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, 2010 年 4 月 12 日

これからの話題

意味論とインタープリタ。

- 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

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.

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

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.

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

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

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-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。

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。

典型的なインタープリタの構成

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 インタープリタ:
   1. read
   2. eval
   3. print

命令型言語 miniC の構文

前回の講義、または、配布資料を参照のこと。
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。

miniC の字句解析

トークン:

- 変数、定数、予約語
- 記号列 ;、(、)、+、=

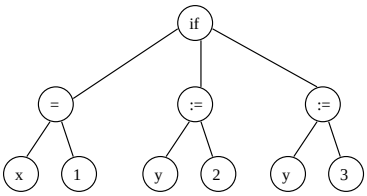
空白文字: スペース、タブ、改行など。

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

miniC の構文解析

トークン列を構文解析木 (parse tree) に変換。



miniC の構文解析

- 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。

曖昧さを持つ文法

- 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
  - 結合の優先度を決める。例: 「* は+ より強い」「* は左結合的」
  - 文法で決める。

曖昧さのない式の構文

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)

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

曖昧さのない式の構文

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)

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

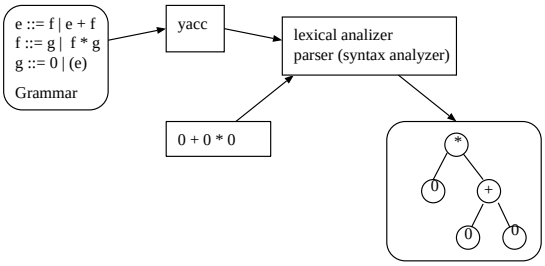
Quiz

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。

構文解析

- 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



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。

次回以降の予告: 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のような結果になるかを (厳密に) 決めること。