

プログラム言語論

亀山幸義

筑波大学コンピュータサイエンス専攻

筑波大学 情報科学類 講義

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 のもの 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);
}
int f () {
    int x;
    x = 3;
    return (g(2));
}
int main () {
    x = 5;
    print f();
}
```

g:----+
y=2
f: f: |
x=3 x=3 |
---- |
main main main |
---- |
glob glob glob<--+
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「関数クローージャ」(後述) を生成することによって静的束縛を実現する。

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 $\dots * 0$ から計算する。

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、

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

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。

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

評価順序とは

先週の課題から:

- 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 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 $\text{fun1 } 3$ 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、 $\text{return } (1+2)+(1+2)$ を得て、最終的に 6 を返す。(名前呼び計算)

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、

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

どの戦略が「最善」か?

- 引数の計算をなるべく繰返さない計算方式が優秀とすれば必要呼びが最善の戦略。
- 実装する際、引数を (値ではなく) 式のまま保存しておく必要がある。名前呼びや必要呼びは必ずしも効率良く実装できない。
- 実用的なプログラム言語の多くにおいて、基本的な関数呼出しは値呼び。

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

- 静的束縛と動的束縛
- Access Link による静的束縛の実装方式
- 3つの評価順序