CA4003 Compiler Construction

A Lexical and Syntax Analyser for the basicL Language

Eoin Murphy 11487358

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.

Lexical Analysis

Handling Comments.

Firstly I began by declaring the comment tokens which are to be the ignored by the tokeniser:

```
SKIP : /* COMMENTS */
{
    <"--" (["a" - "z"] | ["A" - "Z"] | ["0" - "9"] | " ")* ("\n" | "\r" |"\r\n")>
  | "/*" { commentNesting++; } : IN_COMMENT

}
```

The first Form of comments is defined by one which begins with “--” and can follow zero or more Chars, Digit's or spaces and ends with the newline character.

The second form of comments begin with “/*” and end in “*/” the tokeniser ignores all tokens inside of this block. We check to see if we have reached the end of the statement block by checking if we consumed a “/*” then we must have consumed a even number of “*/ this is done by declaring the IN_COMMENT token:

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

This token checks if we have read in a “/*” to increase our internal counter commentNesting and if we consume a “*/” to decrement the counter and once this counter is 0 so we have consumed a even number of “/*” and “*/” then we have reached the end of the comment and to return other wise we just skip all other tokens.

Ignoring Chars.

Some characters we don't want to read in as tokens these being spaces, tabs,and newlines to ignore these again as we did with comments we just place them inside a skip block:

```
SKIP : /*** Ignoring spaces/tabs/newlines ***/
{
    " "
  | "\t"
  | "\n"
  | "\r"
  | "\f"
}
```

Keywords, Operators & Relations.

Declaring keywords, operators and relations was pretty straight forward to do this it was simply token blocks with the name of the token followed by the string that matches to it. To just make it easier to read I wrote two token blocks one for keywords and another for Operators and relations as follows:

```
TOKEN : /* Keywords */
{
    <AND : "and">
  | <BOOL : "bool">
  | <CONST : "const">
  | <DO : "do">
  | <ELSE : "else">
  | <FALSE : "false">
  | <IF : "if">
  | <INT : "int">
  | <MAIN : "main">
  | <NOT : "not">
  | <OR : "or">
  | <RETURN : "return">
  | <THEN : "then">
  | <TRUE : "true">
  | <VAR : "var">
  | <VOID : "void">
  | <WHILE : "while">
  | <BEGIN : "begin">
  | <END : "end">
}
```

```
TOKEN : /* Operators and Relations */
{
    <PLUS_SIGN : "+">
  | <MINUS_SIGN : "-">
  | <MULT_SIGN : "*">
  | <DIV_SIGN : "/">
  | <EQUALS_SIGN : "=">
  | <NOT_EQUALS_SIGN : "!=">
  | <LESS_THAN : "<">
  | <GREATER_THAN : ">">
  | <LESS_THAT_EQUALS : "<=">
  | <GREATER_THAN_EQUALS : ">=">
  | <LEFT_BRACKET : "(">
  | <RIGHT_BRACKET : ")">
  | <COMMA : ",">
  | <SEMICOLON : ";">
  | <TYPE_ASSIGN : ":">
  | <ASSIGN : ":=">
}
```

Identifiers & Numbers.

Identifiers were defined as "*Any other string of letters, digits or underscore character ('_') beginning with a letter.*". I declared Identifiers as follows:

```
TOKEN : /* Identifiers */
{
    <ID : <CHAR> (<DIGIT> | <CHAR> | "_" )* >
  | <#CHAR : ["a" - "z"] | ["A" - "Z"]>
}
```

What this says is a Identifier is a Char followed by zero or more Digits, Chars or underscores as defined by the kleene closure "*" to make it easier I declared a internal token CHAR to define characters and also a DIGIT token to declare digits (which is declared in the Token block for numbers). This token declaration ensures that Identifies must begin with a Char and not a underscore or digit.

Numbers are defined as "*A string of (decimal) digits. Integer numbers are represented by a string of digits* " I defined these as follows:

```
TOKEN : /* Numbers */
{
    <NUM : (<DIGIT>)+>
  | <REAL: ( (<DIGIT>)+ "." (<DIGIT>)*) | ((<DIGIT>)* "." (<DIGIT>)+) >
  | <#DIGIT : ["0" - "9"]>
}
```

This Declaration says that numbers are one or more digits as defined by the internal token DIGITS. We then allow for decimal numbers with the token REAL which says that a can be either a One or more digits followed by a decimal point followed zero or more digits (this can results in real values of "1.11", "2123." etc.) or else it can be defined as zero or more digits followed by a decimal point followed by one or more digits (this allows for real values of ".2", ".222" , "22.2).

Not recognised.

