Project 1: AcDc Compiler

DUE DATE: April 6, 2020

In Chapter 2, we have learned how to build a simple compiler for the *AC* language. Now you may download this compiler (written in C) from the class website on NTU COOL.

Files for this assignment are available in the following directories:
/src contains the C source files and a makefile for building the *AC* compiler,
/test contains a set of sample tests.
You may compile the simple compiler as follows:

```
cd src
make
```

and the AcDc compiler will be generated, you may test it using sample test files in the test directory.

```
./AcDc ../test/sample1.ac output1
./AcDc ../test/sample2.ac output2
```

NOTE:
1. Every token of the AC code are all separated by whitespaces or newlines !
2. Please refer to ACDC_notes.pdf to know more about the details behind code generation

In this assignment, you are required to extend the AcDc compiler in three ways:

1. **Extend the AC language to accept integer multiply (*) and divide (/) operators. You must correctly handle the precedence of * and / operators, which are higher than the + and - operators.**
2. **The AC language supports only single character variable names. You are required to lift this restriction and allow for variable length names. Note that in the test data, the length of a variable name will not exceed 256 characters, and the number of different variables will not exceed 23 (to simplify later code generation).**
3. **Enhance the AcDc compiler with a simple optimization called "constant folding", which evaluates constant expressions at compile time.**

For example, the following expression

a= 10+20-5 + b

should be turned into
a = 25 + b

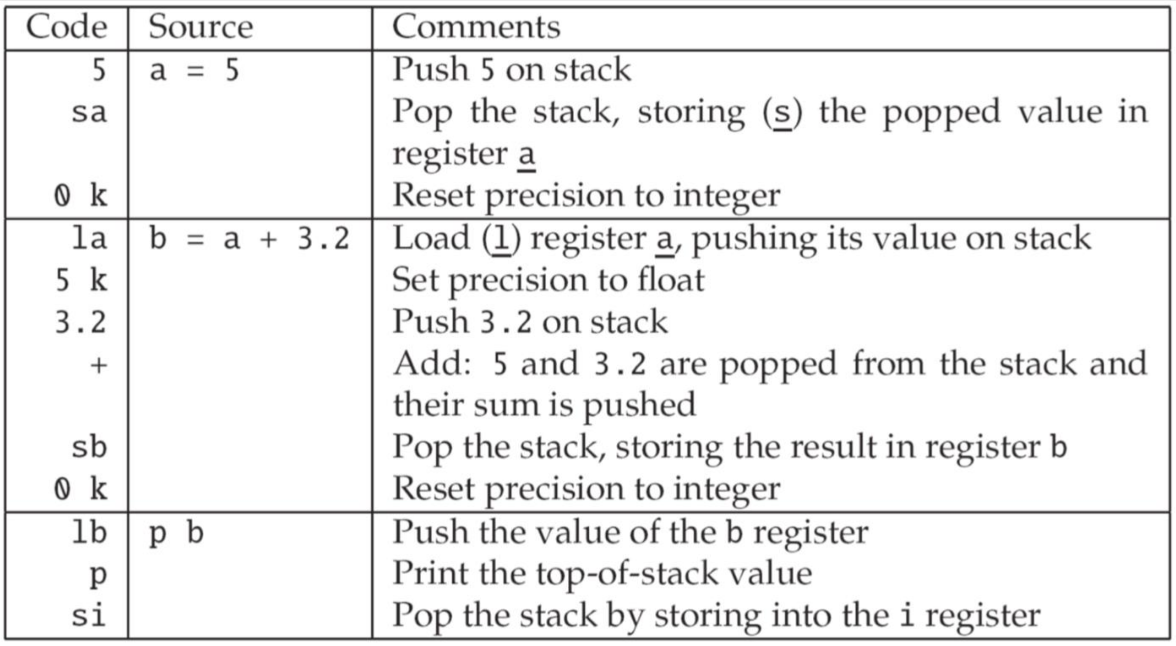
With constant folding at compile time, fewer instructions would be generated for DC.

Note that you are not required to exploit the constant folding opportunities in the following expressions:
a – 100 –50 + 6

Because the order of evaluation for the above expression is actually
(((a-100) – 50) + 6)
Therefore, there are no constant expressions to be folded unless more complicated optimizations such as applying the commutative laws to this expression.

When integer and float constants are mixed in expressions, you need to pay attention to the correctness of constant folding, for example, 1 / 2 = 0, but 1.0 / 2 = 0.5.

Brief Example:



Code	Source	Comments
`5`	`a = 5`	Push 5 on stack
`sa`		Pop the stack, storing (s) the popped value in register a
`0 k`		Reset precision to integer
`la`	`b = a + 3.2`	Load (l) register a, pushing its value on stack
`5 k`		Set precision to float
`3.2`		Push `3.2` on stack
`+`		Add: `5` and `3.2` are popped from the stack and their sum is pushed
`sb`		Pop the stack, storing the result in register b
`0 k`		Reset precision to integer
`lb`	`p b`	Push the value of the b register
`p`		Print the top-of-stack value
`si`		Pop the stack by storing into the `i` register

Submission requirements:

1) DO NOT change the executable name (AcDc).
2) Submit a zip file containing your code. Name it to **studentID_hw1.zip**. Then upload to NTU COOL.
3) Accept late submission for 3 days after the deadline.
4) Late submission penalty is 10 points per day.
5) Wrong submitted format will get 10 points penalty.