CA4003 - Compiler Construction Assignment 1

*“A Lexical and Syntax Analyser in JavaCC”*

* Student Name: Ross Franey
* Student Number: 14302851
* Submission Date: 13/11/2017

**Contents**

* Introduction
* Lexical Analysis
  + Design
  + Implementation
  + Testing
* Syntax Analysis
  + Design
  + Implementation
  + Testing
* External Sources

**Introduction**

The **Aim** of this project was to create a **lexical and syntax analyzer** using JavaCC, which would accept the CCAL language provided in the assignment specification. This document aims to outline how I approached this assignment in terms of the design, implementation and testing/evaluation of the solution.

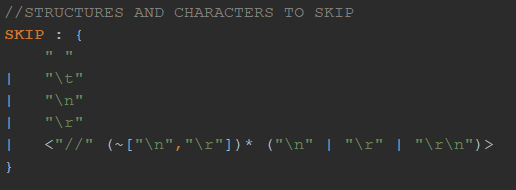
**Lexical Analysis**

**Design:**

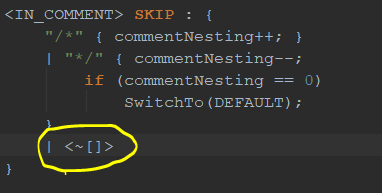
The initial step involved in the compilation of this program was the Lexical Analyzer, or "Tokenizer". The goal in Lexical analysis is to take a stream of characters and convert them into a stream of tokens such as names, keywords, etc. which the program can later use to form valid expressions.

**Implementation:**

1. Skip: In terms of implementing this functionality in JavaCC, I first defined what tokens should be skipped by the parser. Essentially, this means characters that should be ignored while scanning for valid tokens, and includes a white space, new line (\n), carriage return (\r), tab character(\t), and finally an inline comment “//”.



As well as this, a separate definition for nested comments was required. To achieve this, the program searches for a “/\*” symbol, and increments the commentNesting variable. This variable is decremented upon the reading of a “\*/” token. The program then checks to ensure that the variable is 0, and if it is, returns form the IN\_COMMENT state to the DEFAULT state. The symbol following this code highlighted below simply means everything else that has not been matched.

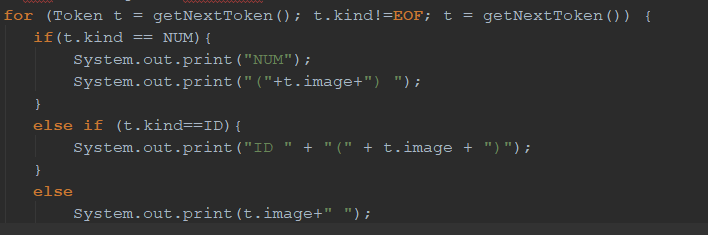


1. Tokens: The next step was to define the “punctuation” tokens as it were. This includes tokens such as comma, semicolon, brackets, Boolean operators, etc.
2. Finally, I created a set of static tokens for Numbers, Strings and Digits and IDs, defining what each would be.

Noteworthy here, in order to ensure that no invalid Number was read in (i.e., starting with 0 or – 0), which was a specification of the assignment, code was written to ensure a Number must start with any digit (0-9) excluding 0, unless it is the number 0 itself.

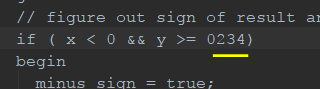
**Testing**:

To test the Lexical analysis, I wrote code for the tokenizer (which has been commented out in the submission file), which reads in each token, printing its kind and image.

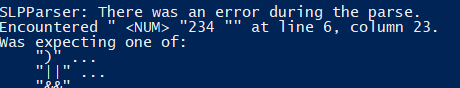


As well as this, in this section, I wanted to test the two comment scenarios (nested and inline), and also ensure that the specification of 0 being an accepted number but not a leading digit in a larger number, was fulfilled.

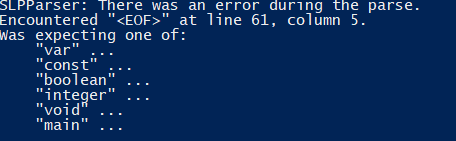
This was achieved by replacing a 0 in the final provided test code in the assignment document, shown below, with “0234”. This also comes directly after a comment and as such is testing both features.



The following error was generated, ensuring that the number constraint was in place.



For the nested comment, I simply nested the entire sample program in between /\* \*/ and received the following:



Meaning the scanner skipped the entire code and read End of File (<EOF>) token.

**Syntax Analysis**

**Design**:

The next aspect of the program is syntax analysis. This involves taking a stream of tokens acquired from the previous lexical analysis, and forming with them a valid "sentence". The rules by which this will be carried out are outlined in a grammar, which can be represented as parse trees.

**Implementation:**

1. Firstly, the reserved words of the language were outlined. This includes things like IF, ELSE, END, etc. As well as a SKIP\_TOKEN. This was included as the term SKIP is already a reserved word in JavaCC and could not be overwritten. These words would be used to build the production rules to follow.
2. I then defined the main unit of my program, the production rules themselves. These are the rules which should be followed in order to generate valid sentences.

The start method is first reached, leads to a program followed by an end of file symbol.

From here, the defined grammar is a translation of that specified in the notes, where Epsilon is denoted by a “?” symbol, meaning 0 or 1 occurrences.

In this section, I have used left factoring to remove expansion conflicts where they arose. This was solved using the course notes, following the principals given and applying them to code rather than alpha-beta symbols. Where factoring occurred, and the empty set was not a prefix of the RHS of the production rule, a new non-terminal was created. These additional non-terminals are denoted in the code with “prime” included their name.

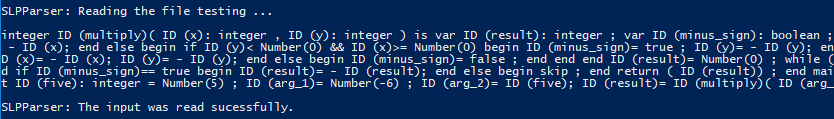
Left recursion was eliminated by generalizing and making use of bracket tokens.

This allowed for the program to run all tests correctly without using a lookahead, meaning the grammar is LL (1).

**Testing:**

Although it is difficult to test this program to the extent of which I would be comfortable typically, the program accepts all test inputs provided for the assignment, and I have modified these examples to contain known errors, all of which my program has detected. I am aware that this is not anywhere close to full coverage, but I believe I have tested the program to the best of my ability considering the scope of the assignment.

An example running the final test given in the assignment specification returned the following result (when my previous test code was uncommented):



This is an example of both the lexical analysis from above, combined with the syntax analysis, which tells me that the input file was successfully read. The input file used for this demonstration can be found at the end of the assignment specification document.

**External Sources**

The following is a list of resources utilized throughout the development of this project which taught some of the subtler features of JavaCC (such as the debugging option, for example), as well as provided interesting and insightful reading material, which lead to a more thorough understanding of the way javaCC, and indeed compilers in general, work in practice/code.

<https://javacc.org>

<http://www.engr.mun.ca/~theo/JavaCC-FAQ/javacc-faq-moz.htm#tth_chAp2>

<https://www.javaworld.com>