

プログラム言語論-第2週-

亀山幸義

筑波大学コンピュータサイエンス専攻

筑波大学 情報科学類講義, 2009年4月20日

前回の概要

- プログラム言語論とは?
- インタープリタとコンパイラ (外から見た姿)
- Futamura Projections (研究レベル)
- プログラム言語の構文, BNF

前回: 「外」から見た Interpreter, Compiler

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.

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

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.

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

これらは, インタープリタとコンパイラの1つの見方。しかし, これでは, インタープリタ等の内部構造が見えてこない。ここで, 典型的な構成を見ておこう。

典型的なコンパイラの構成-構文に関わる部分

- 字句解析
 - 入力された文字列を, 基本単位 (トークン, token) の列に変換。
 - トークン: 自然言語では, 「名詞」「助動詞」などに相当。
- 構文解析 (parse)
 - トークン列を, プログラム言語の文法に沿って, 構文木 (parse tree) に変換。
 - 文法に合わない場合は, 構文エラーを出力。
- 意味解析
 - 構文木に対して, 文脈に依存した構文チェックを行う。
 - (今回述べる「プログラムの意味」に関する処理ではない。)

具体的な内容は, 後で。

1 インタープリタとコンパイラの構成

2 構文の定義と処理

3 プログラムの意味

本日の話題

今週と来週の日標: インタープリタを作る。

- インタープリタとコンパイラの構成
- 構文と意味 (続き)
- ブロック構造と変数スコープ, 関数と手続き, 評価順序
- メモリ管理
- 簡単なプログラム言語のインタープリタ

典型的なコンパイラの構成

- 1 Lexical Analysis (字句解析)
- 2 Syntax Analysis (構文解析)
- 3 Semantic Analysis (意味解析)
- 4 Intermediate Code Generation (中間コード生成)
- 5 Code Optimization (最適化)
- 6 Code Generation (最終的なコードの生成)

参考: John Mitchell, "Concepts in Programming Languages", 2003, pp. 50-52.

典型的なコンパイラの構成-残り

- 1 中間コード生成
 - 中間言語 (intermediate language) へ変換。
 - 中間言語は特定の CPU に依存せず, 最適化がやりやすいもの。
 - 基本的な中間言語: CPS (継続渡しスタイル), SSA (単一静的代入)。
 - うまい中間言語を設計することがコンパイラ設計者の腕の見せ所。
- 2 最適化
 - 中間言語で記述されたコード (プログラム) を高速化する。
 - 意味 (計算結果) を変えずにスピードを速くする。
- 3 コード生成
 - 中間言語のコードを, 最終的なターゲット言語 (通常は, 機械語) のコードに変換する。
 - 機械語コードを生成する場合, CPU のもつレジスタへの割当てなどを含む。

上記についての具体的な内容は, 「プログラム言語処理」等の講義を参照。

- ① 字句解析、構文解析、意味解析
- ② 解釈実行
- ③ 実行結果の出力
- ④ cf. Lisp インタプリタ:
 - ① read
 - ② eval
 - ③ print

(変数) $x ::=$ (英文字で始まる英数字列、予約語でない)
 (定数) $c ::=$ 自然数 | true | false
 (プログラム) $p ::= s \mid p ; s$
 (文) $s ::= x := e \mid \text{while } e \text{ do } s \mid \text{begin } p \text{ end}$
 $\mid \text{if } e \text{ then } s \text{ else } s \mid (s)$
 (式) $e ::= x \mid c \mid e + e \mid e = e \mid (e)$

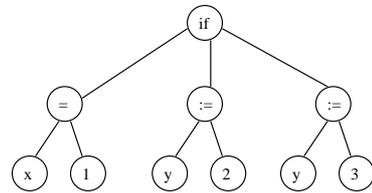
赤字は、先週に比べて追加・加筆した部分。
 予約語: true, while 等、上記の構文定義で使われている英数字列のこと。

- トークン:
- 変数、定数、予約語
 - 記号列 ;, :=, (,), +, =
- 空白文字: スペース、タブ、改行など。

入力文字列の例: "if (x=1) then y:=2 else y:=3"
 対応するトークン列: if, (, x,=,1,), then, y, :=, 2, else, y, :=, 3

提出を要しない課題: miniC の字句解析をおこなうプログラムを、自分の好きなプログラム言語を用いて書きなさい。

トークン列を構文解析木 (parse tree) に変換。



- プログラムの構文解析
 - 文を1つ読む。
 - 次に ; があれば、続けて読む。
 - 次に ファイル終端 があれば、読みこんだ文をリスト状に並べた構文木を返して終了。
- 文の構文解析
 - 次のトークンが、変数、while, begin, if, 左かっこ のどれであるかで場合分け。
 - たとえば、while の場合、次に式を1つ読み、do を読み、文を1つ読み、それらを構文木にしたものを返して終了。
- 式の構文解析
 - 曖昧さ: $x+1=y$ は $(x+1)=y$ か $x+(1=y)$ か。

- 曖昧さ
 - 2つ以上の parse tree が、同じ文字列に対応すること。
 - 例: $1+2*3$
- 曖昧さの解消
 - 結合の優先度を定める。例: 「* は+ より強い」「* は左結合的」
 - 文法で決める。

