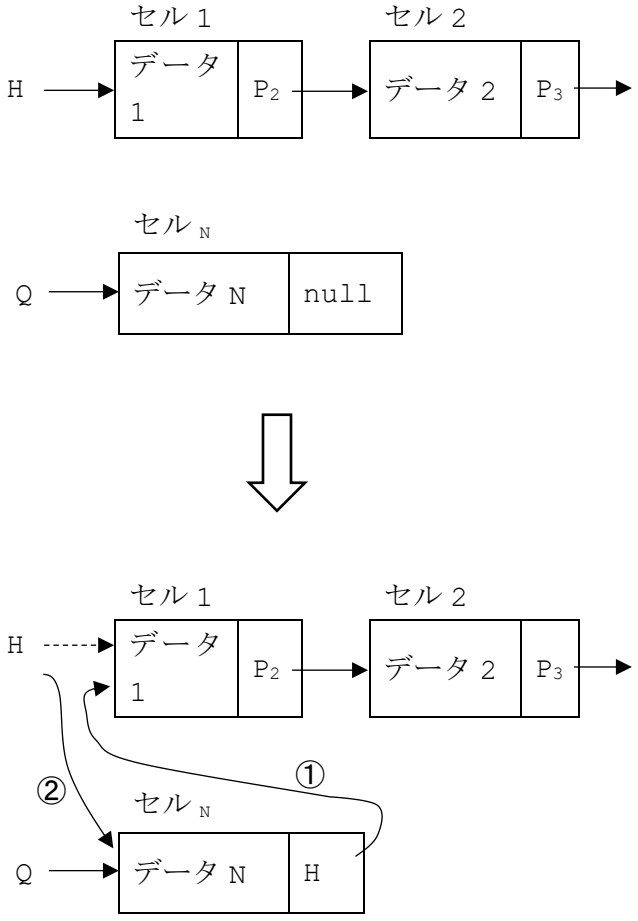


図 2-13　セルの挿入（先頭）

挿入は次の 2 つのステップで行なう。

① Q ―> （セルのポインタ部） に H を格納する
② H に Q を格納する

H が null の場合は H に Q を格納するだけで済む。

（b）連結リストからセルを削除

（b1）リストの間 or 最後のセルの削除

削除したいセルの 1 つ手前のセルを指すポインタを P とする。
P の指すセルの次のセルを削除するには図 2-14 に示す操作を行う。

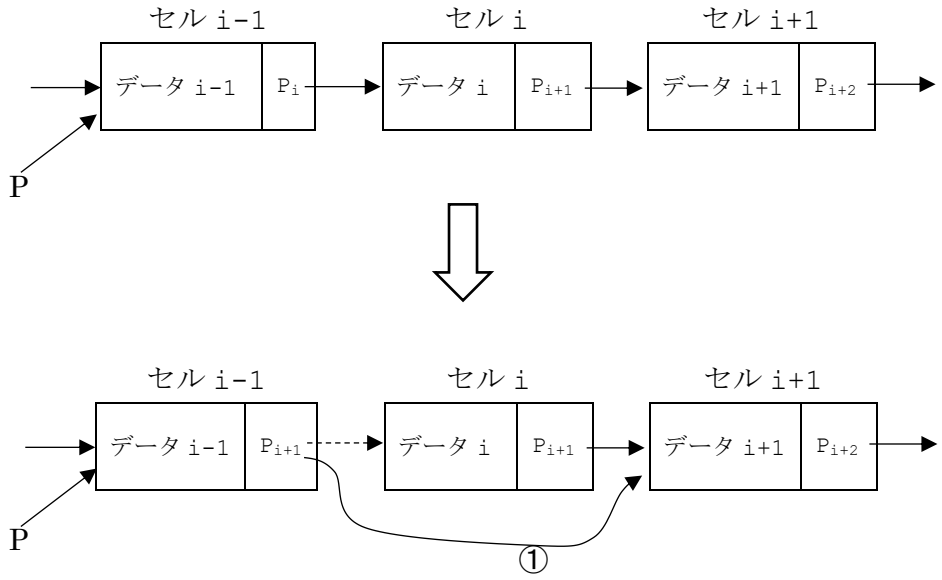


図 2-14　セルの削除

① P ―> （セルのポインタ部） に
　　P ―> （セルのポインタ部）の指すセルのポインタ部を格納する

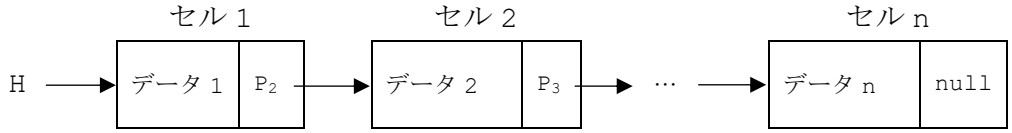
（b2）連結リストの先頭のセルの削除

H が連結リストの先頭のセルを指しているとき、先頭のセルを削除するには次の操作を行う。

① H に、H－> （セルのポインタ部）の指すセルのポインタ部を格納する

（c）連結リストの探索

連結リストの探索とは、連結リストのポインタ部を次々と辿って、各セルのデータ部にデータ d があるか否かを調べることである。探索の操作は図 2-15 に示す通りである。探索は次の操作で行う。ここに、P は作業用のポインタである。



1　　P に H を格納する
2　　P が null でない場合
- P －> （セルのデータ部）に d があるなら
  “あり”として探索を終了
- d がないなら、P に P －> （セルのポインタ部）を格納して、
  2 の処理へ

3　　P が null の場合
“ない”として探索を終了

図 2-15　連結リストの探索

（６）辞書

データ構造の辞書はハッシュ技法と連結リストを用いる。

ハッシュ（hash）とは混ぜ合わせることだが、プログラミングでのハッシュとはデータを混ぜ合わせて元のデータより小さな一定サイズ（ビット数）のデータ（0 以上の整数）に変換することである。変換されたデータのことをハッシュ値という。

ハッシュ値は元のデータより小さいので、元のデータが多数あると、それらの内、ハッシュ値が同じになるものがある。同じハッシュ値になることを衝突という。

ハッシュ値の求め方は、できるだけ衝突を避け、散らばるようにするのが望ましい。

たとえば、英数字列データのハッシュ値を求めるために、文字列の左から順に文字をとり出し、文字コードを整数（C_1，C_2，C_3，…，C_n）と見なし、2 進固定小数点数 w に加算していく。このとき、2 文字目からは w に加算する前にそれぞれ定数 k，k^2，k^3，…をかけて桁上げしておく。その結果の w をハッシュ値の最大サイズ m で割って、その余りを h に設定すると、h には適度に散らばったハッシュ値が得られる。

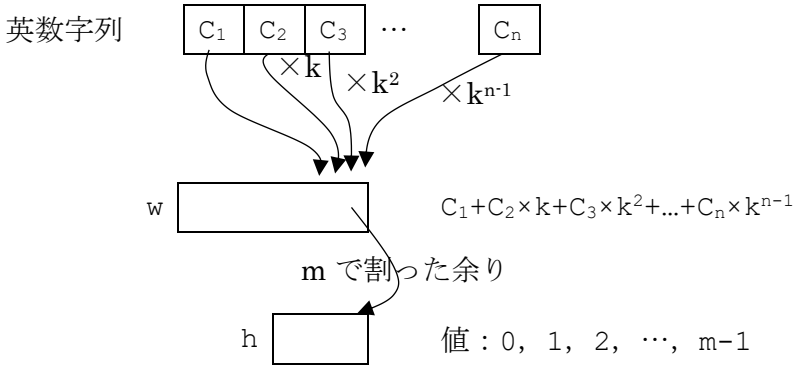


図 2-16　ハッシュ値の計算

プログラミング言語によっては任意のデータをハッシュ値に変換する関数を用意している。

辞書は同じハッシュ値をもつデータの連結リストと、各連結リストの先頭のセルを指すポインタの配列（要素番号が当該連結リストのデータのハッシュ値）からなるデータ構造を用いる。このポインタの配列をハッシュ表という。

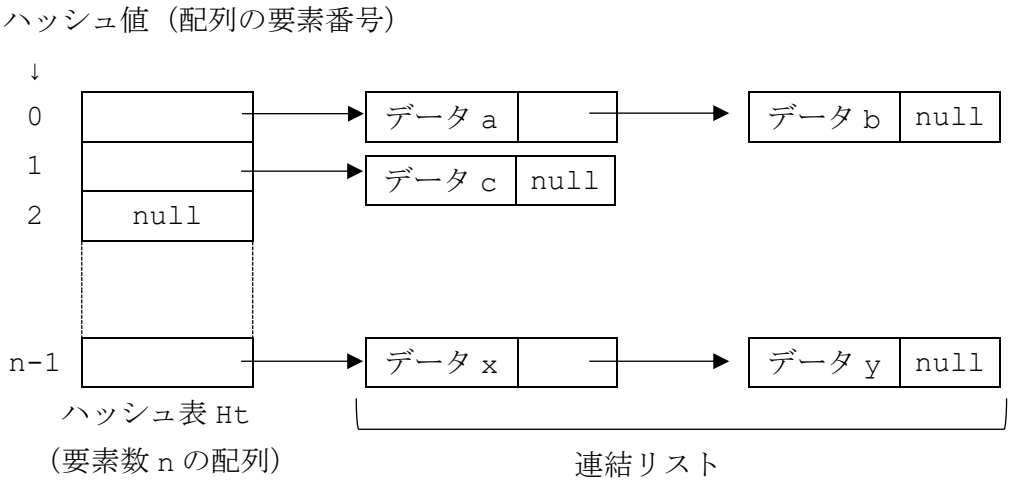


図 2-17　辞書

図 2-17 で、

- データ a とデータ b のハッシュ値＝0
- データ c のハッシュ値＝1
- データ x とデータ y のハッシュ値＝n-1

である。

初期状態ではハッシュ表のすべての要素のポインタ値は null である。

辞書における、データの登録（挿入）、削除、探索は連結リストにおけるやり方と同じであるので、連結リストの説明を参照されたい。

注意すべき点は、データのハッシュ値によって特定される連結リストを使うことである。

プログラミング言語の処理系をつくる際、たくさんのデータや関数の名前を扱うが、そこでは辞書が活用されている。

第３章　プログラムの表現方法

プログラムとは、第２章の冒頭で述べたように、アルゴリズムをマシン命令やプログラミング言語で表現したものである。

マシン命令で表現するプログラムは人にとって扱いにくいため、分かり易さと用途（目的）に合わせたさまざまな表現方法が模索され、数多くのプログラミング言語が登場した。

科学技術計算用の FORTRAN（フォートラン)、事務処理用の COBOL（コボル)、オペレーティング・システムなどのシステム開発に向いた C（シー）はコンピュータが普及し始めた頃から使われている、歴史のある言語である。マイクロ・コンピュータやネットワークの普及と時を同じくして、構造化プログラミング（Structured Programming)、オブジェクト指向（Object Oriented）というプログラミングの考え方が生まれ、それらのアイディアを組込んだ言語として JAVA（ジャバ)、C++（シープラスプラス)、Ruby（ルビー)、Python（パイソン）などが登場した。

