A Lexical and Syntax Analyser for the CCAL Language

CA4003 Compiler Construction

Assignment 1

Name: Hannah O'Connor
ID: 16382283
Date: 04/11/2019

Declaration

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 is a grave and serious offence 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 set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, 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.
I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified.
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 http://www.library.dcu.ie/citing&refguide08.pdf and/or recommended in the assignment guidelines.
I understand that I may be required to discuss with the module lecturer/s the contents of this submission.
I/me/my incorporates we/us/our in the case of group work, which is signed by all of us.

Signed: Hannah O'Connor
Date: 04/11/2019

Introduction

This purpose of this assignment was to implement a Lexical and Syntax Analyser to accept the language CCAL.

I was required to build a parser that would parse the given input via the command line, either from ccl file or standard input. I constructed my parser through coding the language's lexical and syntax descriptions within a JJ file called *'HannahParser.jj'* and invoking JavaCC on it. As a result, I got seven java files, all of which have been attached to my submission. To compile those file I ran *javac *.java* within the same directory.

Once the parser is initialized, the function **program()** is ready to parse any given input. If successful, the terminal will return "Completed Successfully". Else, a ParseException or TokenMgrError will be raised. This means that it can accept code from standard input by running *java HannahParser* or by passing a ccl file as an argument to the parser, e.g *java HannahParser test.ccal*

For my accompanying code, I used JavaCC 5.0.

Within this report, I'll be discussing the main components of my lexical and syntax analyser and how I implemented them. These components include:

- ★ Options
- ★ User Code
- ★ Tokens
- ★ Grammar & Production Rules

Section 1 - Options

For my parser I set one single option. It can be seen below.

```
options {
    IGNORE_CASE = true;
}
```

For the first option I set ***IGNORE_CASE***. This parser needed to be not case sensitive as specified within the CCAL language description. As a result, the parser can accept various case scenarios such as:

- ❏ Main
- ❏ MAIN
- ❏ Main

Those cases will be read the same way by the parser.

Section 2 - User Code

This section relates to the class declaration of my parser. The structure and basis for this section of code was taken from the modules notes on JavaCC, specifically the SLPTokeniser example. The name selected for my parser is simply "*HannahParser*". JavaCC parsers are required to be enclosed within the keywords ***PARSER_BEGIN*** and ***PARSER_END,*** this is where my HannahParser class definition is placed***.***

Within the HannahParser class definition, the main method initialises my parser and retrieves the user input from the command line. This input can either be standard input taken from the terminal, else a ccl file can be passed as an argument to HannahParser. If an error occurs in the process of retrieving the file, the user will be informed that it could not be retrieved via message within the terminal. Otherwise, instructions on how the parser should be used will be printed to the users' screen.

When the parser has been initialised and the given input has been received, the **program()** method from with the production and grammar rules will be called. When processing that input, it will work its way through the relevant rules to parse the input stream correctly until it finished with the <EOF> token (end of file). The determined results will be printed to the screen informing the user if the parser was successful or not. If an error does occur, it will be thrown based on it's type. Within my code, either a **ParserException** will be raised for unsuccessful parsing or **TokenMgrError** for errors relating to the tokens.

For getting input from stdin:

```
 Please enter input...
Call:   program
  Call:   decl_list
```

For file input:

```
Reading from file ../tests/scopes.ccal . . .
HannahParser: program parsed successfully.
```

Section 3 - Tokens

With tokens, they are used to specify that the matched string will be transformed into a token that can then be used to communicate with the parser. For example, `< VAR : "var" >` tells JavaCC that the keyword var is given the symbolic name “VAR”.

There are various subsections to my tokens declarations. These include comments, keywords, punctuation, operators, skip, numbers and identifiers.

Keywords

Within the CCAL language definition, there was a predefined list of what keywords, operators and punctuation should be represented in the form of tokens. To declare them, all that was required was to create a token block with each one taken from the given list, along with its matching string. The keywords shown below are reserved words for the language. I organised each type of token into its own subsection for the simple purpose of producing more readable code. Each defined token block for those mentioned sections can be seen below.

```
TOKEN : { /* Keywords */
      < VAR : "var" >
    | < CONST : "const" >
    | < RETURN : "return" >
    | < INT : "integer" >
    | < BOOLEAN : "boolean" >
    | < VOID : "void" >
    | < MAIN : "main" >
    | < IF : "if" >
    | < ELSE : "else" >
    | < TRUE : "true" >
    | < FALSE : "false" >
    | < WHILE : "while" >
    | < SKIP_TOKEN : "skip" >
}
```

```
TOKEN : { /* Punctuation */
      < COMMA : "," >
    | < COLON : ":" >
    | < SEMICOLON : ";" >
    | < ASSIGN : "=" >
    | < LBRACE : "{" >
    | < RBRACE : "}" >
    | < LPARENS : "(" >
    | < RPARENS : ")" >
```

```
}
```

```
TOKEN : { /* Operators */
      < PLUS_SIGN : "+" >
    | < MINUS_SIGN : "-" >
    | < NOT : "~" >
    | < OR : "||" >
    | < AND : "&&" >
    | < EQUAL : "==" >
    | < NOT_EQUAL : "!=" >
    | < LESS_THAN : "<" >
    | < LESS_THAN_EQUAL : "<=" >
    | < GREATER_THAN : ">" >
    | < GREATER_THAN_EQUAL : ">=" >
}
```

