

# コンパイラ資料（言語の基本機能とその実現方法） 担当：佐々木晃

## 本日の話題

- 演習で用いる原始言語(source language)
- 目的言語(hsm 仮想機械) (target language)
- 翻訳方法(compile / translation)

## 演習で用いる原始言語の説明

### OA. トップレベル

プログラムは、

```
int main() {  
    ...  
}
```

の形をしている。{...}の部分はブロックという。{}の間には「変数宣言(declaration)」「文(statement)」を書くことができる。「文」の中には、「式」(expression)を書くことができるものがある。

プログラムの部品（構文）のうち、宣言、文、式、という分類を意識することは重要である。

### OB. 基本機能（プログラムの意味）

putint 文の機能；() の中の値を表示する。

```
int main() {  
    putint(123);  
}
```

数値は、当面整数型のみである。

putint 文の () の中には算術「式」を書くことができる。その式の値（計算結果）を表示する。

```
int main() {  
    putint(123+456);  
}
```

ブロック内には、文(statement)を並べることができる。上から順番に文が実行される。

```
int main() {  
    putint(123);  
    putint(456);  
}
```

空のブロックも可能である。何も実行されない。

```
int main() {  
}
```

## OC. 変数の機能

□宣言(declaration)について

変数宣言は、メモリ上に値（データ）を格納（記憶）させるための領域を確保し、かつその領域に一意的な名前（変数名）を割り当てることを宣言(manifest とも)する機能である。型(type)は、その領域に格納したい値の種類(kind)を表すものであるが、当面「値」は整数型(int)のみとする。

変数(variables)の宣言。

```
int main() {  
    int a;  
}
```

変数は複数個宣言できる。やり方その1

```
int main() {  
    int a;  
    int b;  
}
```

変数は複数個宣言できる。やり方その2

```
int main() {  
    int a, b;  
}
```

□変数の使用(use)について

宣言された変数を用いて、その変数名に対応する領域に値（データ）を書き込んだり、書き込んだ値を読んだりすることができる。ある変数名に対応する領域に値を書き込むことを変数への代入(assignment)、領域にしまわれた値を読むことを変数値の参照(read)と呼ぶ。(今回はC言語にあるポインタ機能はないがポインタも参照(reference)と呼ばれる。これは記憶領域へのアドレスのことである。)

記憶域への値の書き込み(write)と読み出し(read)をあわせて、記憶域へのアクセス(access)と言う。(access = write + read)

変数への代入：

代入は代入文(assignment statement)で行う。

```
int main() {
    int a;
    a = 1;
}
```

代入文の右辺 (“=”の右側 (right hand side ... rhs)) には算術式(arithmetic expression)を書くことができる。

```
int main() {
    int a;
    a = 1 + 2;
}
```

同じ変数に違う値を代入すると、値は上書きされる。

```
int main() {
    int a;
    a = 1;
    a = 2;
}
```

複数個の変数に代入する例

```
int main() {
    int a;
    int b;
    a = 1;
    b = 2;
}
```

今回の言語では、下記は認められない。すべての変数の宣言が終わったあとから文が始まる。(一度文を置いたら、それより後ろには宣言は置けない)

```
int main() { /* まちがい */
    int a;
    a = 1;
    int b;
    b = 1;
}
```

→構文エラー：4行目：int b; 文法違反です。

□変数の値の参照：

変数を使って代入した値は、同じ変数の名前を用いて参照することができる。

```
int main() {
    int a;
    a = 1;
    putint(a);
}
```

4行目：aの値を参照して、それをbに代入する。

```
int main() {
    int a, b;
    a = 1;
    b = a;
}
```

ここで、参照の際には、値（データ）のコピーが必ず起こっていることに注意すること。  
aにしまわれていたデータ「1」がコピーされてbにしまわれる。

4行目：aの値と定数2を足した結果をbに代入する。

```
int main() {
    int a, b;
    a = 1;
    b = a + 2;
}
```

4行目：式の中で変数は何種類でも何回でも参照できる。

```
int main() {
    int a, b;
    a = 1;
    b = a + a;
}
```

```
int main() {
    int a;
    a = 1;
    putint(a + a);
}
```

○変数の宣言と使用での意味エラー、意味の違反(Semantic error / Semantic Violation)

□宣言していない変数に代入してはいけない。

```
int main() { /* まちがい */
    int a;
    b = 1;
}
```

意味エラー：未定義の変数への代入

□宣言していない変数は参照できない。

```
int main() { /* まちがい */
    putint(b);
}
```

意味エラー：未定義変数の参照(b not declared)

```
int main() { /* まちがい */
    int a;
    putint(b);
}
```

