プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 1

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語る学問 ではなく
- 種々のプログラム言語に共通する概念、あるいはプログラム言語により異なる概念について、個々のプログラム言語にとらわれずに語る学問.
- そんなことは可能だろうか？意味があるのだろうか？

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。ではなく
- 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。

そんなことを言っても、プログラム言語は、素晴しいものが1つあれば十分では？ 誰かが良い言語を開発してくれれば、自分は使うだけでいいのでは？
→ 亀山なりの答えはありますが、面白い論点なので、今日の宿題にします。

質問

「インタープリタ (I) とコンパイラ (C) はどう違うのですか？」

- 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 (interpreter)

shell (たとえば csh や bash) のインタープリタとは,

- それ自身がプログラムである．
- 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,

- それ自身がプログラムである．
- 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 (compiler)

C のコンパイラとは,

- それ自身がプログラムである．
- C プログラムを入力する．
- 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,

- それ自身がプログラムである．
- Java プログラムを入力する．
- 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 (run time) 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．
- ところで，この実行時処理系は，Java byte code の (Java VM の) インタープリタである．

解釈系 (interpreter)

インタープリタは，言語 S で書かれたプログラムを解釈して実行するプログラム (それ自身は言語 L で書かれている) である．

- 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
- あるインタープリタ: S=Scheme, L=Scheme
- もちろん，L は元々は，C なり Java なりで書かれていて，コンパイラを使って機械語に変換したものであろう．
- S=L のとき，meta-circular interpreter という．

$$[\![p]\!]_S(\vec{x}) = [\![\mathtt{int}]\!]_L(p, \vec{x})$$

翻訳系 (compiler) あるいは変換系 (translater)

コンパイラは，言語 S で書かれたプログラムを，言語 T で書かれたプログラムに翻訳 (変換) する (それ自身は言語 L で書かれている) プログラムである．

- 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=byte code, L=機械語
- Java native コンパイラ: S=Java, T=機械語, L=機械語
- クロスコンパイラ: S=C 言語, T=PowerPC の機械語, L=Pentium の機械語
- ある変換器: S=Java, T=Scheme, L=Scheme

$$[\![p]\!]_S(\vec{x}) = [\![[\![\mathtt{comp}]\!]_L(p)]\!]_T(\vec{x})$$

たくさんの言語を意識する必要。

部分評価 (プログラム特化) の話-1

プログラム mix:

$$[\![[\![mix]\!]_L(p,\vec{x})]\!]_L(\vec{y}) = [\![p]\!]_L(\vec{x},\vec{y})$$

プログラム p への入力 x, y のうち、x だけが、あらかじめわかっている時、「y をもらうと $[\![p]\!](x,y)$ を返す」プログラムを作成する。
部分評価 (partial evaluation) あるいは プログラム特化 (program specialization) という。
例. 「x の n 乗」を計算するプログラムがあるとき、$n = 3$ のときに (異なる x に対して) 頻繁に動かしたいなら、その n に特化した高速プログラムが欲しい。

部分評価 (プログラム特化) の話-2

二村射影 (Futamura Projections) [Futamura 1971]
mix の定義式で、$p = \mathtt{int}$, $\vec{x} = p$ とすると：

$$[\![[\![\mathtt{mix}]\!]_L(\mathtt{int},p)]\!]_L(\vec{y}) = [\![\mathtt{int}]\!]_L(p,\vec{y})$$

右辺は、インタープリタの定義式より、$[\![p]\!]_S(\vec{y})$ に等しい。よって、

$$[\![[\![\mathtt{mix}]\!]_L(\mathtt{int},p)]\!]_L(\vec{y}) = [\![p]\!]_S(\vec{y})$$

結局、$[\![\mathtt{mix}]\!]_L(\mathtt{int},p)$ は、p と同じ動作をする (二村の第 1 射影)。

$$[\![\mathtt{mix}]\!]_L(\mathtt{int},p) = p$$

部分評価 (プログラム特化) の話-3

mix の定義式で、$p = \mathtt{mix}$, $\vec{x} = \mathtt{int}$, $\vec{y} = p$ とすると：

$$[\![[\![\mathtt{mix}]\!]_L(\mathtt{mix},\mathtt{int})]\!]_L(\vec{p}) = [\![\mathtt{mix}]\!]_L(\mathtt{int},\vec{p})$$

