University of Puerto Rico
Mayagüez Campus
College of Engineering
Department of Electrical and Computer Engineering

ICOM4029 – Compilers
Professor: Bienvenido Vélez
Technical Assistant: René D. Badía

Laboratory 5 – Syntactic Analysis

The function of the syntactic analyzer, or parser, in the compiler is to take an input sequence of tokens, validate the syntax of the code, and output an abstract syntax tree (AST), or parse tree. The AST is constructed using the syntactic actions of the parser generator. The tool we will use for this purpose is the Java-Based Constructor of Useful Parsers (CUP for short). CUP is a system for generating LALR (left-associative, left-to-right) parsers from simple specifications. It serves the same purpose as the widely used YACC. Its output is a Java source program that contains the necessary functions to perform syntactic analysis (the parser).

I. Preparation

1. CUP Manual

Download or browse to the CUP manual at: http://amadeus.ece.uprm.edu/~rbadia/cupmanual/
(downloaded from: http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html)

2. Lab files

The files needed for this lab can be obtained by doing:

```
mkdir lab5
cd lab5
cp ~icom4029/labfiles/lab5/* .
```

II. CUP General Syntax

CUP File syntax:

```
-Package and Import Specifications
-User Code Components
-Symbol Lists
-Precedence and Associativity declarations
-The Grammar
```

Comments are expressed within '/*' and '*/'. Code strings are expressed within '{:' and ':}'.

III. Example 1 – Simple Expression Evaluator

1. Description

Example 1 is a simple expression evaluator that takes a list of arithmetic expressions ending in ";" and outputs the result of evaluating them.
e.g. For an input:

```
1 + 1;
1 + 2;
```

The output would be:

```
= 2
= 3
```

1. Context-Free Grammar

```
L -> S L | S
S -> E;
E -> E + E | E - E | E * E | E / E | ( E ) | int
```

2. Unambiguous CFG

```
L -> S L | S
S -> E;
E -> A B
A -> ( E ) | int X
B -> + E | - E | ε
X -> * A | / A | ε
```

3. JLex lexer definition (example.lex)

```
/* Scanner definition for the simple expression evaluator */

import java_cup.runtime.Symbol;

%%
%{
    private int curr_lineno = 1;
    int get_curr_lineno() {
      return curr_lineno;
    }
%}
%eofval{
      return new Symbol(TokenConstants.EOF);
%eofval}
%class ExampleLexer
%cup

/* regular expressions */
WHITESPACE = [" "\ \t\r\b\f\v]
NEW_LINE = [\n]
DIGIT = [0-9]
INT_CONST = {DIGIT}+
SEMI = ";"
PLUS = ("+")
MINUS = "-"
MULT = "*"
DIV = "/"
LPAREN = "("
RPAREN = ")"

/* actions */
%%
{NEW_LINE}        {curr_lineno++;}
{WHITESPACE}      {}
{SEMI}            { return new Symbol(TokenConstants.SEMI); }
{PLUS}            { return new Symbol(TokenConstants.PLUS); }
{MINUS}           { return new Symbol(TokenConstants.MINUS); }
{MULT}            { return new Symbol(TokenConstants.MULT); }
{DIV}             { return new Symbol(TokenConstants.DIV); }
{LPAREN}          { return new Symbol(TokenConstants.LPAREN); }
{RPAREN}          { return new Symbol(TokenConstants.RPAREN); }

{INT_CONST}       {
      String int_str = yytext();
      Symbol retSym = new Symbol(TokenConstants.INT_CONST,
            AbstractTable.inttable.addString(int_str));
      return retSym; }
```

```
.                 {
      String error_msg = yytext();
      Symbol ret = new Symbol(TokenConstants.ERROR,
            AbstractTable.stringtable.addString(error_msg));
      return ret; }
```

4. CUP code (example1.cup)

```
// CUP specification for a simple expression evaluator (w/ actions)

import java_cup.runtime.*;

action code {:
    int curr_lineno() {
      return (((ExampleLexer)parser.getScanner()).get_curr_lineno());
    }
:}

parser code {:
    public void syntax_error(Symbol cur_token) {
        int lineno = action_obj.curr_lineno();
        System.out.print("line " + lineno +
                    ": parse error at or near ");
        Utilities.printToken(cur_token);
    }

    public void unrecovered_syntax_error(Symbol cur_token) {
    }
:}

/* Terminals (tokens returned by the scanner). */
terminal                SEMI, PLUS, MINUS, MULT, DIV;
terminal                LPAREN, RPAREN;
terminal AbstractSymbol INT_CONST;
terminal                ERROR;

/* Non-terminals */
non terminal            expr_list, expr_part;
non terminal Integer    expr;

/* Precedences */
precedence left PLUS, MINUS;
precedence left MULT, DIV;

/* The grammar */
expr_list ::= expr_part expr_list
            | expr_part
            ;

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

