プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 1

目次

プログラム言語とは？

- プログラムを記述するため，人工的に作られた言語.
- なぜ必要か？
  - 現代コンピュータ == von Neumann Machine (Turing Machine)
  - プログラム内蔵方式
- 何のために？
  - 人間のため vs コンピュータ (ハードウェア) のため
  - 書きやすさ/理解しやすさ/保守のしやすさ vs 実行性能の高さ
- この授業は，主として,「人間のためのプログラム言語」(高級言語) を扱う．(cf. 低級 (low-level) 言語．)

プログラム言語論とは？

- 1 つ 1 つのプログラム言語について順番に語る のではなく
- 種々のプログラム言語に共通する概念について語る
- また、プログラム言語ごとに異なる概念を統一的に理解する

この授業の目的は？

- プログラムを設計・実装するのが，容易になる，楽しくなる。 ではなく
- 新しいプログラムの設計の指針が得られる。 でもなく
- プログラム言語を設計・実装するのが，容易になる，楽しくなる．
- 新しいプログラム言語の設計の指針が得られる。

目次

質問

「インタープリタ (I) とコンパイラ (C) はどう違うか？」

- I は実行が遅くて，C は速い．(?)
- I はプログラムを 1 行ずつ実行して，C は一気に全部実行する．(?)
- 言語によって決まる。C 言語には C しかないし，shell には I しかない．(?)
- I は，プログラムを受け取って，それを実行して答えを返すプログラムであるが，C は，プログラムを受け取って,「それを実行して答えを返すプログラム」を返すプログラムである．

インタープリタ

shell (たとえば csh や bash) のインタープリタとは,

- shell スクリプト (と何らかの入力データ) を入力とし、
- それを 1 行ずつ順番に実行する (実行結果を出力する).

perl のインタープリタとは,

- perl プログラム (と何らかの入力データ) を入力とし、
- それを全部読みこんでチェックしてから実行する (実行結果を出力する).

コンパイラ

ある C コンパイラ:

- C プログラムを入力として、
- 機械語のプログラムを出力する.
- 出力された機械語プログラムを (何らかの入力データを与えて) 実行すると，もとの C プログラムを実行したのと同じ実行結果になる.

Java のコンパイラとは,

- Java プログラムを入力として、
- Java の byte code で書かれたプログラムを出力する.
- 出力された Java byte code プログラムを (何らかの入力データを与えて) Java の実行時処理系で実行すると，もとの Java プログラムを実行したのと同じ実行結果になる.

若干の定式化

言語 X で書かれたプログラム $prog$ を、入力 $x_1, \cdots, x_n$ に対して走らせたときの結果 (の値) を、

$$[\![prog]\!]_X(x_1, \cdots, x_n)$$

と表すことにする。
(便宜上、プログラムが停止しないときは、「停止しない」という値を返すと仮定する。)

解釈系 (interpreter)

(L 言語で書かれた) インタープリタは，S 言語のプログラムを解釈して実行するプログラムで、以下を満たす。

インタプリタ int が満たすべき式

S 言語の任意のプログラム p と任意の入力 $x_1, \cdots, x_n$ に対して、

$$[\![\mathtt{int}]\!]_L(p, x_1, \cdots, x_n) = [\![p]\!]_S(x_1, \cdots, x_n)$$

- shell インタープリタ: S=shell, L=機械語
- perl インタープリタ: S=perl, L=機械語
- JVM 実行系: S=Java byte code, L=機械語
- S=L のとき，meta-circular interpreter という.

翻訳系 (compiler)

(L 言語で書かれた) コンパイラは，S 言語のプログラムを，T 言語のプログラムに翻訳 (変換) するプログラムで、以下を満たす。

コンパイラ comp が満たすべき式

S 言語の任意のプログラム p と任意の入力 $x_1, \cdots, x_n$ に対して、

$$[\![\mathtt{comp}]\!]_L(p) = q$$

$$[\![q]\!]_T(x_1, \cdots, x_n) = [\![p]\!]_S(x_1, \cdots, x_n)$$

(ここで、q は、T 言語で書かれた、あるプログラム)

- C コンパイラ: S=C 言語, T=機械語, L=機械語
- Java コンパイラ: S=Java, T=JVM バイトコード, L=機械語
- Java native コンパイラ: S=Java, T=機械語, L=機械語
- ある変換器: S=Java, T=Scheme, L=Scheme

