第10章 いろいろなプログラミング言語

この章はプログラムを書くための言語—プログラミング言語—について概観する。本書では Ruby 言語を用いて情報科学の概念や方法を解説してきたが、実際の情報システムの開発や情報科学の研究開発においては、Ruby 以外にも数多くのプログラミング言語が用いられている。

プログラミング言語には様々な特徴を持ったものがある。例えば数学や人工知能の問題を解くといった特定の目的のプログラムを簡単に作れるように工夫されているものや、プログラムの実行速度ができるだけ速くなるように設計されたもの、プログラムを作るための手間が少なくなるように工夫されたものなど様々である。Ruby はそのように沢山あるプログラミング言語の一つに過ぎない。

そこで、この章ではまずプログラミング言語がどのように発展してきたかを概観して、いくつかの代表的なプログラミング言語がどのような特徴を持っているかを紹介する。最後に、Ruby 以外の言語を使って実際にプログラムを動かしてみて、実行方法や、速度の違いを見ておこう。

10.1 プログラミング言語の発展

プログラミング言語とは、言うまでもなくコンピュータが実行するプログラムを書くために作られた「言語」である。

「情報」(川合編, 東京大学出版会)5章 5.3.2 節参照。

10.1.1 機械語とアセンブリ言語

コンピュータのハードウェアが実行しているのは、四則演算のような単純な命令を並べて作られた機械語プログラムと呼ばれるものである。コンピュータが発明された当初は、人間が機械語プログラムを直接作っていた。例えば $(D + C)(D - C)$ という計算をして A にしまうのであれば、これを足し算や掛け算の命令に分解して次のような命令の並びを作ることになる。

命令	アセンブリ言語による表現	メモリ上の表現
D の値を A にしまう	<code>movl %edx, %eax</code>	9, 80
A を C だけ減らす	<code>subl %ecx, %eax</code>	41, 72
C を D だけ増やす	<code>addl %edx, %ecx</code>	1, 81
A を C 倍する	<code>imull %ecx, %eax</code>	15, 47, 65

実際には、右端の列にあるような 9, 80, 41, ... 数値がメモリ上に並べられ、それをハードウェアが読み取って計算を行うことになる。人間がプログラムを書く場合には、中央の列にあるような単語を並べて書き表わす。この書き方はアセンブリ言語と呼ばれ、1 つ 1 つの行が右列の数値として表現された命令に対応している。

上の例を見て分かるように、機械語のプログラムは沢山の命令を並べて 1 つの式や操作を表わすことになるため、人間がこれを直接書き表わすのは非常に労力のいる作業である。

さらに、機械語プログラムはハードウェアの間での互換性がないこともある。つまりハードウェア、正確には CPU, によって全く違った機械語命令を用意しており、それらに割り当てている数値も違うため、ある種類の CPU のために作られた機械語プログラムは、別の種類の CPU では実行できないのが普通である。逆に言えば、機械語でプログラムを作っていると、新しいハードウェア (CPU) のためにはプログラムを全て作り直さなければいけなかった。

10.1.2 高級プログラミング言語の誕生と発展

その後、より人間に分かりやすい書き方でプログラムを作ることができるようになった。これは高級言語と呼ばれるもので、我々が学んだ Ruby もそのひとつである。現在では単にプログラミング言語と言った場合には高級言語のことを意味することが多い。

機械語に対して「高級」だ
という意味である。

最も初期の (高級) プログラミング言語の代表は 1950 年代に発明された FORTRAN, COBOL, LISP などである。FORTRAN は科学技術計算、COBOL は事務処理、LISP は人工知能研究などの記号処理を目的に設計された。これらの言語は今日でも使われている。また、その後には作られた多くのプログラミング言語の基礎にもなっている。

1960 年代から 1970 年代にかけて新しい考え方をとり入れた言語が数多く作られた。代表的なものにはオブジェクト指向の考え方をとり入れた Simula や Smalltalk, 初学者向けに設計された BASIC や Pascal, オペレーティングシステムのようなハードウェアを制御する目的で設計された C, 論理型言語 Prolog, 関数型言語 ML などがある。

