CCAL Assignment 1 Report

CA4003: COMPILER CONSTRUCTION

LEXICAL/SYNTAX ANALYSER

Nigel Guven | 14493422 | 4th November 2019

Word Count: 2951

Table of Contents

1. Assignment Objective

Title: Assignment 1: A Lexical and Syntax Analyser for the CCAL Language

Aim

The aim of this assignment is to implement a lexical and syntax analyser using JavaCC for a simple language called CCAL. Details of this language are found in 'CCAL Definition' found in this directory. The compiler should be defined in JavaCC

"JavaCC provides two important advantages. First, it substantially reduces the time required to produce a compiler. Second, it allows us to produce more reliable compilers. The majority of bugs in hand-written compilers are introduced during tedious process of computing selection sets and grinding out the code from the translation grammar. Because JavaCC will do these tasks and will do them correctly, it produces a compiler with fewer bugs."

J, Dos Reis, Anthony. Compiler Construction using Java, JavaCC and YACC. Chap-13.

2. The Contents of this Directory

This directory should contain the source files, the main of which is MyCCParser.jj which is the grammar file to describe the Lexical and Syntax elements of the CCAL grammar. There are also some files generated by the parser which are shown below. And finally, the document of plagiarism and this specific document which is the assignment report.

- CCAL Definition
  This file provides information on the CCAL Language and was created by the examiner. Contained within is the grammar definition of the CCAL Language. It displays the set of all reserved words, tokens, literals in the first section. The second section describes the production rules of the grammar which define the syntax of the language. A list of semantic operators are provided with their order of precedence. A sample of program files are provided for testing the syntactic correctness of the JavaCC grammar file. These include testing empty statements, variables, scope and functions.

- Declaration on Plagiarism
  This statement is the required signed declaration of originality.

- MyCCParser.jj
  This JavaCC file contains the main grammar file specifications and is the most important file in this directory. It contains, the lexical and syntax analyser for the CCAL assignment language. See chapter 4 for the comprehensive look into the breakdown of this file.

- Test.ccl
  This type '.ccl' document was used to test upon the grammar file and contains a sample program provided from the CCAL Definition document.

- MyCCParser.java
  This file is the translated version of the JavaCC file into a standardized Java file. It contains the parser block described in the former and also demonstrates the benefit of using JavaCC by translating every production in the grammar file into static Java functions for use by the parser. This saves a lot of time for a person trying to develop a compiler spontaneously. There is a compiled class for this java file as well as two added classes for calling tokens and to describe the lookahead functions.

- MyCCParserConstants.java
  This java file describes the lexical tokens and literals described in the grammar file. This file is supplementary to the parser and contains a compiled class.

- MyCCParserTokenManager.java
  This file is generated once the JavaCC file is compiled. It contains the proprietary set of every possible lexical state described within the JavaCC grammar. The token manager is in exactly one lexical state at any moment and considers regular expressions that might match this lexical state. Once a match is found, an item can be placed into one of four regular expressive productions i.e. SKIP, MORE, TOKEN, SPECIAL_TOKEN. This file is supplementary to the parser and contains a compiled class.

- ParseException.java
  This file is generated by JavaCC when a grammar file is compiled and is the same for all grammar files. It checks for parse errors that the main parser file encounters when it is reading through a file. This java file can be modified by the user.

- SimpleCharStream.java
  This file is an adapter class for the Token Manager and delivers characters though some I/O medium to the Token Manager to process into regular expressions. It is the same class for all JavaCC grammars.

- Token.java
  This file is a class representing the token objects which are manipulated by the Token Manager. Each token has an integer variable called kind that represents the token type which is determined by JavaCC, and also a String variable called image which describes the contents of the token itself. The token also operates similar to a linked list whereby it contains a link to the next token as another variable.

- TokenMgrError.java
  The Token Manager Error class contains a simple detection system for checking lexical errors in the Token Manager and the same file generated for all JavaCC grammars. The files that do not include the name of the parser as a prefix are generated with boilerplate code like this file.

3. Tasks

In order to understand how to develop this grammar, I decided to list all of the necessary tasks that I would need in order to achieve full correctness for this assignment language. *This section includes a breakdown of the tasks laid out in the assignment definition and does not include any research made separately in books or in the JavaCC documentation.*

1. Make the CCAL language case-insensitive.
2. Represent any test files with a type ending of '.ccl'.
3. List the reserved word of the language.

var	const	return	integer	boolean	void	main
while		skip	if	else	true	false

4. List the separators of the language.

,	;	:	=
{	}	(	)

5. List the arithmetic and Boolean logic of the language.

+	-	~	||	&&	==	!=
<		<=	>		>=	

6. Create a regular expression for integers that have no leading zero characters but may be preceded by a minus sign.

("-") ("1-9") "[0-9"]) *

7. Create a regular expression for characters and strings whereby the first character is always a letter. Identifiers cannot be reserved words. This is done by order of precedence in the lexical section of the grammar file.