For the ***SKIP_TOKEN*** within the keywords token block, I wasn't able to name it as just '*SKIP*' as it was causing issues and conflicts with my parser due to the SKIP block, which contains the regular expressions for finding unless characters to skip. See the error message below.

```
Reading from file HannahParser.jj . . .
org.javacc.parser.ParseException: Encountered " "SKIP" "SKIP "" at line
128, column 9.
Was expecting one of:
    "EOF" ...
    "#" ...
    <IDENTIFIER> ...
    <STRING_LITERAL> ...
    "<" ...
    "~" ...
    "[" ...
    "(" ...

Detected 1 errors and 0 warnings.
```

Numbers & Identifiers

In the CCAL language description brief, there were precise instructions on how the number and identifier tokens should be represented.

- ❏ Integers can be a string of one or more digits, represented by enclosing <DIGIT> within (..)*. They also don't start with the digit '0. Yet they may start with a minus sign so it was enclosed with (..)? Meaning that it's optional.
- ❏ Identifiers are represented by a string of letters, digits or underscore characters. They must begin with a letter. Identifiers cannot be reserved words.
- ❏ I created the token "Digit" which takes any single digit from 0 to 9.
- ❏ I also created the token "Char" which takes any single lower or upper case letter.

The preceding character "#" in the ***DIGIT*** & ***CHAR*** token definition means that it can only be used in the definition of other tokens and were created and added within this block for that purpose. Again, this was to provide more ease of reading my code and eliminating redunant code.

```
TOKEN : { /* Numbers and identifiers */
      < #DIGIT : ["0"-"9"] >
    | < #CHAR : ["a"-"z", "A"-"Z"] >
    | < INTEGER : (<MINUS_SIGN>)? ["1"-"9"] (<DIGIT>)* | "0" >
    | < ID : <CHAR> (<CHAR>| <DIGIT> | "_")* >
}
```

Skip

Skip is required and an important element to this parser as its what allows for meaningless characters to be ignored by the parser.
Similarly with my token declarations, I split up my skip blocks based on their subsection.

Comments

In relation to comments, within the specification of the language it said the following:

"Comments can appear between any two tokens. There are two forms of comment: one is delimited by / and */ and can be nested; the other begins with // and is delimited by the end of line and this type of comments may not be nested."*

What it means is that comments from any given file or standard input must be ignored. There are two specific types of comments that we're dealing with: Single line and Multi line.

For single line comments, all that was required was a regular expression to be declared to match such a comment. Here it checks for it the line begins with "//" and matching any character in between, until it ends with one of the following: a newline character "\n", a return carriage "\r" or both "\r\n".

For multi-line comments, the treatment of this was quite different. Here these comments must begin with “/*” and end in “*/”. For this, I utilised the notes on multi-line comments. A ***commentNesting*** variable was declared and initialised to 0. The purpose of this variable is so that whenever “/*” is matched, it is incremented by 1. From there, the variable can either be increased again when another match is found, otherwise it is decremented when matched with “*/”. Upon being decremented, there is a check to see if the variable is equal to 0. If so, it changes back to its default.

```
TOKEN_MGR_DECLS :
{
    static int commentNesting = 0;
}
```

```
SKIP : { // Single line comments
    < "//" (~["\n","\r"])* ("\n" | "\r" | "\r\n") >
}

SKIP : /* Multi line comments */
{
    "/*" { commentNesting++; } : IN_COMMENT
}

<IN_COMMENT> SKIP :
{
    "/*" { commentNesting++; }
  | "*/" { commentNesting--;
             if (commentNesting ==0)
                 SwitchTo(DEFAULT);
         }
  /* OTHER (UNKNOWN) CHARACTERS */
  | < ~[] >
}
```

Skip characters

Here, skip is used to specify that the matched strings shown within the below block should be thrown away and not passed to the parser. On the first line, " " matches the space character meaning that it will be skipped and will not be passed to HannahParser. “\t” represents tabs. The next three lines following that are put in place in order to accomodate

line breaks. “\n” is relevant to unix and linux machines, “\r” is for Windows and “\r\n” is for older OS’s.

```
SKIP : /* Skip Characters */
{
    " "
  | "\t"
  | "\n"
  | "\r"
  | "\r\n"
  | "\f"
}
```

Section 4 - Grammar & Productions Rules

For my parser’s grammar and production rules, I defined my syntax based on the brief for the CCAL language. The first that I defined was **program()**, which was the first to be called by the parser from within the main method of the class definition of HannahParser upon retrieving input, as previously mentioned.