プログラムは「２．１　文字列」に述べたように、文字列の系として表現されるが、コンパイラやインタープリタを経由して、最終的には前章に述べたマシン命令やデータのレベルで動作するのである。コンパイラは、文字列の系として表わされたプログラムを、マシン命令やデータに変換するプログラムである。インタープリタは、文字列の系のプログラムを逐次実行するプログラムである。

上記のプログラミング言語は、用途（目的）に合わせた差異はあるものの、プログラムの表現方法の基本は殆ど同じである。これはプログラミングにおけるわたしたちの思考方法に大きな差がないということだろう。

本章では、その“基本”として次をとり上げ、それぞれの表現方法について紹介する。

- 値の文字列表現
- 変数と代入
- 配列と辞書
- 演算
  文字列演算、算術演算、比較演算、論理演算
- 条件判定
- 繰り返し
- 関数
- 例外

“表現方法”は本書独自のものであるが、多くは一般のプログラミング言語に準じた方法を用いている。

本章の説明はプログラミング言語そのものではないが、個別のプログラミング言語を学習するために役立つはずである。

３．１　値の文字列表現

メモリ上の値（2 進法表現）に対応する文字列表現を定義すると、文字列を経由してコンピュータに値を与え、逆にコンピュータ内の値を人が理解し易い文字列表現で受け取ることができる。

文字列表現を定義する値としては、スカラー・データとデータ構造がある。

本節ではスカラー・データの文字列表現について説明する。データ構造の文字列表現については「３．３　配列と辞書」を参照されたい。

スカラー・データに対する文字列表現をリテラルという。

リテラルには、文字列リテラルと数値リテラルの 2 種類がある。また、厳密にはリテラルではないが、似たものに論理データがあるので、論理データについてもここで紹介する。

（１）文字列リテラル

文字列リテラルは文字列を引用符（ ’ ）や 2 重引用符（ ” ）で囲んだものである。

例　
```
’char string’
”char string”
```

’や”で囲まれた文字列の 2 進法表現が実行時の処理対象となる。

引用符（ ’ ）を文字列の中で使いたいときは 2 重引用符（ ” ）で文字リテラルを表わし、2 重引用符（ ” ）を文字列の中で使いたいときは引用符（ ’ ）で文字リテラルを表わす。

例
```
”today’s business”
’what is ”the Internet of things”?’
```

引用符（ ’ ）と 2 重引用符（ ” ）を文字列の中で使いたいときは、逆スラッシュ（ 日本語のフォントでは¥で代用 ）を引用符（ ’ ）や 2 重引用符（ ” ）の前につけて、¥’や¥”のように表わす。逆スラッシュ自身は¥¥のように表わす。

例
```
’what¥’s ”the Internet of things”’
```

（２）数値リテラル

「２．３　数値」で紹介した 3 つの数値データに対し、2 進固定小数点数と 2 進浮動小数点数のリテラルを導入する。10 進固定小数点数のリテラルは本書では導入しないことにする。10 進固定小数点数は事務計算など、お金を扱う場合には重要だが、多くの人にとっては必ずしも必要でないため割愛した。

（ａ）2 進固定小数点数リテラル

2 進固定小数点数リテラルは 10 進数字の並びで表わす。符号+、–をつけることができる。

例　12345　、　+54321　、　-6789

２．３（１）で述べたように、2 進固定小数点数は小数点以下の数を含めることもできるが、リテラルとしては表現しないことにする。従って、2 進固定小数点数リテラルは常に整数のみを扱う。小数は 2 進浮動小数点数リテラルで扱う。

2 進固定小数点数リテラルは２．３（１）で述べた方式でメモリに格納される。

（ｂ）2 進浮動小数点数リテラル

2 進浮動小数点数リテラルには次の 2 つの表現方法がある。

① 小数点を含む 10 進数字の並びで表わす。符号+、–をつけることができる。

例　10.5　+10.5　-10.5
　　0.5　+0.5　-0.5

② 小数点を含む 10 進数字の右に“e±10 進数字”をつけた指数表示で表わす。全体に符号+、–をつけることができる。

例　10.5e2　+10.5e2
　　10.5e-2　-10.5e-2

2 進浮動小数点数リテラルは２．３（２）で述べた方式でメモリに格納される。

（３）論理値

リテラルではないが、リテラルに似たものとして論理値がある。論理値とは命題の真偽を表わす値であり、真偽値ともいう。2 つの値の一方が“真”を意味するなら、他方は“偽”を意味すると定義できるものなら何でもよい。数値の 1 と 0、文字列の on と off、文字列の true と false はどれも論理値になり得る。

プログラミング言語の処理系が文字列 true を数値の 1、文字列 false を数値の 0 に置き換えて扱っても良い。文字列 true および false は文字列データでも数値データでもない。

本書では“真”を文字列 true で、“偽”を文字列 false で表わすことにする。
true と false の両者を合わせて論理データという。

３．２　変数と代入

「２．４　データ構造」の冒頭で述べたメモリ上のバイト列であるフレームに名前をつけよう。名前は英数字からなる文字列とする。

例　a , a1 , a2 , b , b1 , b2 , name , integer , character , variable

名前はフレームに限らず関数（「３．７　関数」参照）などにもつけられ、対象を識別する手段となる。

フレームにはスカラー・データ、配列、辞書、関数を格納することができる。このようなスカラー・データ、データ構造、関数はフレームの“値”とみることができる。関数が“値”になるということには少し違和感があるかもしれない。関数という属性から、“値”としてとらえてもおかしくないことが分かるだろう。“値”としての関数については「３．７　関数」の「(ix)　関数値」を参照されたい。

名前がつけられたフレームで、複数の“値”を入れ替わり格納することができるものを変数と呼ぶ。このフレームの名前を変数名という。

変数はなぜ必要なのだろうか。

あるアルゴリズムが 1 つの値に対してだけでなく、ある範囲の値のすべてについて成り立つように考えるとアルゴリズムの表現の幅が拡がる。変数はこのような値の集合を表わす手段になる。変数はアルゴリズムの表現力を豊かにするのである。

変数に値を格納する操作を代入という。これは、

変数　＝　値

のように記号“=”を用いて表わす。左辺に変数を指定し、右辺には変数に格納すべき値を指定する。この形式の操作を代入文と呼ぶ。

プログラミング言語によっては、変数に予め属性（attribute、スカラー・データなど）を与えておいて、その属性をもつ値だけがその変数に代入できるようにしているものがある。また、そのような制約をもたずに、変数に値を代入する度に、値のもつ属性

を変数の属性にしている言語もある。後者の方が前者よりもプログラミングが簡単になるので、本書では後者を採用することにする。

例
```
a = 10              #  a は 2 進固定小数点数
a = 2.1             #  a は 2 進浮動小数点数
a = ”abc”           #  a は文字列データ
```

注意　各行の右側の文字 # は、同じ行のこの文字以降の文字列がコメントであることを意味する。

上記の例は値がスカラー・データの場合である。値がデータ構造や関数の場合については、「３．３　配列と辞書」、「３．７　関数」を参照されたい。

３．３　配列と辞書

（１）配列

配列の定義は「２．４　データ構造」で与えたが、言い直すと次の通りである。

メモリ上に隣接して配置されている複数のフレームの集まりのことを配列という。

配列を構成するフレームにはメモリ・アドレスの昇順に、0,1,2,…,n-1（n 個のフレーム）なる番号がつけられる。

配列の値の文字列表現は、次のように、複数のデータを括弧[]内にカンマ（,）で区切って並べて表わす。

[データ 1,データ 2,…,データ n]

データの指定は省略できるが、カンマ（,）は省略できない。カンマの数が配列の要素数を決めるからである。すべてのデータの指定を省略する場合、カンマを並べるのを避けるため、次のように表わす。

(n)[]　（n は 10 進数で、配列の要素数である）

変数 a に、データ指定のある文字列表現を代入する代入文を考えよう。

a = [データ 1,データ 2,…,データ n]

この代入文により、変数 a は配列という特性をもち、配列の各要素フレームに、文字列表現のデータが左から順に格納される。フレームのサイズはデータのサイズに依存して決まる。

また、変数 a に、配列の値の文字列表現(n)[]を代入する代入文を考えよう。

a = (n)[]　（n は 10 進数で、配列の要素数である）

この代入の結果、変数 a は配列という特性をもち、各要素はフレームのままで、値は代入されない。

変数 a が配列で、n 個の要素をもつとする。このとき、配列 a の i（0≦i≦n-1）番目の要素は、

a[i]　(0≦i≦n-1)

のように書いて特定する。

配列の要素 a[i]もまた変数であり、スカラー・データ、配列、関数が格納できる。しかし、辞書は格納できない。辞書は複雑な構造をもつデータ構造であり、配列の要素

にすることや辞書内に辞書をもたせるようなことは考えない。単に組合せの巧みさを楽しむだけにすぎず、実益がないと考えるからである。

例　配列の値の文字列表現[30, 20, 50, 10, 100]を変数 a に代入する。

```
a = [30, 20, 50, 10, 100]
```

変数 a は 5 つの要素をもつ配列になり、各要素の値は 2 進固定小数点数である。配列の要素の値はそれぞれ、a[0]は 30、a[1]は 20、a[2]は 50、a[3]は 10、a[4]は 100 である。

上例の a[2]に対し、次の代入を行うと、

```
a[2] = [50, 51, 52, 53, 54]
```

配列 a の値は、

```
[30, 20, [50, 51, 52, 53], 10, 100]
```

である。

配列 a の 2 番目の要素は配列であり、その 1 番目の要素を特定するには、

```
a[2][1]  あるいは  a[2, 1]
```

のように表わす。a[2][1]　あるいは　a[2, 1]の値は 51 である。

配列の要素 a[i] (0≦i≦n-1)への代入は、

```
a[i] = 値
```

のように行う。

配列内の配列の要素 a[i, j]　(0≦i≦n-1, 0≦j≦m-1)への代入は、

```
a[i, j] = 値
```

のように行う。

（2）辞書

２．４（６）で述べたように、辞書は要素数が n のポインタの配列（ハッシュ表）をもち、各要素のポインタはハッシュ値が同じセルの連結リストの先頭のセルを指している。

辞書の値の文字列表現は、次のように、ハッシュ計算に用いるキーと呼ぶ文字列と値との複数の組を、括弧{}内にカンマ（,）で区切って並べて表わす。

```
{キー:値, キー:値, … }
```

キーは文字列データであり、ハッシュ値計算と連結リストの探索に用いられる。連結リストのデータ部にはキーの文字列データと値が格納される。値としてはスカラー・データ、配列が許される。

辞書の値の文字列表現は辞書の初期値を与えるものである。
辞書の初期化は次の代入文で行なう。

```
a = {キー:値, キー:値, … }
```

この代入の結果、変数 a には辞書という特性が与えられる。キーと値の組が指定されていれば、辞書に値を登録する。指定されていなければ、すなわち{}の場合、辞書のハッシュ表のすべての要素は null である。
ハッシュ表の要素数 n は明示的には指定しないが、できるだけ衝突の機会を減らせる値が望ましく、処理系によって決められる。

