

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？

- プログラムについて、もっともらしく論じる学問。(?)
- 1つ1つのプログラムについて語るのではなく、プログラム言語(全体)について語る学問。(?)
- 1つ1つのプログラム言語について語るのではなく、種々のプログラム言語に共通する(あるいは相違する)概念について語る学問。
- そんなことは可能だろうか？そんなことに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。
- 新しい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。

授業の目的については、シラバスも参考にすること。

「インタプリタ (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. (?)
- 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.
- 気分の違い (違いはない). (?)

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) インタプリタである.

インタプリタ (interpreter)

shell (たとえば csh や bash) のインタプリタとは,

- 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).

解釈系 (interpreter)

インタプリタは, 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 インタプリタ: S=shell, L=機械語
- perl インタプリタ: 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.

$$\llbracket p \rrbracket_S(\vec{x}) = \llbracket \text{int} \rrbracket_L(p, \vec{x})$$

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

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

プログラム mix :

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

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 をもらって $[[p]](x, y)$ 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。

二村射影 (Futamura Projections) [Futamura 1971]

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

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

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、

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

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

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

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

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

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)

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

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

$$\llbracket \text{mix} \rrbracket_L(\text{mix}, \text{mix}) \llbracket_L(\vec{int}) = \text{comp}$$

これより、 $\text{cogen} = \llbracket \text{mix} \rrbracket_L(\text{mix}, \text{mix})$ 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)

結局のところ、プログラム言語論とは？

- 個々のプログラムの処理方法ではない。
- 個々のプログラム言語の処理方法ではない。
- プログラム言語における種々の「概念」を論じる。
- 個々のプログラム言語でなく、プログラム言語たちに共通する概念・機能に興味がある。
- プログラム言語の一般的な理解や比較に役立つ。
- 洗練されたプログラミングが可能(?)
- 新しいプログラム言語の設計

こんな机上の空論のような計算をして何が面白いのか？

- 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

John C. Mitchell, "Concepts in Programming Languages", Cambridge University Press, 2003.

上記図書の誤植修正が、Mitchell 自身のホームページに置かれている。
<http://theory.stanford.edu/~jcm/books.html>

最近の本 1: Maurizio Gabbriellini, Simone Martini, "Programming Languages: Principles and Paradigms", Springer, 2011.

最近の本 1: Sestoft, "Programming Language Concepts", Springer, 2012.
 (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)

- 命令型言語 imperative language: プログラムは、「命令の列 (手順、手続き)」である。順序付けられた一連の命令を順番に実行していくことにより、計算が行われる。
 - 例: C, Fortran, COBOL, ...
- 宣言型言語 declarative language: プログラムは、(関数や論理式などの) 宣言である。計算の手順は書かず、計算の意味を記述する。
 - 関数型言語の例: Lisp, Scheme, ML (Standard ML, OCaml), Haskell
 - 論理型言語の例: Prolog

では、Java, Ruby などは?

- プログラムは、文をセミコロンで区切って並べたものである。
- 文は、代入文 ($x=e1;$)、while 文 ($\text{while } e1 \text{ } s1;$)、逐次実行文 ($s1 \dots sn$)、条件文 ($\text{if } e1 \text{ } s1 \text{ else } s2$) のいずれかである。ただし、 x は変数、 e_i は式、 s_i は文である。
- 式は、 $x, c, e1+e2$ のいずれかである。ただし、 x は変数、 c は自然数か真偽値の定数、 $e1, e2$ は式である。

問題点: こんな言葉による説明では、定義がはっきりしない。

命令型言語の例: miniC の構文 (やり直し)

プログラム言語の構文定義には、しばしば、BNF が用いられる。
Backus Normal Form (あるいは Backus-Naur Form):

```
<prog> ::= <dec> | <dec> <prog>
<dec>  ::= <type> <var> "(" <plist> ")" <stment>
        | <type> <var> ";"
<stment> ::= <var> "=" <exp> ";"
        | "while" <exp> <stment>
        | "{" <slist> "}"
        | "if" <exp> <stment> "else" <stat>
        | ...
<slist> ::= <stment> | <stment> <slist>
<exp>  ::= <var> | <const> | <exp> "+" <exp> | ...
...
```

命令型言語の例: miniC の構文 (更にやり直し)

BNF を多少崩した形がよく使われる。

$$\begin{aligned} p &::= d \mid d p \\ d &::= t x (pl) s \mid t x; \\ s &::= x = e; \mid \text{while } e \text{ } s \mid \{s\} \\ &\quad \mid \text{if } e \text{ } s \text{ else } s \mid \dots \\ s/ &::= s \mid ss/ \\ e &::= x \mid c \mid e + e \mid \dots \\ t &::= \text{int} \mid \dots \end{aligned}$$

- BNF は、文脈自由文法 (context-free grammar) を与える簡潔な記法。(「オートマトンと形式言語」を参照)
- 実際のプログラム言語の構文は、文脈自由でないことが多い。
- 例えば、C 言語では、「宣言されない変数を使ってはいけない」という構文上の規則があるが、これは BNF で与えられない。

$$\begin{aligned}
 x &::= \dots \\
 c &::= \dots \\
 e, f &::= x \mid c \mid e + f \mid \lambda x. e \mid e f \mid \text{let } x = e \text{ in } f
 \end{aligned}$$

$\lambda x. e$ ラムダ式:

- $f(x) = e$ となる関数 f のことを $\lambda x. e$ と書く。
- $(\lambda x. e)(x) = e$ が成立する。
- $(\lambda x. e)(f)$ はどう定めたらよいか?

let $x = e$ in f let 式:

- $x = e$ として f を計算する。

- 言語の「構文」:
 - 日本語の場合、日本語の文字から構成される任意の文字列のうち、「日本語の文になっているもの」を定める規則が構文規則。
 - プログラム言語の場合も同様。
- 言語の「意味」:
 - 日本語の文の意味は、何だろう?
 - プログラムの意味は何か?
 - プログラムを計算すると、どういう結果が得られるか。

プログラムの意味を考える

以下のプログラムはどのような**結果**になるか。

```

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

```

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) .