1980 年代から 1990 年に入って、これらの言語を様々な目的のために発展させたり機能を追加した言語が作られた。代表的なものとして、C にオブジェクト指向の考え方をとり入れた C++, 文字列を処理する目的で設計された Perl,

本書で紹介した Ruby, 簡単な処理を手軽に書くためのオブジェクト指向言語 Python, 色々なハードウェアで安全に動作させる目的で設計されたオブジェクト指向言語 Java や C#, web ページの中に簡単な処理を埋め込む目的で設計された JavaScript などがある。今日のソフトウェア開発で主に用いられるのはこれらの言語である。

10.2 プログラミング言語の種類

このようにプログラミング言語は沢山のもので作られているが、プログラムを作るときの考え方や、目的などによっていくつかの分類方法がある。

10.2.1 命令型言語と宣言型言語

Ruby もそうであるが多くの言語のプログラムは、「 x と y を足して z に代入せよ」「 n が正の間は以下の命令を繰り返せ」のように、コンピュータに対する命令を並べたものになっている。このような種類の言語は 命令型 (imperative) 言語と呼ばれる。Ruby 以外でも、前出の中では C, C++, Java などは命令型言語に分類される。これらの言語は、機械語プログラムを人間が書きやすくなるように進化させていったものだとも言える。共通するのは、変数に値を代入することができる、条件分岐や繰り返し命令を使って命令の順序を制御する、関数やメソッド によって一連の命令をひとまとめにするといった機能を使ってプログラムを作る点である。

しばしば手続き型 (procedural) 言語とも呼ばれる。

命令型言語に対比されるのは宣言型 (declarative) 言語である。宣言型言語のプログラムは、物事の性質や関係を書き表わしたものであり、コンピュータはそれを基にして答えを探し出すことが計算が行われる。このよう言語は解き方が明らかでない問題を試行錯誤しながら解く場合などに多く用いられている。また、プログラムとしては性質や関係だけを書くことになるので、多くの場合には命令型言語よりも簡潔にプログラムを書くことができる。

以下、いくつかの宣言型言語のプログラムの例を見てみよう。

関数型言語

関数型言語は、数学の関数を使って関係を表わすような宣言型言語である。

以下に Haskell という関数型言語を用いて練習 4.13 (p.61) で紹介した Eratosthenes の篩を定義した例を示す。

```
1 primes = sieve [2..]
2   where sieve (p:xs)
3         = p : sieve [ x | x <- xs, x `mod` p > 0 ]
```

この例は次のような関係を記述している。

- 素数の列 (`primes` とは、2 から始まる数列 (`[2..]`) を篩 (`sieve`) にかけてのものである。
- 数列を篩にかけるとは、数列の先頭の値 (`p`) に「先頭以外の数列 `xs` のうち、`p` で割り切れないものを残した列」をつなげた (`:`) ものである。数学である条件を満たす値の集合を定義するとき $\{x \mid x \in xs \wedge x \% p > 0\}$ のような書き方をするが、`[ ]` の部分がこれに似せてあることが分かるだろう。

この定義では列が持つべき性質だけを記述しており、どのような順序で値を求めるかについては書いていないことに注意してほしい。宣言型言語は、このような定義から適切な順序で計算を進める機能を持っている。

練習 10.1 (関数型言語のプログラム) 上に示したプログラムを `primes.hs` という名前のファイルに保存せよ。さらに、以下のようにして「ターミナル」から Haskell 言語の処理系を実行し、素数列が求められていることを確かめよ。

```

1 cm12345$ hugs                Haskell 言語を実行する
2 Hugs> :load primes.hs        primes.hs を読み込む
3 Main> take 10 primes         先頭から 10 個取り出す
4 [2,3,5,7,11,13,17,19,23,29]
5 Main> primes !! 100         100 番目を取り出す
6 547
7 Main> :quit                  Haskell 言語を終了する
8 cm12345$ █

