プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 3

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。

ML 言語のプログラム

```
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。)

ブロック構造言語の実行方式

- 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。

プログラム実行時のメモリの状況 (2)

- 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)

プログラムスタック (あるいは，環境スタック)

- 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

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。

関数呼出しの意味論 (1)

```
int f (int x) {
   int y = 10;
   return(x * 2 + y);
}
int main () {
   printf(%d\n", f(20*3));
}
```

ここでは、直感的に理解するために、informal に言葉で定義する。(あとで、抽象機械を使ってきちんと定式化する。)
関数 $f(20*3)$ が呼ばれたときの処理 (C 言語の場合):

- 引数 $20*3$ の値を (現在のスタックのもとで) 計算して、60 を得る。
- 関数 f に対応するスタックフレームを生成して、以下の値を持たせて、スタックに push する。
  - 関数の仮引数 x と実引数 60 の束縛。
  - f の局所変数 y を 10 に束縛。
  - 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.
  - $f(60)$ の戻り値 ($f(60)$ の計算結果) を格納するスペース.

関数呼出しの意味論 (2)

関数 $f(20*3)$ の計算が終わったときの処理 (C 言語の場合):

- スタックの一番上のスタックフレーム (関数 f に対応するスタックフレーム) において
  - 戻り値を格納する。
  - 戻り先アドレスを、Program Counter (コード領域を指す変数) に格納する。
  - Control Link を使って、一番上のスタックフレームを pop する (捨てる)。

しかし、この説明では、詳細がはっきりしない⇒後に抽象機械で定式化

動的束縛と静的束縛

```
int x;
int g (int y) {
  return (x + y);
}
int f () {
  int x;
  x = 10;
  return (g(20));
}
int main () {
  x = 5;
  print f();
}
```

問題: `main` から `f` を呼び、そこから `g` を呼んでいる。関数 `g` の中で参照されている変数 `x` は、その直前の `f` で定義されたものか、大域変数か？

- 大域変数⇒静的束縛 (static binding)
- 直前の f で定義⇒動的束縛 (dynamic binding)

動的束縛と静的束縛

束縛 (binding) = 変数の宣言と使用の間の関係。

- 静的束縛: プログラムの文面上で決まる束縛関係。
  - プログラム上で、変数宣言が有効な範囲 (スコープ) が定まる。
  - x に対する変数宣言は、それが有効なスコープ内の変数 x の使用を束縛する。
  - ただし、「入れ子」の時は、最も内側が有効。
- 動的束縛: プログラムの実行順序で決まる束縛関係。
  - 実行時の関数呼び出しの順序により、有効な時間が定まる。
  - x に対する変数宣言は、それを含む関数等が呼出されてから終了するまでの時間、有効。
  - 変数 x の使用は、その時間に有効な変数宣言により束縛される。
  - ただし、「入れ子」の時は、最後 (最近) のものが有効。

動的束縛と静的束縛の比較

- FUNARG 問題: 昔の Lisp 言語では、インタープリタでは動的束縛、コンパイラでは静的束縛であり、同じプログラムでも意味が異なっていた。
- 現代の多くのプログラム言語の変数束縛は、静的束縛。(人間が見てわかりやすい。コンパイラにとってもやりやすい。)
- 現代でも、意図的に動的束縛にしている事がある。
  - 例: 「標準出力先」を一時的に変更する。
  - 例: オブジェクト指向言語のメソッド名参照は、一種の動的束縛。

様々なプログラム言語

入れ子の関数定義が許される言語:
Scheme 言語:

```
(define (fun1 x)
  (define (fun2 y) (+ x y))
  (define (fun3 x) (fun2 10))
  (fun3 2))
(fun1 5)
```

OCaml 言語:

```
let fun1 x =
  let fun2 y = x + y in
  let fun3 x = fun2 10 in
     fun3 2
in
  fun1 5
```

どちらも静的束縛:上記の計算の答は 15。

静的束縛と動的束縛の実現

- 動的束縛: プログラム実行中に、変数参照があったとき (変数の値を知りたいとき)、Control link を逆順にたどれば良い。
- 静的束縛: Control link では役に立たない.
  - スタック上の位置関係ではなく、プログラムの文面上で「現在のブロックの 1 つ外にあるブロック」が何かを知りたい.
  - これは,「現在のスタックフレームの 1 つ前のスタックフレーム」とは必ずしも一致しない.
  - Control link 以外の情報が必要 ⇒ Access link.

Access Link による静的束縛の実現

- Control link: 1 つ手前のスタックフレームへのポインタ.
- Access link: は文面上で「1 つ外」のブロックに対応するスタックフレームへのポインタ。

(詳細は、例により説明。)

Access Link による静的束縛の実現