意味エラー：未定義変数の参照(b not declared)

□同じ変数名で変数を2個以上宣言できない。

```
int main() { /* まちがい */
    int a;
    int a;
}
```

意味エラー：3行目のaはすでに定義されています。(redefinition of a)

## hsm (スタックマシン型 CPU) の説明

別資料

## hsm (スタックマシン型 CPU) の機械語への翻訳の方法

### □算術式の計算の実現

→前回講義の通り。(逆ポーランド記法による算術式をスタックを用いて計算するやり方)

## □putint 文の翻訳

□準備編：逆ポーランドで出力文を表す。

```
int main(){
  putint(123);
}
```

123 putint (123 をスタックにおく。そしてスタックトップの内容を表示する。)

```
int main(){
  putint(123 + 456);
}
```

123 456 + putint  
(123 をスタックにおく。456 をスタックに置く。加算を適用する。スタックトップの内容を表示する。)

```
int main(){
  putint(123);
  putint(123 + 456);
}
```

123 putint  
123 456 + putint

□本編：

```
int main(){
  putint(123);
}
```

表示したい値をスタックにおいて、WRI 命令を使う。

```
LDC 0, 123 (123)
WRI 0, 0 (putint)
```

ただし、プログラムの最後にきたら停止命令を置く。

```
LDC 0 123
WRI 0 0
HLT 0 0
```

以下では HLT 0 0 省略。

(1) 「式」の部分のコード (式の計算の翻訳)

```
LDC 0 123 (123)
LDC 0 456 (456)
AD 0 0 (+)
```

(2) WRI 命令を使う。式の結果だけがスタックに残っているので、それが表示される。

```
LDC 0 123 (123)
LDC 0 456 (456)
AD 0 0 (+)
WRI 0 0 (putint)
```

□文を続けて書く場合の翻訳方法(文の接続(concatenation))、

```
int main(){
    putint(123);
    putint(123+456);
}
```

1つ目の putint 文に対するコード (翻訳結果) =A

```
LDC 0 123 (123)
WRI 0 0 (putint)
```

2つ目の putint 文に対するコード (翻訳結果) =B

```
LDC 0 123 (123)
LDC 0 456 (456)
AD 0 0 (+)
WRI 0 0 (putint)
```

Aの直後にBを置けばよい。

```
LDC 0 123 (123)
WRI 0 0 (putint)
LDC 0 123 (123)
LDC 0 456 (456)
AD 0 0 (+)
WRI 0 0 (putint)
```

□変数機能の実現

準備編：代入を逆ポーランド記法で表す。

```
int main(){
    int a;
```

```
int b;
a = 123;
b = 456;
a = b;
}
```

123 a := (123 を a に代入する。123 をスタックに乗せ、a に入れる)  
 456 b := (456 を b に代入する。456 をスタックに乗せ、b に入れる)  
 b a := (b の値を a に代入する b の値をスタックに乗せ、a に入れる。)

```
int main() {
  int a;
  int b;
  b = 123
  a = 456 + b * b;
}
```

123 b := (123 を b に代入する。123 をスタックに乗せ b に入れる)  
 456 b b \* + a :=  
 (456 をスタックに乗せ、b の値をスタックに乗せ、b の値をスタックに乗せ、  
 乗算を適用し、加算を適用する。その値を、a に入れる。)

□準備編その2：擬似 hsm コード (hsm の機械語「風」の言葉) で。

```
int main(){
  int a;
  int b;
  a = 123; -> 123 a :=
  b = 456; -> 456 b :=
  a = b; -> b a :=
}
```

DECL a b (二つの変数 a b を使うことを宣言→この命令は実際の hsm にはない)  
 LDC 0 123 (定数 1 2 3 をスタックトップに格納する。ロード)  
 STV 0 a (a にスタックトップの内容を格納する。ストア)  
 LDC 0 456 (定数 4 5 6 をスタックトップに格納する。ロード)  
 STV 0 b (b にスタックトップの内容を格納する。ストア)  
 LDV 0 b (スタックトップに b の内容を置く。ロード)



STV 0 a (a番地にスタックトップの内容を格納する。ストア)

STV 命令や、LDV 命令の第 2 引数では、a, b のような変数名を指定しているが、実際の hsm ではそのような指定はできない。(一種の擬似コードである。)