```

練習 10.2 (関数型言語による整列) 以下は Haskell によって書かれた整列プログラムである。

```

1 qsort [] = []
2 qsort (a:xs) = qsort [x | x <- xs, x <= a ]
3                ++ [a] ++ qsort [ x | x <- xs, a < x ]

```

このプログラムは、1 行目が「空の列 `[]` は整列されている」ことを表わしている。2 行目から 3 行目は、先頭が `a`、残りが `xs` であるような列を整列すると、`xs` の中で `a` 以下の値を残した列を整列したものと、`a` と `xs` の中で `a` より大きい値を残した列を整列したものを順につなげた (`++`) ものであることを示している。

練習 10.1 を参考にして、`qsort [3,1,4,1,5,9,2]` などのような式を計算すると整列された列が求められることを確認せよ。

論理型言語

論理型言語は、論理式を使って関係を表わすような宣言型言語である。以下には Prolog という論理型言語を用いたプログラムを示す。

```

1 mother(kishiyoko, abeshinzo).      %岸洋子は安部晋三の母
2 father(abeshintaro, abeshinzo).   %安部晋太郎は安部晋三の父
3 father(abeshintaro, kishinobuo).  %安部晋太郎は岸信夫の父
4 father(kishinobusuke, kishiyoko). %岸信介は岸洋子の父
5
6 parent(X,Y) :- father(X,Y).       %X が Y の父ならば X は Y の親
7 parent(X,Y) :- mother(X,Y).      %X が Y の母ならば X は Y の親
8 sibling(X,Y)                       %Z が X の親で Z が Y の親ならば
9     :- parent(Z,X), parent(Z,Y).  %X は Y の兄弟
10 grandparent(X,Y)                  %X が Z の親で Z が Y の親ならば
11     :- parent(X,Z), parent(Z,Y).  %X は Y の祖父母

```

このプログラムは「岸洋子は安部晋三の母である」というような事実は `mother(kishiyoko, abeshinzo)` という形で表わし、「Z が X の親でかつ、Z が Y の親であるならば、X と Y は兄弟である」という規則を `sibling(X,Y) :- parent(Z,X), parent(Z,Y).` という形で表わす。このような関係だけを書き表わしておく、あとは自動的に「安部晋太郎と岸信夫は兄弟か?」や「安部晋太郎の祖父母は誰か?」といった質問の答えを自動的に探してくれる。

下は上の定義を読み込んだ上で答えを探す様子である。下線部が入力で、Yes, No や変数 X にあてはまる値などが答として表示されている。

```

1 ?- sibling(abeshinzo,kishinobuo). %兄弟かどうか調べる
2 Yes
3 ?- grandparent(X,abeshinzo).    %祖父母を探す
4 X = kishinobusuke ;                %答が表示。さらに探す
5 No                                  %もういない

```

10.2.2 オブジェクト指向言語

第 8 章で紹介したように、「もの」を中心にプログラムを作ってゆく種類の言語としてオブジェクト指向言語がある。特徴的な機能には、データとそれに対する処理(メソッド)をまとめた単位であるクラス定義、継承による定義の流用などがある。

プログラミング言語が命令型か宣言型であるかという分類と、オブジェクト指向であるかどうかは直交する性質のものなので、例えばオブジェクト指向かつ論理型であるようなプログラミング言語もある。

オブジェクト指向言語は、今日のソフトウェア開発で中心的に用いられている。その理由の 1 つは、オブジェクト指向言語には、色々なプログラムで使われる典型的な定義をまとめたライブラリが充実しているためである。例えばウインドウ、メニュー、ボタンなどを用いたグラフィカルユーザインターフェース (GUI) を持ったプログラムを作る場合、ボタンやメニューなどの定義はほとんどのプログラムで共通している。オブジェクト指向言語では、これらの定義をクラスライブラリとして用意することで、容易にプログラムを完成させることができるようになる。

