CA4003 Compiler Construction – Assignment 1

A Lexical and Syntax Analyser

Declaration on Plagiarism

Assignment Submission Form

Name(s):	Paul McNally
Programme:	CASE 4
Module Code:	CA4003
Assignment Title:	Assignment 1: A Lexical and Syntax Analyser
Submission Date:	15/11/2020
Module Coordinator:	David Sinclair

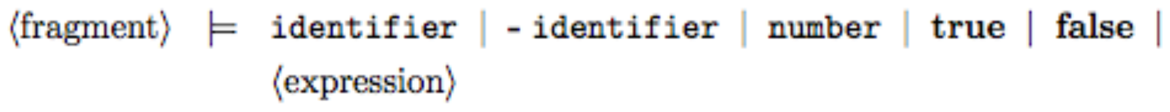
Name(s): Paul McNally Date: 15/11/2020

Grammar file(cal.g4):

As all grammars must always start with a grammar header, this grammar is called
cal as it must match the filename: cal.g4.

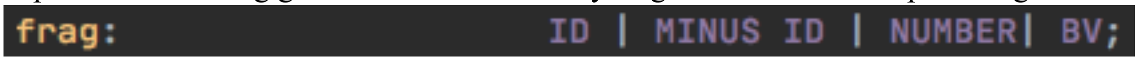
The definitions all follow the assignment language definition given with this assignment. For the epsilons, I used the or operator "|" followed by a semi-colon as ";" as epsilon means empty, this method was sufficient.

In the language definition provided the fragment definition looks like this:



⟨fragment⟩ ⊨ **identifier** | **- identifier** | **number** | **true** | **false** |
⟨expression⟩

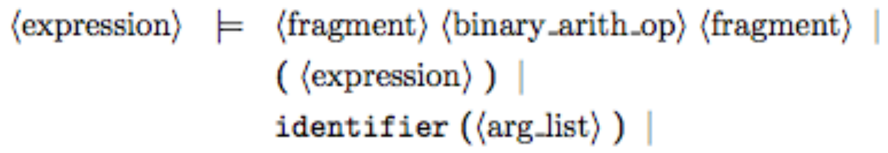
However, when I did this, I ran into a number of problems, the first was that I was unable to call the definition "fragment" as "fragment is used to prefix rules in antlr4 e.g. the fragments for uppercase and lowercase letters of the alphabet, so I instead just called it "frag". The next problem I ran into also was when I include <expression> in the frag definition I received this error, "The following sets of rules are mutually left-recursive [expression, frag]". Removing expression from frag got rid of this error so my frag definition ended up looking like this:



```
frag:                         ID | MINUS ID | NUMBER| BV;
```

("BV" represents the Boolean values true or false.)

Next, for the expression rule:

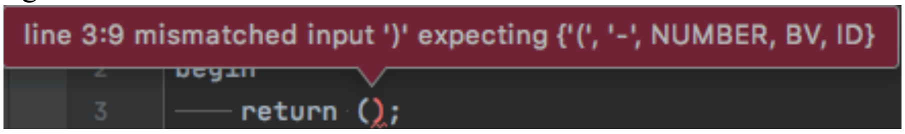


⟨expression⟩ ⊨ ⟨fragment⟩ ⟨binary_arith_op⟩ ⟨fragment⟩ |
(⟨expression⟩) |
identifier (⟨arg_list⟩) |

When I wrote the rule like this in the cal.g4 and attempted to parse the 5th test file:

```
void func () is
begin
   return ();
end
main
begin
   func ();
end
```

I got this error:

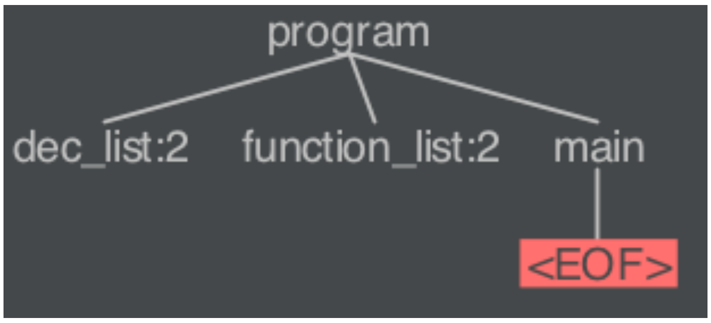


```
line 3:9 mismatched input ')' expecting {'(', '-', NUMBER, BV, ID}
3    return ();
```

To eliminate this error, I added "?" to expression in the function rule, this meant that having an expression was optional and it could be empty (expression | epsilon), so my function rule is now this:

```
function:                   type ID LBR parameter_list RBR IS dec_list BEGIN
statement_block RETURN LBR expression? RBR SEMI END;
```

The final change I made was to the main, initially if I tried to parse an empty file the parse tree would have an end of file error "<EOF>".



In order to get rid of this error, I added an asterisk to around the main rule, the * symbol means main rule can be implemented zero or more times.
Main rule:

```
main:                        (MAIN BEGIN dec_list statement_block END)*;
```

As previously said, there are fragments for all the letters of the alphabet as the language is case insensitive meaning words such as BegIN would match the same as begin.
Example:

```
fragment A :                 [aA];
```

Also added fragment rules for Letter, Digit, Underscore:

```
fragment Letter:           [a-zA-Z];
fragment Digit:              [0-9];
fragment Underscore:       '_';
```

Next in the grammar file I have all the reserved words, it is important to have all the reserved words, if/else, Boolean values in the grammar file before the Identifier is defined in the file, if this does not happen, the parser will pick up the reserved words as an ID. I ran into this problem as I had VOID written into the grammar file after the Identifier rule, so in the parse tree it was reading void as an ID which caused an error when parsing this particular file. I also had to call skip, SKIP_STATEMENT instead of SKIP as skip is a reserved.

After the reserved words, I defined all tokens, all operators and a number. And finally, I added to the grammar file for the parser to recognise and skip whitespace and to recognize comments, multi-line comments and nested comments.

Java files:

Cal.java

When the “cal.java” file is run with the input file as an argument, the program checks to see if the arguments are greater than 0 and if so args [0] is the input file, if a file name isn’t specified it will be from the system input.

The Lexer, “calLexer” is created and this lexer gets its input from a character stream, the character stream which is attached to the input file. The lexer is used to generate a list of tokens. The list of tokens is then thrown into the parser (calParser).

If the file has parsed successfully, there will be an output message saying the file has parsed successfully e.g. file.cal parsed successfully

ErrorListen.java

If there is an error in the input file an instance of the ErrorListen class is called.

```
parser.removeErrorListeners();
ErrorListen errors = new ErrorListen(f.getName());
parser.addErrorListener(errors);
```

There will then be an output message saying the file has not parsed successfully e.g. file.cal has not parsed. The program is then terminated.

```
System.out.println(file + " has not parsed");
System.exit(1);
```