コンピュータ概論B
ー ソフトウェアを中心に ー

#12 コンパイラとインタプリタ

京都産業大学
安田豊

プログラミング言語

- 低水準プログラミング言語と
  - 例えばアセンブリ言語
- 高水準(高級)プログラミング言語
  - 例えば C など
- 低水準言語の使い道
  - 家電製品など極限まで効率を高めたい場合
  - 参考：Internet Week での非接触カード
  - 組み込み機械など

変数とアルゴリズム

- 教科書 pp.117-
- 手続き型言語と非手続き型言語
  - 処理手順を書き下すタイプ
  - 処理手順を意識しないタイプ
- 変数の概念
  - ノイマン型コンピュータの本質
- 問題の「導出を可能にする解法」をアルゴリズムと呼ぶ
  - 手続き型プログラムとはアルゴリズムを「手順重視」で、文法に従い記述したもの

```
K=0
FOR I=0 TO N
  K=K+I
NEXT I
```

- 変数K (ある位置のメモリ) の内容をゼロに
- ラベル10までを変数 I を 1 から N までひとつずつ変化させながら繰り返し
- K に K の内容と I の内容を足したものを記入

FORTRAN

- 科学技術計算用
- 数値演算精度を重視したシステム
- 事務処理計算に(文法上も)特化したCOBOLの対極
- 1955以来長く使われ、多数の数値演算ライブラリが用意されている

```
      K=0
      DO 10 I=0,N,1
        K=K+I
   10 CONTINUE
```

BASIC

- Beginner's All porpose Symbolic Instruction Code
  初心者向け汎用記号命令列(?)
- 1964年から
- 学習が容易
- 簡易な記述
- 長いプログラムをわかりやすく書く仕組みがない
- 8bit マイクロコンピュータと簡易なインタプリタ（後述）とともに 1980 頃大きく普及

```
K=0
FOR I=0 TO N
    K=K+I
NEXT I
```

COBOL

- 事務処理向け
- 自然言語に近く、冗長な記述
  - ビジネスロジックをそのまま記述しバグを減らす
- レコード定義が自然に可能
- 1-Nまでの加算も冗長だが普通に書ける

```
    MOVE ZERO TO K.
    PERFORM LOOP-ADD UNTIL
                    I EQUAL TO N.
....
LOOP-ADD.
    ADD I TO K.
    ADD 1 TO I.
```

```
01 INPUT-RECORD.
   10 ID          PICTURE X(6).
     20 YEAR      PICTURE 9(2).
     20 SEQ       PICTURE 9(4).
   10 LENGTH      PICTURE 9(3)V99.
   10 NAME        PICTURE X(20).

01 OUTPUT-RECORD.
   10 ID          PICTURE X(6).
     20 YEAR      PICTURE 9(2).
     20 SEQ       PICTURE 9(4).
   10 FILLER      PICTURE 9(3)V99.
   10 FILLER      PICTURE X(20).
....
    MOVE ID TO WORKID.
    ADD 1 TO SEQ OF WORKID.
    WRITE OUTPUT-RECORD.
```

C

- システム記述向け
- 1972年に開発（これでも後発）
- 下例が一般的手法
- 右例が再帰呼び出しを利用した記述
  - 0の累計は0と定義
  - 0以上のNの累計はN-1の累計にNを足したものと定義

```
main() {
  int i,k,n;
  n=10;
  k=0;
  for(i=0;i<=n;i++) {
    k+=i;
  };
  printf("%d¥n",k);
};
```

```
int sub(i) {
  if(i==0) {
    return(0);
  } else {
    return(sub(i-1)+i);
  };
}

main() {
  int k,n;
  n=10;
  k=sub(n);
  printf("%d¥n",k);
};
```

非手続き型言語

- データの処理手順を書くのではなく
- データ間の関係を記述する（など）
  - 計算機が自動処理する内容はそこから導き出す
  - アルゴリズムの別の視点からの記述でもある
- 例：論理型言語 Prolog
  - 論理(二者間の関係)を記述すれば
  - 自動的に解を導出することができる
- 例：データフロー
  - 処理するべきデータ間の演算方法を書けば
  - データが揃い次第先へ進み解が出る

