Coursework : Recursive Descent Recogniser

SCC312 Compilers
2013/14
Version 1

1. General Instructions

Section 3 gives a grammar for a simple programming language rather like Ada. The task is to implement part of a compiler for this language using a recursive descent parser.

1.1. Java Classes Provided

You are provided with the following Java classes:

(a) **Token** in a file **Token.java**, to represent a token returned by the lexical analyser stage. This has:
- a set of integer constants (**becomesSymbol**, **beginSymbol**, **identifier**, **leftParenthesis**, and so on) representing the possible types of token in this language
- three public attributes (**symbol**, an int, which is one of the constants declared above; **text**, a String, the characters making up the token; **lineNumber**, an int, the number of the line containing the token)
- two constructors, and a static method **getName** to return the name (a String) of a token provided as the single int argument

(b) **CompilationException** in a file **CompilationException.java** (see below)

(c) **LexicalAnalyser** in a file **LexicalAnalyser.java**, which is the lexical analyser for this programming language. This has:
- a constructor with one String argument, the name of the file from which the tokens are to be read
- a method **getNextToken** (with no arguments), to return the next token read from the source text
- a **main** method, with which the operation of the lexical analyser can be tried out on a suitable file (using a **toString** method supplied in the **Token** class)

(d) **AbstractGenerate** in a file **AbstractGenerate.java**. This is the abstract class you need to make concrete in the Generate class you have to provide.

(e) **AbstractSyntaxAnalyser** in a file **AbstractSyntaxAnalyser.java.** This is the abstract class you need to make concrete in the SyntaxAnalyser class you have to provide.

(f) **Compile** in a file **Compile.java**. This is the driver program for the whole coursework. This driver program calls the **parse** method of the **SyntaxAnalyser** class for each file with a name of the form "program*n*" (integer $n \geq 0$) (these files are in the coursework pack).

These classes can be found in the coursework pack that this document came in.

1.2. Java Classes To Be Implemented

1.2.1 SyntaxAnalyser

Write a Java class **SyntaxAnalyser** .

The **AbstractSyntaxAnalyser** class contains the following methods :

abstract void programItem() throws IOException, CompilationException;
abstract void acceptTerminal(int symbol) throws IOException, CompilationException;

public void parse(PrintStream ps)throws IOException

You have to extend the above class as appropriate.

1.2.2. Generate

The parser should make use of the **Generate** class, which you must also supply by extending the **AbstractGenerate** class. The **AbstractGenerate** class contains the following methods:

```
public void          insertTerminal(Token token);
public void          commenceNonterminal(String nonTerminalName);
public void          finishNonterminal(String nonTerminalName);
public void          reportSuccess();
public abstract      reportError(Token token, String explanatoryMessage)
void                 throws CompilationException;
```

The parser should demonstrate its operation by calling the **Generate** class methods as follows:

- **insertTerminal(Token token)** when it has correctly read a terminal.
- **commenceNonterminal(String nonTerminalName)** and **finishNonterminal(String nonTerminalName)** when it respectively starts and finishes reading a non-terminal. For non-terminals specified in the grammar below, the String **nonTerminalName** should be that specified in the grammar (for example "<procedure list>" or "<assignment statement>"). For new non-terminals introduced by you, the String **nonTerminalName** should be of the form "<new SOMETHING>".
- the void method **reportSuccess()** when it has successfully parsed the file.

Use these methods in a class **Generate** to display a trace (using System.out.println) of the operation of the parser.

Full error recovery is not required for this parser. Instead **parse** should report the first syntax error encountered, by calling **reportError(Token tokenRead, String explanatoryMessage)** in the **Generate** class. Implement a suitable version of this method to indicate what the next erroneous token is, what the parser is trying to recognise at this point, and the line number where the error is recognised. The method should finish by throwing the exception **CompilationException**, which should be caught by the **parse** method in the **SyntaxAnalyser** class.

The **parse** method should return in the normal way after processing a file, whether it reports success or failure, so that it can then be called to start to process the next file (if any).

