# CSC 526, Spring 2020, Assignment 1

**Purpose:** Setup

**Due:** 11:59pm, Wednesday, January 29, 2020

### Get the assignment code

These instructions assume that your course git repository is set up. Change into your course repository directory and enter the following commands.

```
git fetch assignments
git checkout assignments/master -- assignment1
git add assignment1
git commit -a
```

This will copy the `assignment1` directory into your working directory, start tracking the files in the `assignment1` directory, and commit those files to your local git repository.

### Assignment Description

The purpose of this assignment is to implement a simple compiler for a small language. The compiler will take a source file containing a single integer and create an executable that prints the integer.

The code that we generate will be in the form of a C-style function call. This allows us to implement some of the runtime features in C rather than assembly. Also, we will be able to use the gcc compiler to link the runtime to generated assembly code.

We will use the following C program, which we will name `runtime.c`, as our initial runtime.

```
#include <stdio.h>
#include <stdlib.h>

extern int64_t code_entry_point() asm("code_entry_point");

int main(int argc, char** argv) {
    int64_t result = code_entry_point();
    printf("%ld\n", result);
    return 0;
}
```

Here, the main function calls the function `code_entry_point`. This is the code that we will generate. The syntax `asm("code_entry_point")` indicates to a compiler such as `gcc` to not perform any platform-specific name alterations and to use the provided name exactly as it appears.

The goal for this assignment is to write a program that reads in a source program file and writes the generated assembly program text that defines `code_entry_point` to standard out.

Given a source file that contains the program `( 23 )`, the following is an example of an x86-64 assembly program that meets the specifications of the assignment:

```
  .text
  .globl code_entry_point
code_entry_point:
  movq $23, %rax
  retq
```

The assembly code is in AT&T syntax (there is also an Intel syntax) because `gcc` can handle that syntax.

The line by line account of the program means:

- `.text` – code in text form
- `.globl code_entry_point` – this assembly code is globally accessible
- `code_entry_point:` – the code for the symbol starts here
- `movq $23, %rax` – take the constant 23 and put it in the register called `rax`. This is the register that C programs expect to find a function's return value in.
- `retq` – perform the actions related to managing the stack (there will be more detail about this later) and then jump to where the caller of `code_entry_point` left off.

To produce an executable program we need to link the generated assembly with with `runtime.c`. We will use the GNU C compiler for this purpose. The following steps will create the executable program from a source file named 23.int:

```
./your_program 23.int > 23.s   # redirect standard out to a file
gcc 23.s runtime.c -o 23.run   # create executable named 23.run
```

**Syntax**

```
<program> :=
    | (<expr>)

<expr> :=
    | <integer>
```

**Deliverables**

The assignment1 repository contains the files: `Makefile`, `compile.sh`, and `runtime.c`. You need to add your source files and edit `Makefile` to build your program and `compile.sh` to run your program. The `compile.sh` script produces the x86-64 assembly file and an executable file.

**Turning in the Assignment**

To turn in the assignment execute the following git commands from within your repository:

```
git add <file>
git commit -a
git push origin master
```

where a `git add <file>` command is needed for every file that is required for building the assignment executable. Failure to add any required files will result in a failing grade for the assignment.

### Grading Criteria

The program will be graded by executing the `compile.sh` script and running the executable. Your program can assume that all input files are well-formed, that is, your program does not need to perform lexical or syntactic error checking.