Prologによるプログラム例

```
grand(X,Y) :- parent(X,Z), parent(Z,Y).
parent(amy,bob).
parent(amy,jane).
parent(bob,carol).
parent(bob,will).
```

- 手続きを書くのではなくデータの関係を記述
- そこからシステムが解を導出する
- 実際は文頭から手続き的に解く、、、
- 例：
  - amyの親はbobとjane
  - bob の親はcarolとwill
  - 事実を入力すると yes と答え
  - 変数を与えると、あり得る解を返す
  - amyの祖父母は？willが祖父母なのは？

```
% sbprolog
SB-Prolog Version 3.1
| ?- consult('a.pro').
yes
| ?- parent(amy,bob).
yes
| ?- parent(amy,X).
X = bob
X = jane
no
| ?- grand(amy,X).
X = carol
X = will
no
| ?- grand(X,will).
X = amy
no
```

Prolog による1-Nまでの和

```
kei(0,0).
kei(N,K) :- N > 0, N1 is N - 1,
      kei(N1,Kn), K is Kn + N.
```

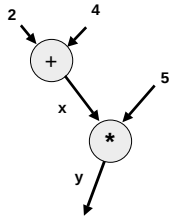
```
| ?- kei(0,0).
yes
| ?- kei(0,100).
no
| ?- kei(0,N).
N = 0
yes
| ?- kei(1,N).
N = 1
yes
| ?- kei(4,N).
N = 10
yes
| ?- kei(1000,N).
N = 500500
yes
```

- 0の累計は0と定義
- 0以上のNの累計はN-1の累計にNを足したものと定義
- 再帰的定義
- Nを指定し、kei(N,K)を評価することでKを導出
- 再帰的手順

データフロー

- 関数型言語の一つ
  - 処理をデータ間の関係として定義
  - データが揃った関数から実行
  - ループは再帰的実行で処理
- 実行順序は指示されない
  - 値が上書きされるようなメモリの使い方ができない
  - ノイマン型である必要がなくなるポイント

{x=2*4, y=x+5}
という計算をフローグラフで表現



データフロー

- 並列処理に有利
  - 順序による制約はノイマン型の致命的弱点
  - 処理の並列度が上がらない
- ハードウェアとの親和性
  - そのままプログラムを回路化できる可能性
  - 演算素子を並列に用意できる
- 並列動作可能な構成
  - ノイマン型では複数用意しても同時に使えない
  - CPU （ノイマン型処理装置）は一点注視タイプ

まとめ

- 低水準言語と高水準言語
  - CPU 依存と非依存
  - 自然言語との距離
- 手続き型言語と非手続き型言語
  - 手続き型：ノイマン型そのもの
  - 非手続き型＝非ノイマン型システムの可能性
    - 非手続き型言語のノイマン型システムによる処理は効率が上がらない場合が多い
  - いずれにしてもアルゴリズムの記述である
- 目的に合わせて多様な言語が存在する
  - データフローチップも画像処理などで実用化

コンパイラとインタプリタ

- プログラミング言語から機械語へ
  - CPUは機械語しか実行できない
  - 機械語によって等価な振舞いをさせなければ
  - どうやって？
- 二つの方法
- コンパイラ
  - 機械語に変換する
- インタプリタ
  - 変換せず真似て動作させる

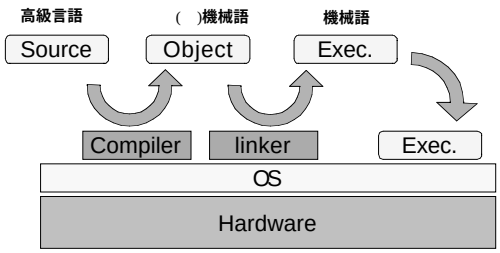
コンパイラ

- Compiler = 編集、編纂
- 機械語に変換して実行
  - 変換前：原始プログラム (Source program)
  - 変換後：目的プログラム (Object program)
  - 多くの OS では目的プログラムをリンク (link) と呼ばれる処理を通してライブラリと結合し、実行可能プログラム (executable program) とし、これを実行する

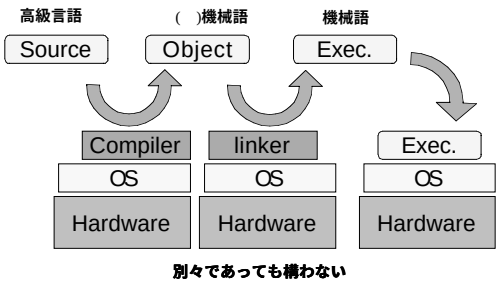
コンパイラ

- 最適化の可能性
  - ループの中の無駄な演算をループの外に
  - 最近の CPU では並列度を上げるためにも使われる（Itanium などでは CPU 中の並列動作可能な演算命令を同時に投入することが可能）
- 変換一回、実行多数回、では高効率
- 秘密保持
  - ソースが得られない場合が多い（逆変換不可）

コンパイラ



コンパイラ

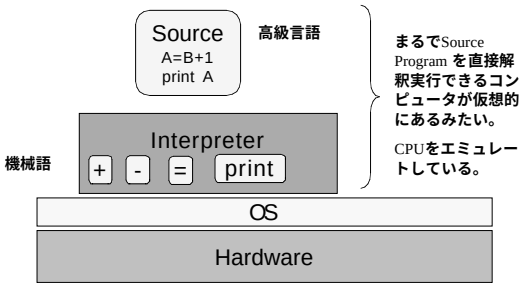


別々であっても構わない

インタプリタ

- プログラムの機能ごとに対応し、部分的に実行可能な機械語コードを予め用意しておく
- 実行したいプログラムを部分的に読み出し、対応する機械語コードを選択して実行
- Interpreter = 通訳、ではあるが変換はしていない
  - 逐語的に処理しているため、と理解するべき
- まるでシミュレーション
  (simulate : 装う / emulation : 真似る)
  - その言語を直接実行できる CPU をソフトウェアで作ったようなもの

インタプリタ



変換はされていない！

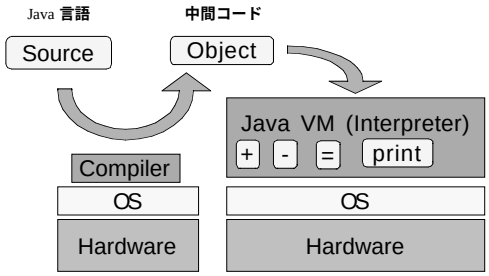
インタプリタ

- 部分的実行が可能
  - 一行ずつテストしながら開発するような作業が可能
- 実行速度が遅い
  - 真似るために必要な機械語は等価変換した機械語より処理量が多い
- 機械語が異なる環境でも動作する
  - Java / iアプリ

Java

- 教科書 pp.119
  - Sun Microsystems が 1995 年に開発
  - C に似た文法
- まず中間プログラムにコンパイル
  - 実行速度を向上させるため
  - バイトコード (Byte Code) と呼ばれる
- インタプリタで実行
  - CPU やハードウェアの差を吸収して実行
  - Java VM (Virtual Machine 仮想計算機) の存在
  - iAPPLや Web で使われる

Java



コンパイラとインタプリタ

- コンパイラ：一括変換して実行
- インタプリタ：逐次真似して実行
- 様々な方式がある
  - Java のような組み合わせ
- Java 自身、VM 高速化のためにいろいろやっている
  - Just In Time コンパイラ：バイトコード実行前に一括して機械語に変換、実行
  - Hot Spot ：バイトコードの一部分（ループの中など）をコンパイルしながら実行
- 工夫の産物である
  - 無意味な分類にとらわれないように