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://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html

### 2. Lab files

Download and extract source code files needed for this lab from the course website. Create a new Eclipse project named lab5 and copy all the Java files to the src directory within the project. In addition copy the **`example.lex`** file containing a JFlex specification for the lexer used in this lab. Your JFlex plugin should automatically generate an **`ExampleLexer.java`** file containing the lexer. You may still have some compiler errors which will get corrected as you work through the lab instructions.

## 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)**

Copy the **`example1.cup`** file from the **`lab5`** directory into the root of your lab5 Eclipse project. Again, the JFlex/Cup plugin will automatically generate various java files. The **`MyParser1.java`** file contains the LALR parser. **`Example1Sym.java`** contains a set of constant declaration for the token ids that the parser expects from the lexer. Here is the CUP specification. Open the file and inspect the definitions.

```
// 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**

Before your parser can run you must make sure that the lexer and the parser agree on the token identifiers that will be returned by the lexer. To accomplish this you need to modify all instances of the **`TokenConstants`** class within the **`example1.lex`** specification to reference **`Example1Sym`** instead. This should be all that is required to run your parser. Copy the **`input.txt`** file from your lab5 directory into the root of your lab5 Eclipse project. Now create a new run configuration for the MyParser1.java application with **`input.txt`** as an argument. You may now run your Example1Parser as a java application.

Which are the results of evaluating:

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

What if we introduce a syntax error at line 2:

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

**5. Extending your expression evaluator with additional operators.**

Modfy your lexer specification and your expression grammar to accommodate expressions of the form

**`x % y`**

where `%` is the modulo (remainder) operator. This operator should have the same precedence as **`*`** and should associate to the left. Test your parser with a few sample expressions including `%`.

## 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

Copy the **`example2.cup`** file from your lab5 directory to the root of the lab5 Eclipse project. Also copy the **`MyParser2.java`** file into the src directory inside the Eclipse project. The JFlex plugin will automatically generate two files: **`Example2Cup.Java`** and **`Example2Sym.java`**. Create a new run configuration for the parser using input.txt as the input file argument and run the parser.

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).

How could you accommodate the addition of the % operator into the second parser in order to generate the appropriate trees for expressions containing this new operator? Proceed to modify your example2.cup specification accordingly.