Compilers HomeWork 1

Part1

First of, we need to remodel our grammar in order to meet the following requirements.

1. **Support priorities**

   To achieve this we need to ensure that, during the production of an expression, choosing to produce a sub-expression containing the & operator, whose precedence happens to be the highest, implies that we can no longer produce a sub-expression containing the ^ operator, except if it is surrounded by parentheses (so it's precedence is now higher). Therefore, we can guarantee that while traversing the parse tree from top to bottom, the operator precedence will not be decreased. The resulting grammar looks like this:

   - exp ➔ exp ^ exp2 | exp2
   - exp2 ➔ exp2 & last | last
   - last ➔ (exp) | 0 | 1 | … | 9

2. Eliminate Right-Recursion

   To do that, for each non-terminal having a left-recursive production rule, we will replace that with a single one by finding the symbol's 'base case rule', a rule containing just terminals, and then place it as the head of the new rule. There will also be a tail for this rule, that will be two non-terminal. Now we are able to produce, using right-recursion, all of the symbols the initial one could produce plus the empty string. So our grammar now looks like this:

   I. exp ➔ exp2 tail
   II. tail ➔ ^ exp2 tail | ε
   III. exp2 ➔ last tail2
   IV. tail2 ➔ & last tail2 | ε
   V. last ➔ (exp) | 0 | 1 | … | 9

Notice that this grammar happens to be LL(1) trivially, because there is only one production rule for each non-terminal, so we can uniquely identify a production just by looking at the first symbol of the string derived from it (lookahead = 1). In another occasion we would have to transform the grammar to LL(1), using left factoring.

Now, that the all requirements are satisfied we can build a lookup table that will help us convert the above grammar into code, by defining a function for each non-terminal and having those calling each other based on the lookahead token.

Lookahead Table

To build this table we need a row for each non-terminal and a column for each terminal. Cell [i, j] will either contain the word 'error' or the number of a production rule if the j-th terminal belongs in the FIRST+ set of the i-th non-terminal. So, we first need to compute the FIRST+ and FOLLOW sets for each non-terminal.

After taking a good look at our grammar, we come up with the following:

- FIRST+ (exp) = FIRST+(exp2) = FIRST+(last) = { (, 0 , 1 , ... , 9}
- FIRST+(tail) = { ^ ,) , EOF }
- FIRST+(tail2) = { & , ^ ,) , EOF}
- FOLLOW(exp) = FOLLOW(tail) = {) , EOF }
- FOLLOW(tail2) = FOLLOW(exp2) = FIRST+(tail) = { ^,) , EOF}
- FOLLOW(last) = FIRST+(tail2) = {&, ^ ,) , EOF}

So the lookup table will probably look like this:

	[0-9]	**^**	**&**	**(**	**)**	**EOF**
exp	I	error	error	I	error	error
exp2	error	II	error	error	ε	ε
tail	III	error	error	III	error	error
tail2	error	ε	IV	error	ε	ε
last	V.ii	error	error	V.i	error	error

To run the calculator type:

- javac Calculator.java
- java Calculator

Then type any arithmetic expression you wish containing just ^ and & as operators or 'q' to exit the Calculator.

Part2

The goal here was given an input file in a simple language L, translate L to java, producing code that can hopefully get compiled and run. To do this, we used **Jflex** and **JavaCUP**.

scanner.flex

First off, the scanner/lexer written in scanner.flex, produced a stream of token from the input string. The form of each token was specified by a regular expression making sure that keywords are defined first, in order to get higher priority than identifiers and not get overlapped by them. For each regular expression recognized, scanner.flex passed up helpful information to the parser, like the code of the token, the position(*yyline*, *yycolumn*) or the content(*yytext()*).

parser.cup

After that, the parser written in parser.cup given a context-free grammar representing language L, produced the actual code by doing the following:

- checking the sequence of tokens the scanner returned, against the grammar
- for each expression recognized, generating code by returning a value in RESULT

Each non-terminal used the RESULT passed from the non-terminals it produced to construct its own RESULT.

make and run

To run the code produced, open the folder named 'Part2': it contains a makefile, scanner.flex, parser.cup, the Main.java file that creates and runs a Parser "on" a Scanner (these classes were produced by *JavaCUP* and *Jflex* respectively). and two directories, In and Out. The first one contains the input files given in the project's description(in1, in2 and in3) and another two, a bit more complicated(in4 and in5) using some interesting combinations of all valid expressions[1]. Folder Out will eventually contain the Main.java produced after parsing the input. It also contains a script, **check.sh**, that does all the work. To run:

1. First type 'make' to produce Parser and Scanner .java and .class files
2. Then type './check.sh <input>' in order to:
   - scan and parse <input> translating it to java
   - get a side-by-side look at the <input> and output file
   - compile Out/Main.java to Bytecode using javac
   - rut Out/Main using JVM

all in one in order to automate and speed up the whole process.

Hopefully, this made things a bit more clear. Thank you.

[1] These are the following and they all return a String: concatenation, function call and if/else statement. Last one, is translated into an equivalent java expression that uses the ternary operator so that we can have if/else statements as conditions in other if/else statements(see In/in5).