CS 5810 `Nonsense` Compiler Project
Due Date: *Monday, September 24, 2018 @ 8am*

You are to write a compiler for the language, `Nonsense`, which is an expression-based language. `Nonsense` is informally described below. The goals of this project are to become familiar with the compiler tools, flex and bison, and use it to build a small translator.

Requirements

1. Write your program in C or C++. It will be tested under 32-bit Ubuntu 16.04 and MUST work there. You will receive no special consideration for programs which work under a different environment.
2. Your code should be well-documented, and follow standard practice of good coding techniques (consistent, easily readable indentation, good choice of variable names, absence of tricky code, etc.)
3. Your compiler will translate any `Nonsense` program into assembly language for the 32-bit `x86`. The output of your compiler (written to `cout`) will be an `x86` assembly language program that generates correct results when assembled and executed.
4. You may use Eclipse to develop your code or your favorite editor.
5. Your program must take as a single command-line argument the name of a `Nonsense` program and emit `x86` assembly language to `stdout` or `cout`.

`Nonsense`

1. The source language is called `Nonsense`. The only data type supported by `Nonsense` is integer, so no declarations are required. Variable names must start with a letter followed by any number of letters and digits. `Nonsense` has no control constructs. A `Nonsense` program begins with a `begin;` statement followed by 0 or more output or assignment statements. Assignment statements consist of a variable on the left hand side and an expression on the right hand side that may use $+, -, *, /$ and ^(exponentiation) with proper precedence. A `Nonsense` program ends with an `end;`.
2. The following is a simple `Nonsense` program.

```
begin;
  x = 3;
  y = x * x + 2;
  z = y - x;
  output(z);
end;
```

`x86` Assembly Language

Your output must run on an `x86`, so you need to know a little about the assembly language. Use the `x86` pdf manual located in the "Project Resources" folder on the CS4131 Blackboard page as an in-depth reference. Basically, what you need to know is:

1. All variables and compiler generated temporaries should be stored on the stack to make the project easier. You will need at most 64 bytes of stack space. There will be at most 5 variables in any `Nonsense` program. To make things easier, your compiler should store the result of all arithmetic operations in a compiler generated temporary location. The only operations that should store into variable locations are assignments.

2. The `x86` has eight general purpose 32-bit registers: `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp` and `%esp`. We will reserve `%ebp` and `%esp` for stack management. You can use the others for holding temporary values. Note that some assembly instructions use some of the the general purpose registers as temporaries. This will be noted as necessary.

3. The top of the stack is held in the temporary register `%esp` and the base of the stack frame is held in the register `%ebp`. The space for variables is addressed with a negative offset off of `%ebp`. For example, the first 32-bit memory location for variables or temporaries is addressed as `%ebp-4`.

4. `x86` assembler uses two-address code. At most one operand may be a memory address, so you will have to use the general purpose registers as temporaries if the operation requires two memory addresses.

5. To accomplish an addition, subtraction or multiplication, use the following code sequence and the `add`, `sub` or `imul` assembly operation, respectively, where the two source operands are stored 4 and 8 bytes off of `%ebp` and the result operand is stored 12 bytes off of `%ebp`.

```
mov %eax, dword ptr [%ebp-4]
add %eax, dword ptr [%ebp-8]
mov dword ptr [%ebp-12], %eax
```

   The first operation loads the first operand into the register `%eax`. The second operation adds the contents of `%eax` to the second operation and stores the result in `%eax`. The last operation stores the final result back in the memory location for the result.

6. To move an immediate value (*e.g.*, 2) into a memory location (*e.g.*, one that is 4 bytes off of `%ebp`), use

```
mov dword ptr [%ebp-4], 2
```

7. To execute an integer divide of the form `y/x`, where `y` has an offset of 8 bytes, `x` has an offset of 4 bytes and the result is stored in a compiler temporary that has an offset of 24 bytes, do the following:

```
mov %eax, dword ptr [%ebp-8]
mov %ebx, dword ptr [%ebp-4]
cdq
idiv %ebx
mov dword ptr [%ebp-24], %eax
```

   Note that the dividend must be in `%eax` and that the quotient is written into `%eax`.

8. To execute an `output` we will use the C function `printf`. To call `printf`, use the following sequence, where the variable to be printed has a 4-byte offset.

```
push dword ptr [%ebp-4]
push offset flat:.io_format
call printf
add %esp, 8
```

The label `.io_format` is defined below.

9. To do exponentiation you may need the following instructions:

```
cmp %eax, %ebx
jge LABEL
jmp LABEL
```

   where `cmp` compares its operands (one may be in memory) and sets the condition code, `jge` jumps to `LABEL` if the condition code is set to greater or equal and `jmp` unconditionally jumps to `LABEL`.

10. In order to set up your program correctly, you must begin it with the following sequence

```
   .intel_syntax
   .section .rodata
.io_format:
   .string "%d\12\0"
   .text
   .globl main;
   .type main, @function
main:
   push %ebp
   mov %ebp, %esp
   sub %esp, 64
```

11. End your code with the following operations:

```
   leave
   ret
```

Grading

I will run your compiler on the small set of `Nonsense` programs found in `TestPrograms.zip` on the eLearning course page. You will be graded on whether your compiler generates correct code. Make sure your compiler does not seg fault on simple `Nonsense` programs (are there any other kind?). There are two facets (at least) to this project. Exponentiation will require more thought/work. So, get everything but exponentiation working first, then work on that. To get full credit for this assignment, you need to get exponentiation to work. However, you can still receive up to 90% of the total credit even if exponentiation is not implemented.

Getting Started

The following is a reasonable plan for doing this project. Get started early.

1. Get symbol table working (any c++ hash data structure should be ok)
2. Test the provided flex and bison files (`ns.l` and `ns.y`, respectively) with no actions to see that it is accepting `Nonsense` programs.
3. Instead of adding the correct actions, I'd add lots of `cout` statements to the bison actions. This would allow me to see output that shows the order of the bison rules being fired. This is something which is not obvious to the beginner and can help a lot.
4. Add in the actions to emit `x86` assembly for everything except exponentiation. Start with output statements and numbers. Then do expressions and assignment statements.
5. Add exponentiation.

What to turn in

Turn in a zip file named `Project1.zip` containing the directory in which your code resides (Eclipse project directory if you use Eclipse). Upload this zip file to the eLearning course site. If you choose not to use Eclipse, there must be a `makefile` in the root directory of your project such that I type make and I get the executable `nsc`.

Assembling and Executing Your Output

To assemble an assembler file that is output by your compiler, simply type `gcc file.s` To execute the program, type `a.out`.

Example Output for the Simple `Nonsense` Program Above

```
    .intel_syntax
    .section .rodata
.io_format:
    .string "%d\12\0"
    .text
    .globl main;
    .type   main,@function
main:
    push    %ebp
    mov %ebp, %esp
    sub %esp,64
    mov %eax, 3
    mov dword ptr [%ebp-4], %eax
    mov %eax, dword ptr [%ebp-4]
    mov %ebx, dword ptr [%ebp-4]
    imul %eax, %ebx
    mov $ebx, 2
    add %eax, %ebx
    mov dword ptr [%ebp-8], %eax
    mov %eax, dword ptr [%ebp-8]
    mov %ebx, dword ptr [%ebp-4]
    sub %eax, %ebx
    mov dword ptr [%ebp-12], %eax
    mov %eax, dword ptr [%ebp-12]
    push %eax
    push offset flat:.io_format
    call printf
    add %esp, 8
    leave
    ret
```