Language Processors Lab 7 (Week8)
The SIMPLE Programming Language: Type-checking and Interpretation

In this lab you will familiarise yourself with the SIMPLE Programming Language, a toy imperative language. You will be asked to extend the language with a new arithmetical expression and a new statement.

Downloading the files

Start a Unix shell window and move to your LanguageProcessors directory. Download the file `lab7.tar.gz` from Moodle or from `http://www.soi.city.ac.uk/~sbbc287/lab7.tar.gz`. Unzip and untar the file with the following commands:

```
gunzip lab7.tar.gz       // this will genenerate lab6.tar
tar -xvf lab7.tar
```

The last command will generate the `lab7` directory. It is possible that, while downloading, the file system automatically unzips, and even expands (untars), the file. If that's the case, you may have to skip one or both of the above commands, though make sure to copy the new directory.

Your new `lab7` directory should contain one directory: `tpl-interpret` . Double check that this is the case.

Load java and javacc by executing the command:

```
module add java javacc
```

The SIMPLE Programming Language

Running the typechecker and Interpreter

Move into the `tpl-interpret` subdirectory, inside the `lab7` directory. **For example**, if you are in the `LanguageProcessors` directory, type:

```
cd lab7/tpl-interpret
```

The `tpl-interpret` directory contains files generated by running the jjtree and javacc command on `TPL.jjt` and `TPL.jj`, respectively. I have deleted some of these generated files in order to avoid version conflicts.

Execute JJTree, JavaCC, and compile the generated java files:

```
jjtree TPL.jjt          javacc TPL.jj          javac *.java
```

Before runing the program, load in your text editor of choice the contents of `TPL.java`. You should be able to figure out that the main method:

- Takes a file as input,
- Parses the contents of the file,
- And, if succesful, executes the following:

```
((SimpleNode)parser.jjtree.rootNode()).dump("")
parser.jjtree.rootNode().identification();
parser.jjtree.rootNode().typecheck();
parser.jjtree.rootNode().interpret();
```

The next question should be: What type of files am I going to pass as argument to the program? You should pass files that contain valid SIMPLE programs. We have included a few of these. Look for all files with extension .tpl by typing:

```
ls *.tpl
```

You should get a few e.g. `fact.tpl`, `rel.tpl`, `types.tpl`, etc.

Now we are ready to test the program. Type:

```
java TPL fact.tpl
```

You will notice that the program:

- Prints the AST of the program taken as input.
- Executes typecheck and identification. However, since there were no type errors, nothing was displayed.
- Executes the factorial program: it asks a number as input and then prints the value of its factorial.

Now try with sqrt.tpl. What does it do?
You can also write your own SIMPLE program!

Adding a new operator and expression

Recall that a few weeks ago you were asked to add the Power operator to your calculator. We can do the same for SIMPLE.

Task: Extend SIMPLE with the power `^` operator. Assume that the power operator has precedence over plus and minus, that is, `2 - 3 ^ 4` should be parsed as `2 - (3 ^ 4)`.

Solution: Do the following:

- You must first modify the SIMPLE syntax so it accepts the new operator. The power operator should also be part of a different type of expression and, consequently, a new type of AST node should be created (by jjtree). Open the `TPL.jjt` file and change the specification for `MultiplicativeExpression` so it looks like:

```
void MultiplicativeExpression() #void :
{}
{
  PrimaryExpression()
  (
    "*" PrimaryExpression() #Mul(2)
   |
    "/" PrimaryExpression() #Div(2)
   |
    "^" PrimaryExpression() #Pow(2)
  )*
}
```

- execute `jjtree TPL.jjt`. JJTree should generate an `ASTPow.java` file. Double check that the file has been generated.
- The new `ASTPow.java` file does not have its own implementations for `identification`, `typecheck` and `interpret`. These implementations (if needed), should be very similar to the other multiplicative expresions e.g. to the ones found in `ASTMul.java`. Have a look and try to figure out the correct version for `ASTPow.java`.

- Load `ASTPow.java` to your text editor. The implementation for `identification` and `typecheck` should be the same as ASTMul, for example, since both require their operands to be integers.

  Add the following method definitions (you can copy them from ASTMul) to the end of the class definition of `ASTPow`:

```
public void identification () {
  jjtGetChild(0).identification();
  jjtGetChild(1).identification();
}

public void typecheck () {
  jjtGetChild(0).typecheck();
  jjtGetChild(1).typecheck();
  if (!(jjtGetChild(0).GetNodeType() == TPLTypes.intType &&
        jjtGetChild(1).GetNodeType() == TPLTypes.intType))
     System.out.println("TPL Typechecker: mult of non-ints.");
  NodeType = TPLTypes.intType;
}
```

- The structure of `interpret` should be very similar to the one used in `ASTMul`. The only diference should be the operation itself: instead of multiplying the numbers you should call `Math.pow`. Type the following after the methods added above:

```
public void interpret () {
  int left, right;
  jjtGetChild(0).interpret();
  jjtGetChild(1).interpret();

  right = ((Integer)stack.pop()).intValue();
  left = ((Integer)stack.pop()).intValue();

  stack.push(new Integer((int)Math.pow(left,right)));
}
```

- Run `javacc TPL.jj` and compile the new file with `javac *.java`.
- In order to test our new construct we need a SIMPLE program that uses the power operation. Create a new file `cube.tpl` containing the following:

```
int n;
int c;

read n;
c  = n ^ 3;
write c;
```

- Run the program with `java TPL cube.tpl`.
  Does it work? It should!

Adding a new statement

The SIMPLE Programming language contains one statement for building loops: the while statement. Supposse we want to extend SIMPLE with a `repeat` stament that has the following syntax:

```
RepeatStatement  -->   "repeat" "(" Expression ")" StatementBlock
```

Thus, we should be able to write programs like this:

```
.....
x = y * 4;
repeat (x+10) { y = y*2; write(y); }
```

A Repeat statement first evaluates the expression in parenthesis (the result must be an integer), say to integer n, and then evaluates the block of statements n times.

Task: Add the repeat statement to SIMPLE. Hint: Check the implementation of the while statement.

Solution: Out on Wednesday.