目次

C 言語プログラム

質問: このプログラムを実行すると何か印刷されるか？

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

答の可能性. (1) "1 2", (2) "2 1" , (3) "2 2"

答. (1) も (2) もあり得る (処理系次第; C 言語は複数の引数を左右どちから計算するかは決めていない (unspecified))。どころか、2 つの++を計算してから、引数の計算結果を関数に渡す、ことも許している。

TYPES メイリングリスト

(Robbert Krebbers 氏の記事より引用, 2013/4/24)
Let me then notice that in the case of C, it is worse than just non-determinism. There are also so called sequence point violations, which happen if you modify an object more than once (or read after you've modified it) in between two sequence points. For example

```
int x = 0;
int main() {
  printf("%d ", (x = 3) + (x = 4));
  printf("%d\n", x);
  return 0;
}
```

not just randomly prints "7 3" or "7 4", but instead gives rise to undefined behavior, and could print arbitrary nonsense. When compiled with gcc at my machine, it for example prints "8 4".

プログラムの意味を定める

「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 (あるいは「形式的に」) 決めること。

言葉による informal な説明しかないプログラム言語では··

- 処理系 (インタプリタ、コンパイラ) を設計できない。
- 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。

さまざまな意味論

- 操作的意味論 (operational semantics)
- 公理的意味論 (axiomatic semantics)
- 表示的意味論 (外延的意味論; denotational semantics)

意味論の厳密な定義があるプログラム言語：

- Scheme:
  `Revised^7 Report on the Algorithmic Language Scheme` (R7RS)
- Standard ML (SML): `The Definition of Standard ML`
- いずれも、言語のコア部分のみ→ 完全な定義は難しい。

目次

小さなプログラム言語

自然数に対する加算と乗算のみを許すプログラム言語 L_1

この言語の具体構文：

$$n ::= 0 \mid 1 \mid 2 \mid \cdots \quad \text{(自然数定数)}$$
$$e ::= n \mid (e) \mid e + e \mid e * e \quad \text{(式)}$$

式の例: 5, 4+3+2, ((1+(2*3)))

プログラム言語 L_1 の抽象構文：

$$n ::= 0 \mid 1 \mid 2 \mid \cdots$$
$$e ::= \mathtt{Int}(n) \mid \mathtt{Plus}(e, e) \mid \mathtt{Times}(e, e)$$

抽象構文は、構文木に相当。具体構文の ((1)) と (1) と 1 は、すべて、$\mathtt{Int}(1)$ という抽象構文に対応。

L_1 の表示的意味

式を自然数に対応付ける。

$$[\![\mathtt{Int}(n)]\!] \stackrel{\text{def}}{=} n$$
$$[\![\mathtt{Plus}(e_1, e_2)]\!] \stackrel{\text{def}}{=} [\![e_1]\!] \dot{+} [\![e_2]\!]$$
$$[\![\mathtt{Times}(e_1, e_2)]\!] \stackrel{\text{def}}{=} [\![e_1]\!] \dot{*} [\![e_2]\!]$$

右辺の $\dot{+}$ は、自然数の加算を表す。$\dot{*}$ も同様。

例

- $[\![\mathtt{Plus}(\mathtt{Int}(1), \mathtt{Times}(\mathtt{Int}(2), \mathtt{Int}(3)))]\!] =$
  $[\![\mathtt{Int}(1)]\!] \dot{+} [\![\mathtt{Times}(\mathtt{Int}(2), \mathtt{Int}(3))]\!] = 1 \dot{+} ([\![\mathtt{Int}(2)]\!] \dot{*} [\![\mathtt{Int}(3)]\!]) = 7$

L_1 の操作的意味

式 e に対する「計算 (操作)」を形式的に記述する。
ここでは、式 e を計算した結果 (値) が n であることを $e \downarrow n$ と書く。

$$\frac{}{\mathtt{Int}(n) \downarrow n}$$

$$\frac{e_1 \downarrow n_1 \quad e_2 \downarrow n_2 \quad (n_1 \dot{+} n_2 = n)}{\mathtt{Plus}(e_1, e_2) \downarrow n} \qquad \frac{e_1 \downarrow n_1 \quad e_2 \downarrow n_2 \quad (n_1 \dot{*} n_2 = n)}{\mathtt{Times}(e_1, e_2) \downarrow n}$$

