LINGI2132 - Assignment 2

Now that you are used to j-- and its environment, more complex aspects of compilers can be considered. With this assignment, you will go through the whole compilation process in deeper details and really understand how it works.

Warning

This assignment is challenging (much more than assignment 1). As this assignment represents 3 points of the overall grade of the course, do not neglect it.

Preliminary remarks

- Do not remove initial tests provided in j--.
- Please use UTF-8 encoding.
- Minimal changes must be applied to j--. For instance, you do not have to manage the *break* statement (see below).
- Do not remove the junit.jar from the lib directory.
- The reference book will be very helpful, use it !
- Remove the tests you have from assignment 1. They will make fail the tests using JavaCC (see below).

Paper exercises

Lexical Analysis

Here is a regExp:

(a|b)*a(a|b)

1. Give 5 different strings belonging to the language described by this regExp.
2. Construct the NFA from this regExp (Thompson Construction).
3. Transform the NFA into a DFA (justify important steps, ε-closures).
4. Minimize the DFA (Hopcroft Algorithm).

Parsing

Given the following grammar (small letters are terminal symbols):

1. S ::= u B D z
2. B ::= B v
3. B ::= w
4. D ::= E F
5. E ::= y
6. E ::= ε
7. F ::= x
8. F ::= ε

1. Give evidence that this grammar is not LL(1).
2. Modify the grammar as little as possible to make an LL(1) grammar that accepts the same language.
3. Calculate first and follow sets for this grammar (the LL(1) one, not the original one of course). Provide the details of calculation (final result is not enough).
4. Construct the final LL(1) parsing table.

Programming : recursive descent

In order to get used to recursive descent parsing process, we ask you to implement a parser for a small grammar. We provide an eclipse project on iCampus (Assignment2/recursive-descent-assignment.zip). The grammar is provided in comments in the *Grammar.java* file.

To achieve this part of the assignment, you have to :

- Import this project in your workspace.
- Implement the *parse* method of the *Parser* class.
- Write some tests to ensure your program is correct. A test (*Test* class) is provided to you as an example.
- Once your implementation is complete, export your project as a **.zip** file and submit it on iCampus (Assignement2_(code_recursive_descent))

Programming directly in Java bytecode

In order to get used to compilation for the JVM, you must understand what the final output is. For this, we ask you to write a *.java* program to "generate" a *.class* file, or said differently, to generate bytecode (almost) manually. To do this, you will have to use the *CLEmitter* class provided by j--. The source code of the class to be compiled is provided on iCampus (Assignment2/ClassToGenerate.java).

More practically, we ask you to create the following class, where you have to implement the generateClass() method, in the j-- project (please ensure the class is in the correct package).

```
package lingi2132;

public class Generator {
   private String outputDir;

   public Generator(String outputDir) {
      this.outputDir = outputDir;
   }

   public void generateClass() {
   //TO IMPLEMENT
   }

   public static void main(String [] args) {
      Generator gen = new Generator(args[0]);
      gen.generateClass();

   }
}
```

In order to be able to compile your Java bytecode generator, you must add the following ant task in the file *build.xml* of your j-- project. The .class file will here be generated in the */tmp/* directory. You can change this directory according to your will in order to generate it elsewhere.

```
<target name="testGenerator" depends="compile,jar">
        <echo message="Running LINGI2132 assignment ..."/>
        <javac srcdir="${basedir}/src/lingi2132/"
```

```
        destdir="${CLASS_DIR}"
        includeantruntime="false"
        debug="on">
        </javac>
         <java fork="true" failonerror="yes" classname="lingi2132.Generator" classpath=
"${basedir}/${CLASS_DIR}">
                    <arg line="/tmp/"/>
         </java>
</target>
```

Hints

- Good documentation is provided in the book. Use it !
- Ctrl-click and Ctrl-Shift-R will be very helpful in Eclipse in order to navigate in the j-- source code.

Programming : j--

Lexical Analysis

This assignment considers the handwritten compiler and the generated one (using JavaCC). We **strongly** recommend to implement first the handwritten part. However, do not neglect the JavaCC part, it can take time to get used to JavaCC.

In this part, you have to be able to scan and ignore Java multi-line comments.

Hand written Compiler

Modify the handwritten scanner to scan and ignore Java multi-line comments.

