Programming assignment 2

Parsing

This document contains a description of an example parser with instructions to test it in section 2 and the assignment proper in sections 3 and 4.

This assignment will give you experience in implementing parsers, both by hand coding a recursive descent parser and by generating a parser using a parser generator (JavaCC). You will also construct a context-free grammar for a small language.

1 Preparations*

Read through this guide. Study the parsing example CalcParser. Make sure that you understand the `CalcParser.jjt` specification. Browse through the relevant technical documentation so you know where to look for details later.

Relevant parts of the literature:

- Appel: Ch 3 - 3.2
- Lectures F3 and F4.

2 The CalcParse demo*

There is a small parser example that extends the scanner from the first demo. The instructions for downloading and running the example are the same as in the first demo. The specification now resides in the file `Parser.jjt` which will be input to a preprocessor to JavaCC called JJTree. It will add code to build a rudimentary parse tree.

2.1 Grammar

We are considering the same kind of expression as in the CalcScan demo. The grammar for the language is

exp	→	factor (`"*"` factor)*
factor	→	let `|` numeral `|` id
let	→	`"let"` id `"="` exp `"in"` exp `"end"`
numeral	→	`<NUMERAL>`
id	→	`<ID>`

2.2 Parser.jjt*

The scanner part of the specification is the same as in the CalcScan demo.

The parser specification is a mixture of Java code and grammar notation. Each grammar rule is replaced using the following template

```
<method header> : {<method initials>}
{
  <grammar production>
}
```

The <method header> is a pure Java method header with return type, method name and parameters. It is recommended to use the same name as in the left hand part of the production. The <method initials> will appear in the beginning of the method body. It is typically used for declaration of local variables.

The <grammar production> is essentially the right hand part of the grammar rule with argument lists after the nonterminals. For more details, see [6].

```
void exp() : {}
{
  factor() ("*" factor())*
}

void factor() : {}
{
  let()
| numeral()
| id()
}

void let() : {}
{
  <LET> id() "=" exp() <IN> exp() <END>
}

void numeral() : {}
{
  <NUMERAL>
}

void id() : {}
{
  <ID>
}
```

The generated parser will build a parse tree where all nodes are `SimpleNode` objects. Every method invocation will create such a node. This node will have children created by methods invoked in the method body.

We would like the parser to check that there is no further tokens when it has seen a complete expression. The easiest way to accomplish that is to extend the grammar with a new start symbol and require that the expression is followed by an end of file token. start → exp <EOF> The local name of the node created by a method is `jjtThis`. It is possible to insert Java code inside <grammar production> provided that it is surrounded by braces. We use this to return this node. It will be the root of the complete parse tree.

```
SimpleNode start() : {}
{
  exp() <EOF>
  { return jjtThis; }
}
```

Download the example try it out!

2.3 build.xml*

JJTree requires an extra task in the `build` target. It will use the `jjt` specification to produce a `jj` specification and some supporting Java files in the parser directory. The `jj` file will serve as input to JavaCC.

```
<project name="Parser" default="build">

    <target name="build">
        <mkdir dir="${parser.directory}" />
        <jjtree target="${specification.directory}/${parser.name}.jjt"
                outputdirectory="${parser.directory}"
                javacchome="${lib}"
                buildnodefiles="true"
                static="false"
                nodepackage="${parser.package}" />
        <javacc target="${parser.directory}/${parser.name}.jj"
                outputdirectory="${parser.directory}"
                javacchome="${lib}"
                static="false" />
        <antcall target="compile" />
    </target>

</project>
```

3 Assignment A– The MiniS language

MiniS is a very small language with just a few statements and expressions. A program is just a single statement. The context-free grammar is shown below. You may assume that `<ID>` and `<NUMERAL>` are the same as in Assignment 1.

program	→	stmt
stmt	→	forStmt | ifStmt | assignment
forStmt	→	`"for"` `<ID>` `"="` expr `"until"` expr `"do"` stmt `"od"`
ifStmt	→	`"if"` expr `"then"` stmt `"fi"`
assignment	→	`<ID>` `"="` expr
expr	→	`<ID>` | `<NUMERAL>` | `"not"` expr

Figure 1: Context-free grammar for MiniS

3.1 Example program for MiniS*

Write an example program in MiniS that uses all constructs in the grammar.

3.2 Scanner for MiniS*

Generate a scanner for MiniS using JavaCC. It is convenient to make a copy of the Assignment 1 project and adapt it for this language. We assume that the name of the parser is `MiniSParser`.