Any other tokens that is not recognised I just put in its own token OTHER so I can handle it appropriately

Running the lexical analysis.

To run the tokeniser for the basicL language you simply run the interpreter with the -L option as follows:

java BASICLInterpreter inputfile -L

This will output all tokens recognised along with identifiers and numbers with their values and those not recognised. The output Tokeniser for the sum_primes.bl file is as follows:

```
C:\Windows\system32\cmd.exe
C:\Users\Eoin\Documents\GitHub\Complier_Construction_CA4003\assignment_one>java
BASICLInterpreter sum_primes.bl -L
BASICL Interpreter:  Reading from file sum_primes.bl...
-----------------------------------------
-------------Lexical Analysis-------------
-----------------------------------------
var ID(result) : int ; const ID(N) : int = NUM(100) ; bool ID(is_prime) ( ID(x)
: int ) begin var ID(i) : int , ID(res) : bool ; if ID(x) = NUM(1) or ID(x) = NU
M(2) then ID(res) := true ; else begin ID(i) := NUM(2) ; ID(res) := true ; while
 ( ID(i) * ID(i) < ID(x) ) and ID(res) = true do begin if ID(x) - ( ID(x) / ID(i
) ) * ID(i) = NUM(0) then begin ID(res) := false ; ID(i) := ID(i) + NUM(1) ; end
 ; else ID(i) := ID(i) + NUM(1) ; end ; end ; return ID(res) ; end main begin va
r ID(i) : int , ID(sum) : int ; ID(sum) := NUM(0) ; ID(i) := NUM(2) ; while ID(i
) <= ID(N) do begin if ID(is_prime) ( ID(i) ) then ID(sum) := ID(sum) + NUM(1) ;
 else begin end ; end ; ID(result) := ID(sum) ; end
C:\Users\Eoin\Documents\GitHub\Complier_Construction_CA4003\assignment_one>
```

Syntax Analyser

Defining the grammar.

When defining the grammar for the BasicL language I had to make some slight modifications to the grammar along with some major changes to account for left recursion and to keep it as a LL(1) Grammar.

One of the first changes I had to make was to the Decl function which had a kleene star surrounding it. This kleene star meant that in other functions where it was called with a kleene star a error would occur as a result of it being able to parse to a empty string for example in the Program function:

```
void Program() : {}
{
  ( Decl() )* // fix due to it decl
  ( Function() )*
  Main_Prog()
}
```

To fix this I just removed the kleene star in Decl so Decl now became:

```
void Decl() : {}
{
  ( Var_decl() | Const_decl() )
}
```

In the original grammar there was left recursion present in Expression and Condition:

To remove the left recursion in expression I performed left factoring on it. Expression1 is the prime of Expression. along with this change I edited Fragment to Remove the original call back to fragment as this call did not seem needed and was a source of error I also allowed for expression to be housed in Brackets the resulting change is as follows:

```
void Expression() : {}
{
    <ID> [ Arg_list() ] Expression1() | Fragment() Expression1()
}

void Expression1() : {}
{
  Opp() Expression() | {}
}

void Opp() : {}
{
  <PLUS_SIGN> | <MINUS_SIGN> | <MULT_SIGN> | <DIV_SIGN>
}

void Fragment() : {}
{

    <TRUE>
  | <FALSE>
  | <NUM>
  | <REAL>
  | <LEFT_BRACKET> Expression() <RIGHT_BRACKET>
}

void BoolOpp() : {}
{
    <EQUALS_SIGN>
  | <NOT_EQUALS_SIGN>
  | <LESS_THAN>
  | <GREATER_THAN>
  | <GREATER_THAN_EQUALS>
  | <LESS_THAT_EQUALS>
  | <AND>
  | <OR>
}
```

Removing the Left recursion in Condition was a lot more challenging this left recursion was a result of the statement Condition(<and> | <OR>) Condition:

The fix I implemented to this was to have condition be a expression followed by zero or more bool operators which include the AND and OR tokens. The reason for using a kleene star rather then a alteration was because of the fact a condition can be a function call as described bellow.

I tried first to have the function call declared in Condition but this resulted in a choice conflict due to it being matched in Expression also. This looked as follows:

```
void Condition() : {}
{
    <NOT> Condition()
  | [<LEFT_BRACKET>] Expression() ( ( BoolOpp() ) Expression()  [<RIGHT_BRACKET>] ) +
  | Arg_list()
}
```

I realised that if I used a kleene closure it would allow it to be matched with just an expression which can be a function call thus allowing condition to be a function call.

```
void Condition() : {}
{
    <NOT> Condition()
  | [<LEFT_BRACKET>] Expression() ( ( BoolOpp() ) Expression()  [<RIGHT_BRACKET>] ) *
}
```

The only problem with this is that condition can parse the following syntax as correct:

“if 1 + 1”

I feel though that is acceptable as one of the things I will have to do later in the generation of this compiler is to check that a condition can evaluate to a boolean value. Because even if I did use the above with a alteration instead of kleene star the problem that a function could return a value other then a boolean is still present.

