Presentation of Compiler Design

Presented by:

Genevieve Plante-Brisebois

40003112

Section 1: Design

**Lexer**:

For the design of the lexer, the lexical elements were used as is to do the tokenization:

**id::** = Letter alphanum\*

alphanum:: = letter | digit| \_

**integer::=** nonzero digit\* |0

**float::=** integer fraction [e[+|-] integer]

fraction::= .digit\* nonzero | .0

letter::= a.. z |A..Z

digit: 0+1..9

nonzero::=1..9

operator::= == | <> | < |> | <= | >= | + | - | \* | / | = | && | || | !

punctuation :: = ; | , | . | : | :: | ( | ) | { | } | [ | ]

keywords::= if | then | else | for | class | integer | float | read | write |return | main

comments::=/\*|\*/ |//

Using the DFA that is the result of this lead to the following possible token:

* id
* integer
* float
* keyword
* punctuation
* comment
* operator

The structure of the lexer made use of libraries to make regular expressions for the most basic types of tokens and later the tokens validation was made with a series of Boolean expressions representing their regular expression and token.

As for the parsing of the document which leads to the tokenization, the maximum number of increments before it is possible to get a valid token has been used as a benchmark and if by that time the tokens were still not valid the lexer would back track and do the tokenization from that point on.

The final lexer method returns an arraylist of tokens. The tokens are a structure built for the purpose of the program. It includes the token type, the data(content of the token) and the location of the token.

The error recovery system is that when it finds an erroneous token, there is an error message sent to the console and to a file for the tokenization errors. It includes what it is (invalid token) what the error is and the location of the error.

There is an output to files of AtoCC format, token stream, errors. The input file to read from is flexible.

**Parser:**

The first step of the parser was to make the grammar in the LL1 format. That has been done with the help of the kfg edit and an online tool for small LL1 conversions.

For the parser, I have chosen recursive decent model as the rules of the grammar were straight forward.

However it has brought the issue that with the current structure it needs to be a perfect program to be able to get the AST at the end of the line.

There are flaws in some of the methods. There is often an issue with the factors, however I have not been able to get to the bottom of the issue.

For the AST I build a Node list structure. Each node has its parent, array list of children node, the token (id need be) and a production. Should the node be a leaf its production will be written as “terminal” and there will be the token data in the node. Error handling has yet to be efficient. As it stands if the program is not “perfect” it will not go through.

**Expr**

**Factor**

have been noted as with issues from debugging with sample programs. The structure of the functions

for the parser is as follow:

Root or parent node as a parameter

* initialize a temp as the same of the variable pointer(which points to the current token we are looking at) . This allows us to back track in the case that there is an error and we stop return a Boolean false result.
* A marker called child\_marker so that if we need to take away some nodes that we added while processing the function, we do not take away all the children nodes but only the onces that were added locally.
* If else statements to reproduce the decisions that follow the grammar. If a function, let us say the program, needs to check on the class declarations, it will call the class declaration function to do so.

**Semantic analyzer:**

**Section 2: Tools used in the project**

Word for documentation

Version control : github <https://github.com/GenevievePlanteBrisebois/COMP442-CompilerDesign>

Eclipse as IDE

Java as language

Multiple java libraries

KFG edit for LL1 conversion

LL1 converter for individual LL1 conversions