となり、右辺は第 1 射影なので、(言語 S で書かれている)p と、同じ動作をするプログラム (言語 L で書かれている) である。つまり、言語 S から言語 L へのコンパイラ (言語 L で書かれている) ができた。(二村の第 2 射影)

$$\mathtt{comp} = [\![\mathtt{mix}]\!]_L(\mathtt{mix},\mathtt{int})$$

部分評価 (プログラム特化) の話-4

同様に、mix の定義式で、$p = \mathtt{mix}, \vec{x} = \mathtt{mix}$ とすると以下が得られる。

$$[\![[\![\mathtt{mix}]\!]_L(\mathtt{mix}, \mathtt{mix})]\!]_L(\vec{int}) = \mathtt{comp}$$

これより、$\mathtt{cogen} = [\![\mathtt{mix}]\!]_L(\mathtt{mix}, \mathtt{mix})$ は「インタープリタが与えられると、コンパイラを生成するプログラム」(コンパイラジェネレータ) であるといえる。(二村の第 3 射影)

部分評価 (プログラム特化) の話-5

こんな机上の空論のような計算をして何が面白いか？

- 実際に、効率よい「コンパイラジェネレータ (cogen)」を生成する mix が、(Scheme 言語や C 言語に対して) 実装された!
- 人手で作成した cogen より効率が良い (ことがある)!
- 理論と実装の結びつきの好例。

ちなみに、、、現在は再び「手動で cogen を書く」ことへの関心が高まっている。
⇒マルチステージプログラミング言語の研究: たとえば、Walid Taha, "Gentle Introduction to Multi-Stage Programming" などを参照。

参考書

この授業で以前に参考にしていた本:

- John C. Mitchell, "Concepts in Programming Languages", Cambridge University Press, 2003.
- Maurizio Gabbrielli, Simone Martini, "Programming Languages: Principles and Paradigms", Springer, 2011.

今年のたね本:

- Sestoft, "Programming Language Concepts", Springer, 2012.

Springer の 2 冊は、筑波大学のネットワークから無料でアクセス可能 (電子図書館から LINK へ)

プログラムの意味

質問: 以下のプログラムはどのような結果になるだろうか。

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

補足: C 言語の++x は、「x の現在値が N のとき、x の新しい値を N+1 にして、N+1 を返す」式。

演習

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

答えの可能性. "1 2", "2 1", "2 2", ...
答え。どれにでもなり得る。C 言語の仕様書では，複数の引数を、左右どちらから評価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 2 つの引数を積む、ということも許している。

“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".

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)
- 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、小さなプログラム言語に対する意味論を与える.

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

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)))

上記のプログラム言語の抽象構文：

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

抽象構文: 構文木に相当するもの，((1)) と (1) と 1 は，異なる (具体) 構文を持つが，構文木としては同じと考えたい．曖昧さはない．

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。

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

ただし、$\dot{+}$ は、式の構文にでてくる $+$ の記号ではなく、2 つの自然数の足し算をあらわす。$\dot{*}$ も同様

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

操作的意味論

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

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

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

例題。$\mathtt{Plus}(\mathtt{Int}(1), \mathtt{Times}(\mathtt{Int}(2), \mathtt{Int}(3))) \downarrow 7$ の導出は以下の通り。

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

小さな言語のまとめ

言語を「定義」するとは？

- 構文を厳密に決める。
  - 具体構文: BNF などを使う
  - 抽象構文: 構文木、曖昧さなし
- 意味を厳密に決める。
  - いろいろな方法 (数学的な記述や論理を使う、インタプリタを使う、etc.)

意味の記述につかう「メタ言語」として、今年は、Sestoft の教科書に従い F# のサブセット (実は、OCaml のサブセットでもある) を使う。OCaml の知識 (「ソフトウェア技法」の授業など) があると有用だが、知らない人向けに前提で授業を進める予定。(これは OCaml の授業ではないので、最小限の利用にとどめる。数学的記法よりは OCaml という程度。)