例: $\mathtt{Plus}(\mathtt{Int}(1), \mathtt{Times}(\mathtt{Int}(2), \mathtt{Int}(3))) \downarrow 7$ の導出

$$\frac{\dfrac{}{\mathtt{Int}(1) \downarrow 1} \quad \dfrac{\dfrac{}{\mathtt{Int}(2) \downarrow 2} \quad \dfrac{}{\mathtt{Int}(3) \downarrow 3} \quad (2 \dot{*} 3 = 6)}{\mathtt{Times}(\mathtt{Int}(2), \mathtt{Int}(3)) \downarrow 6} \quad (1 \dot{+} 6 = 7)}{\mathtt{Plus}(\mathtt{Int}(1), \mathtt{Times}(\mathtt{Int}(2), \mathtt{Int}(3))) \downarrow 7}$$

操作的意味からインタプリタへ

操作的意味	インタプリタ
$\dfrac{}{\mathtt{Int}(n) \downarrow n}$	入力が $\mathtt{Int}(n)$ であったら、n を返す。
$\dfrac{e_1 \downarrow n_1 \ e_2 \downarrow n_2 \ (n_1 \dot{+} n_2 = n)}{\mathtt{Plus}(e_1, e_2) \downarrow n}$	入力が $e_1 + e_2$ であったら、(1) e_1 を計算して結果を n_1 とし、(2) e_2 を計算して結果を n_2 とし、(3) $n_1 \dot{+} n_2$ を計算して結果を返す。
(かけ算の規則)	(同様)

この授業では、操作的意味論を数学的な表記で与えるかわりに、インタプリタとして与える。

インタプリタによる意味の記述

言語 L_1 のインタプリタ in OCaml:

```
let rec eval expr =
  match expr with
    | Int(n)        → n
    | Plus(e1,e2)   → (eval e1) + (eval e2)
    | Times(e1,e2)  → (eval e1) * (eval e2)
```

実行例:

```
eval @@ Int(3)   ==>  3
eval @@ Plus(Int(-3),Times(Int(4),Int(5)))   ==>  17
eval @@ Plus(Int(-3))   ==> (error)
```


コンパイラによる意味の記述

以下の 3 命令をもつスタック機械 S:

- RCst n: 整数 n をスタックに push
- RAdd: スタックの上 2 つの要素を pop し，その和を push
- RMul: スタックの上 2 つの要素を pop し，その積を push

言語 L_1 をスタック機械 S の命令列へ変換した例:

```
Plus(Int(-3),Times(Int(4),Int(5)))
==>
RCst -3; RCst 4; RCst 5; RMul; RAdd

Plus(Times(Int(-3),Int(4)),Int(5))
==>
RCst -3; RCst 4; RMul; RCst 5; RAdd
```

まとめ

プログラム言語を「定める」＝構文と意味を決める

- 構文を厳密に決める。
  - 具体構文: 文字列としてのプログラムや式を定める。
  - 抽象構文: 構文木を定める。
- 意味を厳密に決める。
  - 数学や論理を使う。
  - または、インタプリタを定める。
  - または、コンパイラを定める。(この場合、コンパイル先の言語の意味が別途決まっていることが必要)

この授業での記述

対象言語= 論じる対象であるプログラム言語

- C, ML, Lisp, Java, Ruby, Prolog など

メタ言語= 対象言語を記述したり論じたりするための言語

- 構文の記述: BNF (Backus Normal Form, Backus-Naur Form)
- 意味の記述: 数式、OCaml(または F#)

授業関連情報

教科書

Sestoft 著: "Programming Language Concepts", Springer, 2012.

筑波大学のネットワークから大学電子図書館経由で無料でアクセス可能
(授業ウェブページからリンクあり)

成績評価

授業への出席を前提として、演習レポート 30%, 試験 70% とする。

授業ホームページ・連絡先

(web) http://www.cs.tsukuba.ac.jp/~kam/plm/
(TA) plm@logic.cs.tsukuba.ac.jp (注. 授業後に修正しました)
(Instructor) kam@cs.tsukuba.ac.jp, Room SB1008