辞書 a への新しい項目の登録は次の代入文で行なう。

```
a{キー} = 値
```

キーと一致する項目が既に存在するならば、その項目の値はこの代入によって更新される。

辞書 a に、項目{キー : 値}が登録されているとする。このとき、辞書のデータの参照は、

```
a{キー}
```

で行なう。

辞書の項目の削除は多くの場合必要ないので用意しない。

辞書内を探索し、目的の項目が存在するかを確認するために、存在確認演算 in を使用する。in 演算については、３．４（４）の存在確認演算を参照されたい。

３．４　演算

演算は数の計算、文字列の編集、データ比較というごく基本的な機能をコンパクトに表現する手段である。

演算として次の 5 種類を用意する。

- 文字列演算
- 算術演算
- 比較演算
- 存在確認演算
- 論理演算

演算を表わす記号や文字列は、

+, -, *, /, ==, >=, >,…, in, and, or, not　など

である。これらを演算子と呼ぶ。

演算子の演算対象は文字列データ、数値データ、論理データ、辞書である。
演算結果の値は文字列データ、数値データ、論理データである。そのため、演算結果が演算の対象にできる。このような演算の組合せを演算式と呼ぶ。

演算子には優先順位がある。優先順位とは、演算式の中の隣りあう演算子の評価順のことである。

たとえば、算術式、

3 + 5 × 2

では、かけ算（×）がたし算（+）より優先順位が高いので、5 と 2 をかけ算（×）し、3 とその計算結果をたし算（+）する。

このような優先順位が、本書で導入するすべての演算子に対して与えられる。表 3-1 は演算子の優先順位を示している。表中の上の升にある演算子ほど優先順位が高い。同じ升内の演算子は同じ順位である。

表 3-1 では演算子毎に演算対象と演算結果のデータの種類（文字列データ、数値データ、論理データ）を示している。
表 3-1 から演算式としてどのような組み合わせが可能か読み取れる。

表 3-1 演算子

演算子	意味	演算対象	結果
+,-	単項の正負	+a,-a	
		a は 2 進固定小数点数	2 進固定小数点数
		a は 2 進浮動小数点数	2 進浮動小数点数
**	べき乗	a**b	
		a,b 共に 2 進固定小数点数 (b は非負の整数)	2 進固定小数点数
		a,b は 2 進固定小数点数と 2 進浮動小数点数の任意の組合せ ☆	2 進浮動小数点数
*,/,%	乗算,除算,剰余	乗算 a*b	
		a,b 共に 2 進固定小数点数	2 進固定小数点数
		a,b の少なくとも 1 つが 2 進浮動小数点数 ☆	2 進浮動小数点数
		除算 a/b	
		a,b は 2 進固定小数点数と 2 進浮動小数点数の任意の組合せ ☆	2 進浮動小数点数
		剰余 a%b	
		a,b 共に 2 進固定小数点数 (a,b は整数)	2 進固定小数点数
+,-	加算,減算	a±b	
		a,b 共に 2 進固定小数点数	2 進固定小数点数
		a,b の少なくとも 1 つが 2 進浮動小数点数 ☆	2 進浮動小数点数
||	連結	a||b	
		a,b 共に文字列データ	文字列データ
>,>=,<=,<	大小比較	a>b,a>=b,a<=b,a<b	
		a,b は 2 進固定小数点数と 2 進浮動小数点数の任意の組合せ ☆	論理データ
==,!=	等値比較	a==b,a!=b	
		a,b は 2 進固定小数点数と 2 進浮動小数点数の任意の組合せ ☆	論理データ
		a,b 共に文字列データ	論理データ
		a,b 共に論理データ	論理データ

in	存在確認	a in b	
		a は文字列データ、b は辞書	論理データ
not	論理否定	not a	
		a は論理データ	論理データ
and	論理積	a and b	
		a,b 共に論理データ	論理データ
or	論理和	a or b	
		a,b 共に論理データ	論理データ

表 3-1 中の星印☆のついた個所では、暗黙に 2 進固定小数点数から浮動小数点数への変換が行われる。

演算式の中の演算の優先順位は括弧 ( ) を用いて変更できる。
たとえば、3+5*2 の計算順は「5*2 を行い、3 にその結果をたす」である。3+5 を先に行うには括弧を用いて(3+5)*2 とする。

演算式の評価は、演算子の優先順位に従って行ったり来たりする。コンピュータは逐次処理するように設計されているので、評価に先立ち優先順位に合うように演算順を並べ替える方が効率的である。この変換は２．４　データ構造」の「(４)　スタック」の例で述べた逆ポーランド記法への変換で行なうことができる。

表 3-1 に記した演算についてもう少し詳しく述べよう。

（１）文字列演算　a||b

a,b は共に文字列データが許される。
2 つの文字列データ a,b を連結して 1 つの文字列データを生成する。
従って、連結の結果は再び連結することができる。

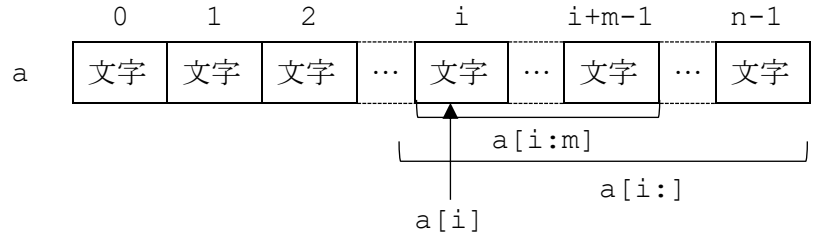
文字列データ || 文字列データ || … || 文字列データ

例　連結演算 "abc" || "def" の結果は "abcdef" である。

編集の対象が文字列リテラルや文字列変数だけでは柔軟性に欠ける。文字列の 1 部をとり出し他の文字列と組み合わせるなど、きめ細かな操作が求められる。これを可能とするのが、部分文字列である。

文字列データを 1 つ以上の文字が並んだ配列と見立て、配列要素を特定する記法に似た記法を使用して部分文字列をとり出す。

n 個の文字からなる文字列データ a の各文字を特定するために、0 番目から n-1 番目までの番号を付与する。



文字列データ a の部分文字列のとり出しは次のように行う。

① 0 から数えて i 番目の文字
a[i] (i≧0)
② i 番目の文字から右へ m 個の部分文字列
a[i:m] (m≧1)
i 番目の文字から最後の文字までの部分文字列は m を省略して、
a[i:]
と指定

ここで、i、m は 2 進固定小数点数である。

例 変数 a が値として "abcdef" をもつとき、a に対する部分文字列の値はそれぞれ次の通りである。

部分文字列指定	値
a[3]	"d"
a[1:2]	"bc"
a[2:]	"cdef"

utf-8 の場合、英数字や記号などの ASCII 文字は 1 バイトで、日本語文字は 3 バイトで表現される。これらの文字が混在する文字列の長さはをどのようにカウントするのか。utf-8 コードでは各バイトの左端ビットで ASCII 文字か日本語文字かを区別できるので（これについては、２．２（２）の utf-8 を参照)、処理系はこれらを区別して文字列の長さを計算する。

（２）算術演算

算術演算には単項の正負（+,-)、べき乗（**)、乗算（*)、除算（/)、剰余（%)、加算（+)、減算（—）がある。

それぞれについて詳細に述べる。

（ａ）単項の正負　+a,-a

ａは２進固定小数点数あるいは２進浮動小数点数である。

単項演算は数学の規則に従う。

結果の値はａの値の種類と同じである。

（ｂ）べき乗　a**b　（ａを底、ｂを指数という）

a,bは２進固定小数点数と２進浮動小数点数の任意の組合せが許される。

べき乗の計算は数学の規則に従う。

結果の値の種類は、a,b共に２進固定小数点数（ただしｂは非負）のとき、２進固定小数点数である。それ以外の場合は２進浮動小数点数である。

例

べき乗指定	値
16**3	4096
16**0.5	4.0
16**0	1
16**-1	0.0625
3**0.5	1.7320508075688772
4.1**0.5	2.0248456731316584

（ｃ）乗算　a*b

a,bは２進固定小数点数と２進浮動小数点数の任意の組合せが許される。

乗算の計算は数学の規則に従う。

結果の値の種類は、

- a,b共に２進固定小数点数なら２進固定小数点数
- a,bのうち少なくとも１つが２進浮動小数点数なら２進浮動小数点数

である。

例

乗算指定	値
2*3	6
2*3.14	6.28
1.732*4	6.928
1.732*3.14	5.43848

（d）除算　a/b

a,b は 2 進固定小数点数と 2 進浮動小数点数の任意の組合せが許される。

除算の計算は数学の規則に従う。

結果の値の種類は 2 進浮動小数点数である。

例

除算指定	値
2/3	0.6666666666666666
2.1/3.1	0.6774193548389097

（e）剰余　a%b

a,b は 2 進固定小数点数のみが許される。

剰余は数学の mod である。

結果の値の種類は 2 進固定小数点数である。

例

剰余指定	値
2%3	2
4%3	1

プログラミングにおける剰余演算の用途は広い。本書で紹介しているアルゴリズムだけでも待ち行列の操作やハッシュ値の計算、うるう年の判定で剰余演算が使用されている。

（f）加算　a+b,　減算　a-b

a,b は 2 進固定小数点数と 2 進浮動小数点数の任意の組合せが許される。

加算（+)、減算（-）の計算は数学の規則に従う。

結果の種類は、

- a,b 共に 2 進固定小数点数なら 2 進固定小数点数
- a,b のうち少なくとも 1 つが 2 進浮動小数点数なら 2 進浮動小数点数

である。

例

加算減算指定	値
2+3	5
2-3	-1
2.1+3.1	5.2
3.1-2	1.1
2-3.1	-1.1

（３）比較演算

比較演算には、より大 (>)、より大か等しい (≧)、より小か等しい (≦)、より小 (<)、等しい（==)、等しくない（!=）がある。

比較演算は大小比較と等値比較に分かれる。

（ａ）大小比較　a>b, a>=b, a<=b, a<b

a,b は 2 進固定小数点数と 2 進浮動小数点数の任意の組合せが許される。

大小比較は数学の規則に従う。

大小比較の結果の種類は論理データ（true,false）である。比較の条件を満たすならば true であり、そうでなければ false である。

例

a の値	比較	値
3.1	a>3	true
3	a>=3.1	false
3	a<=3.1	true
3.1	a<3	false

（b）等値比較　a==b, a!=b

a,b には次の 3 通りがある。

- a,b は 2 進固定小数点数と 2 進浮動小数点数の任意の組合せ
- a,b 共に文字列データ
- a,b 共に論理データ

結果の値の種類は論理データである。

例

a の値	比較	値
3	a==3	true
3 以外		false
3.1	a==3.1	true
3.1 以外		false
”abc”	a==”abc”	true
”abc”以外		false
true	a==true	true
false		false
false	a!=true	true
true		false

（４）存在確認演算　a in b

a はキーとなる文字列データ、b は辞書が許される。

存在確認演算は辞書内にキーと一致する項目があるか否かを調べる。

結果の値の種類は論理データである。

例

存在確認指定	値
a in b	true（b 内にキーa の項目が存在する） false（b 内にキーa の項目が存在しない）

（５）論理演算 not a, a and b, a or b

a,b は論理データが許される。

論理データ（true, false）に対する 3 つの演算：否定、論理積、論理和を次のように定義する。

（a）否定 not a

a の値	not a
true	false
false	true

（b）論理積 a and b

a の値	b の値	a and b
true	true	true
true	false	false
false	true	false
false	false	false

（c）論理和 a or b

a の値	b の値	a or b
true	true	true
true	false	true
false	true	true
false	false	false

論理演算の間にはいくつかの関係式が成り立つので紹介する。

論理データ a,b,c の間で次の関係式が成り立つ。

（ⅰ） a or a = a , a and a = a

（ⅱ） a or b = b or a , a and b = b and a

（ⅲ） c or (a or b) = (c or a) or b （結合則）
c and (a and b) = (c and a) and b （結合則）

（ⅳ） c or (a and b) = (c or a) and (c or b) （分配則）
c and (a or b) = (c and a) or (c and b) （分配則）

（ｖ）　a or (a and b) = a　（吸収則）
　　a and (a or b) = a　（吸収則）

（iii）を結合則、（iv）を分配則、（ｖ）を吸収則という。

（iv）の１つ目の式についてのみ証明を与えておこう。

【c or (a and b) = (c or a) and (c or b)の証明】

cがtrueのとき　左辺は定義（ｃ）からtrueで、
右辺は定義（ｃ）、（ｂ）からtrueであり、
上の式は成立つ。

cがfalseのとき　-- a,bが共にtrueなら、
左辺は定義（ｂ）、（ｃ）からtrueで、
右辺は定義（ｃ）、（ｂ）からtrueであり、
上の式は成立つ。

-- a,bの少なくとも１つがtrueなら、
左辺は定義（b）、（ｃ）からfalseで、
右辺は定義（ｃ）、（ｂ）からfalseであり、
上の式は成立つ。

-- a,bが共にfalseなら、
左辺は定義（ｃ）、（ｂ）からfalseで、
右辺は定義（ｃ）、（ｂ）からfalseであり、
上の式は成立つ。

このことから（iv）の１つ目の式は常に成り立つ。

（ｉ）～（ｖ）の他の関係式についても同様に証明できる。

もう１つの関係式はnot,and,orの関係性を表わすものである。

not(a or b) = not a and not b　・・・　☆1
not(a and b) = not a or not b　・・・　☆2

この２つの式はド・モルガンの規則と呼ばれている。

ド・モルガンの規則の☆1について証明を与えておこう。

【not(a or b) = not a and not bの証明】

a,bの少なくとも１つがtrueのとき

左辺は定義（ｃ）、（ｂ）から false で、
右辺は定義（ａ）、（ｂ）から false であり、
☆1 は成立つ。

「 a,b の少なくとも 1 つが true のとき」以外、すなわち a,b 共に false のとき

左辺は定義（ｃ）、（ａ）から true で、
右辺は定義（ａ）、（ｃ）から true であり、
☆1 は成立つ

このことから☆1 は常に成り立つ。

☆2 についても同様に証明できる。

上記の関係式により、論理演算を使ったアルゴリズムを簡潔にできる。良い例が「３．５　条件判定」の「例　うるう年の判定」にあるので参照されたい。

（６）演算つき代入

３．２で代入は、

```
変数 = 値
```

と表わすと述べた。値を変数に加算あるいは減算するという例は大変多く見受けられる。

たとえば変数 a に 1 を加えるとき、

```
a = a + 1
```

と書くが、これを

```
a += 1
```

のように “+=” を用いて簡潔に表わすことにする。

同様に、

```
a = a - 1
```

は、

```
a -= 1
```

のように表わす。

このような代入を、単項の正負を除くすべての算術演算および文字列演算に拡げて、次のように表わすことにする。

変数　{ **｜*｜/｜%｜+｜-｜|| } = 値
（値は 2 進固定小数点数、2 進浮動小数点数、文字列データ）

このような代入を演算つき代入という。

例

演算の種類	演算つき代入	代入結果の値
べき乗	a = 16	a の値：4096
	a **= 3	
乗算	a = 2	a の値：6
	a *= 3	
除算	a = 2	a の値：
	a /= 3	0.6666666666666666
剰余	a = 2	a の値：2
	a %= 3	
加算	a = 2	a の値：5
	a += 3	
減算	a = 2	a の値：-1
	a -= 3	
文字列の連結	a = ”abc”	a の値：”abcdef”
	a ||= ”def”	

３．５　条件判定

論理的な文書では問題を整理するための手段の一つとして“場合分け”が使われる。“場合分け”によって問題が分析的にとらえられ、考えの抜けが防止でき、分かり易くなるからである。

プログラミングの条件判定という表現方法は、この“場合分け”とほぼ同じである。プログラミングを進める上で、問題を“場合分け”すると、殆どの場合そのまま条件判定という表現に置き換えることができる。

条件判定は if 文で行なう。if 文として次の２通りを用意する。

形式１　if―else―end
形式２　if―elif―else―end

それぞれについて説明しよう。

（１）形式１　if―else―end

```
if (論理データ)
      # 論理データが true のときの処理（true ブロック）
else
      # 論理データが false のときの処理（false ブロック）
end
```

解説

（ｉ）形式１の if 文の構文は次の通りである。

if に続く括弧()で囲まれた“論理データ”により条件判定の条件を指定する。“論理データ”は、評価された値が論理データとなる演算式で指定する。“(論理データ)”の後ろには、“論理データ”の値が true のときに実行する文の列（true ブロックと呼ぶ）を指定する。“論理データ”の値が false のときに実行する文の列があれば else の後ろ（false ブロックと呼ぶ）に指定する。“論理データ”の値が false のとき実行する文の列がなければ else は省略する。

if 文は end で閉じる。

true ブロック、false ブロック内の文は空文（何も書かない）でもよい。

（ⅱ）形式 1 の if 文は図 3-1 のフローチャートに示したように動作する。

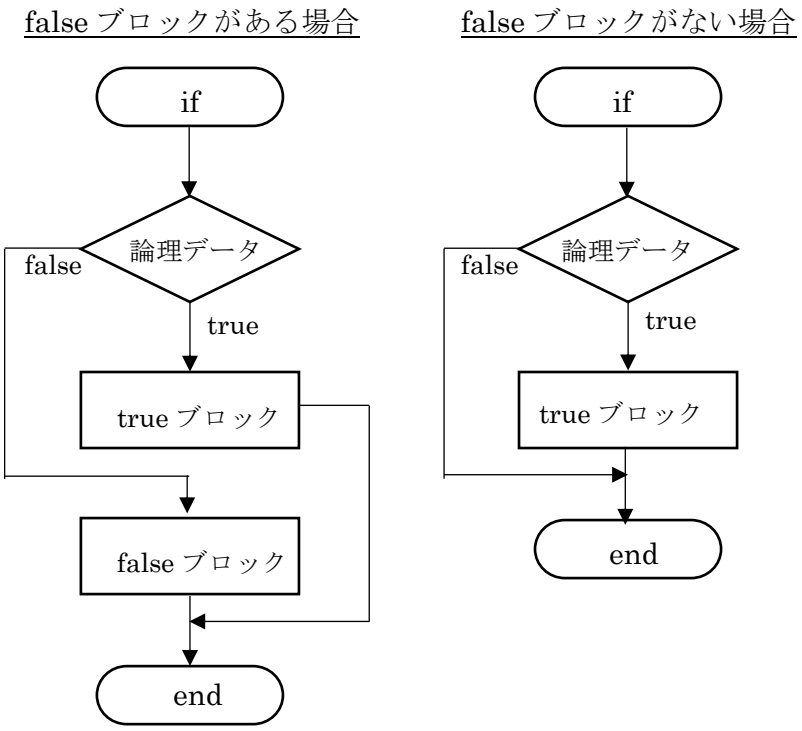


図 3-1　形式 1 の if 文のフローチャート

例　　変数 a,b は 2 進固定小数点数とする。この 2 つの変数の比較演算 a == b の結果は論理データである。a と b の値が等しければ true で、等しくなければ false である。次の if 文はこの比較演算の結果を判定し、それぞれの処理を遂行する。

```
if (a == b)
    # 比較演算：a == b の結果が true のときの処理
else
    # 比較演算：a == b の結果が false のときの処理
end
```

（２）形式 2　if―elif―else―end

```
if (論理データ 1)
      # 論理データ 1 が true のときの処理（true ブロック 1）
elif (論理データ 2)
      # 論理データ 2 が true のときの処理（true ブロック 2）
elif (論理データ 3)
      # 論理データ 3 が true のときの処理（true ブロック 3）
  ⋮
else
      # 論理データ 1、論理データ 2、論理データ 3、… がすべて false のと
      # きの処理（false ブロック）
end
```

解説

（ｉ）形式 2 の if 文は、形式 1 の if 文の else ブロックに形式 1 の if 文が入れ子になっていると考えることができる。
すなわち、次の通りである。

```
if (論理データ 1)  # true ブロック 1
else if (論理データ 2)  # true ブロック 2
else if (論理データ 3)  # true ブロック 3
   ⋮
```

論理データ 1、論理データ 2、論理データ 3、…のすべてが false のときに実行する文の列があれば else の後ろ（false ブロックと呼ぶ）に指定する。このような文がなければ else は省略する。
if 文は end で閉じる。

true ブロック i、false ブロック内の文は空文（何も書かない）でもよい。

（ⅱ）形式 2 の if 文は図 3-2 のフローチャートに示したように動作する。

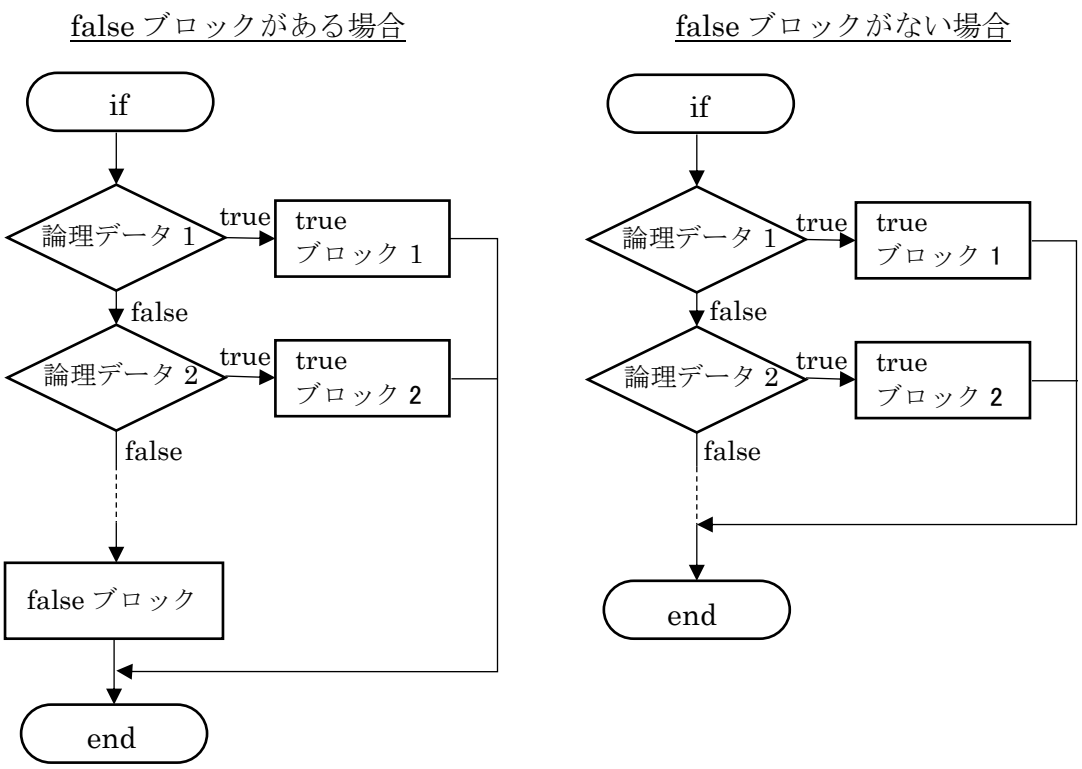


図 3-2　形式 2 の if 文のフローチャート

例　変数 shingo が信号機の色を意味する文字、

"青", "黄", "赤"

をもつものとする。

このとき、次の if 文は各色の判断を示している。

```
if (shingo == "青")
      # 進んでも良い。
elif (shingo == "黄")
      # 止まれ。安全に止まれないときは進んでもよい。
elif (shingo == "赤")
      # 止まれ。
end
```

例　うるう年の判定

西暦（グリゴリオ歴）では、うるう年を次のように決めている。

①　西暦年号が 4 で割り切れる

② ただし、例外として、西暦年号が 100 で割り切れ、かつ 400 で割り切れない年は平年とする

上の①と②を満たす西暦年号はうるう年である。

この取り決めを素直に演算式で表わすと、次のようになる。

（＊1） (Y%4 == 0)and not((Y%100 == 0)and(Y%400 != 0))
（ここで、Y は西暦年号をもつ 2 進固定小数点数）

従って、うるう年を判定するプログラムは、

```
if ((Y%4 == 0)and not((Y%100 == 0)and(Y%400 != 0)))
        # うるう年である
else
        # うるう年ではない
end
```

と書ける。

このままで、まったく問題ないが、上の演算式はもう少し簡潔に書けるのである。つまり、入れ子が外せる。

うるう年判定の演算式（＊1）の 2 項に対し、ド・モルガンの規則を適用して次のように書き直す。

```
not(Y%100 == 0)or not(Y%400 != 0)
```

この式の各項の not 演算を比較演算に適用して、次のように書き直す。

```
(Y%100 != 0)or(Y%400 == 0)
```

この結果を使って（＊1）の式を書き直すと、次の式（＊2）が得られる。

（＊2） (Y%4 == 0)and((Y%100 != 0)or(Y%400 == 0))

式（＊2）は分配則を使ってまだ入れ子はずしができる。

```
(Y%4 == 0)and(Y%100 != 0)or(Y%4 == 0)and(Y%400 == 0)
```

ここで、右の(Y%4 == 0)and(Y%400 == 0)は、(Y%400 == 0)に等しい。なぜなら、Y が 400 で割り切れるならば常に 4 で割り切れるからである。

このようにして（＊1）は、入れ子をすべてはずして、（＊3）のように書き直すことができた。

（＊3）　(Y%400 == 0)or(Y%4 == 0)and(Y%100 != 0)

この式は左から順に評価すれば比較的簡潔に結果が求められる。

先に示したうるう年を判定するプログラムは、（＊3）を使って次のように書き換えることができる。

```
if ((Y%400 == 0)or(Y%4 == 0)and(Y%100 != 0))
        # うるう年である
else
        # うるう年ではない
end
```

また、うるう年を判定するプログラムは単純な演算式と if 文を使って次のように表わすこともできる。

```
if (Y%400 == 0)
        # うるう年である
elif (Y%100 == 0)
        # うるう年ではない
elif (Y%4 == 0)
        # うるう年である
else
        # うるう年ではない
end
```

３．６　繰り返し

同じ種類の対象が複数あり、それぞれについて何がしかの処理を、その数だけ繰り返し行う状況は数多くある。

たとえば、次のような「繰り返し」処理がある。

① 配列
配列 a の要素を、

a[i]　0≦i≦n-1　　（n：配列 a の要素数）

としたとき、i を移動させながら各要素に対する処理を要素の数だけ繰り返す。

② データの系列
データの系列から 1 つのデータをとり出し、各データに対する処理を繰り返す。
反対に、1 つのデータをデータの系列に追加する処理を繰り返す。

③ 数学の計算式

- 多項式
  $a_0 + a_1x + \cdots + a_nx^n$
  （加算を n 回繰り返す）
- 階乗　n!
  1・2・…・(n-1)・n

繰り返し処理として次の while 文と for 文を用意する。

```
while―end
for―end
```

それぞれについて説明しよう。

（１）while―end

```
while (論理データ)
    # 論理データが true のときの処理（true ブロック）
end
```

解説

（ｉ）while 文の構文は次の通りである。

while に続く括弧 () で囲まれた“論理データ”により繰り返し条件を指定する。“論理データ”は、評価された値が論理データとなる演算式で指定する。“(論理データ)”の後ろには、“論理データ”の値が true のときに実行する文の列（true ブロックと呼ぶ）を指定する。“論理データ”の値が false のときに while 文を終了する。
while 文は end で閉じる。

（ii）while 文は図 3-3 のフローチャートに示したように動作する。

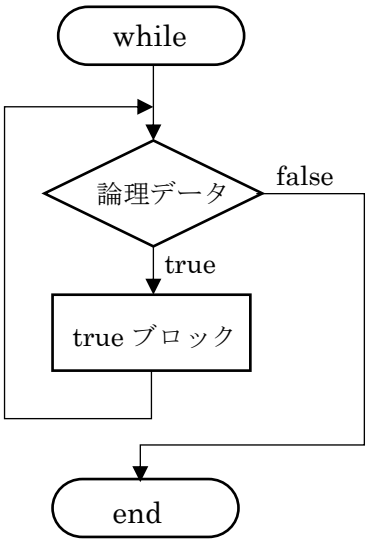


図 3-3　while 文のフローチャート

（iii）true ブロックの処理の途中で while 文を終了させたいとき、break 文を使用する。

```
break
```

（iv）true ブロックの処理の途中で打ち切り、while 文を継続したいとき、continue 文を使用する。

```
continue
```

例　ハッシュ値の計算

ハッシュ値の計算アルゴリズムについては、「２．４　データ構造」の「(３）辞書」で述べたが、そのアルゴリズムをプログラムに書き直してみよう。

```
string = ”abcde12345”      # 英数字列
ln = 10                    # string の長さ
```

```
hv = 0                          # ハッシュ値
kw = 1                          # 桁上げ計算用
i = 0                           # string を配列と見立てた要素番号
while (i < ln)
    hv += charint(string[i])*kw
    kw *= 5                     # 5 は桁上げ用定数
    i += 1
end
hv %= 100                       # 得られたハッシュ値
```

注意 charint()は utf-8 の 1 文字（0x20～0x7E の文字）の 2 進法表現を 2 進固定小数点数とみなす組み込み関数である。「３．７　関数」の「（４）組み込み関数」を参照。

例　データの系列

空白文字で区切られた 10 進数字列の系列、

```
12345 80 456 77 8910 …  E
```

があり、系列の最後は文字 E とする。また、この系列の左から順番に文字列をとり出す関数 getitem()があるとする。

このとき、系列内の 10 進数字列の値が 99 以上の数字列の数を求めるプログラムを考えよう。

```
num = 0                         # 値が 99 以上の数字列の数
while (true)                    # 常に繰り返す
    str = getitem()             # 系列から文字列を 1 つとり出す
    if (str[0] == "E")          # 系列は終わった
        break                   # while を終了する
    end
    w = numeric(str)            # 10 進数字列を 2 進固定小数点数に変換
    if (w < 99)
        continue                # while を継続
    end
    num += 1                    # count up
end
```

プログラム終了後の num の値が求める値である。

注意 numeric()は 10 進数字列（文字列）を 2 進固定小数点数に変換する組み込み関数である。

例　階乗　n!

階乗の値を求めるプログラムを書こう。

```
number = 5                  # 階乗値を求める数
factorial = 1               # 階乗の値
i = 1
while (i <= number)         # 階乗計算を繰り返す
    factorial *= i
    i += 1
end
```

プログラム終了後の factorial の値が 5!の値である。

（2）for―end

```
for 変数 (開始値, 終了値, ステップ値)
    # 繰り返し処理
end
```

解説

（ｉ）for 文の構文は次の通りである。
“開始値”、“終了値”、“ステップ値”は 2 進固定小数点数である。
“変数”は省略できる。

for 文は次のプログラムと同等である。

```
&sw = true
while(true)
```

```
    if (&sw == true)
        &sw = false
        “変数” = “開始値”
    else
        “変数” += “ステップ値”
    end
    if (“変数” >= “終了値”)
        break
    end
    繰り返し処理
end
```

・&sw は処理系が生成する変数。
・“変数”が省略された場合、処理系は適当な変数を割り当てる。

for に続いて“変数”と括弧 () に囲まれた繰り返し条件を指定する。その後ろには繰り返しで実行する文の列を指定する。
“変数”が指定されているとき、繰り返しの最初で“変数”に“開始値”が代入され、2 回目以降の繰り返しでは“ステップ値”が加算される。その直後に、“変数”が“終了値”より大きいか比較し、大きい場合 for 文を終了する。そうでなければ、繰り返し処理を実行する。繰り返し処理は“変数”が“終了値”に等しいか、“終了値”より大きくなるまで繰り返される。
“変数”が省略された場合、処理系が割り当てる変数が使われる。
“変数”が指定された場合、繰り返し処理の中で“変数”を参照することができる。“変数”の値を変更してはならない。
for 文は end で閉じる。