また、大規模な情報システムが扱う対象は必然的に複雑なるため、そのようなシステムを作る場合にもオブジェクト指向言語が用いられることが多い。人間が複雑な対象を整理するときは階層的に分類することが多くあるが、クラス定義の継承関係は、この階層性を直接表現できるのが、大きな理由の 1 つである。

10.2.3 スクリプト言語

スクリプト言語とは、簡単な処理を簡単に書けるように工夫された言語である。「スクリプト」という呼び方は、複雑な処理をするプログラム群を制御する台本 (script) のようなプログラムを書くために用いられることに由来している。実際、スクリプト言語の多くは、別のプログラムを起動したり文字列の処理をするための機能が用意されており、例えば「デジタルカメラから読み込んだ画像を、画像処理ソフトウェアを使って縮小して、電子メールソフトウェアを使って指定されたメールアドレスに送信する」のような処理を簡単に書くことができたりする。

スクリプト言語は、インタプリタ実行形式 (後述) のものが多い。これは、同じプログラムが異なるハードウェアでも動作できるようにする互換性のためと、プログラムを作って即座に実行できるようにするためといった理由による。しかしインタプリタ実行のために実行速度は、他のコンパイル実行形式 (後述) の言語と比べるとあまり速くない。

10.3 プログラムの実行のされ方

コンピュータのハードウェアが直接実行できるのは機械語命令で書かれたプログラムだけであるため、高級言語のプログラムは間接的に実行されることになる。この実行のされ方は、大きく 2 つに分けられる。

1 つはコンパイル実行というもので、本格的なソフトウェアはこの方法で実行されるものが多い。もう 1 つはインタプリタ実行というもので、プログラムを作ったその場で実行してみるような場面ではこの方法が採られること

が多い。Ruby の処理系はインタプリタ実行に分類される。さらにまた、この 2 つの方法を組み合わせたような方法もある。

10.3.1 コンパイル実行方式

コンパイル実行方式では、まず最初に高級言語で書かれたプログラム全体を機械語プログラムに翻訳する。この翻訳のことをコンパイル、翻訳をするプログラムのことをコンパイラと言う。そして、翻訳された機械語プログラムを実行すると、ハードウェアは中に書かれている命令を 1 つずつ実行してゆき、元のプログラムが表わしているような計算を行う。

この方式は、次に説明するインタプリタ実行方式に比べると実行が速い。一方、翻訳という作業が入るために、プログラムを作ってから実行するまでの手間がかかる。また、プログラムの動きに異常があった場合、翻訳されたプログラムの動きから元の高級言語のプログラムの誤りを見つけなければいけないため、誤りを発見することが難しくなっている。そのため、この方式は一度プログラムを作って何度も実行するような場合に用いられる。

この方式で翻訳された機械語プログラムは、異なる種類の CPU では実行できない。従って例えば Cell Broadband Engine と呼ばれる種類の CPU で実行するために翻訳された機械語プログラムを、Intel 社の x86 と呼ばれる種類の CPU を持ったパーソナルコンピュータに読み込ませても実行はできず、高級言語で書かれたプログラムを翻訳し直す必要がある。

実行速度が重要なソフトウェア開発に用いられるプログラミング言語処理系の多くは、コンパイル実行方式をとっている。代表的なものとしては FORTRAN, C, C++ などはこの方式を取っている。

10.3.2 インタプリタ実行方式

インタプリタ実行方式では、CPU はインタプリタと呼ばれる機械語プログラムを実行する。インタプリタは、高級言語プログラムの命令を 1 つずつ読み込み、それに応じた処理を行う。

例えば $1+1$ という数式であれば、インタプリタプログラムは

1. 1 文字目「1」を見ると数字なので、それを数値 1 に変換し、変数 a にしよう
2. 2 文字目「+」を見ると足し算記号なので、それを変数 o にしよう
3. 3 文字目「1」を見ると数字なので、それを数値 1 に変換し、変数 b にしよう
4. ここで式の終わりに達したので、変数 o を見ると、+ なので、変数 a と b の値を加え、それを答えとする

