

# プログラム言語論 ブロック構造言語の意味と処理

亀山幸義

筑波大学 情報科学類 講義

## 演習

```
#include <stdio.h>
void foo (int y, int z) {
    printf("%d %d\n", y, z);
}
main () {
    int x = 0;
    foo(++x, ++x);
}
```

答え: "1 2"と"2 1" と"2 2"のどれにでもなり得る (処理系依存). C言語の仕様書では、「左から」とも「右から」とも決めていない (unspecified) **どころか、2つの++を計算してから2つの引数を積む、ということも許している。**

## 前回

- プログラム言語「論」
- コンパイラとインタープリタ
- プログラム言語の構文,BNF
- プログラム言語の意味論とは？

## 前回: miniC 言語に対する意味論

- 状態 (state): 実行時のある瞬間の処理系の状況,
- 式 (expression) の意味: 状態ごとに計算結果の値 (value) を決める .
  - 例: 式  $x + 30$  の, 状態  $[x \mapsto 10, y \mapsto 20]$  における値は 40.
- 文 (statement) の意味: 状態ごとに, その文を実行した後の状態を決める .
  - 例: 文  $x = y + 30;$  の, 状態  $[x \mapsto 10, y \mapsto 20]$  における次状態は状態  $[x \mapsto 50, y \mapsto 20]$ .
- プログラムの意味: 「どの変数も値を持たない」状態から始めて, プログラム中の文を順番に実行して最後に得られる状態 .

## 疑問

「状態」はそんなに単純に決まるのか？

```
int factorial (int x) {
    if (x == 0) return 1;
    else return x * factorial(x-1);
}
```

- factorial(10) を計算している途中では,  $x = 10, 9, \dots, 0$  といった値を全て記憶しているはず .
- 単純に  $[x \mapsto 10]$  といった状態では, 記述できない .
- 拡張が必要 .

## microC から miniC へ

- microC: 関数呼び出しなどのない, C言語の非常に小さなサブセット
- miniC 言語 (演習で使用する言語):
  - 関数呼び出しなどのブロック構造がある .
  - 環境 (状態) に含まれる変数が, 実行時に変化する . (静的束縛, 動的束縛)
  - miniC の意味論を理解するためには, 環境の変化を記述する仕組みが必要 .
- ここでは, 形式的な意味論ではなく, 環境を表すスタックを用いた素朴なインタープリタに基づいた意味論を, miniC に対して展開する .

## C言語のプログラム

```
#include <stdio.h>;
int x, *s;
int data[100];
int sort (int *s) {
    int y;
    ...x...
}
int main () {
    int x;
    ... sort( .. ) ...
    {int x = 10; ...}
}
```

```
#include <stdio.h>;
int x, *s;
int data[100];
int sort (int *s) {
    int y;
    ...x...
}
int main () {
    int x;
    ... sort( .. ) ...
    {int x = 10; ...}
}
```

(正確には, 1つのブロックは, { から } まで)

## ブロック構造

- ALGOL 以来, 多くのプログラム言語が採用 .
- プログラムのテキスト (文面) に対する概念 .
- 変数の有効範囲 (スコープ) と密接に関連 .
- 入れ子構造をなす .

入れ子 (nest)

- 「2つのブロックが, 共通部分をもてば, 必ず, 片方が他方を包含する .

```

let rec eval_exp =
  let apply_binop ope exp1 exp2 =
    ...
  in
  match exp with
  | ...
  | Plus(e1,e2) -> apply_binop (+) e1 e2
  | Times(e1,e2) -> apply_binop ( * ) e1 e2

```

C 言語と違い、入れ子になった関数定義が許される。(eval\_exp の中で、apply\_binop が定義されている。)

- 1つのブロックが、実行時に何度も呼ばれることがある。
- ブロックの実行開始と実行終了は、Last-in, First-Out (First-in, Last-Out とも言う)。
- **スタック**

これ以降では、スタックに基づく形式意味論は、省略して、スタックに基づく実行方式を学ぶ。

- Register (CPU のレジスタ)
- Program Counter (コード領域を指す変数)
- Code (プログラムのコードを格納する領域)
- Environment Pointer (スタックを指す変数)
- Data:
  - Stack (スタック)
  - Heap (ヒープ)

- ブロック構造を持つプログラム言語の処理系で使用。
- ブロックに局所的な変数たちの値を格納。

```

int f (int y) {
  int z = 10;          ----
  return y+z;         z=10
}                    y=11
main () {
  int x = 10;         x=10 x=10 x=10
  x = f(x+1);       ---- ---- ---- ----
}

```

スタックフレーム (stack frame, activation record)

- スタックに積まれる、ひとまとまりのデータ。
- スタック全体は、0 個以上のスタックフレームから構成。
- 典型的なスタックフレームの中身 (関数ブロックの場合)
  - 局所変数 (関数の引数、関数で定義された変数) の値
  - 計算の途中結果
  - 関数の戻り先アドレス (コード領域の番地)
  - 関数が返す値
  - 1つ前のスタックフレームへのポインタ (Control link)
  - 値を参照する変数を探すためのリンク (Access link)

演習で使う処理系では、show 関数により、「スタックフレームごとの局所変数とその値」が表示される。

int f(int x, bool y) {int z; ...}  
環境  $\sigma$  のもとで  $f(e_1, e_2)$  が呼ばれたときの処理:

- 引数  $e_1, e_2$  を現在の環境  $\sigma$  で計算する。
- それらの結果を  $v_1, v_2$  とする。
- 環境スタックに新しいスタックフレームを追加する。
- Environment Pointer が新しいスタックフレームを指すようにする。
- 新しいスタックフレームに以下の値を格納:
  - Control link: 1つ前のスタックフレームへのポインタ。
  - Access link: 値を参照する変数を探すためのリンク。
  - 戻り先アドレス: 関数の計算終了後に戻ってくるべきコード領域の番地。
  - 戻り値を格納するスペース。
  - 関数の実引数  $v_1, v_2$
  - 関数の局所変数  $z$  を格納するスペース

int f(int x, bool y) {int z; ...}  
関数呼び出し  $f(e_1, e_2)$  の中で、return  $e$ ; が実行されたときの処理:

- その時点での状態  $\sigma$  のもとで  $e$  を計算し、その値をスタックフレーム内の「戻り値を格納するスペース」にいれる。
- スタックフレームに保存しておいた戻りアドレスに飛ぶ。(Program Counter にそのアドレスをいれる。)
- 現在のスタックフレームをはずす。(Control link をたどり、Environment Pointer が1つ前のスタックフレームを指すようにする。)
- (局所変数はすべて失われる。)

学習したこと

- ブロック構造をもつプログラム言語
- スタックを用いたインタープリタ
- 関数呼び出しの意味論