（ⅱ）for 文は図 3-4 のフローチャートに示したように動作する。

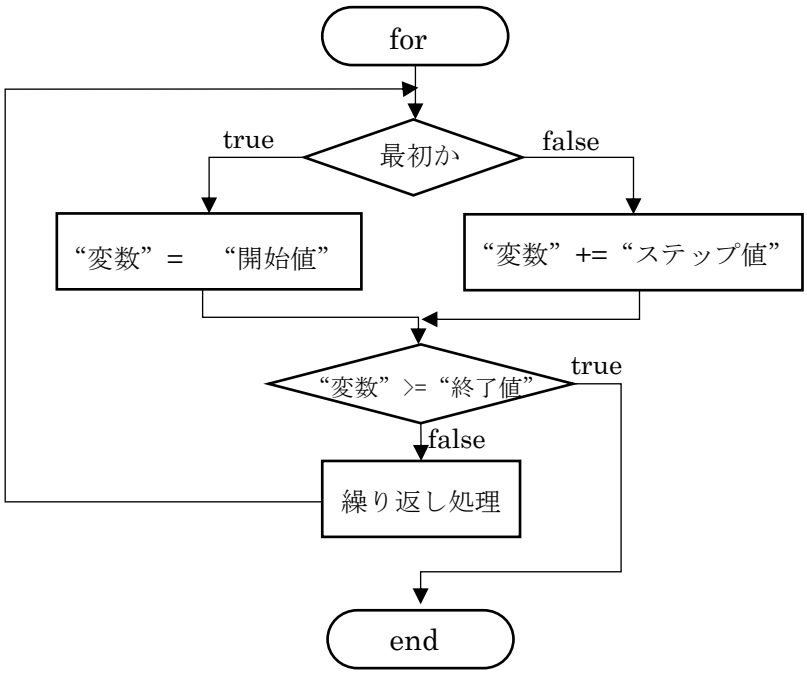


図 3-4　for 文のフローチャート

（ⅲ）true ブロックの処理の途中で while 文を終了させたいとき、break 文を使用する。

```
break
```

（ⅳ）true ブロックの処理の途中で処理を打ち切り、while 文を継続したいとき、continue 文を使用する。

```
continue
```

例　多項式の計算

多項式：　$a_0 + a_1x + \cdots + a_nx^n$ に関して、係数 a_i（$0 \leqq i \leqq n$）は数値データで、配列 A に先頭から順番に格納されているとする。また、x は数値データ、n は整数とする。

x が x0 のときの多項式の値を求めるプログラムを考えてみよう。

多項式の計算では加算の繰り返しと乗算の繰り返しを用いる。

```
pv = 0
for i (0, n+1, 1)
    x = 1
    for (0, i, 1)
        x *= x0
    end
    pv += A[i]*x
end
```

pv の値が求める値である。

内側の for 文の代わりにべき乗演算を使うと多項式の計算はもっと簡潔に表現できる。

```
pv = 0
for i (0, n+1, 1)
    pv += A[i]*x0**i
end
```

例　1000 までの素数を求める

第２章の冒頭で述べた 1000 までの素数を求めるプログラムを考えよう。
奇数の最初の素数は 3 であり、これは予め素数配列に登録しておく。調査は奇数 5 から始める。

```
primen = (500)[]                # 素数配列
primen[0] = 3                   # 素数配列の先頭の要素を 3 とする
top = 1                         # トップの次の要素番号
candi = 5                       # 最初の候補となる数
while (candi < 1001)
  decis = 0                     # 0：candi は素数
                                # 1：candi は素数ではない
  for i (0, top, 1)
    if (candi%primen[i]==0)
      decis = 1
```

```
      break
    end
  end
  if (decis == 0)
    primen[top] = candi
    top += 1
  end
  candi += 2
end
```

求める素数は 2 と素数配列 primen 内の要素番号 0 から top-1 までの要素の値である。

３．７　関数

プログラミング言語の関数は、関数という用語を数学から借りているが、その考え方は数学とはかなり異なる。

数学の関数は y=f(x)　(a≦x≦b) のように書き、y を x の関数という。ここで、関数 y は区間 a≦x≦b 内の各数に対し、1 つの値を確定する規準をもつと仮定している。

プログラミング言語の関数は、

```
func y ( [x] )        ([ ] は x が省略できることを示す記号)
    # 機能を実現する処理
end
```

のように書き、y を関数という。

ここで、関数 y は次の規準を満たすものとする。

①　関数 y は引数 x をもつことができる。
②　関数 y を呼び出す度に、関数 y は内部の処理で、ある値を確定する。
③　関数 y は戻り値をもつことができる。
④　関数 y はプログラム内の適切な個所で呼び出すことができる。

このような規準により、プログラミング言語の関数は、数学の関数（の一部）を疑似的に表現する能力をもつだけでなく、プログラムの表現の幅を拡げる能力をもつ。後者は、たとえば次のようなことを意味している。

すこし大きなプログラムの場合、一本調子で先頭から最後まで書き下すのでなく、次のような体系化を行うのが望ましい。

（a）プログラムの機能を、それぞれがまとまりのある部品に分解して、部品の１つひとつを関数として表現する。
（b）関数間の関係を明らかにして、階層化、並列化、共通化し、プログラムを１つのシステムとして構成する。

関数はこのようなプログラムの体系化に威力を発揮する。

上記の体系化については「第４章　プログラムの構造」で述べる。

また、プログラミング言語で用意していない、ＯＳ（オペレーティング・システム）やＤＢ（データ・ベース）などの外部機能を、関数を用いてプログラム内で利用可能にすることができる。関数は言語機能を拡張する能力をもっているのである。
多くのプログラミング言語は関数を導入することによって、言語仕様をコンパクトにしている。

関数には次の種類がある。

① 利用者定義関数
② 組み込み関数
③ 外部関数

利用者定義関数は、利用者がプログラム内で定義し、プログラム内で利用するものである。
プログラミング言語は関数を定義する手段と、利用する手段を用意する。

組み込み関数（built-in）関数はプログラミング言語の機能の１部として予め用意されたものである。
前節の説明で使われた charint、numeric は組み込み関数の１つである。

外部関数は、ＯＳやＤＢなどの外部機能を関数として利用できるようにしたものである。
利用者定義関数も、多くのプログラムで共有するために、外部関数として構成することができる。外部関数の利用には、モジュールともいうファイルに存在する関数をプログラムに読み込む import 文を利用する。
外部関数やモジュールについては「第４章　モジュール」を参照されたい。

（１）利用者定義関数

利用者が定めた機能を関数として定義し、その機能を使用するために、関数定義と関数呼出しを用意する。

関数定義　func―end

```
func 関数名 (［引数 1, 引数 2, … ］) ［static］
     # 機能を実現する処理
end
```

関数呼出し

```
関数名 (［パラメータ 1, パラメータ 2, … ］)
```

（括弧［ ］で囲まれた指定は省略できる）

解説

（ｉ）関数は func 文で定義する。func 文は func で始まり end で終わる。

（ⅱ）func の後ろに“関数名”を指定する。“関数名”は変数と同じように、英数字からなる文字列である。“関数名”の後ろには括弧 () で囲まれた“引数 i”

（i=1,2,…）をカンマ（,）で区切って指定する。引数は省略できるが、括弧は省略できない。その後ろには static オプションを指定する。
これらに続いて関数で定義したい機能を実現する処理である文の列を指定する。

(iii) 関数呼出しは、“関数名”とその後ろに括弧 () で囲まれた“パラメータ i”(i=1,2,…) をカンマ（,）で区切って指定する。“パラメータ i”は省略できるが、括弧は省略できない。

(iv) 関数呼出しのパラメータ i（i=1,2,…）は関数定義の引数 i に対応し、呼び出しの際にパラメータ i が引数 i に渡される。この動作をパラメータ・パッシングといい、いくつかの方法がある。代表的な方法は次の２つである。

① 値渡し　call by value
　関数呼出しの際、パラメータの値が対応する引数（変数）に代入される。
　関数内で引数（変数）が変更されても、対応するパラメータに影響を与えない。
　本書では、スカラー・データと論理データ、関数値が対象となる。関数値については後述の「(ix)　関数値」を参照されたい。

② 参照渡し　call by reference
　関数呼出しの際、パラメータの値が収められたフレームのアドレスが、対応する引数に代入される。引数の参照ではこのアドレスが指すデータを使用する。その結果、関数内の引数の変更は、対応するパラメータの変更となる。
　本書では、配列、辞書が対象となる。

(v) 関数呼出しにより開始した関数を終了させるには関数定義の end あるいは return 文、

```
return [(値)]
```

を用いる。戻り値となる値は括弧で囲まれたスカラー・データあるいは論理データである。戻り値の指定は省略できる。

戻り値がある関数呼出しは演算子が使えるところで使用できる。戻り値がない関数呼出しは文と同じように使用できる。

(vi) static オプションは、関数内で使われる変数の寿命に関係している。static オプションをもたない場合、変数は関数が呼び出されたときに生成され（メモリが割り当てられる)、関数が終了すると寿命が尽きる（メモリが解放される)。

static オプションをもつ場合、変数は関数が呼び出されたときに生成されるのは同じだが、関数が終了しても生き続ける。このことが重要な意味をもつ。詳しくは続く説明で紹介する。

(vii) 関数定義の中で func 文を使うことができる。つまり、関数を入れ子にすることができる。その結果、名前のスコープという考え方が生まれる。スコープとは名前がどこから見えるかということである。ここで問題にする名前は変数の名前と関数の名前である。

プログラム内のすべての名前が、プログラム内のどこからでも見えると便利なこともあるが、制約を与える方が良いこともある。名前が使える場所を制約すると、プログラミングで考慮しなければならない範囲が狭まり、不用意な誤りを減らすことができる。多くのプログラミング言語はこうした制約を与えている。

(a) 変数名のスコープ

関数内で定義される変数はその関数内でしか参照できない。内側や外側の関数では参照できない。このような制約があるので、異なる関数では同じ名前が使える。

内側の関数でその関数を含む外側の関数で定義された変数の参照は global 文を使ってグローバル宣言することにより可能となる。

```
global  (変数, 変数, …)
```

(変数は global が宣言された関数を含む外側の関数で定義されたもの)

例

```
func a()
     x = 0
     func b()
          global (x)
          y = x + 1
     end
end
```

関数 b は関数 a の内側にある。関数 b 内で、関数 a で定義された変数 x を参照するために、グローバル宣言している。

当然だが、内側の関数内で定義された変数を外側の関数で参照することはできない。

(b) 関数名のスコープ

複数の関数の定義は並べることができる。また、この関数はそれぞれ内部に関数を入れ子にできる。

一番外側の関数の名前はプログラムのどこからでも見える。この関数をグローバル関数という。

入れ子になっている内側の関数の関数名は、その関数を直接含む関数の到るところから見えるがそれより外側からは見えない。
このような制約を設けることによって、プログラムが複雑に絡み合うのを抑止する。

```
┌ func a
│   …
└ end
┌ func b
│   …
│   func b1
│     …
│   end
│   func b2
│     …
│   end
…
└ end
 ⋮
┌ func z
│     …
└ end
┌ func main
│     …
└ end
```

左記の a, b,… ,z, main がグローバル関数である。

b1, b2 はグローバル関数ではない。すなわち、ローカル関数である。

グローバル関数の名前はプログラム内でただ 1 つでなければならない。

プログラム内のグローバル関数の内、1 つだけ、関数名として main が指定できる。関数 main はプログラム内で最初に実行する関数である。

static オプションはグローバル関数にだけ指定できる。

(viii) 関数内関数の呼出し

static オプションをもつ関数が 1 度呼び出されていれば、その関数が直接含む内側の関数は、ピリオド（.）を使って、

＜static オプションをもつ関数名＞.＜直ぐ内側の関数名＞

と指定することにより、グローバル関数と同じように扱える。つまり、static オプションをもつ関数に直接含まれる関数は、グローバル関数を経由して呼出すことができる。

このようにして呼び出された内側の関数では、すぐ外側の static オプションをもつ関数内の変数を global 文を使ってグローバル宣言することにより参照することができる。

（ix）関数値

“関数値”とは“値”としての関数である。関数名とその呼出しで関数内の変数全体に割り当てられた領域のメモリ・アドレスの対からなる構造体である。関数の戻り値とは異なることに注意されたい。