というような計算をする。

インタプリタ実行方式はコンパイル実行方式よりも実行速度が遅い。その理由はこのように単純な計算であっても、式や演算子の種類が何であるかを 1 つ 1 つ判断しながら計算を進めてゆくためである。一方、コンパイル作業が不要なので、プログラムを即座に実行することができ、プログラムを簡単に作って実行したい場合に適している。

また、CPU の種類ごとにインタプリタを用意しておけば、同じプログラムを色々なコンピュータで実行することができる。

インタプリタ実行方式は、学習用の処理系やスクリプト言語処理系などで用いられている。

10.3.3 仮想機械による実行方式 *

コンパイル実行とインタプリタ実行を組み合わせた実行方式もある。その 1 つは仮想機械実行方式とよばれるものである。この方式では、架空のコンピュータである「仮想機械」を設計しておき、

- 高級言語のプログラムを仮想的機械の機械語プログラムに翻訳する。
- CPU は「仮想機械」というプログラムを実行する。このプログラムは、仮想的なコンピュータのシミュレータであり、仮想的機械の機械語プログラムを読み込んで実行することができる。

という方法でプログラムを実行するものである。

この方式は 2 つの実行方式の利点を兼ね備えている。つまり、

- インタプリタ方式と同じように、CPU の種類ごとに仮想機械を用意しておけば同じ(仮想機械)機械語プログラムを色々なコンピュータで動かすことができる
- インタプリタ方式より実行速度が速く、必要なメモリの量も小さい

という特徴がある。

仮想機械による実行方式を用いているプログラミング言語処理系には Pascal, Smalltalk, Java などがある。特に最近では仮想機械の作り方を工夫によって、コンパイル実行方式に遜色のない実行速度が達成されてきている。

10.4 C プログラムの紹介

Ruby 以外のプログラミング言語として C 言語のプログラムと、その実行方法を紹介しておこう。C 言語は 1970 年代初頭に UNIX オペレーティングシステムとそのアプリケーションソフトウェアを開発するために設計された

命令型言語である。C 言語の特徴は、高級言語であるにも関わらずハードウェアを直接操作することができる点である。それでいて高級言語であるため、色々な種類の CPU で実行できる機械語プログラムを得ることができた。C 言語が発明された当時、オペレーティングシステムのプログラムの多くの部分でハードウェアを制御する必要があるためにアセンブリ言語によって作られていたことを考えると画期的な特徴だった。