Also to allow for conditions to be held in Brackets I added the optional tokens left_bracket and Right_bracket.

Another slight change I had to make to the grammar was in Statement:

```
void Statment() : {}
{
    <ID> (<ASSIGN> Expression() | Arg_list() )
  | <BEGIN> ( Statment() <SEMICOLON> )* <END>
  | <IF> Condition() <THEN> Statment() <SEMICOLON> <ELSE> Statment()
  | <WHILE> Condition() <DO> Statment()
  | {} // this will allow you to just stick aload of semicolons in the file and will parse correctly !!!
}
```

I modified the first statement to a id followed by either a assign with an expression or a function as when the two were separate this resulted in a choice conflict which could be fixed easily with a LOOKAHEAD(2) in it but that would make it a LL(2) grammar which I did not want so moving them to this keeps it as a LL(1) Grammar by removing the need for a LOOKAHEAD.

As you can see in the comment above the I did identify a problem with the grammar in that it allows for some one to include multiple semicolons in the file one after another and still parse correctly this is due to the call to statement in Function and Main_Prog:

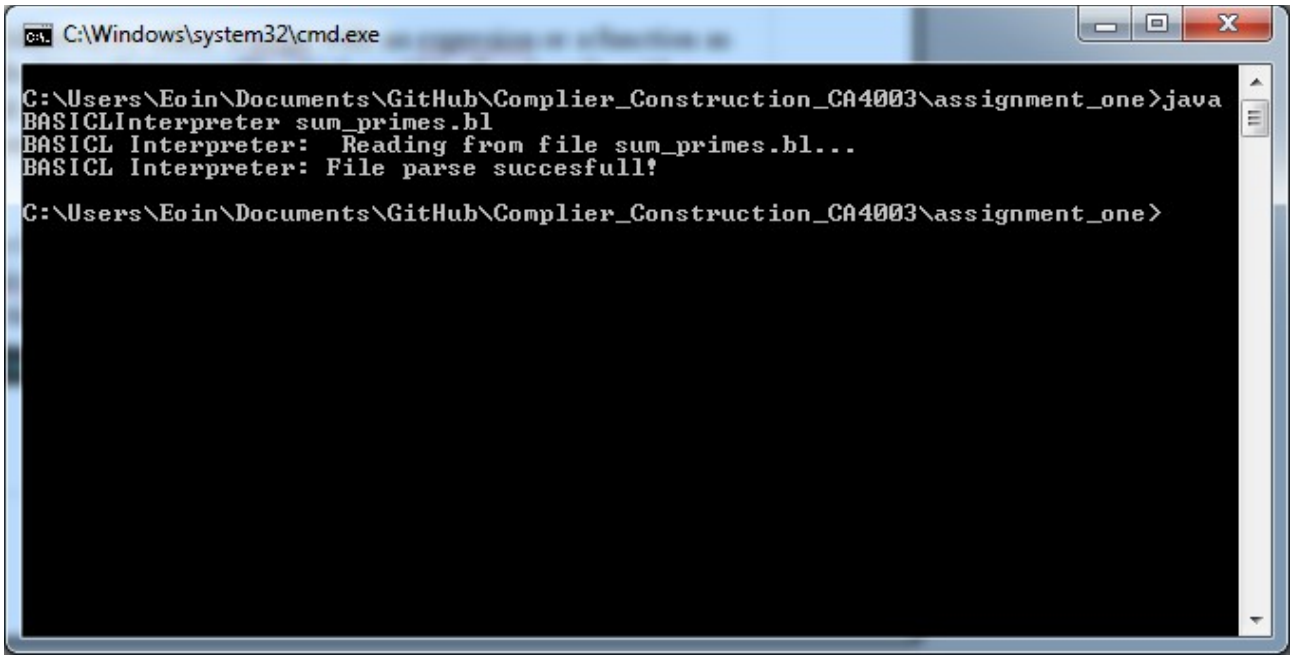
```
( Statment() <SEMICOLON>)*
```

Running the Syntax analyser.

To run the Syntax analyser you simply call the interpreter as follows:

java BASICLInterpreter inputfile

If the file is parsed correctly and the syntax is correct the output will look as follows:



If a error is detected it will point to the line that caused the error: