プログラム言語論
意味論

亀山幸義

筑波大学 情報科学類

No. 2

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 と文脈自由文法

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。

意味を厳密に考える必要性

「プログラムの意味を決める」とは、プログラムを実行するとどのような結果になるかを (厳密に) 決めること。

言葉による説明しかないプログラム言語では‥。

- コンパイラが正しいかどうか確かめられない。
- プログラムの性質を解析・検証できない。
- プログラムの保守・再利用もできない。

プログラム言語の意味論

- 厳密な意味論がある言語: Scheme, Standard ML
- 言葉による意味論がある言語: C, Java, etc.
- 言葉による意味論もない言語: Ruby, etc.

さまざまな意味論の与え方

- 操作的意味論 (operational semantics)
  - small-step semantics (structural operational semantics)
  - bigl-step semantics (natural semantics)
  - abstract machine semantics
- 公理的意味論 (axiomatic semantics)
- 表示的意味論 (外延的意味論; denotational semantics)

ここでは、big-step と表示的意味論と抽象機械意味論。

小さな小さなプログラム言語

自然数の加減算

```
n ::= 0 | 1 | 2 |...
e ::= n | (e+e) | (e-e)
```

例: (1+(2-3))=1 (負の数はないので 0 にする。)

表示的意味論: プログラムの各要素を、何らかの数学的要素に対応付ける。

$$
\begin{aligned}
[\![式]\!] &= 自然数 \\
[\![n]\!] &= n \\
[\![(e_1 + e_2)]\!] &= [\![e_1]\!] + [\![e_2]\!] \text{ 右辺は自然数の加算} \\
[\![(e_1 - e_2)]\!] &= [\![e_1]\!] - [\![e_2]\!]
\end{aligned}
$$

big-step 意味論

式 e を計算した結果が n であることを $e \downarrow n$ と書く。

$$\frac{}{n \downarrow n}$$

$$\frac{e_1 \downarrow n_1 \quad e_2 \downarrow n_2}{(e_1 + e_2) \downarrow n} \quad \text{ただし } n_1 + n_2 = n \text{ とする}$$

$$\frac{e_1 \downarrow n_1 \quad e_2 \downarrow n_2}{(e_1 - e_2) \downarrow n} \quad \text{ただし } n_1 - n_2 = n \text{ とする}$$

これらを使って、

$$\frac{\dfrac{}{1 \downarrow 1} \quad \dfrac{\dfrac{}{2 \downarrow 2} \quad \dfrac{}{3 \downarrow 3}}{(2-3) \downarrow 0}}{(1 + (2-3)) \downarrow 1}$$

抽象機械意味論

CK 機械 (C=code, K=continuation or stack)

$$
\begin{aligned}
< e >_{init} &\rightarrow < e,\ End >_e \\
< n,\ K >_e &\rightarrow < K,\ n >_c \\
< (e_1 \oplus e_2),\ K >_e &\rightarrow < e_1,\ (\Box \oplus e_2) :: K >_c \\
< End,\ n >_c &\rightarrow n \\
< (\Box \oplus e) :: K,\ n >_c &\rightarrow < e,\ (n \oplus \Box) :: K >_e \\
< (n \oplus \Box) :: K,\ n' >_c &\rightarrow < K,\ m >_e \quad \text{ただし } n \oplus n' = m
\end{aligned}
$$

ただし、$\oplus = +, -$

抽象機械での実行

$$
\begin{aligned}
< (1+(2-3)) >_{init} &\rightarrow < (1+(2-3)),\ End >_e \\
&\rightarrow < 1,\ (\Box+(2-3)) :: End >_e \\
&\rightarrow < (\Box+(2-3)) :: End,\ 1 >_c \\
&\rightarrow < (2-3),\ (1+\Box) :: End >_e \\
&\rightarrow < 2,\ (\Box-3) :: (1+\Box) :: End >_e \\
&\rightarrow < (\Box-3) :: (1+\Box) :: End,\ 2 >_c \\
&\rightarrow < 3,\ (2-\Box) :: (1+\Box) :: End >_e \\
&\rightarrow < (2-\Box) :: (1+\Box) :: End,\ 3 >_c \\
&\rightarrow < (1+\Box) :: End,\ 0 >_c \\
&\rightarrow < End,\ 1 >_c \\
&\rightarrow 1
\end{aligned}
$$

K はスタック (残りの計算がつまれている)

抽象機械 (Abstract Machine)

- (本当の) 機械: machine (computer)
  - ハードウェアで実現
- 仮想機械: virtual machine
  - インタープリタで実現される。
  - 通常、instruction set を持ち、本当の機械に近い。
- 抽象機械: abstract machine
  - インタープリタで実現される。
  - 抽象度が高い (高級言語に、より近い)。
  - 通常、instruction set を持たない。

代表的な抽象機械

- SECD machine [Landin 1964]
- CEK machine [Felleisen 1989]
- WAM, CHAM, Krivine machine, ...

(形式) 意味論

ともかく、プログラムの意味を厳密に決めるもの。

- 操作的意味論 (operational semantics)
  - プログラムの動作を記述。
- 公理的意味論 (axiomatic semantics)
  - プログラムが満たす性質を記述。
- 表示的意味論 (外延的意味論; denotational semantics)
  - プログラムが表している (数学的な) ものを記述。

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, bool y) {int z; ...}`
環境 σ 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 σ 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

関数呼出しの意味論

`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 σ のもとで e を計算し，その値をスタックフレーム内の「返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．)

まとめ

- プログラムの意味論
- 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