Generated Compiler

Modify the j--.jj file in order to make the generated scanner ignore multi-line comments. The section 2.9 in the book will help you.

Do not forget to modify the lexicalgrammar file describing formally the current lexical grammar of your j-- language (see Documentation section below).

Parsing

1. Modify the Parser (manual parsing) and j--.jj (JavaCC parser) to parse and return nodes for the conditional expression, for example (a > b) ? a:b . Because there is an expression hierarchy in the Java grammar, implementing support for level 12 (*conditionalExpression*, see appendix C in the book), you would normally have to support all levels below 12 too. In the case of this assignment, we do not ask you to do so. Consider the grammar rule 12 to be :

   conditionalExpression ::=
   conditional**And**Expression
   [? assignmentExpression **:** conditionalExpression]

   Also, modify the level 13 grammar rule to consider *conditionalExpression* instead of just *conditionalAndExpression*.

2. Modify the Parser (manual parsing) and j--.jj (JavaCC parser) to parse and return nodes for the for statement, including both the basic for statement and the enhanced for statement. In the case of the enhanced for loop, you only have to manage the case where *Expression* is an array of ints.
   The grammar for the enhanced for statement is not provided in the appendix C, but you can find it here : http://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.14.2

Read section 3.5 in the book for the generated parser.

Do not forget to modify the grammar file describing formally the current grammar of your j--language (see Documentation section below).

Semantic Analysis + JVM-Code Generation

We ask you to implement to semantic analysis and the JVM-Code generation for :

- conditional expressions.
- for loops (basic ones and enhanced ones). However, in the case of enhanced for loops, we will restrict the tests to int array expressions (so you may do more if you want, but it

will not be tested). Concretely, only statements of the form

for (*Type identifier* : *Expression*) *Statement* ,

where *Expression* is an array of *int* .

Caution : The type checking must be as effective as possible : if an error in a source code can be seen during compilation, it must be reported during this process, not during execution ! Here, you can see the importance of fail tests. In general, if you have been too permissive during the parsing, you have to handle potential errors during semantic analysis.

Testing

We ask you to provide some pass and fail tests. We will also add tests to which you do not have access. **If one test fails, the functionality** (e.g., handwritten scanning of multi-line comments) **is considered as not working, so cover all potential cases !**

Also, **remove** the tests you have from assignment 1. They will make fail the tests using JavaCC.

Do not forget that the tests you are writing must be written using the j-- language, which is way less permissive than Java ! If you hesitate, check carefully the j-- grammar.

Documentation

The “lexicalgrammar” and “grammar” files serve as documentation for this project. You **must** modify them according to your modifications. We will consider that what you implemented is what is documented in those files. So document correctly your work, and especially, do not forget this part, because nothing means nothing implemented … Provide a short explanation of this in your narrative.

Static Code Analysis

For this part, the instructions are the same as in the previous assignment.

Narrative

As for the previous assignment, you have to provide a short narrative that explains what you have done. Be sure that your narrative clearly and concisely describes your work.

Submission

Instructions

To submit your work, we ask you to export your Eclipse project as a **zip** file (right click on the project export... General Archive File). You **have to** use the following name convention for your zip file : j--_gr_*grnumber*.zip , where *grnumber* is your group number on iCampus.

You have to submit this zip file on iCampus (in Assignment_2(code_j--)).

Submit also a **brief** report (pdf format) on iCampus (in Assignment_2(report)), that includes your narrative and the solutions of the paper exercises.

Finally, you have to submit the recursive descent parser project on iCampus (in Assignement2_(code_recursive_descent)).

Script

You can reuse the script provided for the assignment 1, in order to check that your submission for the j-- part can be tested by our script.

Grades

This assignment represents 3 points of the overall grade of the course, so do not neglect it.

Narrative and documentation (/2.5)

- Clear discussion of what you did, didn't do, and why.
- Clear discussion of development: choices available to you, choices taken, and why.

- Clear discussion of testing: coverage, tests chosen, test results.

Paper exercises (/5)

Programming (/12.5)

For this part, the grades will be binary : either your programs pass our tests (and of course yours too!), either they do not. So it is in your interest to write as many **useful** tests as possible. An implementation is good only if it is correct in any case.