10.4.1 C プログラムの特徴

C プログラムの特徴を見るために、第 6.1.3 節で紹介した Monte Carlo 法による円周率の計算を行うプログラムを C 言語で書き直したのを見てみよう (ファイル 10.1)。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double montecarlo(int n) { /*def montecarlo(n) */
5     int m, i;
6     double x, y;
7     m = 0; /* m=0 */
8     for (i=1; i<=n; i++) { /* for i in 1..n */
9         x = drand48(); /* x = rand() */
10        y = drand48(); /* y = rand() */
11        if (x*x + y*y < 1.0) { /* if x*x + y*y < 1.0 */
12            m = m + 1; /* m = m + 1 */
13        } /* end */
14    } /* end */
15    return m*1.0/n; /* m*1.0/n */
16 } /*end */
17
18 int main(int argc, char** argv) {
19     srand48(time(NULL));
20     printf("%f\n", 4*montecarlo(1000000));
21 }

```

ファイル 10.1: mc.c

多くのプログラミング言語には、複雑な処理を行うための関数やデータの定義を集めたライブラリがある。C 言語にも色々なライブラリが用意されている。実際、文字列の表示や数学関数などは全てライブラリとして用意され

ている。1 行目、2 行目の#include で始まる行は、文字例の表示や乱数を使うための標準的なライブラリを使うための準備である。(Ruby でも数学関数を使うためには include(Math) を実行しなければいけないかったが、それと同じである。)

4 行目から 16 行目までが Monte Carlo 法によって四分円の面積を求めている関数である。右側に並べてあるのは Ruby による定義 (p.94 のファイル 6.2 に示したもの) である。(C 言語では /* */ という記号に挟まれた部分が注釈として扱われる。)

C と Ruby の定義を見比べると、細かな点を除けばかなりの部分が似ている一方で、C の 4 行目の定義に double, int という語が加っていたり、5 行目、5 行目のように Ruby にはない命令がある。これらは型に関する宣言であり、変数にどのような種類の値をしまうか、関数がどのような種類の値を受け取り、返すかを表わしている。例えば 5 行目は変数 m, i には整数 (int) をしまうと宣言している。

型に関する宣言は何に用いられるのだろうか。第 6.3.1 節 (p.97) で見たように、実数と整数ではメモリ上の表現が異なり、その大きさも 32 ビットであったり 64 ビットであったりと一通りではない。また、足し算や掛け算のような演算は機械語命令では「整数の足し算」と「実数の足し算」で別の命令が用意されている。コンパイル実行方式の場合、プログラムを実行する前に変数をしまう場所を決めたり、計算式を機械語命令の並びに翻訳することになるが、その際に変数の型を用いて大きさや命令を選択している。例えば変数 m は 32 ビット整数 (int) と宣言されているので、翻訳された機械語プログラムの中では 32 ビット長のレジスタによって表わされ、m+1 という式は整数の足し算命令によって計算されるといった具合である。

レジスタとは CPU の中にあり、一時的に用いられるデータをしまうための回路である。

18 行目からの関数 main は、プログラムを実行したときに実行される関数である。ここでは乱数を使うための準備をして、montecarlo の計算を表示している。

10.4.2 C プログラムの実行

C プログラムは、コンパイル実行するのが一般的である。ここでは、Mac OS X や Unix OS 上で GCC という C 言語のコンパイラを用いて C プログラムを実行する手順を紹介する。

プログラムを作る: エディタを使ってプログラムを作成する。ここではファイル 10.1 の内容を mc.c という名前のファイルに作ることにする。

コンパイルする: 「ターミナル」で次のように入力する。

```
1 cm12345$ gcc -O mc.c
2 cm12345$ █
```

もし入力に間違いがあった場合には、エラーメッセージが行番号とともに表示される。その場合は、エディタに戻って誤りを修正する。

何も表示されない場合は機械語のプログラムが a.out という名前のファイルの中に作られている。

実行する: 続けて「ターミナル」で次のように入力する。

```
1 cm12345$ ./a.out
2 3.140104
3 cm12345$ █
```

練習 10.3 (C プログラムの修正と実行) a) プログラムを実行する際に ./a.out と入力するかわりに time ./a.out と入力すると、実行に要した時間を測定することができる。ファイル 10.1 に示したプログラムをコンパイルして実行し、実行に要した時間を測定せよ。

b) ファイル 10.1 に示したプログラムは 100 万回の繰り返しを行っているが、1000 万回行う場合の時間を測定せよ。

10.4.3 コンパイル実行方式と誤り

インタプリタ実行方式と比べると、コンパイル実行方式はプログラムを作成してから実行するまでの手間が多い。その一方で、コンパイルの際に誤りを見つけることができるので、場合によってはプログラムを実行しなくても間違いを修正することもできる。

どのような誤りを見つけることができるかの例をいくつか見てみよう。まずファイル 10.1 の 7 行目を $k = 0$; と変更したファイル mc1.c を作り、それをコンパイルしてみよう。

```
1 cm12345$ gcc -O mc1.c
2 mc1.c: In function 'montecarlo':
3 mc1.c:7: error: 'k' undeclared (first use in this function)
4 mc1.c:7: error: (Each undeclared identifier is reported only once
5 mc1.c:7: error: for each function it appears in.)
```

コンパイラは 7 行目で宣言されていない (undeclared) 変数 k が使われているとして、誤りを報告する。

さらに、この行を $m = "abc"$; と変更したファイル mc2.c を作り、それをコンパイルしてみよう。

