Introduction to Language Theory and Compilation Exercises

Session 12: `yacc/bison` parser generator

Remainder

In the previous session, we saw how to generate a scanner with JFlex and how to interact with a parser generator (CUP). In this remainder, we will explain briefly the CUP specification but the manual is available at `http://www2.cs.tum.edu/projects/cup/manual.html`.

The first lines of a CUP file are designed for the additional Java code source. Thus, you can add imports and some code (variables, functions) into the parser. In order to add some piece of code, you need to encapsulate your code in this way

```
import static java.Math.*;
parser code {:
   your java code
:}
```

The next part of a CUP file is dedicated to the declaration of symbols that are terminals and non-terminals. These terminals are the lexical units returned by the JFlex specification. For each of these symbols, you can define a type, for instance **Integer** for your integer numbers.

```
terminal Integer JFLEX_LEXICAL_UNIT_NAME_FOR_INT;
terminal PLUS, MINUS, DIVIDE, TIMES;
non terminal java.util.Map<?,?> StartSymbol;
```

CUP can also handle associativity and priority if the operators rules of your grammar is ambiguous. This is done with the **precedence** option left (resp. right) for left (resp. right) associativity. The first declaration has the lowest priority. For instance:

```
precedence left PLUS, MINUS;
precedence left  DIVIDE, TIMES;
```

The last part contains your grammar in the form

$$symbol := rule1|rule2;$$

For instance:

```
StartSymbol := Expression | ;
Expression := JFLEX_LEXICAL_UNIT_NAME_FOR_INT PLUS JFLEX_LEXICAL_UNIT_NAME_FOR_INT
| JFLEX_LEXICAL_UNIT_NAME_FOR_INT MINUS JFLEX_LEXICAL_UNIT_NAME_FOR_INT
| JFLEX_LEXICAL_UNIT_NAME_FOR_INT DIVIDE JFLEX_LEXICAL_UNIT_NAME_FOR_INT
| JFLEX_LEXICAL_UNIT_NAME_FOR_INT TIMES JFLEX_LEXICAL_UNIT_NAME_FOR_INT
;
```

The CUP generated parser can add action inside the grammar. You can name symbols by using the **:** and add actions into **{: Java code :}**. For instance:

```
Expression := JFLEX_LEXICAL_UNIT_NAME_FOR_INT:a PLUS JFLEX_LEXICAL_UNIT_NAME_FOR_INT:b
            {: RESULT=new Integer(a.intValue()+b.intValue()) :}
;
```

Exercise : polynomial manipulation

You have received a `lex` and a `yacc` specification (they can also be downloaded off the Web site).

1. Informally describe the accepted language of the compiler we'd generate from the specifications.
2. Adjust the specification so it only accepts polynomials of a single variable. We input a polynomial per line, but there can only be one variable used on each line.
3. Add the necessary code to show the first derivative of a polynomial. For example, if `2x^3+2x^2+5` was given on input, we would output :
   `First derivative:  6x^2+4x`
4. Add a way to recognize polynomial products and adjust the derivative calculation. For example, if `(3x^2+6x)*(9x+4)` is given on input, we would output:
   `First derivative:  (3x^2+6x)*(9)+(6x+6)*(9x+4)`
5. Add a way to evaluate a polynomial and its first derivative for a given value. The user should be able to input the variable value, followed by a semicolon, followed by the polynomial (all this on the same line). For example :

```
2 ; (3x^2+6x)*(9x+4)
First derivative : (3x^2+6x)*(9)+(6x+6)*(9x+4)
p(2) = 528, p'(2) = 612
```