["a-z"] ["A-Z"] (["a-z"] ["A-Z"] | ("0-9") | "_")

8. Create a regular expression to check for comments both single-line and multi-line. Add a variable to check for nested comments.

9. Write the productions in the form of nonterminals having a capitalized first letter. Bold characters and strings represent tokens.

<CONST> **<IDENTIFIER>** **<COLON>** Type() **<ASSIGNMENT>** Expression()

Items in between '<>' are terminals and Items that are suffixed by '()' are nonterminals and may be expanded upon.

10. Instantiate the empty variable as '{}'.

11. Remove left recursion where the JavaCC states in the command line.

```
Reading from file MyCCParser.jj . . .
Error: Line 374, Column 1: Left recursion detected: "Expression... --> Expression..."
Detected 1 errors and 0 warnings.
```

12. Add Lookahead functions where JavaCC states in the command line as seen below.

```
Warning: Choice conflict involving two expansions at
         line 418, column 18 and line 420, column 17 respectively.
         A common prefix is: "(" "("
         Consider using a lookahead of 3 or more for earlier expansion.
```

13. Remove Lookahead functions where possible by disambiguation of the grammar.

Testing files

14. Test the compiler for the simple non-empty file made of main function
15. Test for comments and nested comments
16. Test the compiler using functions
17. Test the compiler for scope.
18. Test the compiler for using functions, operators and variables.

4. Parser File

The main ‘MyParser.jj’ file is partitioned into five sections, four of which are important to the examiner. The first section is made up of metadata and displays details about the author of the program.

4.1 Options

The options section is essentially an optional list of options which can enhance performance or user defined functionalities which change how the compiler works. Options may also be specified in the regular expressions and context-free productions as well as on the command line following javacc.

I have two such options enabled. The first is the debug parser which prints the flow of the compiler to the command line. This is especially helpful for finding out where there are problems in the set of productions whereby an exception occurs.

The second option is output directory which is used by the programmer to tell the compiler where to place all of the generated Java files. I have it currently selected to the same directory as the compiler but had been previously been testing with the command in a subdirectory. I kept this command in to show that I had used it but it can be removed as well.

```
options
{
    DEBUG_PARSER = true;
    OUTPUT_DIRECTORY = ".";
}
```

4.2 Parser Code Block

The next section is the Java compilation unit which is enclosed in a parser code block surrounded by two statements, 'PARSER_BEGIN' and 'PARSER_END'. Two parentheses follow these statements which must match the name of the generated parser. This file can be left empty but in the case of the CCAL grammar which I have written, I have composed a file I/O reader which takes in a filetype preferably of '.ccl'. When JavaCC compiles this '.jj' file, this block of code becomes the main parser function as a '.java' file.

```
public class MyCCParser
{
    public static void main(String [] args)
    {
        MyCCParser parser;

        if(args.length == 0)
        {
            parser = new MyCCParser(System.in);
        }
        else if (args.length == 1)
        {
            try
            {
                parser = new MyCCParser(new java.io.FileInputStream(args[0]));
            }
            catch (java.io.FileNotFoundException e)
            {
                System.out.println("Parser: File not found");
                return;
            }
        }
        else
        {
            return;
        }
        try
        {
            parser.Program();
            System.out.println("PARSED SUCCESSFULLY!");
        }
        catch (ParseException error)
        {
            System.out.println("Encountered errors during parse: ");
            System.out.println(error.getMessage());
        }

    }
}

PARSER_END(MyCCParser)
```

When the parser reads in a file or a set of arguments from the command line, it checks their trueness to the grammar of the CCAL language. It runs the following command to start the first production which then runs through the file and outputs either a successful or unsuccessful parse.

```
parser.Program();
```

When compiling a '.jj' file, JavaCC will not perform detailed error-checking on this compilation unit making it likely that there could be bugs to be found in the generated java files which are harder to find. This code block can also use import statements, but I have not seen the use for any in this grammar.

4.3 Lexical Manager

The Lexical Manager section defines the tokens of the languages. This is divided up into 8 sections, each of which describe a token type. The Lexical Manager operates by order of precedence so I have written the lexical analyser so that important items like reserved words and comments are matched to their specific sections to halt the token manager from listing a reserved word as a variable or placing commented statements as tokenized string literals.

Token Manager Declarations

The first items are the token manager declarations which are variables written in Java and inside of the 'TOKEN_MGR_DECLS' like so:

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

These declarations are used by the token manager and are usable within lexical productions. There can only be one token manager declaration in a JavaCC grammar file. I have on item in this block which is used a counter for filtering out nested comment structures.