関数が一度も呼び出されていない状態では、関数内の変数全体にメモリは割り当てられていないので対になるメモリ・アドレスはない。

関数が呼び出されたとき、関数内の変数全体にメモリが割り当てられ、関数名とそのメモリ・アドレスの対が“関数値”となる。

static オプションをもたない関数の呼出しの場合、関数呼出しで関数内の変数全体のメモリ・アドレスが確保されるが、戻った段階でそのメモリは解放されるので、一度も呼び出されていない状態になる。

static オプションをもつ関数の呼出しの場合、関数呼出しで関数内の変数全体のメモリ・アドレスが確保され、戻った段階でもそのメモリは解放されないので、そのメモリ・アドレスが関数名と対になってその呼出しの“関数値”となる。

“関数値”は変数に代入でき、その変数を使って関数呼出しができる。

“関数値”のメモリ・アドレスが重要な意味をもつのは「(viii) 関数内関数の呼出し」の場合である。

“関数値”が関数名をもたないとき、値は null（ヌル）であるという。変数 x は組み込み関数 null() を使用して null にすることができる。

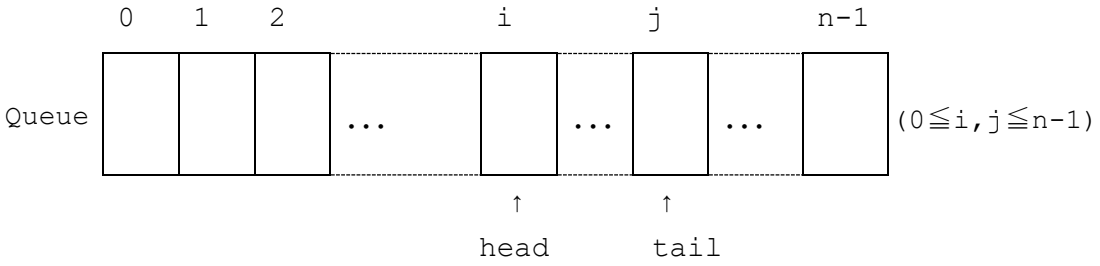
```
x = null()
```

（２）static オプションつき関数の利用

static オプションつき関数は複数の関数間で共有データを扱うために必要となる。

（ｉ）待ち行列

「２．４　データ構造」で述べた待ち行列に対するプログラムを考えよう。



待ち行列 Queue への要素の登録（enqueue）と取り出し（dequeue）は次のプログラムで行なう。

```
func fqueue () static
   Queue = (n)[]                          # 要素数 n の配列
   head = 0                               # 先頭の要素番号を 0 にする
   tail = 0                               # 末尾の要素番号を 0 にする
   return
# enqueue
  func enqueue(d)
   global (Queue,head,tail)
       tail += 1
       tail %= n
       if (((tail+1)%n)==head)
            # オーバーフローメッセージを出力
            return
       else
            queue[tail] = d
            return
       end
  end    # enqueue 関数の end
# dequeue
  func dequeue()
```

```
    global (Queue,head,tail)
        if (head == tail)
            return
        end
        rv = Queue[head]
        head += 1
        head %= n
        return (rv)
    end   # dequeue 関数の end
end     # fqueue 関数の end
```

上の関数 fqueue を利用するプログラムは次の通りである。

```
fqueue()  # 待ち行列を初期化する
      ⋮
fqueue.enqueue(e)  # 待ち行列にデータ e を登録する
      ⋮
w = fqueue.dequeue() # 待ち行列からデータを取り出す
```

（ⅱ） 連結リスト

「２．４　データ構造」で述べた連結リスト（図 3-5）に対するプログラムを考えよう。「２．４　データ構造」では連結リストをつくるためにポインタ・データと構造体を用いた。

ここでは、ポインタ・データの代わりに“関数値”を用い、構造体の代わりに static オプション付きの関数を用いる。

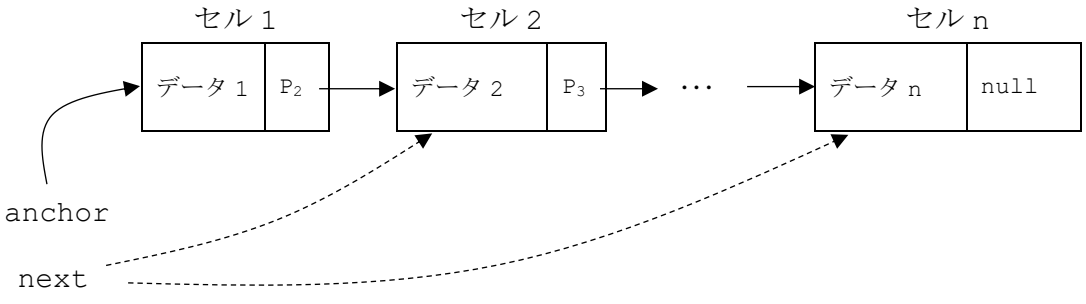


図 3-5　連結リスト

```
# cell はセルを確保する関数
func cell(d) static
    next = null()
    data = d
    return
  func set(e)
     global (next)
     next = e
     return
  end
end
# consolist は連結リストを作成する関数
func consolist() static
   anchor = null()                    # 連結リストのトップを指す
   next = null()                      # 作業用の関数値
   first = true                       # 1 回目か否かの判定に使用
  func register(d)
     global(anchor,next,first)
     cell(d)                          # セルを確保
     if (first == true)
        first = false                 # 1 回目
        anchor = cell                 # 1 回目のセルをセット
        next = cell                   # 1 回目のセルをセット
     else                             # 2 回目以降
        next.set(cell)                # 前のセルに現在のセルを指
        next = cell                   # すリンクを張る
     end
  end
end
```

上の関数 consolist を利用するプログラムは次の通りである。

```
     consolist()          # 連結リストを初期化する
          ⋮
     consolist.register(e)  # 連結リストにデータを登録する
                            # e は登録するデータ
          ⋮
```

（３）再帰呼び出し

static オプションをもたない関数は内部から自分自身を呼び出すことができる。このことを再帰呼び出し（recursive call）という。

static オプションをもたない関数は呼出しの度に戻りアドレスと関数内のすべての変数をコール・スタックに配置するしくみを採るので、再帰呼び出しが可能となる。コール･スタックについては「第６章　プログラムのメモリ割り当て」を参照されたい。

注意　static オプションつきの関数も原理的には再帰呼び出しが可能だが、関数が終了しても変数に割り付けられたメモリが残るので、再帰呼び出しを不可とすることにする。

再帰呼び出しの簡単な例は階乗計算である。

例　階乗　n!

以前に while 文の例として階乗計算を取り上げたが、ここでは、再帰呼び出しを使って階乗を計算しよう。

```
func factorial(n)
     if (n == 0)
         return (1)
     else
         return  (n*factorial(n-1))
     end
end
```

5 の階乗を求めるには、上の関数 factorial を呼び出せばよい。そのとき、パラメータには 5 を指定する。

```
result = factorial(5)
```

関数 factorial が最初に呼び出されたとき、変数 n には 5 が渡され、再帰呼び出しで次々と引数 n に 4、3、2、1、0 が渡される。このとき、変数 n は、名前は同じでも呼出し毎に異なる変数である。

n の値が 0 になったとき、呼び出されていた関数は順に戻りはじめ、最初の呼出しに戻ってきたときの戻り値は次を計算した結果となる。

```
5*(4*(3*(2*(1*1))))
```

この結果は 120 である。
従って、result の値は 120 である。

（４）組み込み関数

組み込み関数（built-in）関数はプログラミング言語の機能の１部として予め用意されたものである。func 文による関数定義は必要ない。
一般に、プログラミング言語では数多くの組み込み関数を用意している。
ここでは、本書の説明に必要な関数のみを記載する。

```
chrint(x)
```

・x は文字データ（utf-8 の場合 0x20～0x7E の文字）
・x を 2 進固定小数点数とみなし、戻り値とする

```
float(x)
```

・x は 2 進固定小数点数
・x を 2 進浮動小数点数に変換し、戻り値とする
例　2　→　2.0

```
int(x)
```

・x は 2 進浮動小数点数
・x の小数部を切り捨て値を 2 進固定小数点数に変換し、戻り値とする
例　23.5　→　23

```
null()
```

・パラメータは指定しない
・関数名をもたない関数値（null）を戻り値とする

```
numeric(x)
```

・x は 10 進数の文字列データ（左に符号+、-付いたものも可）
・x を 2 進固定小数点数に変換し、戻り値とする
例　”127”　→　127　（0x7F）

```
str(x)
```

・x は 2 進固定小数点数あるいは 2 進浮動小数点数
・x が 2 進固定小数点数の場合、x を 2 進固定小数点数リテラル（文字列データ）に変換し、戻り値とする
・x が 2 進浮動小数点数の場合、x を 2 進浮動小数点数リテラル（文字列データ）に変換し、戻り値とする

３．８　例外

プログラムをコンピュータ上で動作させるとき、さまざまな不具合に遭遇する。
コンピュータ自身や、接続する装置の障害に対してはＯＳが対処するが、ＯＳの上で動作するプログラムに起因する障害に対しては、プログラム自身で対処する必要がある。

プログラム自身に起因する障害として想定されるものは次である。

・ＯＳやＤＢ機能の操作上の障害
　ファイル・アクセスのプログラム・ミスなどが該当する。

・プログラムが想定しているデータとの不適合
　プログラム実行時に外部から受け取った文字列を数値に変換する際、不正な文字列があるときなどが該当する。

これらの障害が発生すると、ＯＳやＤＢはプログラムに対して障害メッセージを通知するので、プログラムはそのメッセージをキャッチして後処理を行う。

このような例外を想定したプログラムは try 文、

```
try―except―end
```

を使ってプログラミングする。

```
try
  # 例外が発生する可能性がある処理
except
  # 例外が発生したときの後処理
end
```

try の後ろに例外が発生する可能性がある処理を書き、except の後ろには例外が発生したときの後処理を配置する。

本書は例外が発生する可能性があるＯＳやＤＢを扱う外部関数や実行時に外部から情報を入力する関数などを導入していないので、これ以上の説明は省略する。

第３章では、プログラムの表現方法について簡潔な定義を与えた。ここで用いた表現方法は一般に用いられているものだが、この方法にこだわる必要はない。読者はそれぞれの目的に合わせて独自の定義を与えてもかまわない。プログラミング言語とはそういう類のものである。

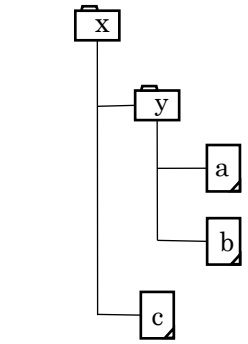
第４章　モジュール

モジュールとは何がしかの機能をまとめたものである。関数もある種の機能をまとめたものなのでモジュールとみなすことができる。複数の関数があって、それらが何がしかの機能を実現するために使われるとすれば、それらの関数をまとめたものはモジュールである。

