CSC4180 Assignment 4: Design a compiler for C

```
119010181 Lin Xinhao
```

In this assigment, I finally implemented a compiler front-end for C-minus. It does not rely on any parser or scanner tools like bison and flex, so it should earn the bonus I suppose.

How do you implement the Scanner?

I just used the scanner in assignment 2, a DFA-like program which will transfer file inputs into a list of tokens. Additionally, I let the generated token become a pair<token_type,string_val> which will carry the INT_NUM and ID variables to the latter part of the compiler.

How do you implement the Parser?

The same one as I did in assignment 3, a parser which generate First set, DFA and lookup-tables by reading production rules.

How the code is generated?

Similar to the generator in assignment 1, the generator function is called each time a reduction process is executed: in the parser, we maintain a op_stack which has a stack with all the current un-reduced Ts and NTs (when op_stack only contain a NT which is S derived by program, the next go-to state will be the acception of the whole input program), and these Ts and NTs will be loaded into a production and generate a new LHS NT.

So, each time a reduction process is executed, I pass the executing reduction rule (aka the production rule) and the list of to-reduce-Ts and to-reduce-NTs to the **generator(production,reduction_list)**.

The generator I implemented is quite ugly. It has a case for every possible production rule, and will produce their corresponding LHS non-terminals with some extra values that needs to carry.

Here is a list of the main carrying values that pass through the whole parsing process:

- a mips_code list. It is the necessary mips codes that needed in such recognized expressions/statements/declarations It is initially empty for those NT which doesn't need mips code to generate. And it will gradually grow by combining the reduced NT's mips codes and its own codes. In assignment 1 we don't need to pass such thing because the code doesn't contain jump cases so that it is sequential. However, this time we need to deal with cases like if(xxx1){xxx2}else{xxx3}, and the xxx1, xxx2, xxx3 are acutally nested inside the 'if(){}else{}''s mips codes.
- necessary info for some NTs, like in opeartor NT, it need to specify which kind of operator ('+' or '-'?) it is.

Moreover, I need to maintain a stack pointer $SP and frame pointer $FP. $FP is fixed because this project doesn't implement function call, and I just assume the whole input program is a function call. $SP keeping growing down and up to trace the temporary values pushed inside the memory during expression and statement execution. Noticing that the $SP will restore back to the initial value when the program is over, just like what real stack do. And if it did not restore, then it means some part of my code is wrong and I failed to find the bug.

You may notice that my generated mips code are very long, that's because I only use register to do the operations, and didn't use them to store the temporary values. I used stack instead. (it is extremely in-efficient, but means I don't need to worry the register-shortage)

Moreover, the declared IDs are stored in a compiler's map, they should be the shift amount relative to the $SP address, but I just used absolute address instead.

And a label_name map, a register allocation function sets and register_occupation map is also needed. And that's all.

About the result

The generated MIPS codes are output into the terminal, but it is usually too long. So I also stored it into a file named as 'output.txt', you can use this file to run on Mars.