COMS W4115
Programming Languages and Translators
Lecture 7: Parsing Context-Free Grammars
February 13, 2013

Outline

1. Yacc: a language for specifying syntax-directed translators
2. The pumping lemma for context-free languages
3. The parsing problem for context-free grammars
4. Top-down parsing
5. Transformations on grammars

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

- Yacc is popular language, first implemented by Steve Johnson of Bell Labs, for implementing syntax-directed translators.
- Bison is a gnu version of Yacc, upward 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. (From ALSU, p. 292, Fig. 4.59, 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;
}
```

- 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!

2. The Pumping Lemma for Context-Free Languages

- The pumping lemma for context-free languages can be used to show certain languages are not context free.

  The pumping lemma: If L is a context-free language, then there exists a constant *n* such that if z is any string in L of length *n* or more, then z can be written as uvwxy subject to the following conditions:
  1. The length of vwx is less than or equal to *n*.
  2. The length of vx is one or more. (That is, not both of v and x can be empty.)
  3. For all *i* â‰¥ 0, uv^iwx^iy is in L.
- A typical proof using the pumping lemma to show a language L is not context free proceeds by assuming L is context free, and then finding a long string in L which, when pumped, yields a string not in L, thereby deriving a contradiction.
- Examples of non-context-free languages:
  - {$a^nb^nc^n$ | n â‰¥ 0 }
  - {`ww` | `w` is in (a|b)* }
  - {$a^mb^na^mb^n$ | n â‰¥ 0 }

3. The Parsing Problem for Context-Free Grammars

- The parsing problem for context-free grammars is given a CFG G and an input string w to construct all parse trees for w according to G, if w is in L(G).
- The Cocke-Younger-Kasami algorithm is a dynamic programming algorithm that given a Chomsky Normal Form grammar G and an input string w will create in $O(|w|^3)$ time a table from which all parse trees for w according to G can be constructed.
- For compiler applications two styles of parsing algorithms are common: top-down parsing and bottom-up parsing.

4. Top-Down Parsing

- Top-down parsing consists of trying to construct a parse tree for an input string starting from the root and creating the nodes of the parse tree in preorder.
- Equivalently, top-down parsing consists of trying to find a leftmost derivation for the input string.
- Consider grammar *G*:

```
S â†’ + S S | * S S | a
```

- Leftmost derivation for + a * a a:

```
S â‡’ + S S
  â‡’ + a S
  â‡’ + a * S S
  â‡’ + a * a S
  â‡’ + a * a a
```

- Recursive-descent parsing
- Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input string.
- One procedure is associated with each nonterminal of the grammar. See Fig. 4.13, p. 219.
- The sequence of successful procedure calls defines the parse tree.
- Nonrecursive predictive parsing

- A nonrecursive predictive parser uses an explicit stack.
- See Fig. 4.19, p. 227, for a model of table-driven predictive parser.
- Parsing table for *G*:

```
                     Input Symbol
Nonterminal    a        +          *        $

    S        S â†’ a    S â†’ +SS    S â†’ *SS
```

- Moves made by this predictive parser on input `+a*aa`. (The top of the stack is to the left.)

```
    Stack        Input   Output
     S$         +a*aa$
     +SS$       +a*aa$   S â†’ +SS
     SS$         a*aa$
     aS$         a*aa$   S â†’ a
     S$           *aa$
     *SS$         *aa$   S â†’ *SS
     SS$           aa$
     aS$           aa$   S â†’ a
     S$             a$
     a$             a$   S â†’ a
     $               $
```

- Note that these moves trace out a leftmost derivation for the input.

5. Transformations on Grammars

- Two common language-preserving transformations are often applied to grammars to try to make them parsable by top-down methods. These are eliminating left recursion and left factoring.
- Eliminating left recursion:
  - Replace

```
expr â†’  expr + term
     |  term
```

by

```
expr  â†’  term expr'

expr' â†’  + term expr'
      |  Îµ
```

- Left factoring:
- Replace

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

by

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

stmt' â†’ else stmt
      | Îµ
```

6. Practice Problems

1. Write down a CFG for regular expressions over the alphabet {a, b}. Show a parse tree for the regular expression `a | b*a`.
2. Using the nonterminals `stmt` and `expr`, design context-free grammar productions to model
   a. C while-statements
   b. C for-statements
   c. C do-while statements
3. Consider grammar *G*:

```
S â†’  S S + | S S * | a
```

   a. What language does this grammar generate?
   b. Eliminate the left recursion from this grammar.
4. Use the pumping lemma to show that {$a^n b^n c^n$ | n â‰¥ 0 } is not context free.

7. Reading

- ALSU, Sections 4.3, 4.4, 4.9.
- See The Lex & Yacc Page for lex and yacc tutorials and manuals.
- Another nice Lex & Yacc tutorial

---

aho@cs.columbia.edu