```
1 cm12345$ gcc -O mc2.c
2 mc2.c: In function 'montecarlo':
```

```
3 mc2.c:7: warning: assignment makes integer from
   pointer without a cast
```

コンパイラは (m への) 代入 (assignment) によって、文字列 (pointer) を整数に変換してしまうとして、誤りの可能性を報告する。

C 言語では文字列は pointer という型のデータとして表わされている。

このように C 言語のようなコンパイル実行方式を採用する言語では、変数や関数の名前間違いや、値の種類に関する間違いをプログラムを実行する前にある程度発見することができる。

練習 10.4 (インタプリタ実行方式での間違いの発見) Ruby で書かれた Monte Carlo 法による円周率計算プログラム (p.94 のファイル 6.2) に、上で示したような変更を加えた場合、どのような誤りがいつ報告されるかを調べよ。プログラムのどの行で誤りが起きたと報告されるか?

10.4.4 実行方式と速度

コンパイル実行方式は、高級言語のプログラムを無駄のない機械語プログラムへと翻訳してから実行をするため、インタプリタ実行方式よりも高速である。実際に速度の違いがどの程度あるかを比べてみよう。

まず C 言語で書かれた Monte Carlo 法による円周率計算プログラム (ファイル 10.1) の実行時間を測ってみよう。

```
1 cm12345$ gcc -O mc.c
2 cm12345$ time ./a.out
3 3.143172
4
5 real    0m0.257s
6 user    0m0.244s
7 sys     0m0.004s
```

最後の 3 行が実行時間を示しているが、real と書かれた行が実際にかかった時間である。

次に Ruby での実行時間を測ってみよう。

```
1 cm12345$ irb
2 irb(main):001:0> load("./montecarlo.rb")
3 => true
4 irb(main):002:0> require("benchmark") #時間測定の準備
5 => true
6 irb(main):003:0> puts(Benchmark.measure{
7     montecarlo(1000000) })
8 5.660000 1.500000 7.160000 ( 7.166609)
```

ただしコンピュータは複数のプログラムを交替で動かすことがある。そのような場合には、「実際の」時間全てをこの C プログラムの実行に使っていたとは限らないことに注意せよ。

```
9 => nil
```

表示された数値のうち () 内のものが実際にかかった時間であり、C プログラムを測定したときの real で書かれた行に対応している。

つまり、このコンピュータでは、C プログラムの方が 30 倍近く速かったことになる。もちろんハードウェア、コンパイラ、インタプリタ、動かすプログラムの性質などによってこの差は大きく変化するので、一般的に言えるのはコンパイル実行方式の方がインタプリタ実行方式よりも速いことが多いということだけである。

練習 10.5 (実行方式と速度の比較) 以下に示すのは再帰的アルゴリズムによって Fibonacci 数を計算するプログラム (p.18 に示したファイル 5.1) を C 言語で書いたものである。コンパイルして `time ./a.out 35` のように `./a.out` に続けて整数 k を書いて実行すると `fib(k)` を計算する時間を測ることができる。

実際に計算時間を測定して、Ruby の場合 (66 に示したファイル 5.1) との実行時間の違いを調べよ。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int fibr(int k) {          /* def fibr(k)          */
5     if (k==0 || k==1) {   /*   if k==0 || k==1 */
6         return 1;        /*       1          */
7     }
8     else {                /*   else          */
9         return fibr(k-1) + /*   fibr(k-1) +   */
10            fibr(k-2);    /*   fibr(k-2)    */
11     }                    /*   end          */
12 }                        /* end            */
13
14 int main(int argc, char** argv) {
15     printf("%d\n", fibr(atoi(argv[1])));
16 }
```

ファイル 10.2: fib.c

10.4.5 機械語プログラムへのコンパイル*

機械語プログラムがどのようなものかを見るために、C プログラムからコンパイルされたプログラムを解読してみよう。多くのコンパイラは高級言語のプログラムをコンパイルする際に、アセンブリ言語のプログラムを経由して機械語プログラムに翻訳をしている。アセンブリ言語プログラムの命令は機械語の命令と一対一で対応しているが、人間が読むことができるテキスト形式で表現されているので、これを見るとどのような機械語プログラムが作られているかが分かる。