□本編:hsm コード

hsm には (スタック型の仮想機械では) 変数の機能はない。(一般の CPU では、レジスタを変数のように扱うことができるが、名前は勝手に付けられないし、個数も限られる。) a に入れるとか、b に入れる、といことができない。従ってなんらかの方法で同等のことを実現させなければならない。hsm の場合記憶域はスタックだけなので、スタックを用いる。

→それぞれの変数の値は、スタック上の決まった位置 (番地) にしまうことにする。

→計算を行う途中ではそれを壊さないようにする。すなわち、メモリの確保が必要。これは、スタックを底上げ (という言葉はあまり正確でないが。。) して実現する。

```
int main(){
  int a;
  int b;
  a = 123;
  b = 456;
  a = b;
}
```

1. int a という宣言があるので、スタックの底から数えて 0 番目の位置 (0 番地) に a の値を置くと決める。
2. 次に int b という宣言があるので、スタックの底から数えて 1 番目の位置 (1 番地) に b の値を置くと決める。
3. この二つで宣言は終わりなので、変数に対応する記憶域は 2 個分確保すればよい。2 個分スタックを底上げする。(正確に言えば、int は 4byte(32bit)だから、8byte(64bit)の領域を確保する)

PUSH 0 2 (二つの変数をスタックの底から数えて 2 個分置けるようにする。)

LDC 0 123 (定数 1 2 3 をスタックトップに格納する。ロード)

STV 0 0 (0 番地にスタックトップの内容を格納する。ストア)

LDC 0 456 (定数 4 5 6 をスタックトップに格納する。ロード)

STV 0 1 (1 番地にスタックトップの内容を格納する。ストア)

LDV 0 1 (スタックトップに 1 番地の内容を置く。ロード)

STV 0 0 (0 番地にスタックトップの内容を格納する。ストア)

POP 0 2 (確保した領域を返す。)

## 代入機能に関する翻訳機能のまとめ

### □代入

ある変数  $x$  への値の代入はストア命令 (STV 命令) で実現できる。ただし、

STV 0 「変数  $x$  の値を置くと決めた番地」

となる。

なお STV 命令でスタックトップの値が、指定された番地にコピーされたあとは、スタックトップは1つ下にずれることに注意 (スタックトップの値は消費(consume)されたと考えてよい。)

### □参照

変数  $x$  の値の参照は、ロード命令(LDV 命令)によって実現する。ただし、

LDV 0 「変数  $x$  の値を置くと決めた番地」

### □例 :

- (1) ソースプログラム
- (2) 逆ポーランド
- (3) 擬似 hsm コード
- (4) hsm コード

- (1) ソースプログラム

```
int main(){
    int a, b;
    a = 123;
    b = 789 + a;
}
```

- (2) 逆ポーランド

```
123 a :=
789 a + b :=
```

- (3) 擬似 hsm コード

```
DECL a b
LDC 0 123 (123)
STV 0 a (a :=)
LDC 0 789 (789)
LDV 0 a (a)
AD 0 0 (+)
```

```
STV 0 b (b :=)
```

○(4) hsm コード

```
PUSH 0 2  
LDC 0 123 (123)  
STV 0 0 (a :=)  
LDC 0 789 (789)  
LDV 0 0 (a)  
AD 0 0 (+)  
STV 0 1 (b :=)  
POP 0 2
```

○(1) ソースプログラム

```
int main(){  
    int a;  
    a = 123;  
    putint( a + a );  
}
```

○(2) 逆ポーランド

```
123 a :=  
a a + putint
```

○(3) 擬似 hsm コード

```
DECL a  
LDC 0 123 (123)  
STV 0 a (a :=)  
LDV 0 a (a)  
LDV 0 a (a)  
AD 0 0 (+)  
WRI 0 0 (putint)
```

○(4) hsm コード

```
PUSH 0 1  
LDC 0 123 (123)  
STV 0 0 (a :=)  
LDV 0 0 (a)  
LDV 0 0 (a)
```

```
AD 00 (+)
WRI 00 (putint)
POP 01
```

### □記号表

int a, b;のように a,b 二つの変数が宣言されたとする。このとき、スタックの底から数えて0番目の位置(0番地)にaの値を置き、スタックの底から数えて1番目の位置(1番地)にbの値を置く  
と決めることにしよう。これを記号表として(名前表、シンボルテーブルなどとも)まとめると、

つづり		番地
a		0
b		1

となる。これは、変数から番地への写像(Map)となっている。

表を見ることで、

「変数 a の値を置くと決めた番地」=0

「変数 b の値を置くと決めた番地」=1

とすぐに知ることができる。番地は、宣言した順番に割り当てればよい。(他の順番でも別によい。)