# Notes on my implementation of the CM20029 coursework's second part

## Description

The implementation has two files:

- parser.h, the file that contains all the C definitions
- parser.lex, the file that contains the LEX definitions and the program "core"

The parser contains the lexer program described in the first part, but with a small difference: when it sees a typing error, the lexer stops instead of telling that there are other errors in the file... This has been done since a typing error can result in very different interpretations of the tree, creating an incredible amount of parsing errors...

Our parse tree has 3 components, which are all pointers: the pointers pointing to the left and right branches (that will be NULL when the node is a leaf) and a pointer pointing to a token in the hash table. This is a very easy method, that saves at the same time lots of memory space (compared to a copy of all the token information that's already present in the hash table) and has the advantage of linking the things that are the same to the same physical place. On the other hand, in the case one would also need to take out elements from the hash table, some inconsistencies may appear; but we consider that if something is put in the hash table it is to stay there.

The parser is an LL(1) recursive descent parser: as I will detail in a moment it uses left-to-right parsing and looks one step ahead (or back) in the list of tokens. This look has been implemented using a variable called "rollback", and if this variable is TRUE then it means that the last token hasn't been used yet, so the current token information should be used instead of recalling yylex(). We need this since, for example, an expression may be a term or a term + term, therefore we'll need to see if there is a + after the term and, if not, not forget about the thing that was after that term.

The "rollback" variable is indeed the only specific part of my recursive parsing algorithm, the rest is as usual: all functions in the left hand side of the grammar just call functions corresponding to the right hand rules of the grammar, or simply verify the term's type against simple (terminal) rules (such as "a factor is either a variable, an integer or a string"). Each function returns the part of tree that corresponds to its execution (or NULL if the input is meaningless), and that subtree is added to the main tree. Therefore, trees are built as soon as possible and then added together...

The footprint (on paper) of my code is way bigger than the lexer, and this is mainly due to the error handling: most of the code is formed by if statements, and indeed if one takes away all the verification part the parser becomes much shorter!

The grammar has normally been left unchanged, since it did not contain any "dangerous" rule.

Another one note before I finish: as I've told before, trees have 3 pointers: one to the left branch, one to the right branch and one to the token. Therefore, to represent an if ... else statement we have the following: we have to nodes, the first with an "if" token, a left branch with the (conditional) expression and a right tree that has the else part. The "else" has its token pointing to the "else" token, its left branch pointing to the first goal of the if .. else statement and its right branch pointing to the second goal.

I think this is all I want to say about the way I've implemented my program.

## What's good with it?

Well, the good point is that it seems to be working: it can read tokens, see if those tokens form a valid sentence in the language, form a subtree based on this sentence, and then give it to the function that has called it in order to have the bigger tree (recursion). It seems not to reject any valid sentence.

As indicated before, the tree does not have any copy of the tokens but pointers to the hash table, which indeed makes the hash table become really “useful”.

## What's bad with it?

Again, we use malloc() multiple times, and perhaps an array and some smaller pointers pointing to its elements may have been faster. Just like in the first part, some variables do not use the most optimal type, for example “rollback” is a boolean but is marked as an integer. The indent counter should normally not be *very* big, in all cases a DWORD would have been enough.

## Testing

- parser1.txt:

```
if ( Variable +"string" * 123)
        342 % "other string";
else
        if ( Jo78N_09 +"look" * 1209)
                19 % 12;
        else
                ( 1 + 9 / 123 ) + "to" * Se4rch;
        ; // first if..else
; // second if..else
```

- Which gives this in output:

```
if
  + Variable
    * "string" 123
  else
    % 342 "other string"
    if
      + Jo78N_09
        * "look" 1209
      else
        % 19 12
        +
          + 1
            / 9 123
          * "to" Se4rch
```

We can see that the input has been parsed correctly.

– If we try to parse any bizarre word, the lexer stops instantly:

```
> parser
1.2
ERROR IN LINE 1 - you've written "." and I do not understand it

     The lexer will now stop
```

– It can also give some “somehow meaningful” error messages:

```
> parser
if ( Variable

Your if statement doesn't have an closing bracket!

Your text file is meaningless or I am bugful... Stopped at line 1
```

– Other meaningful error message example:

```
> parser
if ( Variable +"string" * 123)
        342 % "other string";
else
        if ( Jo78N_09 +"look" * 1209
                19 % 12;
        else
                ( 1 + 9 / 123 ) + "to" * Se4rch;
        ; // first if..else
; // second if..else

Your if statement doesn't have an closing bracket!

Your text file is meaningless or I am bugful... Stopped at line 5
```

– Finally, an input where the goal is followed by some code other than comments or spacers:

```
> parser
if( _try ) 123 ;;
// should not stop here
_but_here

ERROR: your code has more than one goal!

Your text file is meaningless or I am bugful... Stopped at line 3
```