```
expr      ::= expr:e1 PLUS expr:e2
              {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
              | expr:e1 MINUS expr:e2
              {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
              | expr:e1 MULT expr:e2
              {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
              | expr:e1 DIV expr:e2
              {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
              | INT_CONST:i
              {: RESULT = new Integer(i.getString()); :}
              | LPAREN expr:e RPAREN
              {: RESULT = e; :}
              ;
```

5. Compiling and Running

To create the parser using CUP (all in one line):

```
java -classpath /home/courses/icom4029/icom4029/cool/lib:. java_cup.Main -
parser parser1 -symbols TokenConstants -expect 100 -nopositions <
example1.cup
```

To create the lexer using JLex:

```
jlex example.lex
```

To compile the java files (all in one line):

```
javac -classpath /home/courses/icom4029/icom4029/cool/lib:. example.lex.java
Lexer.java MyParser1.java parser1.java TokenConstants.java Utilities.java
```

To run the lexer:

```
./lexer input.txt
```

To run the parser:

```
./parser1 input.txt
```

The last output should be:

```
= 3
= -5
= 2
```

Which are the results of evaluating:

```
1 + 2;
1 - 2 * 3;
6/(1+2);
```

If we introduce a syntax error at line 2:

```
1 -- 2 * 3;
```

Run the parser with input2:

```
./parser1 input2.txt
```

The output should state that there is a syntax, or parse, error:

```
= 3
line 2: parse error at or near '-'
```

IV. Example 2 – Parse Tree Generation

1. Description

This example makes use of the same lexer as before and parses the same language as Example 1. The only difference is that now, the output generated will be an AST or parse tree. We will use the AST implementation classes provided by the *cool* distribution. The "`RESULT`" of each action is now an object of a class whose parent is `TreeNode`.

2. CUP code (example2.cup)

The "non terminals" and "grammar" sections are the only ones that change:

```
...

/* Non-terminals */
non terminal Expressions      expr_list;
non terminal Expression       expr;
...

/* The grammar */
expr_list ::= expr:e SEMI
              {: RESULT = (new Expressions(curr_lineno())).appendElement(e); :}
            | expr_list:el expr:e SEMI
              {: RESULT = el.appendElement(e); :}
            ;

expr      ::= expr:e1 PLUS expr:e2
              {: RESULT = new plus(curr_lineno(), e1, e2); :}
            | expr:e1 MINUS expr:e2
              {: RESULT = new sub(curr_lineno(), e1, e2); :}
            | expr:e1 MULT expr:e2
              {: RESULT = new mul(curr_lineno(), e1, e2); :}
            | expr:e1 DIV expr:e2
              {: RESULT = new divide(curr_lineno(), e1, e2); :}
            | LPAREN expr:e RPAREN
              {: RESULT = e; :}
            | INT_CONST:i
              {: RESULT = new int_const(curr_lineno(), i); :}
            ;
```

3. Compiling and Running

Create this new parser:

```
java -classpath /home/courses/icom4029/icom4029/cool/lib:. java_cup.Main -
parser parser2 -symbols TokenConstants -expect 100 -nopositions <
example2.cup
```

Compile the new java files:

```
javac -classpath /home/courses/icom4029/icom4029/cool/lib:. MyParser2.java
parser2.java
```

Run the parser:

```
./parser2 input.txt
```

The output should be:

```
list
  plus
    int_const
      1
    int_const
      2
  sub
    int_const
      1
    mul
      int_const
        2
      int_const
        3
  divide
    int_const
      6
    plus
      int_const
        1
      int_const
        2
(end_of_list)
```

This is just an example of an AST. The one created by the parser of the second phase (PA3) of the *cool* compiler will be similar, only that it will contain type information for each node, which will be corrected on the semantic analysis phase (PA4).

V. PA2 Questions

?