```
int x;
int g (int y) {                          ----
 return (x + y);                         g:----+
}                                        y=2   |
int f () {                         ----  ----  |
 int x;                            f:    f:    |
 x = 3;                            x=3   x=3   |
 return (g(2));              ----  ----  ----  |
}                            main  main  main  |
int main () {          ----  ----  ----  ----  |
 x = 5;                glob  glob  glob  glob<-+
 print f();            x=?   x=5   x=5   x=5
                       ----  ----  ----  ----
}
```

C 言語の場合、実装は比較的容易 (入れ子の関数定義を許さないため)。

Access Link による静的束縛の実現

```
(define (fun1 x)
  (define (fun2 y) (+ x y))
  (define (fun3 x) (fun2 10))
  (fun3 2))
(fun1 5)
```

```
                        ----
                        fun2 --+
                        y=10   |
                  ----  ----   |
                  fun3  fun3   |access
                  x=2   x=2    | link
            ----  ----  ----   |
            fun1  fun1  fun1 <-+
            x=5   x=5   x=5
----        ----  ----  ----
glob        glob  glob  glob
```


注: 関数型言語の処理系は、通常は「関数クロージャ」(後述) を生成する

評価順序とは

- MiniC 処理系の各モードについて以下の事を調べよ。
  - 静的束縛であるか、動的束縛であるか。
  - 複数の引数がある関数呼出しでは、左の引数を最初に計算するか最後か。

評価順序 (evaluation order, 計算の順序): 1 つのプログラムにおいて、どの部分 (部分プログラム) から計算するか。
評価戦略 (evaluation strategy) とも言う。

評価順序-例 1

式 $((1+2)*(3+4))*0$ の計算方式はいろいろある。

- 最初に $(1+2)$ から計算する。
- 最初に $(3+4)$ から計算する。
- 最初に $(1+2)$ と $(3+4)$ を 2 つ同時に計算する。
- 最初に $\cdots * 0$ から計算する。

評価順序-例 2

```
int fun1 (int x) {
   return x+x;
}
main () {
   print (fun1 (1+2));
}
```

- (1+2) から計算して 3 を得て、次に fun1 3 を計算して、6 を得る。(値呼び計算)
- 式 1+2 のまま、fun1 の仮引数 x に代入して、return (1+2)+(1+2) を得て、最終的に 6 を返す。(名前呼び計算)

値呼び計算 call by value

関数呼出し $f(e)$ の計算 (静的束縛言語と仮定)。

- まず，e を計算して，値 v を得る.
- f が仮引数 x を取る関数のとき、環境 σ に $x = v$ を追加.
- その環境で f の本体を計算して，その結果を全体の答えとする.

```
int fun1 (int x) { return x+x; }
int fun2 (int x) { return 0; }
```

という定義のもとで、

- fun1(power(2,10)) の計算では、「2 の 10 乗」は 1 回だけ計算される。
- fun2(power(2,10)) の計算では、「2 の 10 乗」は 1 回だけ計算される。

多くのプログラム言語 (C, Java, Scheme, ML 等) の関数呼出しが値呼び。

名前呼び計算 call by name

関数呼出し $f(e)$ の計算 (静的束縛言語と仮定)。

- f が仮引数 x を取る関数のとき、環境 σ に $x = e$ を追加.
- その環境で f の本体を計算して，その結果を全体の答えとする.