Control Characters and Spaces

In order for the token manager to gather tokens that are useful, we need to remove any items that may hinder this process. In order to essentially clean the code, we must use the SKIP production type and list any unwanted whitespaces, newlines and tabs so that the token manager will know not to add these to its list during file parsing.

```
SKIP:
{
      " "
    | "\t"
    | "\n"
    | "\r"
    | "\f"
}
```

Commented Statements

We need to eliminate comments from being read in as string identifiers. Luckily because comments are identified by using '//' or '/*-*/' in CCAL, we can filter any such item by finding these tags first and then removing any item following them or in between two. The main problem is having to eliminate nested comments. In order to do this, I created a variable to count the depth of a set of nested comments so that if a '/*' is encountered then the variable will increase and decrease for '*/'. When the counter is at zero the parser will know that the comment has been processed and can then ignore it.

```
/** Comments, Single, Multi or Nested */

SKIP:
{
      "/*" { nestedCounter++; }:COMMENT_NEST
    |
      "//":SINGLE_COMMENT
}

<COMMENT_NEST>
SKIP:
{
      "/*"
      {
          nestedCounter++;
      }
    |
      "*/"
      {
          nestedCounter--;
          if (nestedCounter == 0)
              SwitchTo(DEFAULT);
      }
    |
      <~[]>
}

<SINGLE_COMMENT>
SKIP:
{
      <"\n" | "\r" | "\r\n" >:DEFAULT
    |
      <~[]>
}
```

Reserved Words

Once all of the commented items that may appear are removed and control characters are eliminated, the set of reserved words can be added to the file which will match to an identifier and can be used in the context-free grammar productions. Tokens are generally of the format:

< IDENTIFIER: ”keyword” >

```
/** Reserved Words */

TOKEN [IGNORE_CASE]:
{
      < VAR: "var" >
    | < CONST: "const" >
    | < RETURN: "return" >
    | < INTEGER: "integer" >
    | < BOOLEAN: "boolean" >
    | < VOID: "void" >
    | < MAIN: "main" >
    | < IF: "if" >
    | < ELSE: "else" >
    | < TRUE: "true" >
    | < FALSE: "false" >
    | < WHILE: "while" >
    | < NOP: "skip" >
}
```

I have used a localized version of the JavaCC option ‘IGNORE_CASE’ which allows for reserved words to be case-insensitive. I could have written this in the options section but this option affects performance, so it is better to be used judiciously as a localized type.

Literals

```
TOKEN:
{
      < NUMBER:   ("0" |("-")? ["1"-"9"] (["0"-"9"])*) >
    |
      < IDENTIFIER: <LETTER>(<LETTER> | <NUMBER> | "_")* >
    |
      <#LETTER: ["a"-"z"] | ["A"-"Z"]>
}
```

This section defines the nomenclature of variable items in the grammar. The grammar rules of CCAL state that a number cannot start with a ‘0’ but can be negative and that string identifiers can have underscores but must start with a letter. Therefore, regular expressions have been written to apply these specifications.

Separators, Arithmetic/Boolean logic and Unrecognized Tokens

Separators define structure between objects and functions. Arithmetic and Boolean logic describe the functional aspect of a program to be used for manipulating variable items. Unrecognized tokens such as non-listed characters like '#' or '$' and also Unicode characters must be caught and removed from the grammar.

Note that '< ~[]>' means to ignore the entire set of characters whereby tilde is used as the identifier and the empty square brackets identifies the entire set of Unicode characters on a computer. Given that it is at lowest precedence in the lexical manager, it will catch any items that have so far been unrecognized by the token manager.

```
/** Separators */

TOKEN:
{
      < COMMA: ",">
    | < SEMICOLON: ";" >
    | < COLON: ":" >
    | < ASSIGNMENT: "=" >
    | < LBRACE: "{" >
    | < RBRACE: "}" >
    | < LPAREN: "(">
    | < RPAREN: ")">
}

/** Arithmetic and Boolean Logic */

TOKEN:
{
      < ADD: "+">
    | < SUB: "-" >
    | < NEG: "~" >
    | < BOOL_OR: "||">
    | < BOOL_AND: "&&" >
    | < BOOL_EQUALS: "==" >
    | < BOOL_NOT_EQUALS: "!=" >
    | < BOOL_LESS: "<" >
    | < BOOL_LESS_EQUALS: "<=" >
    | < BOOL_GREATER: ">" >
    | < BOOL_GREATER_EQUALS: ">=" >
}

/** Set of All Unrecognised Tokens */

TOKEN :
{
    < UNICODE_CHAR : ~[] >
}
```

4.4 Grammar Productions

This section describes the set of Context-Free grammar productions in Backus-Naur Form. These are essential in describing the syntax of the grammar. There are 23 such productions which extend from the definition set of 20 expansions. The productions in the CCAL Definition file are written in a type that must be translated into a JavaCC expansion unit. See the example below:

⟨function⟩ ⊨ ⟨type⟩ identifier (⟨parameter_list⟩)
{
⟨decl_list⟩
⟨statement_bock⟩
return (⟨expression⟩ | ϵ) ;
}

```
void Function():
{}
{
    Type() <IDENTIFIER> <LPAREN> Parameter_List() <RPAREN>
    <LBRACE>
    Decl_List()
    Statement_Block()
    <RETURN> <LPAREN>
    (
        Expression()
    |
        {}
    )
    <RPAREN> <SEMICOLON>
    <RBRACE>
}
```

Any items enclosed with '<>' are nonterminals and there should always be at most, one of these on the left-hand side of the of the grammar production. As can be seen above, nonterminals should be translated to JavaCC code by suffixing brackets and colon to the end of the production name. An empty set of chain brackets allow for manipulation within this function by writing Java code to describe some item within the block.

Items in bold define tokens used by the lexical manager and should be referenced based on their token identification in the former section. For example, 'return' is a reserved keyword and in order to be written into the production, it should be referenced by its keyword identification '<RETURN>'.

Non-terminals on the right-hand side of the production should have their own production description at some point in the grammar. Any non-terminals that are not provided productions are essentially ambiguous which makes for poor grammar rules. All productions should follow through until there are only terminals at the end of the syntactic tree structure.

When a program is compiled, it checks to make sure that there is no left recursive functions. Left recursive functions are non-ending solutions where the recursive feature repeats infinitely. JavaCC will not allow left recursion, so I created some new expansions in order to avoid this.
For one such example the conditional operator, the parser would check if there was a second operator following a boolean statement. This would call the same expansion unit and in order to work around this, I removed that second statement and replaced it with two productions for boolean or/boolean and followed by a nonterminal as can be seen on the right.

```
void Condition():
{}
{
        (<NEG> Condition() Condition_Prime())
    |
        LOOKAHEAD(3) (<LPAREN> Condition() <RPAREN> Condition_Prime())
    |
        (Expression() Comp_Op() Expression() Condition_Prime())
}

// 16B.

void Condition_Prime():
{}
{
        Conditional_Or_Expression()
    |
        Conditional_And_Expression()
    |
        {}
}

// 16C.

void Conditional_Or_Expression():
{}
{
    <BOOL_OR> Condition()
}

// 16D.

void Conditional_And_Expression():
{}
{
    <BOOL_AND> Condition()
}
```

There were lookahead warnings that occurred once I had removed left recursion from my productions. I placed three lookaheads at choice points in the grammar where the compiler is unsure and managed to remove another two lookaheads by shortening the production by removing the choice point, instead opting for square brackets which mean that the contents inside are optional and these are checked when the parser runs through a piece of code which activates this production.

```
void Nemp_Arg_List():
{}
{
    <IDENTIFIER> [<COMMA> Nemp_Arg_List()]
}
```

Some productions had choice points that either included a choice points between a non-terminal or were empty and could be either defined square brackets or placed as a choice point with '{}', which is the lambda production or empty string.

JavaCC creates recursive-descent parsers which must look at symbols to decide which grammar production to use. By default, it looks ahead at one symbol which is why these are called LL(1) parsers. At choice points however, the compiler does not know which branch to follow and may provide a lookahead warning with a value of a certain type to issue a production. I have one such production which requires a Lookahead value of 3.

```
void Expression():
{}
{
        LOOKAHEAD(3) (Fragment() Binary_Arith_Op())
    |
        (<LPAREN> Expression() <RPAREN> Binary_Arith_Op())
    |
        (Fragment() <ASSIGNMENT> Fragment() <SEMICOLON> [Expression()])
}
```

This production requires a lookahead of 3 in that it reads into the 'Fragment()' nonterminal and then does the same for the following nonterminals in order to make a decision of which path it will choose.

5. Conclusions

I found this assignment and it required an extensive look into JavaCC. Luckily there is plenty of documentation both online and in books. I used two books as well as the JavaCC documentation which provides examples and tutorials on describing the power of JavaCC, Lookaheads and the token manager. I found the grammar written on the Java language which is in the JavaCC folder to be of great help in writing my own parser.

I also found the supplementary compiler provided on the Compiler Construction website to be very beneficial as a reference of what kind of tasks must be done and how the layout should. This was the JJTree/LLVM compiler.

My parser has worked for all the test cases and when provided with debug parser options, I can view the full extent of the process. I hope that it may pass test cases and also fail on items that should not throw up an error when parsing a '.ccl' file or similar document.

6. References

https://javacc.org/doc

https://www.computing.dcu.ie/~davids/courses/CA4003/jjtree_and_llvm.html

Modern Compiler Implementation in Java
Appel, Andrew W.

Compiler Construction Using Java, JavaCC and Yacc
Dos Reis, Anthony.

https://www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf