CS 164 Programming Language and Compilers **Fall 2015**

Section 3: CFG, LL Parsing, and Cup

TAs: Melanie Cebula, Ben Mehne, Matt Miller

- **Announcement**
  - WA2 due Mon (9/28) in class.
  - PA3 due Tue (10/06) at 11:59pm.
  - WA3 out Wed (9/23)
- **Outline**
  1. Understanding **context free grammar**.
  2. Understanding **LL/Top-Down parsing**.
  3. **Cup parser** introduction.

---

1 Context Free Grammar

A Context Free Grammar: $G = (N, T, S, \delta)$, where N is a set of non-terminals, T is a set of terminals, S is a start symbol (a non-terminal), and δ is a set of production rules.

Ex.1 Let's consider following CFG : $S \rightarrow SS \mid (S) \mid \epsilon$

(a) How do we tell if some sequence is in this language?

Answer: Begin with non-terminal S. If there exists a path of derivations (repeatedly replacing some non-terminal A by one of its productions until there are only non-terminals left), the sequence is in this language.

(b) What language does this grammar represents?

Answer: Balanced parenthesis.

(c) What aspect of this grammar is problematic for parsing?

Answer: Ambiguity (Multiple parse trees exist for this grammar).

(d) Write an improved grammar that handles this problem:

Answer: $S \rightarrow (S)S \mid \epsilon$

Ex.2 CFG Practice

(a) Write a CFG for 00^*11.
Answer: $S \rightarrow 0S'11 \quad S' \rightarrow 0S' | \epsilon$

(b) Show the derivation of 00011 for your grammar.
Answer: $S \rightarrow 0S'11 \rightarrow 00S'11 \rightarrow 000S'11 \rightarrow 000\epsilon11 \rightarrow 000111$

2 LL/Top-Down Parsing

An LL(k) grammar is a CFG used by a parser that scans input left-to-right ("L"), leftmost derivation ("L"), and uses k tokens of lookahead to predict the correct production. Note: We will look at other parsers, like LR/Bottom-Up parsing next week. LR grammars are an alternative to LL grammars because they are limited (we can't have left recursion because we would need infinite lookahead). We can discover ambiguity through LL conflicts by creating and observing the following sets:

FIRST(A): the set of all terminals that could occur first in an expansion of the terminal or nonterminal A (include ϵ if A can expand to ϵ

FOLLOW(A): the set of all terminals that could follow an occurrence of the terminal or nonterminal A in a (partial) derivation

There are two main types of LL(1) conflicts:

FIRST/FIRST: The FIRST sets of two different productions for some non-terminal intersect.

FIRST/FOLLOW: The FIRST set of a grammar rule contains an epsilon and the intersection with its FOLLOW set is not empty.

Ex.3 Are the following grammars LL(1)? Justify your answer using FIRST and FOLLOW sets. (Thanks to Karen Lemone, at WPI, for these problems.)

(a) $A \rightarrow dA \mid dB \mid f \qquad B \rightarrow g$

Answer: No, FIRST/FIRST conflict. The FIRST and FOLLOW sets for this grammar are:

FIRST(`d A`) = {`'d'`} FIRST(`d B`) = {`'d'`} FIRST(`f`) = {`'f'`} FIRST(`g`) = {`'g'`}

FOLLOW(`A`) = {$} FOLLOW(`B`) = {$}

(b) $S \rightarrow Xd \qquad X \rightarrow C \mid Ba \qquad C \rightarrow \epsilon \qquad B \rightarrow d$

Answer: No, FIRST/FOLLOW conflict. FIRST(X) contains ϵ and the intersection of FIRST(X) and FOLLOW(X) is not empty. The FIRST and FOLLOW sets for this grammar are:

FIRST(`X d`) = {`'d'`} FIRST(`C`) = {ϵ} FIRST(`B a`) = {`'d'`}

FIRST(ϵ) = {ϵ} FIRST(`d`) = {`'d'`}

FOLLOW(`S`) = {$} FOLLOW(`X`) = {`'d'`} FOLLOW(`C`) = {`'d'`} FOLLOW(`B`) = {`'a'`}

Ex.4 For the grammar from **Ex.3**(a): Rewrite the grammar so that it is LL(1) by introducing the non-terminal $AB \rightarrow A \mid B$.

Answer:

$$A \rightarrow dAB|f$$
$$B \rightarrow g$$
$$AB \rightarrow A|B$$

3 Cup Parser Introduction

1. Let's consider the following example cup file.

   Example: `ycalc.cup`

```
import java_cup.runtime.*;

terminal PLUS, TIMES;
terminal Integer NUMBER;

non terminal Object expr_part;
non terminal expr, factor, term;

precedence left PLUS;
precedence left TIMES;

expr_part ::= expr:e
                {: System.out.println(" = " + e); :}
                ;

   expr      ::= expr:e PLUS factor:f
                {: RESULT = new Integer(e.intValue() + f.intValue()); :}
                |
                factor:f
                {: RESULT = new Integer(f.intValue()); :}
                ;

   factor    ::= factor:f TIMES term:t
                {: RESULT = new Integer(f.intValue() * t.intValue()); :}
                |
                term:t
                {: RESULT = new Integer(t.intValue()); :}
                ;

   term      ::= NUMBER:n
                {: RESULT = n; :}
                ;
```

2. You can link the parser with a lexer.

   Example: `lcalc.lex`

```
import java_cup.runtime.*;
%%

%class Lexer

%line
%column
```

```
%cup

%{
    private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
    }

    private Symbol symbol(int type, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
    }
%}

LineTerminator = \r|\n|\r\n
WhiteSpace     = {LineTerminator} | [ \t\f]
dec_int_lit = 0 | [1-9][0-9]*

%%
<YYINITIAL> {

    "+"                { System.out.print(" + "); return symbol(sym.PLUS); }
    "*"                { System.out.print(" * "); return symbol(sym.TIMES); }

    {dec_int_lit}      { System.out.print(yytext());
                         return symbol(sym.NUMBER, new Integer(yytext())); }

    {WhiteSpace}       { /* do nothing */ }
}

/* No token was found for the input. */
[^]                    { throw new Error("Illegal character <"+yytext()+">"); }
```

This parser works! Give it a try.

```
% java -cp .:path/to/coolc.jar jflex.Main lcalc.flex
% java -cp .:path/to/coolc.jar java_cup.Main ycalc.cup
% javac -cp .:path/to/coolc.jar Main.java
% java -cp .:path/to/coolc.jar Main test.txt
```