3.3 Hand coded recursive descent parser

Implement a recursive descent parser for MiniS by hand, not using JavaCC. You should use the scanner for MiniS that you have generated with JavaCC. Your parser can get tokens by calling the method `getNextToken()` in the generated scanner. Token constants are defined in the file MiniSParserConstants.java, which is created when generating the scanner. Normally, JavaCC is used for generating both a scanner and a parser, in which case the scanner constants are also in this file.

You can get inspiration for your implementation of the parser by studying the example in the beginning of section 3.2 in the textbook (similar examples were presented in lecture F4).

The parser should only decide if an input program is correct according to the grammar — you should not build an abstract syntax tree. In case of erroneous input the program should indicate the first error with some explanation. The grammar is so simple that you should be able to implement the parser right away, without constructing a table first.

As usual, work with small increments and write automated tests. Write test cases for both correct and incorrect programs.

4 Assignment B — Generate a parser with JavaCC

Figure 2 shows an example program in a small procedural language. The operators have standard precedences, i.e., `*` has higher precedence than `+` which in turn has higher precedence than `<`.

```
procedure main(integer n)
begin
  integer sum = 0;
  integer k;
  for k = 1 until n do
    sum = sum + k * 2 + 1;
  od;
  if sum + 2 * n < sum * 2 then
    print(sum);
  else
    print(0);
  fi;
end;
```

Figure 2: Example program in a small procedural language

4.1 Design of the CFG*

Design a context-free grammar for a small procedural language such that the program in Figure 2 is correct. It will be easier to work with small increments if the start symbol generates a procedure as above, a single statement, or a single expression. JavaCC will complain since it cannot decide if it is looking at the start of a statement or an expression just by inspecting the first token. Follow the suggestion to use `LOOKAHEAD(2)` in front of the statement. The grammar should be written in EBNF form. It is fine to add more constructs than shown in the example, but do not make the language too large. Try to find good names for the non-terminals.

4.2 Generate a parser

Implement a parser for your language using JavaCC and JJTree. Look at the `CalcParser.jjt` file for inspiration. JJTree is a preprocessor for JavaCC with which you can build abstract syntax trees during parsing. When using JJTree, the specification must be written in a `.jjt` file instead of a `.jj` file.

Again, it is convenient to start from a copy of Assignment 1. It is important that you use the build file from the CalcParse demo. Also, copy the file `TestParser.java` from the demo. You have make some renaming in `build.xml`.

The parser should print the syntax tree that is generated by JJTree for your input program. Take a look at the CalcParse example to get going. Notice that you will not implement a separate scanner here; the generated parser will include the scanner and the parser will itself call `getNextToken`. Work as before in small increments and automate your test cases. Modify your grammar when needed.

You should at least construct the following test cases:

- Scanning of a program with typical test cases for all kinds of correct tokens.
- Scanning of a program with typical erroneous tokens.
- One test case for each grammar construct. If there are iterations using * then there should be one additional test so that you cover both zero and nonzero iterations.
- Two test cases with programs that contain likely syntax errors.
- One test case with a program containing text after a complete correct program (trailing garbage).
- The program in Figure 2.

5 Topics for discussion

1. Why should there be an `<EOF>` in the grammar?
2. There is a `doc` target in the build file. What is the purpose?
3. As can be seen in the `exp` method on page 2 you may write the literal string `"*"` instead of the symbolic token name `<TIMES>`. Such strings will be considered to be reserved words by the scanner generator even if they are not defined as symbolic tokens. When do we need to define symbolic names, and when is it good to that even if it is not required?
4. Does it matter if `+` and `*` are left or right associative? How about `-` and `/`?
5. The Quick guide [3] provides an example on how to skip a single line comment. Could it be made simpler?
6. Java has comments surrounded by `/*` and `*/`. How would you skip such comments?

References

[1] A. W. Appel with J. Palsberg: *Modern Compiler Implementation in Java*, 2nd Edition, Cambridge University Press, 2002, ISBN: 0-521-82060-X. Especially 2.1 and 2.2.

[2] L. Andersson, Lecture slides from Lectures F3-4, `http://cs.lth.se/EDA180`.

[3] G. Hedin: *Quick guide - Scanning in JavaCC*, `http://cs.lth.se/eda180`, located 2009–12–07.

[4] *Apache Ant*, `http://ant.apache.org`, located 2009–12–04.

[5] *Apache Ant Manual*, `http://ant.apache.org/manual/index.html`, located 2009–12–05.

[6] *Javacc*, `https://javacc.dev.java.net`, located 2009–12–04.