１つのモジュールを１つのファイルに対応づけると、モジュールを移動の単位にすることができる。モジュール名はファイル名として使われる。

１台のコンピュータの中には多数のファイルが存在する。ファイルはフォルダ（ディレクトリともいう）という容器に格納し、フォルダはファイルや他のフォルダを格納できる。

図 4-1 に示すように、フォルダやファイルは名前をもつ。図中の x,y はフォルダ名、a,b,c はファイル名である。



フォルダ（ディレクトリ）

ファイル

図 4-1　フォルダとファイル

フォルダ x に格納されているファイル c を特定するために、スラッシュ（/）を用いて、

x/c

という文字列を使用する。

フォルダ y に格納されているファイル b を特定するために、

x/y/b

という文字列を使用する。

このような文字列 x/c、x/y/b をパス（path、通り道）という。

ここでは、パス指定にスラッシュ（/）を用いたが、逆スラッシュ（ 日本語のフォントでは¥で代用）やピリオド（.）を用いることもある。

プログラムを作成し、一連の関数をいくつかのファイルにまとめ、コンピュータ上のフォルダに格納すると、コンパイラやインタープリタはそのファイルを処理するためにファイルの保管場所を知る必要がある。その手段がパスである。

プログラミングで複数の関数を定義し、ある関数から別の関数を呼び出すことができる。この 2 つの関数が同一ファイル内の関数であれば特段の指定なしに関数呼出しができる。しかし、呼び出すべき関数が別のファイルであれば、import 文を使って関数から呼び出せるようにする必要がある。

import 文では次のようにパスと関数名を指定する。

```
import (パス [,関数名,関数名,…] )
```

呼出す関数と呼び出される関数を含むファイルが同じフォルダに格納されている場合、パスはファイル名のみである。

関数名の指定が省略された場合には、ファイル内のすべての関数が呼出しの対象になる。

多くのプログラミング言語は、ＯＳやＤＢの機能を外部関数として利用できるようにするライブラリを用意している。プログラムでこれらの外部関数を利用するために、import 文あるいは同様の機能を用意している。

第５章　プログラムの構造

少し大きなプログラムの場合、プログラムをいくつかのモジュールに分け、それらをつなげて構成すると、プログラム制作の見通しが良くなる。モジュールのレベルでプログラムの全体像を描くのが望ましい。
このような観点からプログラミングについて考えよう。

どのようなプログラムも始まりと終わりがある。始まりではプログラム全体の実行環境を整え、終わりではプログラムの後かたづけをしてプログラムを終了する。始まりの実行環境が整ったところで、プログラムの目的とする処理を遂行する。

図 5-1 はプログラムのモジュール構造の例である。

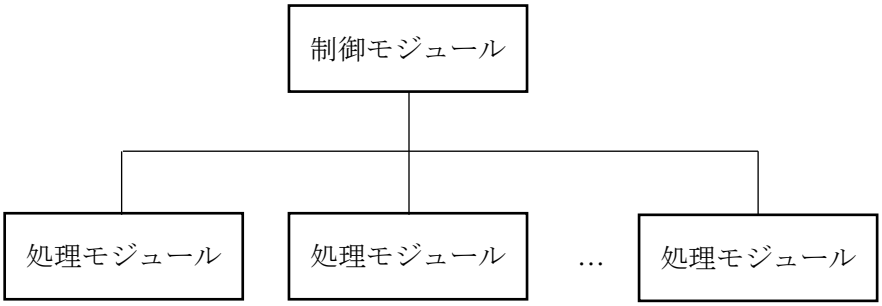


図 5-1　モジュール構造の例

図 5-1 の制御モジュールでは次の機能を用意する。

- プログラム全体の実行環境の準備
- 処理モジュールの呼出し
- プログラムの後かたづけ
- プログラムの終了
- プログラムの共通処理

実行環境の準備では、プログラム全体で共有するデータの初期化、ファイルのオープンなどを行う。
制御モジュールは処理モジュールの呼出しを管理する。順番に呼び出すのか、ユーザからの要求に応じて必要な処理モジュールを呼び出すのか、プログラムの処理手順を決める。
プログラムの後かたづけでは、プログラムが正常に終了する場合や、例外が発生して異常終了する場合に合わせて、ファイルのクローズ、メッセージの出力などを行う。
プログラムの終了はプログラムの制御を呼出し元に返す。
プログラムの共通処理は、

- プログラム全体で使うファイルやＤＢの操作
- 複数のモジュール間で使用する共通データの操作

これについては「３．７（２）static オプションつき関数の利用」を参照されたい。

などである。

制御モジュールを本書の関数を使って表現すると次の通りである。

```
func main() static
    # プログラムの実行環境の準備
    # 処理モジュールの呼出し
    # プログラムの後かたづけ
    # プログラムの終了
    # プログラムの共通処理
    func db()
        try
        # ＤＢの操作
        except
        # 例外発生時の処理
        # プログラムの後かたづけ
        # プログラムの終了
        end
    end
    func comd()
        # 共通データの操作
    end
end
```

第６章　プログラムのメモリ割り当て

第３章で述べたプログラムは文字列の並び、すなわち文字列の系として表現されたものである。この文字列の系のプログラムを原始プログラムと呼ぶ。原始プログラムはそのままではコンピュータ上では動作しないが、人はプログラムが何をしようとしているのか理解できる。原始プログラムをコンピュータ上で動作させるには、２進法表現に変換する必要がある。

2 進法表現に変換する方式として、コンパイラ方式とインタープリタ方式がある。

（1）コンパイラ方式

まず、コンパイラ方式について述べよう。

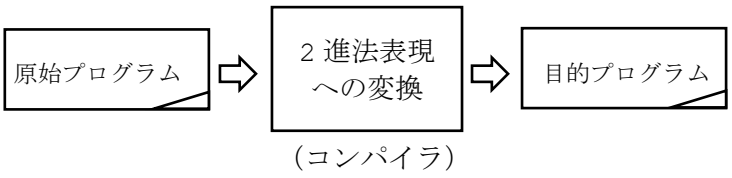


図 6-1　2 進法表現への変換（コンパイラ）

図 6-1 に示すように、コンパイラは原始プログラムを入力して、2 進法表現に変換し、目的プログラムを出力する。

目的プログラムはコンピュータ上で実行可能なマシン命令の列と、マシン命令の処理対象となるデータの列からなる。

ＯＳが目的プログラムをコンピュータのメモリに配置すること（ロードという）によって実行可能となる。

ロードされたプログラムのメモリ上の配置は一般に次の３つの領域からなる。

- 静的領域
- コール・スタック領域
- ヒープ領域

これらの領域の概要は次の通りである。

（ⅰ）静的領域

静的領域は目的プログラム内のマシン命令列が配置されるメモリ内の領域である。プログラムの実行中に変更されることはない。変更されることがない文字列データや数値などもこの領域に配置される。

（ⅱ）コール・スタック

コール・スタックは、関数呼出しの際、戻りアドレスと static オプションをもたない関数内の変数が配置されるメモリ内の領域である。

static オプションをもつ関数の呼出しの場合、コール・スタックには戻りアドレスだけが配置される。この場合、関数内の変数はヒープ領域に配置される。

関数が終了すると、上記で確保されていたメモリ領域を解放する。コール・スタックの遷移については図 6-1 を参照。

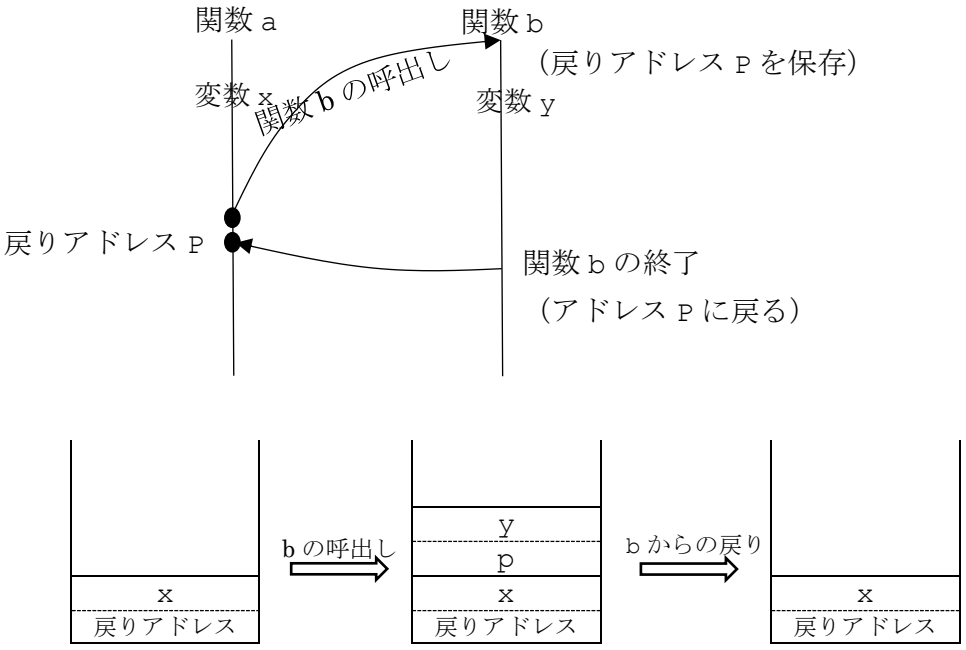


図 6-1　コール・スタックの遷移

（iii）ヒープ領域

ヒープ領域は、関数呼出しの際、static オプションをもつ関数内の変数が配置されるメモリ内の領域である。

ヒープ領域は、関数が終了しても解放されずに残る。このような性質をもつため、ヒープ領域はプログラム全体で共有するデータを配置する目的で使われる。この領域が解放されるのはプログラムが終了したときである。

（2）インタープリタ方式

原始プログラムをコンピュータ上で動作させるもう 1 つの方式として、インタープリタ方式がある。インタープリタは原始プログラムを入力して自身のメモリ内で原始プログラムの文を逐次実行する。

インタープリタ方式については、図 6-2 に示すようなモデルがある。

インタープリタは原始プログラムを入力して各文に対応する解釈ルーチンをもつ。次々と解釈すべき文をとり出すのが制御機構である。とり出した各文を解釈実行するのが解釈ルーチンである。制御機構や解釈ルーチン群は静的領域に配置される。各ルーチンの処理対象となるデータはコンパイラ方式と同じように扱う。つまり、コール・スタックとヒープ領域とが使われる。

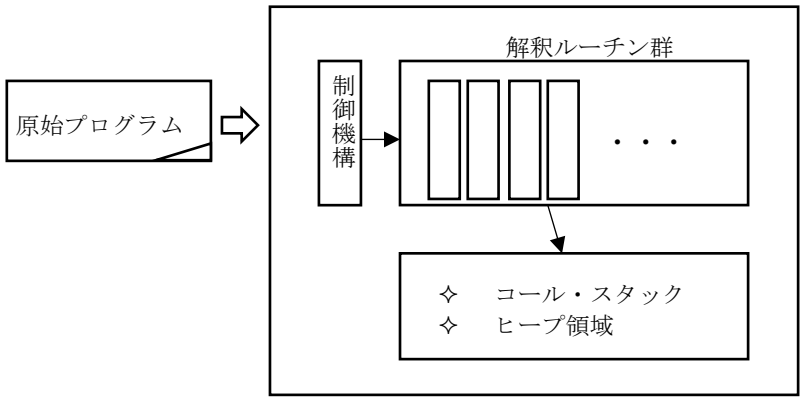


図 6-2　2 進法表現への変換（インタープリタ）

発行年月　２０１８年８月
発行者　　株式会社アイティネット　高橋 哲夫