COMS W4115

Programming Languages and Translators

Lecture 6: Context-Free Grammars

February 11, 2013

Lecture Outline

- Context-free grammars
- Derivations and parse trees
- Ambiguity
- Examples of context-free grammars
- Yacc: a language for specifying syntax-directed translators

1. Context-Free Grammars (CFG's)

- CFG's are very useful for representing the syntactic structure of programming languages.
- A CFG is sometimes called Backus-Naur Form (BNF).
- A context-free grammar consists of

1. A finite set of terminal symbols,
2. A finite nonempty set of nonterminal symbols,
3. One distinguished nonterminal called the start symbol, and
4. A finite set of rewrite rules, called productions, each of the form A â†' Î± where A is a nonterminal and Î± is a string (possibly empty) of terminals and nonterminals.

- Consider the context-free grammar G with the productions

```
E â†’ E + T | T
T â†’ T * F | F
F â†’ ( E ) | id
```

- The terminal symbols are the alphabet from which strings are formed. In this grammar the set of terminal symbols is { id, +, *, (,) }. The terminal symbols are the token names.
- The nonterminal symbols are syntactic variables that denote sets of strings of terminal symbols. In this grammar the set of nonterminal symbols is { E, T, F}.
- The start symbol is E.

2. Derivations and Parse Trees

- *L*(G), the language generated by a grammar G, consists of all strings of terminal symbols that can be derived from the start symbol of G.
- A leftmost derivation expands the leftmost nonterminal in each sentential form:

```
E â‡’ E + T
  â‡’ T + T
  â‡’ F + T
  â‡’ id + T
  â‡’ id + T * F
  â‡’ id + F * F
  â‡’ id + id * F
  â‡’ id + id * id
```

- A rightmost derivation expands the rightmost nonterminal in each sentential form:

```
E â‡’ E + T
  â‡’ E + T * F
  â‡’ E + T * id
  â‡’ E + F * id
  â‡’ E + id * id
  â‡’ T + id * id
  â‡’ F + id * id
  â‡’ id + id * id
```

- Note that these two derivations have the same parse tree.

3. Ambiguity

- Consider the context-free grammar G with the productions

```
E â†’ E + E | E * E | ( E ) | id
```

This grammar has the following leftmost derivation for `id + id * id`

```
E â‡’ E + E
  â‡’ id + E
  â‡’ id + E * E
  â‡’ id + id * E
  â‡’ id + id * id
```

This grammar also has the following leftmost derivation for `id + id * id`

```
E â‡’ E * E
  â‡’ E + E * E
  â‡’ id +  E * E
  â‡’ id + id * E
  â‡’ id + id * id
```

- These derivations have different parse trees.
- A grammar is *ambiguous* if there is a sentence with two or more parse trees.
- The problem is that the grammar above does not specify
- the precedence of the + and * operators, or
- the associativity of the + and * operators
- However, the grammar in section (3) generates the same language and is unambiguous because it makes * of higher precedence than +, and makes both operators left associative.
- A context-free language is *inherently ambiguous* if it cannot be generated by any unambiguous context-free grammar.
- The context-free language { $a^m b^m a^n b^n$ | m > 0 and n > 0} âˆª { $a^m b^n a^n b^m$ | m > 0 and n > 0} is inherently ambiguous.
- Most (all?) natural languages are inherently ambiguous but no programming languages are inherently ambiguous.
- Unfortunately, there is no algorithm to determine whether a CFG is ambiguous; that is, the problem of determining whether a CFG is ambiguous is undecidable.
- We can, however, give some practically useful sufficient conditions to guarantee that a CFG is unambiguous.

4. Examples of Context-Free Grammars

- Nonempty palindromes of a's and b's. (A palindrome is a string that reads the same forwards as backwards; e.g., `abba`.)
  CFG: `S â†’ a S b | b S a | a a | b b | a | b`
  Note that the language generated by this grammar is not regular. Can you prove this using the pumping lemma for regular languages?

- Strings with an equal number of a's and b's:
  CFG: `S â†’ a S a | b S b | S S | Îµ`
  Note that this grammar is ambiguous. Can you find an equivalent unambiguous grammar?

- If- and if-else statements:

```
stmt â†’ if ( expr ) stmt else stmt
     | if (expr) stmt
     | other
```

Note that this grammar is ambiguous.

- Some typical programming language constructs:

```
stmt â†’ expr ;
```

```
     | if (expr) stmt
     | for ( optexpr; optexpr; optexpr;) stmt
     | other
optexpr â†’ Îµ
     | expr
```

5. Yacc: a Language for Specifying Syntax-Directed Translators

- Yacc is popular language, created by Steve Johnson of Bell Labs, for implementing syntax-directed translators.
- Bison is a gnu version of Yacc, upwards compatible with the original Yacc, written by Charles Donnelly and Richard Stallman. Many other versions of Yacc are also available.
- The original Yacc used C for semantic actions. Yacc has been rewritten for many other languages including Java, ML, OCaml, and Python.
- Yacc specifications
  - A Yacc program has three parts:

```
 declarations
%%
translation rules
%%
supporting C-routines
```

The declarations part may be empty and the last part (%% followed by the supporting C-routines) may be omitted.

- Here is a Yacc program for a desk calculator that adds and multiplies numbers. (See ALSU, p. 292, Fig. 4.59 for a more advanced desk calculator.)

```
%{
#include <ctype.h>

#include <stdio.h>
#define YYSTYPE double
%}

%token NUMBER
%left '+'
%left '*'

%%

lines : lines expr '\n'   { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;

expr  : expr '+' expr      { $$ = $1 + $3; }
      | expr '*' expr      { $$ = $1 * $3; }
      | '(' expr ')'       { $$ = $2; }
      | NUMBER
      ;

%%
/* the lexical analyzer; returns <token-name, yylval> */
int yylex() {
  int c;
  while ((c = getchar()) == ' ');
  if ((c == '.') || (isdigit(c))) {
    ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}
```

- The declarations

```
%left '+'
%left '*'
```

make the operator + left associative and of lower precedence than the left-associative operator `*`.

- On Linux, we can make a desk calculator from this Yacc program as follows:

1. Put the yacc program in a file, say `desk.y`.
2. Invoke `yacc desk.y` to create the yacc output file `y.tab.c`.
3. Compile this output file with a C compiler by typing `gcc y.tab.c -ly` to get `a.out`. (The library -ly contains the Yacc parsing program.)
4. `a.out` is the desk calculator. Try it!

6. Practice Problems

1. Let G be the grammar S â†' a S b S | b S a S | Îµ.
   a. What language is generated by this grammar?
   b. Draw all parse trees for the sentence `abab`.
   c. Is this grammar ambiguous?
2. Let G be the grammar S â†' a S b | Îµ. Prove that *L*(G) = { a^nb^n | *n* â‰¥ 0 }.
3. Consider a sentence of the form `id + id + ... + id` where there are *n* plus signs. Let G be the grammar in section (3) above. How many parse trees are there in G for this sentence when *n* equals
   a. 1
   b. 2
   c. 3
   d. 4
   e. *m*?
4. Write down a CFG for regular expressions over the alphabet {a, b}. Show a parse tree for the regular expression `a | b*a`.

7. Reading

- ALSU Sects. 4.1-4.2, 4.9
- [A nice Lex & Yacc tutorial]

---

aho@cs.columbia.edu