I declared the rules exactly as to how they were described. However, upon attempting to generate my parser, I ran into several issues with left recursion, receiving a lot of warnings from JavaCC.

Left Recursion

When first trying to generate my parser, below shows the warning messages for left recursion for my code. As an attempt to rectify this, I used the modules notes to guide me in finding a solution.

```
Reading from file myParser.jj . . .
Error: Line 237, Column 1: Left recursion detected: "expression... -->
fragment... --> expression..."
Error: Line 259, Column 1: Left recursion detected: "condition... -->
condition..."
Detected 2 errors and 0 warnings.
```

It was present within **expression()** and **condition()**. The original versions can be seen below before I refactored to remove the left recursion.

```
void fragment() : {} {
      <ID>
    | <MINUS_SIGN> <ID>
    | <INTEGER>
```

```
    | <TRUE>
    | <FALSE>
    | expression()
```

```
void expression() : {} {
      fragment() binary_arith_op() fragment()
    | ( expression() )
    | <ID> ( arg_list() )
    | fragment()
}
```

```
void condition() : {} {
      <NOT> condition()
    | <LPARENS> condition() <RPARENS>
    | expression() comp_op() expression()
    | condition() (<AND>|<OR>) condition()
}
```

For **expression()**, it contained indirect left recursion. What this means is with the original **expression()**, it calls upon **fragment()** which calls upon **expression()**. This leads to the possibility of an infinite loop.
With expression I used kleene star closure, meaning that within the parentheses can exist 0 or more times. **Expression()** will either calls upon **fragment()** or **binary_arith_op() fragment()** too.

Upon changing **expression()**, it ultimately led me to needing to change **fragment()** by removing expression() from it.

```
void expression() : {} {
      fragment() (binary_arith_op() fragment())*
}
```

```
void fragment() : {}
{
  (<MINUS_SIGN>)* <ID> (<LPARENS> arg_list() <RPARENS>)*
| <INTEGER>
| <TRUE>
| <FALSE>
}
```

For **condition()**, it contained direct left recursion. With this meaning that **condition()** was able to continuously call upon itself and could also possibly result in an infinite loop. The solution can be seen below.

```
void other_condition() : {}
{
      and_or() condition() other_condition() | {}
}

void condition() : {}
{
  <NOT> condition() other_condition()
| <LPARENS> condition() <RPARENS> other_condition()
| fragment() comp_op() expression() other_condition()
}
```

Choice Conflicts

After spending time attempting to resolve the warnings about the existing left recursion, I was left with several choice conflicts.

From my gathered understanding, choice conflicts occur when the parser attempts to apply a production rule and it must choose between two possible paths. It means that based on the time of analysis, JavaCC is unable to determine what correct production to apply based only on a one-symbol lookahead. Choice points don't just occur with "|" but with "(..)*" also.

```
Warning: Choice conflict involving two expansions at
         line 214, column 5 and line 214, column 31 respectively.
         A common prefix is: <ID> ";"
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 229, column 5 and line 230, column 7 respectively.
         A common prefix is: <ID>
         Consider using a lookahead of 2 for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 244, column 7 and line 246, column 7 respectively.
         A common prefix is: "(" "("
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 256, column 7 and line 261, column 7 respectively.
         A common prefix is: <ID>
         Consider using a lookahead of 2 for earlier expansion.
Warning: Choice conflict in (...)* construct at line 266, column 19.
         Expansion nested within construct and expansion following
construct
```

```
         have common prefixes, one of which is: "||"
         Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict involving two expansions at
         line 271, column 7 and line 272, column 7 respectively.
         A common prefix is: "(" "("
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Choice conflict involving two expansions at
         line 291, column 7 and line 292, column 7 respectively.
         A common prefix is: <ID>
```

Examples from my code:

The default choice for making a decision between two possible paths would be to look at the next token in the input stream but when the next token is a <ID> then either choice would be appropriate. This means that the first choice will be the default and can make the other unreachable. You see this scenario occur within the original **nemp_arg_list()**.

```
void nemp_arg_list() : {} {
      <ID>
    | <ID> <COMMA> nemp_arg_list()
}
```

My solution can be seen below. Here it is possible to choose <ID> and {} or <ID> and <COMMA> nemp_arg_list().

```
void nemp_arg_list() : {}
{
    <ID> (<COMMA> nemp_arg_list() | {})
}
```

