Coursework : Recursive Descent Recogniser

SCC312 Compilers 19/20
Version 1.3

1. General Instructions

Section 3 gives a grammar for a simple programming language rather like Ada. The task is to implement a syntax analyser (SA) for this language using a recursive descent parser. The analyser's sole function is make sure a user's source program is syntactically correct, and the SA should generate appropriate and helpful error messages where required. The SA should terminate on encountering and reporting the first error. To be more precise, you are expected to build a Syntax Recogniser as its purpose to recognise its input as a valid sentence in the language specified by the grammar.

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 ZIP folder that this document came in. To help you build and run your work, as well as generate this ZIP file to submit, some shell scripts have been included:

1.2 Building the Code

For Windows Users:

Running 'compile.bat' will compile all java source files
Running 'execute.bat' will run the compiler against all supplied test programs

For Mac/Linux Users:

A makefile has been included which has the following functions (where '$>' is your terminal prompt:

$> make
Compile all required java sources

$> make run
Run the compiler against all supplied test programs

$> make package
Build a submission .zip file - this is a 'beta' feature, check your submission file has been created successfully before actually submitting it!

Note; The makefile requires a working install of the following: zip, java, javac, and make to work correctly.

1.3 Java Classes To Be Implemented

1.2.1 SyntaxAnalyser

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

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

```
abstract void    _statementPart_()
                      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. **Please note that the parse method is provided for you.**

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 void   reportError(Token token, String explanatoryMessage)
                       throws CompilationException;
public Variable        getVariable( String identifier );
public void            addVariable( Variable v );
public void            removeVariable( Variable v );
```

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>".
- **addVariable(Variable v)** and **removeVariable(Variable v)** when creating or destroying variables - during an `<assignment statemnent>`, for example. The **getVariable(String identifier)** is included as a suggestion and/or hint, but can be omitted if your implementation does not require it.
  These methods can be left blank until you attempt the 'Challenge Marks' - **see section 1.3**
- 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.

Error recovery is not required for this parser. Instead **parse** should report at 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 eventually be caught by the **parse** method in the **SyntaxAnalyser** class. As the exception reaches each of your parse methods, you should use it as an opportunity to report where in the parse tree the error occurred. **Hint**: Look at the constructor for **CompilationException.**

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).

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**.

You should strive to make your error messages as helpful and as accurate as possible. As mentioned in the lectures, consider how the structure of the program can be used to get context for errors. When using if statements, if you have more than 2 branches, please use a switch statement instead.

1.3 Challenge Marks - A little bit of semantic analysis

Thus far we've only looked at the syntax of the input files, but to achieve the final few marks for this work, you'll need to implement a small portion of a semantic analyser - specifically, your implementation should be able to do the following:

1. Report the declaration and destruction of variables using the `addVariable()` and `removeVariable()` methods in Generate
2. Determine if the operations used in expressions and factors are syntactically correct, but actually not possible, reporting an error where these are found.
   Using a variable before it has been created, or attempting to do anything other than concatenate strings with '+' (all other string operations are invalid).

   See section 3 of this task for which operations apply to which variable types.
3. Handle temporary iterator variables in **`<for statement>`** blocks.

While the rest of our Ada variant handles variable identifiers in a single, global, persistent namespace, the <for statement> creates a new *temporary variable* which only exists for the lifetime of the for loop, and will forget the variable at the end of the for loop. By example:

```
for( temp := 1; temp < 5; temp := temp + 1 ) do     -- 'temp' is created here
      call put( temp )
end loop ;                                          -- 'temp' is forgotten here
```

Note that the 'temp' variable is only *created* here because it didn't already exist in the program. If the variable did already exist, then the for loop would be merely *using* the variable and is not *declaring* it, and as such, the variable would not be destroyed after the loop.

Consider what happens in the following code snippet:

```
b := 4 ;                                        -- 'b' declared here
for( a := 1; a < 5; a := a + 1 ) do             -- 'a' is declared here
      for( b := 1; b < 5; b := b + 1 ) do
            call put( a, b )
      end loop ;
end loop ;          -- 'a' is removed but 'b' should continue to exist after here
```

In this case, the variable 'b' exists before the initial for loop, gets used in the inner loop (not redeclared) then continues to exist beyond both loops (it is not destroyed). To achieve full marks, your implementations will have to handle situations such as these.

1.4 Mark Distribution

Marks will be allocated approximately along the following scheme (further detailed breakdowns at during the actual marking, this should be used as an overview):

- SyntaxAnalyser (structure, design and implementation)	32 marks
- Generate (methods used, implemented and extension)	7 marks
- For each program (x13), either: - Success on syntactically correct programs: - Errors on syntactically incorrect programs:	1 mark (for each program)
- Detailed (correct) error messages:	2 marks
- Recursive errors (stack traces):	2 marks
- Temporary variables handled correctly:	2 marks
- Undeclared variables caught and reported:	2 marks
- Invalid operations caught and reported:	2 marks
- Code quality and comments:	2 marks
Total:	64 marks

Naturally, these categories will be combined to a final percentage and letter grade, as per usual.

1.5 Test Data

The source texts to be analysed can be found in the "Programs Folder" provided. The output from "program0" is provided as a guide as to what is expected in the way of output, so there is no need to include the results of recognizing "program0".

To make it easier to see where sections start and end, I have indented the output for program0, and while you are not required to do the same in your output, consider it an unofficial challenge from me to work out how I did this automatically for all programs without altering the AbstractGenerate class :)

2. Submission of Work

You should submit:

Listings of the code you have written (the classes **SyntaxAnalyser** and **Generate**, suitably laid out and commented), and all the output from running your code over test files, **both output.txt and res.txt** should be submitted. If you are using Mac or Linux, the makefile will help you do this.

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".

Deadline: 1600 (4pm), Friday, Week 19

WARNING : You ***must not*** change any of the pre-supplied Java classes. The 2 classes you submit will be compiled and tested with the pre-supplied classes. If they fail to compile or run because they depend on some alteration you have made to the pre-supplied classes, you will receive a mark of zero.

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> |
                    <for 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>

<for statement> ::= for ( <assignment statement> ; <condition> ; <assignment
statement> ) do <statement list> end loop

<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 <statement part>.

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.

3.1 Operation Rules

Operation	For Numbers	For Strings
+	Add	Concatenate
-	Subtract	Not valid
*****	Multiply	Not valid
/	Divide	Not valid