Your **SyntaxAnalyser** class should include at an appropriate place a comment line which includes the string "author" and your name.

You may include in your **Generate** class either or both the constructor methods **Generate()** and **Generate(String)**, but no other methods than those specified in **AbstractGenerate**.

Marks will be allocated approximately as follows:

the classes **SyntaxAnalyser** and **Generate**	26
selected output (see below) from running the program over the test data provided	18
a report discussing:	32

- the overall structure and main design decisions in your implementation
- the issues involved in dealing with the non-LL(1) nature of this grammar (note: there are **two** points to consider here)
- the proposed alternative version of the <procedure statement>
- any improvements you can suggest in the grammar and lexical analyser supplied
- your comments on the errors found by your parser in the test programs. That is, for each test program, is your syntax analyser correct in saying that there is an error (according to the grammar supplied) or that the program is syntactically correct according to the grammar? How helpful is the error message supplied (is it indicating the error correctly, in the correct place, or is it at least doing as well as could be expected)?

There are some more requirements for the report in section 3.

2. Submission of Work

You should submit:

your problem report (covering **all** of the items listed above)

listings of the code you have written (the classes **SyntaxAnalyser** and **Generate**, suitably laid out and commented)

all the output from running your code over test file "program8", and the last ten lines or so of output from each of the other test files (1 to 7) (that is, to include the error or success message). This should be in one text document. *Please note we have provided sample output from our worked solution on "program0"; you should use this as a guideline for the output your recogniser produces, and as a check for the results of your recogniser on "program0". There is no need to include your results for "program0".*

3 Grammar Rules for part of a Simple Programming Language

```
<statement part> ::= begin <statement list> end

<statement list> ::= <statement> |
          <statement list> ; <statement>

<statement> ::= <assignment statement> |
          <if statement> |
<while statement> |
<procedure statement> |
<until statement>

<assignment statement> ::= identifier := <expression> |
          identifier := stringConstant

<if statement> ::= if <condition> then <statement list> end if |
if <condition> then <statement list> else <statement list> end if

<while statement> ::= while <condition> loop <statement list> end loop

<procedure statement> ::= call identifier (<argument list> )

<until statement> ::= do <statement list> until <condition>

<argument list> ::= identifier |
          <argument list> , identifier

<condition> ::= identifier <conditional operator> identifier |
identifier <conditional operator> numberConstant |
identifier <conditional operator> stringConstant

<conditional operator> ::= > | >= | = | /= | < | <=

<expression> ::= <term> |
<expression> + <term> |
<expression> - <term>

<term> ::= <factor> |
<term> * <factor> |
<term> / <factor>

<factor> ::= identifier |
          numberConstant |
( <expression> )
```

An "identifier" is a sequence of one or more letters (a to z, A to Z) and digits (0 to 9), starting with a letter, and excluding all the reserved words shown in **bold** above (**procedure**, **is**, **integer**, etc). Have a look at the **initialiseScanner** method in **LexicalAnalyser.java**.

A "numberConstant" is a sequence of one or more digits (in which case it is of type "integer"), perhaps followed by a decimal point and one or more digits (in which case it is of type "float"). A "stringConstant" is a sequence of one or more printable characters (except ") with a " character at each end. Comments start with the symbol -- and terminate at the end of the line.

The distinguished symbol is <program>.

This simple language has no boolean or character data types; no arrays or records; no functions; the actual parameters of all procedures must be identifiers, and are called by reference; only simple boolean expressions (no not, and or or); only simple numerical expressions (no unary minus).

The grammar as written is not LL(1); it has left-recursive rules of the form:

```
<X list> ::= <X> | <X list> separator <X>
```

and rules of the form:

<something> ::= α X β | α Y γ

where α, β and γ are strings of terminals and/or non-terminals (α non-null) and X and Y are different terminal symbols. In your report you should explain how you handle **both** these problems in transforming the grammar into a recursive descent parser.

Also discuss in your report what problems there might be for the parser if the <procedure statement> had been defined as follows:

```
<procedure statement> ::= identifier ( <argument list> )
```