目標: 0 と + と * からなる式の曖昧さのない文法を定義。ただし、+ は * より結合力が弱い。まだ、+ も * も左結合的とする。

例 1 (曖昧さあり)

$e ::= 0 \mid e + e \mid e * e$

例 2 (少し改善; *は+より強い)

$e ::= f \mid e + e$
 $f ::= g \mid f * f$
 $g ::= 0$

例 3 (更に改善; 左結合的)

$e ::= f \mid e + f$
 $f ::= g \mid f * g$
 $g ::= 0$

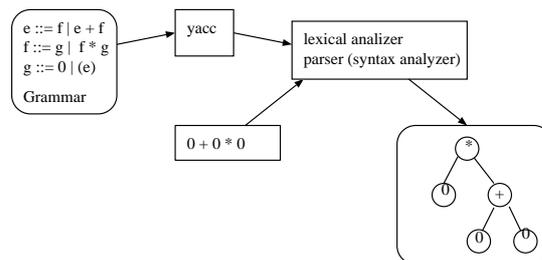
例 4 (かっこ式もいれる)

$e ::= f \mid e + f$
 $f ::= g \mid f * g$
 $g ::= 0 \mid (e)$

- 1 式の構文に「=」を加えるとどうなるか？ただし、「=」は、+ より * より弱く結合する。
- 2 式の構文に「!」(n! で、「n の階乗」)をあらわす 1 引数の記号)を加えるとどうなるか？ただし、「!」は、+, *, = より強く結合する。

例: 「 $x + x! = x$ 」は、「 $(x + (x!)) = x$ 」と parse する。

- 広い範囲の文法に対する効率的な構文解析の方法は、伝統的なコンパイラ構成論の主題。
- LR(k) や LL(k) など。
- 現在では、「文法を記述すると、自動的に構文解析器を出力してくれる」プログラムが使える。例: yacc/lex



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);
}
```

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うなるかを定めること。

意味を厳密に考える必要性

「言葉」(自然言語)による説明(厳密でない意味論)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。

miniC の意味

操作的意味論 (operational semantics) を与えよう。

miniC の意味論の登場人物

- 値(あたい): 式の計算結果; 自然数か真偽値。
- 状態 (state): 各変数がどのような値を持っているか。
 - 変数の集合(無限集合)から、値の集合への部分関数。
 - 例 1: 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」)
 - 例 2: $x := 0$ を実行した直後の状態; 変数 x が値 0 を持ち、他の全ての変数は値を持っていない。
 - $0 + true$ のような式を実行したときをあらわすため、**wrong** という特別な状態があるものとする。

プログラムの実行状態を表すためには、「現在の状態」と「残りの(未実行の)プログラム」があればよい。
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、parse tree)である。

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。

- $\sigma = wrong$ のときは、 $\sigma[s] = wrong$ とする。
- s が代入文 ($x := e$) のとき: $\sigma[s] = \sigma[x \rightarrow \sigma(e)]$ 。
- s が ($begin\ s_1\ end$) のとき: $\sigma[s] = \sigma[s_1]$ である。
- s が ($begin\ s_1;\ p\ end$) のとき: $\sigma[s_1] = \tau$ とすると、 $\sigma[s] = \tau[begin\ p\ end]$ である。
- s が ($if\ e\ then\ s_1\ else\ s_2$) のとき:
 - $\sigma(e) = true$ のとき: $\sigma[s] = \sigma[s_1]$
 - $\sigma(e) = false$ のとき: $\sigma[s] = \sigma[s_2]$
 - 上記以外のとき: $\sigma[s] = wrong$ 。

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。

- $\sigma = wrong$ 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、 $\sigma(e_i)$ の一方が自然数でないとき、 $\sigma(e)$ は値を持たない。
- e が $(e_1 = e_2)$ のとき: $\sigma(e_1)$ と $\sigma(e_2)$ がともに自然数で等しいとき、 $\sigma(e) = true$ とし、ともに自然数で等しくないとき、 $\sigma(e) = false$ とし、それ以外の場合、 $\sigma(e)$ は値をもたない。

例: σ を $x = 1, y = 2$ という状態とすると、 $\sigma((x + y) + 1) = 4$ であり、 $\sigma((x + y) + true)$ は値を持たない。

文の意味

- s が ($while\ e\ do\ s_1$) のとき:
 - $\sigma(e) = true$ のとき、 $\sigma[s] = \sigma[begin\ s_1;\ s\ end]$ とする。
 - $\sigma(e) = false$ のとき、 $\sigma[s] = \sigma$ とする。
 - 上記以外のとき、 $\sigma[s] = wrong$ とする。

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。
(提出を求めない) 課題。以下のプログラム p に対して、 $\sigma_0[p]$ は
なにか？

```
x := 1; y := 0;
n := 5; f := 0;
while (f = 0) do
  begin
    y := x + y;
    x := x + 1;
    if (x = n) then f := 1
    else f := 0
  end
```

(提出を求めない) 課題。プログラムが無限ループするとき、それ
に対する操作的意味はどうなるか？

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。
- 全てのケースにおいてどのように計算するかを書いている。
- ⇒意味論の定義があれば、インタープリタが構成できるの
ではないか？

教訓: 言語を語るときは、常に、構文と意味の両方を考える必要
がある。

来週は、2限は講義、3限は演習を行います。