# CA4003 – Assignment 1:

# A Lexical and Syntax Analyser for the basicL Language

Aoife De Buitléar

12376926

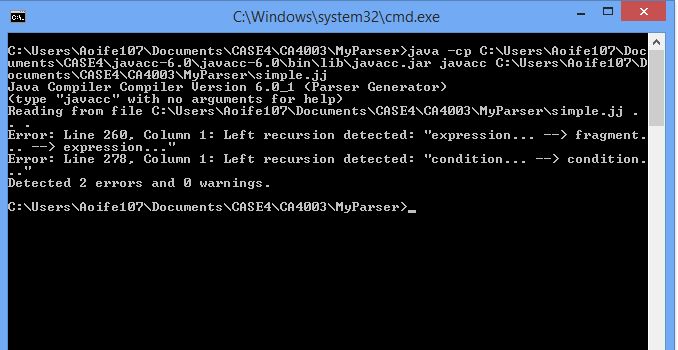
This Assignment is my own work. I have not received assistance beyond what is normal. I am aware that copying or giving a copy of work may have severe consequences.

The first thing I did when coding my parser was to declare are tokens and variables to be recognised by the parser. I based most of this directly from the class notes on Lexical Analysis.

After this, I created my java program to read in the .bl file for parsing. This is just a basic program that checks for the input file and calls the parser program from within my grammar.

I had not designed my grammar at this point. I wrote the grammar directly from the specification for Assignment 1 on the module website. I then compiled this using the “java –cp C:\Users\Me\Documents\javacc-6.0\bin\lib\javacc.jar javacc C:\Users\Me\MyParser\AoifesParser.jj” command. I knew the program would not compile and parse the .bl language straight away and that there would be more work involved but I wanted to visualise how I was to debug through the code.

The first time I compiled the basic program I received two errors which were due to left recursion.



I read through the notes in order to find a solution for this. I broke down the functions that were causing problems to try and avoid this error. I did this using a page in the notes for “Top-down Parsing” entitled “Eliminating Left-recursion”:

To remove left-recursion, we can transform the grammar.   
Consider the grammar fragment:  
 A ::= Aα | β where α and β do not start with A.   
We can rewrite this as: A ::= βA ′ A ′ ::= αA ′ | ǫ where A ′ is a new non-terminal   
This fragment contains no left-recursion.  
  
I applied this rule to my code and began to understand the logic behind each piece of the grammar and how to avoid left recursion occurring again.

The breakdown of my code can be seen in the AoifesParser.jj but here is an example used with the expression() function.

***Original function:***

void expression() : {}

{

fragment()(assignments() fragment())\*

|<LPAREN>expression()<RPAREN>

|ident\_list() <LPAREN> arg\_list() <RPAREN>

}

***Became:***

void expression() :

{}

{

fragmentCheck() expressionNew()

}

void fragmentCheck() :

{}

{

fragment() fragmentNew()

}

void fragment() :

{}

{

<IDENTIFIER>

| <TRUE>

| <FALSE>

| <NUMBER>

| (<ADD>|<SUB>)fragment()

|{}

/\*NOTES FOR DOC\*//\*change expression() to epsilon i.e. {} due to rule (see notes)\*/

}

void expressionNew() :

{}

{

<ADD> expression()

|<SUB> expression()

| <IDENTIFIER> <LPAREN> arg\_list() <RPAREN>

| {}

}

When expression() is called, it checks fragmentCheck() which in turn checks fragment(). Then fragmentcheck() goes to fragmentNew() and finally returns back to expression to check expressionNew().

I did this iteratively throughout my code, working through each function bit by bit and attempting to avoid left recursion. Upon running my program with the sum\_primes.bl file – ignoring any current warnings that were occurring upon compilation – I could see where my parser was getting stuck.

This really helped me with my debugging and I was able to effectively add and alter functions in order to parse through the .bl file correctly.

I created my own .bl file entitled simple\_largest.bl which I tested my program on. I also tested my program on another .bl file that a class mate created and allowed me to use.

/\*

This is a comment

\*/

--simple language

var result:int;

const MyConst:int = 10;

bool is\_largest(x:int) --largest is true if x is larger than integer

begin

var integer:int,largest:bool;

if x > integer

then

largest := true;

else

largest := false;

return largest;

end

main

begin

var i:int, num:int;

num := 5;

i:=1;

while i <= N do

begin

if is\_largest(i)

then

num := i;

else

begin

num := 5;

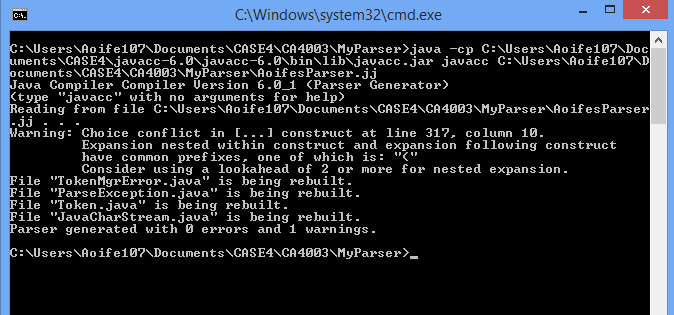
end;

end;

--integer := 5;

result := num;

end --end of my simple language

The way my parser is currently, it does fully parse .bl files. However, I receive a warning upon compilation of the .jj file.   
The warning is as follows:  


I did attempt to use LOOKAHEAD on my .jj file but to no prevail.

This warning is regarding my condition() function which I did have a lot of trouble with.

Through the same method as before, I divided up the original condition function to avoid left recursion issues that I was having.

void condition() :

{}

{

//This is where lookahead is being requested

<NOT> condition()

|[<LPAREN>] conditionNew() [<RPAREN>] cond()

}

void conditionNew() :

{}

{

expression() (assignments() expression()

| <LPAREN> expression() <RPAREN> conditionNew()

| expression())

}

/\*Created new function to hold assignments for ease of use\*/

void assignments() :

{}

{

<EQ> | <NEQ> | <LT> | <GT> | <LTEQ> | <GTEQ>

}

void cond() :

{}

{

<AND> condition()

|<OR> condition()

| {}

}

From running my parser with the sum\_primes file I realised that some conditional instances had not been catered for, for example, there could be brackets around a condition or expression and so I added |[<LPAREN>] conditionNew() [<RPAREN>] cond()

ConditionNew acts as A’ which is a modified version of what was originally in condition(). A condition can be an expression followed by either an assignment and another expression or an expression with brackets around it and then a following condition, or just an expression. These all had to be catered for as well as all of these having brackets around them as a whole.

I created an assignments function simply for ease of use, it enabled me to call the function without having all of the mess. It made the code much cleaner and easier to read in my opinion.

Finally I have the cond function which in simple the expected Operators at the end of the condition. I added in the epsilon as an option as sometimes there was no Operator at the end of a condition, particularly when it ended with an expression, e.g. if x = 1 or x = 2 then

IF from statements knows to go to condition function to see that x is an identifier, = is an assignment, 1 is a number – this must now be looped to check if there is a more to the IF statement and otherwise will continue to check for THEN. OR is an operator, x is an identifier and 2 is a number, the parser then comes across THEN and knows to loop through statement().

However, the parser does work.

I parsed the sum\_primes.bl with debugging enabled in order to see if my parser was recognising all tokens. I saved the result of this to a txt file which I have included in this .zip file as sum\_primes\_parsed.txt and another for my own file simple\_largest\_results.txt.

I plan to resolve the warning I am receiving before continuing with Part 2 of the assignment for fears that it may affect the parser later on.