With **statement(**), the choice conflict said that the common prefix was <ID>. What this means was that with this case, upon expanding and looking for <ID>, it doesn't know which is a starting an assignment and which is a variable. That is why it needs to consider the next token too. So I refactor the first two lines and created **other_condition()** where it is called as <ID> other_condition().

```
void statement() : {} {
      <ID> <ASSIGN> expression() <SEMICOLON>
    | <ID> <LPARENS> arg_list() <RPARENS> <SEMICOLON>
    | <LBRACE> statement_block() <RBRACE>
    | <IF> condition() <LBRACE> statement_block() <RBRACE>
      <ELSE> <LBRACE> statement_block() <RBRACE>
```

```
    | <WHILE> condition() <LBRACE> statement_block() <RBRACE>
    | <SKIP_TOKEN> <SEMICOLON>
```

My solution:

```
void other_statement() : {} {
    <ASSIGN> expression() <SEMICOLON>
    | <LPARENS> arg_list() <RPARENS> <SEMICOLON>
}

void statement() : {} {
      <ID> statement_one()
    | <LBRACE> statement_block() <RBRACE>
    | <IF> condition() <LBRACE> statement_block() <RBRACE> <ELSE>
<LBRACE> statement_block() <RBRACE>
    | <WHILE> condition() <LBRACE> statement_block() <RBRACE>
    | <SKIP_TOKEN> <SEMICOLON>
}
```

Taking **nemp_parameter_list()** as an example, it was more of a simple fix. Here, I was able to just merge the two choices, given that the second option contained part of the first, making use of the optional regex brackets.

```
void nemp_parameter_list() : {} {
    <ID> <COLON> type() | <ID> <COLON> type() <COMMA>
nemp_parameter_list()
}
```

My solution:

```
void nemp_parameter_list() : {} {
   <ID> <COLON> type() (<COMMA> nemp_parameter_list())?
}
```

Testing

Following the completion of my parser, the removing of those choice conflicts and eliminating the left-recursion, I moved on to the final step, testing. I ran the seven example tests written in the CCAL language from within the assignment brief. This gave me a better insight into hidden issues within my code.

The case sensitivity and comments tests ran successfully on the first go. However, I had more of a challenge with the latter two tests, scopes and functions. I had encountered many unsuccessful parsing errors before reaching success, yet, most of the errors that were responsible were simply due to bad syntax within my rules and the misplacement of some rules too. For example, I had specified the <LBRACE> rather than <LPARENS> and for **parameter_list()**, this was my original syntax:

```
void parameter_list() : {} {
    nemp_parameter_list() | "" }
```

Before changing it to:

```
void parameter_list() : {} {
    (nemp_parameter_list())?
}
```

In attempts to reach a solution, I made use of setting ***DEBUG_PARSER.*** This way, I was provided with more logging which ultimately aided me in pinpointing the direct source of the error messages I was receiving.

When first running scope_test.ccl:

```
Encountered " "integer" "integer "" at line 1, column 7.
Was expecting one of:
    "boolean" ...
    "void" ...
    <INTEGER> ...
```

With this, the error seemed to caused by **type()** and so I changed it to accomodate for the token <INT> also.

```
void type() : {} {
    (<INTEGER> | <BOOLEAN> | <VOID> | <INT>)
}
```

```
Encountered " "return" "return "" at line 8, column 3.
Was expecting one of:
    "if" ...
    "while" ...
    "skip" ...
    "{" ...
```

```
    <ID> ...
```

Solution was to surround the following non-terminals with (...)?. Theses included: decl_list(), function_list(), parameter_list().

Successful running of my tests:

```
Reading from file ../tests/case_senstive_1.ccal . . .
HannahParser: program parsed successfully.
```

```
Reading from file ../tests/case_senstive_2.ccal . . .
HannahParser: program parsed successfully.
```

```
Reading from file ../tests/case_senstive_3.ccal . . .
HannahParser: program parsed successfully.
```

```
Reading from file ../tests/comments_1.ccal . . .
HannahParser: program parsed successfully.
```

```
Reading from file ../tests/scopes.ccal . . .
HannahParser: program parsed successfully.
```

```
Reading from file ../tests/functions.ccal . . .
HannahParser: program parsed successfully.
```

To run HannahParser

I create a simple bash script called **start.sh** which can generate the parser, compile the relevant java file and run the seven tests against the parser.

To run the script, make sure you're in the same directory as the parser and enter:

```
» sh start.sh
```

References & Sources

- Modern Compiler Implementation in Java, Second Edition. Online Link to PDF: https://eden.dei.uc.pt/~amilcar/pdf/CompilerInJava.pdf
- https://cs.lmu.edu/~ray/notes/javacc/
- http://www.engr.mun.ca/~theo/JavaCC-FAQ/javacc-faq-moz.htm