```
int fun1 (int x) { return x+x; }
int fun2 (int x) { return 0; }
```

という定義のもとで、

- fun1(power(2,10)) の計算では、「2 の 10 乗」は 2 回計算される。
- fun2(power(2,10)) の計算では、「2 の 10 乗」は 0 回計算される。

C 言語のマクロ展開は、名前呼びの一種と考えられる。

必要呼び計算 call by need

値呼びと名前呼びの「良いとこどり」: 名前呼びと同様に計算するが、引数の値を 1 回計算したらその結果を覚えておいて、2 回目以降の計算で使う。

- fun1(power(2,10)) の計算では、「2 の 10 乗」は 1 回計算される。
- fun2(power(2,10)) の計算では、「2 の 10 乗」は 0 回計算される。

ある種のプログラム言語 (Haskell 等) の関数呼出しは必要呼び。
cf. Java の Just-in-Time Compiler: 各クラスは、それが必要になるまで、compile しない。ただし 1 度 compile したら、2 回目以降の呼出しでは compiled code を使う。

その他の戦略

戦略はいろいろあり得る.

- 並行戦略: 同時に計算可能な複数の部分をすべて同時に計算する戦略.
  - 例: $(1+2)*(3+4) \to 3*7$.
- 非決定的な (non-deterministic) 戦略: 「次の状態」が必ずしも一意的でない.
  - 例: $(1+2)*(3+4) \to (1+2)*7$.
  - 例: $(1+2)*(3+4) \to 3*(3+4)$.
  - $\leftrightarrow$ 決定的な戦略: 次の状態が常に一意的に決まる戦略.
- cf. 計算結果が「決定的」: 途中の段階では複数の状態に分岐することがあるが，最終的な計算結果が一意的.

どの戦略が「最善」か？

- 演算の回数 (足し算など) を行なう回数を最小にする戦略 (関数呼び出しそのものの計算時間は考えない) $\Rightarrow$ 最善は必要呼び.
- 実装においては，変数に束縛されるものが値に限定されている方が，一般の式を許す方式より，効率がよくなる．$\Rightarrow$ 最善は値呼.
- 「なるべく有限時間で停止する」戦略は，名前呼びと必要呼び.

多くのプログラム言語の関数呼出しは，値呼びを採用．ただし，プログラム変換などの言語処理においては，必要呼び (や名前呼び) も採用されることがある.

C 言語のマクロと関数

```
#define foo(x) (x+x)
int goo(int x) {
  return x+x;
}
int main () {
  int y = 0;
  y = foo(power(2,10));
  y = goo(power(2,10));
}
```

マクロ展開は、名前呼びと見なせる。関数呼び出しは、値呼びである。

まとめ (1)

束縛 (変数束縛, variable binding)

- 変数の宣言 (declaration) と，変数の使用 (use) の間の関係
- くだけて言えば「この変数は，プログラム中のどこで宣言された変数のことか」を決めるもの

静的 vs 動的, コンパイル時 vs 実行時

- 静的，コンパイル時: 実行するより前の時点 (
- 動的，実行時: 実行中の (どこかの) 時点

ブロック (block): プログラムの文面上の概念

- プログラムのテキスト (文面) 上の概念．(実行時の概念ではない．)
- ALGOL60 言語で用いられ，現代の多くの言語で採用.
- 手続きや関数に対応するブロックと，それ以外のブロック (関数内での小さなブロックなど) がある.
- ブロックは「入れ子 (nest)」構造を持つ.

スタックフレーム (stack frame, 環境 (environment)): 実行時の概念

- スタックフレームは，そのブロックの局所変数の束縛や Control Link 等から構成される.
- ブロックが実行されるごとに，1 つのスタックフレームが生成される.
- スタックは，スタックフレームがいくつか積まれたものである

まとめ (2)

スコープ規則 (scope rule, 環境と束縛のルール);

- 静的スコープ (static scope, lexical scope; 静的束縛); プログラムの字面上 (ブロックの位置関係) で，スコープが決まる．
- 動的スコープ (dynamic scope; 動的束縛)； プログラムを実行した順番で，スコープが決まる．

多くのプログラム言語の変数束縛は静的スコープである．ただし，動的スコープが有効な場面もある．

まとめ (3)

ブロック構造言語に対するスタックにもとづく実装における 2 つの Link:

- Control Link
  - スタックに積まれた 1 つのブロックから，その 1 つ下 (1 つ前) のブロックへのポインタ
  - 現在実行中のブロックの実行終了後に、スタックフレームを 1 つ捨てるために使う。
- Access Link
  - 「プログラムの文面上で、1 つ外のブロック」に対応するスタックフレームへのポインタ (必ずしも 1 つ下 (前) のスタックフレームではない．)
  - 静的束縛において、非局所変数の値を探すために使う。
  - “display” 技法で効率良く実装。(Gabbrielli and Martini, pp.109-111)

Control Link/Access Link の具体的な動作については、Gabbrielli and Martini, Chapter 5 を参照のこと．