CA4003 COMPILER CONSTRUCTION

Assignment 1: A Lexical & Syntax Analyser for Cal

October 2022

Katie Clancy
19452724

1.1 DECLARATION OF PLAIGERAISM

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged, and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the referencing guidelines found at:
https://www.dcu.ie/ovpaa/policies-and-regulations

Name: Katie Clancy
Date: 30/10/2022

1.2 REPORT

The grammar

In ‘cal.g4’, I wrote the grammar for the Cal language by taking lots of indication from ‘draw.g4’ & the various tutorials provided.

I defined the fragments, tokens, operators and reserved words and then went on to add in digits, which according to the language definition can be positive or negative integers.
I made note of the requirements for certain fragments too, like the language not being case sensitive, meaning I would have to implement upper and lower case letters in individual fragments, and also the required reserved words.
I created the identifier, which must start with a letter which can be followed by a string of letters, digits or underscores. These consisted of fragments (see below). To use these fragments, I made an identifier rule (so that the identifier will begin with a letter and be followed by more than 0 letters, underscores or integers).

```
fragment Letter:        [A-Za-z]+;
//fragment Digit:       [0-9];
fragment UnderScore:    '_';
```

```
ID:             Letter (Letter | Digit | '_')*;
```

I made sure to include whitespace and comment management as per the language definition also.
The comment management included single line and mlti-line comment directions. Single line comments are contained by a double forward-slash, while multi-line comments are contained in between a forward slash and a star, finished in the opposite order:

```
// this is a simple comment

/* this is also
   a simple
   comment */
```

White space is defined as space, tab, newline and instructions say to skip over these.

```
Comment: '/*' (COMMENT|.)*? '*/' -> skip;

Line_Comment: '//' .*? '\n' -> skip;

WS:             [ \t\n\r]+ -> skip;
```

Also, I included the parsing rules as per the language definition, e.g:

```
function:           type ID LBR parameter_list RBR
                    Is decl_list
                    Begin
                    statement_block
                    Return LBR expression
                    | RBR SEMI
                    End;
```

Once I initialised my grammar so the automatically generated files could be created & I could test if my grammar rules were correct (by using the grun command) and see the syntax tree, I began working on the semantic analyser ('cal.java').

The Java Parser

Again, I followed direction given in the tutorials to put together the parser.

Its structure follows the structure of the parser written in the tutorials, with a few things added.

The first element I added was the error handler:

```
parser.setErrorHandler(new DefaultErrorStrategy() {
```

I did some investigating and found that the best error handler to use in this case would be a 'DefaultErrorHandler', where I could include a 'try-catch' expression to determine the parsing status outcome.

```
try {
    ParseTree tree = parser.prog();
    System.out.println("Parsed Successfully! ....");
} catch (Exception e) {
    System.out.println(e);
    System.out.println("Parse Unsuccessfull :( ...");
}
```

To test the functionality of my parser, I used the following command to feed a test file into it:

```
(base) katieclancy@MBP-6568 cal % java calParser < test1.cal
```

I conducted several tests (as per the language definition) to make sure each part of my grammar was right, and also to make sure my parser was working as expected:

basic.cal

```
1  main
2  begin
3  end
```

basicTwo.cal

```
1  main
2  begin
3  eND
4
```

basicThree.cal

```
1   MAIN
2   begin
3   end
```

comments.cal

```
1   main
2   begin
3       // a simple comment
4       /* a simple /* nested */ comment */
5   end
```

simpleFunction.cal

```
1   void function () is
2   begin
3       return();
4   end
5   main
6   begin
7       function();
8   end
```

complexFunction.cal

```
integer mply (x:integer , y:integer) is
    variable result:integer;
    variable minus_sign:boolean;
begin
    // this section will determine the sign of result and convert args to non-negative values

    if(x < 0 & y >= 0)

    begin
       minus_sign := true;
        x := -x ;
    end

    else

    begin
        if y < 0 & x >= 0
        begin
            minus_sign := true;
            y := -y ;
        end
        else
        begin
            if ( x < 1 ) & y < 0
            begin
                minus_sign := false;
                x := -x;
                y := -y;
            end
            else
            begin
                minus_sign := false;
            end
        end
    end

    result := 0;
    while ( y > 0)
    begin
        result := result + x;
        y := y - 1;
    end

    if minus_sign = true
    begin
        result := -result;
    end

    else
    begin
        skip ;
    end

        return (result);
end

main

begin
    variable arg_1 :integer;
    variable arg_2 :integer;
    variable result :integer;
    constant five :integer := -50;

    arg_1 := -6;
    arg_2 := five;

    result := mply(arg_1, arg_2);
end
```

scopeTest.cal

```
variable i:integer;

integer test_fn (x:integer) is
   variable i:integer;
begin
   i := 2;
   return (x);
end

main
begin
   variable i:integer;

   i := 1;
   i := test_fn (i);
end
```

1.3 COPY OF CODE

cal.g4

```
grammar cal;

prog:                   decl_list function_list main;

decl_list:              decl SEMI decl_list |;

decl:                   var_decl | const_decl;

var_decl:               Variable ID COLON type;

const_decl:             Constant ID COLON type ASSIGN expression;

function_list:      function function_list |;

function:               type ID LBR parameter_list RBR
                                Is decl_list
                                Begin
                                statement_block
                                Return LBR expression
                        | RBR SEMI
                        End;

type:                   TypeInteger | TypeBoolean | TypeVoid;

parameter_list:     nemp_parameter_list |;

nemp_para_list:     ID COLON type
                                        | ID COLON type COMMA nemp_para_list;

main:                   Main Begin decl_list statement_block End;

statement_block:    LBR statement statement_block RBR |;

statement:              ID ASSIGN expression SEMI
                                        | ID LBR arg_list RBR SEMI
                                                | Begin statement_block End
                                        | If condition Begin statement_block End Else Begin
statement_block End
                                        | While condition Begin statement_block End
                                        | Skip SEMI;

expression:             bit binary_arith_op bit
                                        | LBR expression RBR
                                        | ID LBR arg_list RBR
                                        | bit;

binary_arith_op:    PLUS | MINUS;

bit:                    ID | MINUS ID | Digit | True | False;

condition:              TILDE condition
                                        | LBR condition RBR
                                        | expression comp_op expression
                                        | condition LBR OR OR AND RBR condition;

comp_op:                EQUAL | NOTEQUAL | LESS | LESSEQUAL | GREATER | GREATEREQUAL;

arg_list:               nemp_arg_list |;
```

```
nemp_arg_list:          ID | ID COMMA nemp_arg_list;


fragment A:             'a' | 'A';
fragment B:             'b' | 'B';
fragment C:             'c' | 'C';
fragment D:             'd' | 'D';
fragment E:             'e' | 'E';
fragment F:             'f' | 'F';
fragment G:             'g' | 'G';
fragment H:             'h' | 'H';
fragment I:             'i' | 'I';
fragment J:             'j' | 'J';
fragment K:             'k' | 'K';
fragment L:             'l' | 'L';
fragment M:             'm' | 'M';
fragment N:             'n' | 'N';
fragment O:             'o' | 'O';
fragment P:             'p' | 'P';
fragment Q:             'q' | 'Q';
fragment R:             'r' | 'R';
fragment S:             's' | 'S';
fragment T:             't' | 'T';
fragment U:             'u' | 'U';
fragment V:             'v' | 'V';
fragment W:             'w' | 'W';
fragment X:             'x' | 'X';
fragment Y:             'y' | 'Y';
fragment Z:             'z' | 'Z';

/* tokens */

COLON:                  ':';
SEMI:                   ';';
COMMA:                  ',';
AND:                    '&';
OR:                     '|';
MINUS:                  '-';
ADD:                    '+';
EQL:                    '=';
NEQL:                   '!=';
GREATER:                '>';
LESS:                   '<';
GREATEREQL:             '>=';
LESSEQL:                '<=';
TILDE:                  '~';
ASSIGN:                 ':=';
LBR:                    '(';
RBR:                    ')';
FWDS:                   '/';
COMMENT:                '//';
MLINECOMM:              '/*';

/* res words */

Variable:               VARIABLE;
Constant:               CONSTANT;
Return:                 RETURN;
Integer:                INTEGER;
Boolean:                BOOLEAN;
Void:                   VOID;
Main:                   MAIN;
If:                     IF;
Else:                   ELSE;
True:                   TRUE;
False:                  FALSE;
While:                  WHILE;
```

```
Begin:                          BEGIN;
End:                            END;
Is:                                     IS;
Skip:                           SKIP;

fragment Letter:        [A-Za-z]+;
//fragment Digit:       [0-9];
fragment UnderScore:    '_';


Digit:                  [0-9];

Integer:        (MINUS? [1-9] Digit*) | '0'+;

Letter:         [a-zA-Z]+;

ID:                     Letter (Letter | Digit | '_')*;

Comment: '/*' (COMMENT|.)*? '*/' -> skip;

Line_Comment: '//' .*? '\n' -> skip;

WS:                     [ \t\n\r]+ -> skip;
```

cal.java

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import org.antlr.v4.runtime.CharStreams;
import java.io.FileInputStream;
import java.io.InputStream;
import org.antlr.v4.runtime.misc.ParseCancellationException;
import org.antlr.v4.runtime.DefaultErrorStrategy;

public class cal {
   public static void main(String[] args) throws Exception {

        String inputFile = null;
        if (args.length > 0)
           inputFile = args[0];

        InputStream is = System.in;
        if (inputFile != null)
           is = new FileInputStream(inputFile);

        calLexer lexer = new calLexer(CharStreams.fromStream(is));
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        calParser parser = new calParser(tokens);

        //ParseTree tree = parser.prog ();
        //System.out.println (tree.toStringTree(parser));

        //calDisplayVisitor calVis = new calDisplayVisitor ();
        //calVis.visit (tree);

        parser.setErrorHandler(new DefaultErrorStrategy() {

           @Override
           public void recover(Parser check, RecognitionException store) {
              for (ParserRuleContext cxt = check.getContext(); cxt != null; cxt = cxt
```

```
                .getParent()) {
            cxt.exception = store;
          }

          throw new ParseCancellationException(store);

        }

        @Override
        public Token recoverInline(Parser check) {
          InputMismatchException store = new InputMismatchException(check);
          for (ParserRuleContext cxt = check.getContext(); cxt != null; cxt = cxt
              .getParent()) {
            cxt.exception = store;
          }

          throw new ParseCancellationException(store);
        }
      });

      try {
        ParseTree tree = parser.prog();
        System.out.println("Parsed Successfully! ....");
      } catch (Exception e) {
        System.out.println(e);
        System.out.println("Parse Unsuccessfull :( ...");
      }
    }

}
```