まずは「ターミナル」で C プログラムをコンパイルする際に以下のように `-S` という指示を加えてコンパイラを実行してみよう。

```
1 cm12345$ gcc -O -S mc.c
```

すると `mc.s` というファイルができる。定義が長いので関数 `montecarlo` に対応する部分だけを抜き出したものが以下である。それぞれの命令についての解説はしないが、右側に C 言語プログラム中の式や命令の対応を書き加えてある。

```
1 montecarlo:
2     pushl   %ebp
3     movl   %esp, %ebp
4     pushl   %edi
5     pushl   %esi
6     pushl   %ebx
7     subl   $12, %esp
8     movl   8(%ebp), %esi
9     movl   $0, %edi
10    testl  %esi, %esi
11    jle    .L4
12    movl   $0, %edi      m = 0;
13    movl   $1, %ebx      i = 1;
14 .L5:
15    call   drand48      x = drand48();
16    fstpl  -24(%ebp)
17    call   drand48      y = drand48();
18    fldl  -24(%ebp)
19    fmul  %st(0), %st   y*y
20    fxch  %st(1)
21    fmul  %st(0), %st   x*x
22    faddp %st, %st(1)  x*x + y*y
23    fldl  1.0
24    fucomp %st, %st    compare x*x+y*y and 1.0
25    fnstsw %ax
26    sahf
27    jbe   .L6          jump L6 if ≤
28    addl  $1, %edi      m = m + 1;
29 .L6:
30    addl  $1, %ebx      i++
31    cmpl  %ebx, %esi    compare i and n
32    jge   .L5          jump L5 if ≥
```

```

33 .L4:
34     pushl   %edi                m
35     fildl   (%esp)
36     movl   %esi, (%esp)       n
37     fildl   (%esp)
38     fdivrp  %st, %st(1)      /
39     addl   $16, %esp
40     popl   %ebx
41     popl   %esi
42     popl   %edi
43     popl   %ebp
44     ret

```

これを見て分かるように、10 行少々だった C プログラムは、40 個くらいの機械語の命令に翻訳されている。これは C 言語では $x*x + y*y < 1.0$ のような 1 行で書かれていた数式の計算は、実際には乗算、加算、比較などの命令の並びに翻訳されていたりするためである。

練習 10.6 (機械語プログラムの編集と実行) 上のようにして得られた `mc.s` はアセンブリ言語のプログラムであるので、これを編集すればプログラムの動きが変わるはずである。次のようにしてこのことを確認せよ。

- `mc.s` の 27 行目の `jbe .L6` という命令は、「直前の $x*x + y*y$ と 1.0 との比較によって、後者 1.0 の方が小さいか等しい場合には .L6 へ進め」という命令である。エディタを使ってこの行を `ja .L6` という命令に変更せよ。この命令は「直前の比較の後者が大きい等しい場合には .L6 へ進め」という意味になる。
- 「ターミナル」で次のように入力して、編集されたアセンブリ言語プログラムを機械語に変換せよ。ファイル名が `mc.c` ではなくて `mc.s` であることに注意せよ。

```

1 cm12345$ gcc mc.s

```

- 機械語プログラム `a.out` を実行し、その結果が、ファイル 10.1 の 11 行目の不等号を反転させたプログラムと同じであることを確認せよ。

練習 10.7 (機械語プログラムの編集) Fibonacci 数を計算する C 言語プログラム `fib.c` をアセンブリ言語プログラムに変換した上で、 $\text{fib}(k-1) + \text{fib}(k-2)$ を行う足し算を引き算に変更せよ。 k の値を変えて $\text{fib}(k)$ をさせ、変更結果を確認せよ。

なお、Intel x86 という種類の CPU では足し算は `addl`、引き算は `subl` という命令になる。