プログラム言語論-1

亀山幸義

筑波大学コンピュータサイエンス専攻

第 1 週

Who am I ?

- 亀山幸義の基本データ
  - 筑波大学 大学院システム情報工学研究科 コンピュータサイエンス専攻
  - 総合研究棟 B-1008 (10 階)
  - URL: `http://logic.cs.tsukuba.ac.jp/~kam`
  - 授業ホーム: `../~kam/plm/`
  - E-mail: `kam@cs.tsukuba.ac.jp`
- 研究テーマ
  - プログラム理論・プログラム言語論
    - 関数型プログラム言語、型システム など。
  - ソフトウェア検証・システム検証
    - コンピュータによる定理証明、モデル検査による検証。

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？

- 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．

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．(?)
- 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)．(?)

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

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

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。

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

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`

最近出た本: Maurizio Gabbrielli, Simone Martini, "Programming Languages: Principles and Paradigms", Springer, 2011.

手続き型言語 (命令型言語) vs 宣言型言語

- 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？

命令型言語の例: miniC の構文

- 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 (while e1 s1;)、逐次実行文 (s1 ... sn)、条件文 (if e1 s1 else s2) のいずれかである。ただし、x は変数、ei は式、si 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。

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

- 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。

関数型言語の例: miniML の構文

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

$\lambda x.e$ 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？

$\texttt{let}\ x = e\ \texttt{in}\ f$ let 式:

- $x = e$ として f を計算する。

構文 (syntax) と意味 (semantics)

- 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) .

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.

miniC 言語の意味と実装

操作的意味論 (operational semantics) に基づいて，以下の手順で展開する．

- miniC に対する意味論を与える．
  - 関数呼び出しがなく，main 関数だけ．print 文なども省略．
  - 式の意味．
  - 文の意味．
- miniC への拡張．
  - 関数呼び出し．静的束縛 vs 動的束縛．
  - スタックを用いたインタープリタ．

miniC 言語の (のサブセットの) 意味

意味論の登場人物: 「値」と「状態」

- 値 (あたい,value): 式の計算結果のこと，miniC では整数値と真偽値．
- 状態 (state): 各変数も持つ値を示すもの．
  - 変数の集合から、値の集合への部分関数。
  - 初期状態 []. すべての変数が値を持たない。(対応する値が「未定義」)
  - 状態 $[x \mapsto 0, y \mapsto \texttt{true}]$. 変数 x が値 0 を持ち、変数 y が値 true を持ち，他の全ての変数は値を持っていない状態．
  - 状態 wrong. 例えば, 式 $0 + \mathit{true}$ を実行した後の状態．

注意. 次ページ以降の式や文は、構文解析が終了しているもの (従って、文字列ではなく構文木) と仮定する．

式の意味

「状態 σ における式 e の意味 (状態 σ で e を計算したときの値)」を、$\sigma(e)$ と書き、e の種類に応じた場合分けで定義する。

- $\sigma = \texttt{wrong}$ のとき、$\sigma(e)$ は値を持たない。
- e が変数 x のとき: $\sigma(e)$ は、σ における x の値とする。
- e が定数 c のとき: $\sigma(e) = c$.
- e が $(e_1 + e_2)$ のとき: $\sigma(e) = \sigma(e_1) + \sigma(e_2)$. ただし、$\sigma(e_i)$ の少なくとも一方が整数でないとき、$\sigma(e)$ は値を持たない。
- e が $(e_1 == e_2)$ のとき: $\sigma(e_1)$ と $\sigma(e_2)$ がともに整数で等しいとき、$\sigma(e) = \texttt{true}$ とし、ともに整数で等しくないとき、$\sigma(e) = \texttt{false}$ とし、それ以外の場合、$\sigma(e)$ は値をもたない。
- 以下同様．

例: 状態 $\sigma = [x \mapsto 1,\ y \mapsto 2]$ に対して，$\sigma((x + 1) == y) = \texttt{true}$ であり、$\sigma((x + \texttt{true}) == y)$ は値を持たない。

文の意味

「状態 σ における文 s の意味 (σ において s を実行したときに得られる状態)」を、$\sigma[\![s]\!]$ と書き、s の種類に応じた場合分けで定義する。

- $\sigma = \texttt{wrong}$ のときは、$\sigma[\![s]\!] = \texttt{wrong}$ とする。
- s が代入文 $x = e;$ のとき： $\sigma[\![s]\!] = \sigma[x \mapsto \sigma(e)]$。
  - 状態 $\sigma[x \mapsto v]$ とは，σ とほぼ同様の状態だが，変数 x に対する値が v となる状態のこと．
- s が複合文 $\{s_1\ s_2\ \cdots\ s_n\}$ のとき：
  $\sigma[\![s]\!] = ((\sigma[\![s_1]\!])[\![s_2]\!]) \cdots [\![s_n]\!]$ である．($n = 0$ のときは $\sigma[\![\{\}]\!] = \sigma$)
- s が $(\mathit{if}\ e\ s_1\ \mathit{else}\ s_2)$ のとき:
  - $\sigma(e) = \texttt{true}$ のとき： $\sigma[\![s]\!] = \sigma[\![s_1]\!]$
  - $\sigma(e) = \texttt{false}$ のとき： $\sigma[\![s]\!] = \sigma[\![s_2]\!]$
  - 上記以外のとき： $\sigma[\![s]\!] = \texttt{wrong}$.

文の意味

- s が $\mathit{while}\ e\ s_1$ のとき：
  - $\sigma(e) = \texttt{true}$ のとき、$\sigma[\![s]\!] = (\sigma[\![s_1]\!])[\![s]\!]$ とする。
  - $\sigma(e) = \texttt{false}$ のとき、$\sigma[\![s]\!] = \sigma$ とする。
  - 上記以外のとき、$\sigma[\![s]\!] = \texttt{wrong}$ とする。

注意. 上記の「定義」は，数学的な定義としては完全なものではない。
なぜなら、while 文が無限ループするとき，上記の定義も無限に展開されるので。 ⇒ 不動点意味論 (fixpoint semantics), 表示的意味論 (denotational semantics)

文の意味: 例題

初期状態 σ_0 を、「すべての変数が、値を持たない」状態とする。

以下の文 s に対して、$\sigma_0[\![s]\!]$ はなにか？

```
{ x = 1; y = 0;
  n = 5; f = 0;
  while (f == 0) {
    y = x + y;
    x = x + 1;
    if (x == n)
        f = 1;
    else f = 0;
  }
}
```

miniC 言語の意味論のまとめ

- $\sigma(e)$: 状態 σ における式 e の計算結果 (値).
- $\sigma[\![s]\!]$: 状態 σ において文 s を実行した後の状態．
- miniC プログラムの意味:
  - 初期状態は，全ての変数が値を持たない状態．
  - main 関数に含まれる文全体を 1 つの文と見なして，上記の定義を使う．
  - 最終的に得られた状態が，このプログラムの意味．

意味論 (semantics) とは何なのか？

- 操作的意味論は、プログラム言語の各要素の計算方法を、(自然言語で書くかわりに) 厳密に書きくだしたもの。
- 全ての場合においてどのように計算するかをきちんと書いている。(⇔ 自然言語による意味の記述)
- プログラム言語の処理系も、全ての場合においてどのように計算するかを記述。
- 意味論の記述とインタープリタの記述は、非常に近い。