Notes on my implementation of the CM20029 coursework's first part

Description

The implementation has two files:

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

The lexer.h file has definitions for isatty, yywrap and exit; just to make the Microsoft Visual C++ compiler happy. The name of the compiler used indeed indicates that testing has been done on Windows-based computers...

The hash function is the Open Hash function as discussed in the lecture. I did not do any "special" testing for that function, the hash table (obtained at the end) shows its correct operation anyways. It has 6 lines, so is not very complicated!

A token has multiple components: a type, a lexeme and a pointer to the next token. This indeed gives an idea of the organization of the hash table: we will actually have lists in all rows of the hash table. This is a method that's very easy to implement and maintain.

I did not do any "search table for token" function, but actually the newToken() function does that as well, and its first few lines (the while loop) could be exported as one "search" function.

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

What's good with it?

The best part of this is that the printed source code is actually 3 pages, most of which are comments... If you look at functions, you'll see that they all have not more than 10 steps; so is easy to understand and to modify later.

Another good part is that is seems to be working: it can recognize valid tokens (or tell that a token is invalid), it can tell what is wrong when something is wrong (but does not stop) and does not crash on the computers on campus... Note that on "normal" applications, we would expect the lexer to stop when a lexeme is not understood.

What's bad with it?

malloc() has been used in multiple cases even though it's been told during lectures that this is evil... I'm sorry about it! In addition, for the moment, the lexer just puts everything in the hash table, including tokens like "if" and it says "it is of type T_IF and its lexeme is 'if'!" ... which is just redundant information...

Otherwise, another more serious problem is that the token type is actually a number smaller than 278, which is smaller than 512, which is 2^5 ... but, we use a whole integer for it; which is probably 32 bits! A similar argument can also used for the hash function, that does 32 bit shifts whereas we only need the last 6 bits... But, in all cases, our processors are also 32-bit so it's not that bad, but still may be considered as a waste of space if you really deeply care about this.

Testing

As I told you, testing has been done on Windows-based computers, but nothing should prohibit the software from running on *nix machines as well... The symbol table printing prints the index of the table followed by the number in the list, so for instance [0 , 5] means that that object is the 5th element in the list in the 0th index of the hash table. We have two files for testing:

- lexer1.txt:

```
123  a  A  abAb  +123 if do while int void return ) , ; = / || &&
 -123 "hello"
```

- Which gives this in output ([...] is where I've cut it down a bit):

```
-- Now reading your input --

Object found on line 1: type 257, lexeme: "123" (291)
At line 1, you've written "a"; and I do not understand it
Object found on line 1: type 256, lexeme: "A" (1)
At line 1, you've written "a"; and I do not understand it
At line 1, you've written "b"; and I do not understand it
       [...]
Object found on line 2: type 258, lexeme: ""hello"" (536)

-- Now printing the symbol table --

[   1,  0 ] Type 256, lexeme "A"
[   9,  0 ] Type 269, lexeme ")"
[  11,  0 ] Type 271, lexeme ";"
[  11,  1 ] Type 266, lexeme "+"
[  12,  0 ] Type 270, lexeme ","
[  13,  0 ] Type 267, lexeme "-"
[  13,  1 ] Type 272, lexeme "="
[  15,  0 ] Type 274, lexeme "/"
[  18,  0 ] Type 256, lexeme "Ab"
[  79,  0 ] Type 264, lexeme "do"
[ 102,  0 ] Type 277, lexeme "&&"
[ 150,  0 ] Type 259, lexeme "if"
[ 204,  0 ] Type 276, lexeme "||"
[ 291,  0 ] Type 257, lexeme "123"
[ 506,  0 ] Type 261, lexeme "while"
[ 536,  0 ] Type 258, lexeme ""hello""
[ 538,  0 ] Type 262, lexeme "int"
[ 648,  0 ] Type 263, lexeme "void"
[ 889,  0 ] Type 265, lexeme "return"
```

We can see that the input has been lexed correctly, and the symbol table is OK.

- lexer2.txt:

```
if       6
// comment
"string1" "string2"
notvariable Variable _otherVAR14BLE
else "string \\|| 3"
while int void
do ++ -- / () { , ; = *
&& || %%
1 2 3 4 56.78 1 1 2 3 33 5 1
```

- Which gives that output ([...] is where I've cut it down a bit):

```
-- Now reading your input --

Object found on line 1: type 259, lexeme: "if" (150)
Object found on line 1: type 257, lexeme: "6" (6)
Object found on line 3: type 258, lexeme: ""string1"" (742)
Object found on line 3: type 258, lexeme: ""string2"" (758)
At line 4, you've written "n"; and I do not understand it
At line 4, you've written "o"; and I do not understand it
[...]
Object found on line 9: type 257, lexeme: "33" (51)
Object found on line 9: type 257, lexeme: "5" (5)
Object found on line 9: type 257, lexeme: "1" (1)

-- Now printing the symbol table --

[   1,  0 ] Type 257, lexeme "1"
[   2,  0 ] Type 257, lexeme "2"
[   3,  0 ] Type 257, lexeme "3"
[   3,  1 ] Type 258, lexeme ""string \\|| 3""
[...]
[ 648,  0 ] Type 263, lexeme "void"
[ 674,  0 ] Type 260, lexeme "else"
[ 691,  0 ] Type 256, lexeme "Variable"
[ 742,  0 ] Type 258, lexeme ""string1""
[ 758,  0 ] Type 258, lexeme ""string2""
```

We can see that the input has been lexed correctly: all things that should be understood indeed are, nothing is put in the hash table twice, and all things that should not be understood are not.

Note that non-understood characters have the same effect as spacers in this lexing process...