Lab 3: Generating Assembly Code

CS 429: Fall 2017

Assigned: October 20, 2017

Due: November 3, 2017

Last updated: October 19, 2017 at 17:58

1 Introduction

Henrique Fingler (`hfingler@cs.utexas.edu` *Piazza is always better for asking questions*) is the lead TA for this lab. Please see him for detailed or specific questions about Lab 3.

This lab is designed to give you experience in C programming and x86-64 assembly. You will be writing a very basic compiler—in other words, a C program that produces x86-64 assembly code.

Your program will take in a *reverse Polish notation (RPN) arithmetic expression* and produce assembly code for a function that computes the value of that expression.

These instructions are quite long. Please read them carefully. Make sure that you are turning in correctly named files and generating your tarball submission in the specified way.

2 Logistics

This lab should only be developed and tested on a UTCS lab machine. Versions of `gcc` vary greatly between different platforms (e.g., Mac vs. UTCS machines), therefore it is imperative that you work only with UTCS machines.

This is an *individual* project. You may not share your work on lab assignments with other students, but feel free to ask instructors for help (e.g., during office hours or discussion section). Unless it's an implementation-specific question (i.e., private to instructors), please post it on Piazza publicly so that students with similar questions can benefit as well.

You will turn in a tarball (a `.tar.gz` file) containing the C source code for your program, a `Makefile`, and a `README`. See section 8 for descriptions of what these files do and how to name them, as well as how to create a `.tar.gz` file in the required manner.

Any clarifications or corrections for this lab will be posted on Piazza.

Please do not leave notes on submissions or ask grading questions on Canvas. Instead, post a private note on Piazza.

3 Background

Reverse Polish notation (RPN) is an alternative way to represent arithmetic expressions like those you might type into a calculator. At one point in time, it was popular to use RPN in physical calculators because it was easier to implement a calculator that read RPN as input. Similarly, it will be easier for you to do this assignment with RPN than with the normal style of writing arithmetic expressions.

In RPN, arithmetic operators such as `+`, `*`, and `-` come *after* their two operands, rather than in between them. Here are some examples of normal notation vs. RPN.

"Normal" Notation	**RPN**
`(1 + 1)`	`1 1 +`
`(3 * (4 + 7))`	`3 4 7 + *`
`(((8 + 9) * (4 + (3 * 10))) - 5)`	`8 9 + 4 3 10 * + * 5 -`

Notice that parentheses are not required to write expressions in RPN. Since each operator has a fixed number of arguments (*arity*) of two, the design of a calculator with RPN input is simplified. For example, nested parentheses don't need to be kept track of. In fact, a RPN calculator can be implemented with a stack data structure using two very simple rules.

A RPN calculator operates on a token-by-token basis, where each token is either a number or an operator. The input is a sequence of tokens, and one token is parsed at a time. The following two rules govern how a sequence of tokens describing an expression is evaluated.

1. When a number n is parsed, the calculator pushes n on its stack.
2. When an operator $\odot$ is parsed, the calculator pops two numbers off the stack (y from the first pop, x from the second pop), performs the operation $z = (x \odot y)$, then pushes z on its stack.

Here is an example of how a RPN calculator processes the last example from above. The left column is what the stack looks like *after* parsing each token listed in the right column. In this diagram, the bottom of the stack is on the left hand side.

```
8               Token parsed: 8  => Push 8.
8 9             Token parsed: 9  => Push 9.
17              Token parsed: +  => Perform (8 + 9), push the result.
17 4            Token parsed: 4  => Push 4.
17 4 3          Token parsed: 3  => Push 3.
17 4 3 10       Token parsed: 10 => Push 10.
17 4 30         Token parsed: *  => Perform (3 * 10), push the result.
17 34           Token parsed: +  => Perform (4 + 30), push the result.
578             Token parsed: *  => Perform (17 * 24), push the result.
578 5           Token parsed: 5  => Push 5.
573             Token parsed: -  => Perform (578 - 5), push the result.
```

Explanation: The first token parsed is `8`. This token is a number, therefore it is pushed on the stack. The second token is `9`. This token is a number, therefore it is pushed on the stack. The third token is `+`. Therefore, two numbers are popped from the stack: `9` from the first pop and `8` from the second pop. `8 + 9` is computed, and the result `(17)` is pushed on the stack. This pattern continues until the expression calculation is completed or an error occurs. Errors and other conditions are described later on.

4 The RPN compiler

We define in this section a simple compiler that takes an expression via the command line and dumps assembly code to `stdout` (standard output). The assembly code will entail a function called `compute` that accepts as arguments three values for the variables `x`, `y`, and `z` respectively, and returns the result of the expression. Your compiler should return `0` if successful. (Note that a C program does not return `0` by default—you must explicitly code this.) If unsucessful, return a non-zero number (see more below).

The input to your program is a RPN expression (sequence of tokens) given on the command line. This expression may contain any number of integer constants, the variables `x`, `y`, and `z`, and any number of operators. Integer constants may be positive, zero, or negative, meaning that they will consist of numeric digits, possibly with a minus sign at the beginning. The following operators are allowed: `+`, `t`, `-`, and `n` (plus, times, minus, unary minus). Expressions are written in RPN, as described earlier, with the addition of a single unary operator: unary minus. The operator mappings we'll use are listed below.

Operators	Corresponding Symbol We'll Use
`+`	`+`
`-`	`-`
`*`	`t`
`Unary Minus`	`n`

4.1 Unary minus

The unary minus operator will be `n`. For instance, the value `5` with unary minus applied to it (i.e., `-(5)`) will be written as follows: `5 n`. Notice that the `n` operator comes after the number it affects, just as the other operators come after the numbers they affect. However, the unary minus only takes a single number as an argument, rather than two. For example, the expression `(-(5) + -(3))` is written with RPN as `5 n 3 n +`. Note that numbers can be negative without using the unary minus at all. Think of the unary minus as an additional feature that sits upon an implementation which already allows positive and negative numbers. For instance, the following expressions are both valid RPN inputs:

`-1 -1 +` $\Rightarrow (-1 + -1) = -2$

`-1 n -1 +` $\Rightarrow (-(-1) + -1) = 0$

4.2 Implementation details

A few sample invocations of your program might be:

```
$ .\compiler x 4 + 2 t 4 -
[assembly code is printed here]
$ .\compiler x 4 + 2 4 t -
[assembly code is printed here]
```

As an example, if variable `x` has value 3, these expressions should equal 10 and -1, respectively. If variable `x` has value 4, the expressions should equal 12 and 0, respectively.

Therefore, the assembly code produced by the first run of your compiler should be a function that returns the number 10 when called with argument `x` set to 3, and the number 12 when called with argument `x` set to 4. The assembly code produced by the second run should be a function that returns the number -1 with argument `x` set to 3, and the number 0 with argument `x` set to 4.

The output of your program will be x86-64 assembly code, printed to `stdout` (standard output). This assembly code should contain a function named `compute` which accepts three arguments (`x`, `y`, and `z`) according to the standard x86-64 calling convention. `compute` should be designed to return the value of the input RPN expression where the values of `x`, `y`, and `z` have been substituted in. More on that below.

First, think about how RPN is evaluated in a bit more detail. Given an RPN expression, do the following, parsing one token (command line argument) at a time:

1. If you encounter a variable, push its value on the runtime stack.
2. If you encounter a constant, push it on the runtime stack.
3. If you encounter an operator that takes two arguments, pop the right number of arguments (i.e., two), apply the operator, and push the result back on the runtime stack.
4. If you encounter an operator that takes one argument, pop the right number of arguments (i.e., one), apply the operator, and push the result back on the runtime stack.
5. If there is a single numeric value on the runtime stack and the input is exhausted, then this is the answer to the expression, so simply pop and return it.

Note that we are directly using "the stack" as our RPN calculation stack, rather than making a separate stack data structure.

Various problems might occur while evaluating an expression. For example,

1. An unrecognizable symbol is encountered (i.e., something other than `+`, `-`, `t`, or `n`).
2. There may not be enough arguments on the stack for the operator you are attempting to apply.
3. There may be more than one value left on the stack when you are finished (i.e., when the input is exhausted).

To help identify some of these situations, maintain an integer variable to keep track of how many values are on currently on the stack when seeing if the expression is valid in your C code.

If you encounter any of these errors, print a useful error message *to stderr, not stdout*, and terminate with a **nonzero** exit code. Examples of illegal inputs are:

- `a` (because the variable `a` is not allowed—only `x`, `y`, and `z` are allowed)
- `4 +` (because `+` needs two arguments)
- `3 4` (because it will leave two values on the stack)
- `3 4 4 2 +` (because it will leave three values on the stack)
- `+` (because `+` needs two arguments)
- `4 5 n` (`n` is a unary operator, therefore two values are left on the stack)

Aside: It's important to print to `stderr` rather than `stdout` so the user can separate the assembly code output from any error messages produced. For example, when redirecting `stdout` to a file using the special shell directive "`>`", `stderr` messages are still printed to the screen, while the assembly code output goes to the file specified. Sample invocations are listed below.

```
$ ./compiler 1 2 3 4 + + + > output1.s
$ ./compiler 1 2 3 4 t - 5 6 7 + + - t 8 9 t + > output2.s
$ ./compiler 2 t 2 > output3.s
Error: The t operator requires two arguments
$ ./compiler 2 2 t > output4.s
```

In all four invocations, any assembly instructions printed to `stdout` will be silently put in the specified files (i.e., `output1.s`, `output2.s`, `output3.s`, and `output4.s`), which we are naming with the `.s` extension because `.s` is used as the extension for assembly code files. Notice in the third invocation there is an error message. The error message didn't get sent to the `output3.s` file, but rather to the screen, because "`>`" only redirects `stdout`, not `stderr`.

Note that if an expression is not legal, it doesn't matter if assembly output is given or not.

Now, imagine parsing a legal RPN expression and generating an x86-64 subroutine that can evaluate it. Our expressions may contain only variables `x`, `y`, and `z`. We assume that these are passed (in order) to your program via the usual calling mechanism. That is, `x`, `y`, and `z` are passed in the registers `%rdi`, `%rsi`, and `%rdx`, respectively. You should return your answer in register `%rax`, which again is specified by the standard calling conventions.

Consider the following RPN expression:

```
x 4 + 2 t 4 -
```

A first attempt at compiling this might be to generate the following sequence of x86-64 instructions (probably **without the comments**):

```
    .globl compute                # Note: this marks that the label "compute"
                                  # is a function name that can be used from
                                  # other code. This line is required.
  compute:
    pushq %rdi                    # push x on the stack
    pushq $4                      # push 4 on the stack
    popq  %r10                    # . add the top two numbers
    popq  %r11                    # |
    addq  %r10, %r11              # |
    pushq %r11                    # push the result
    pushq $2                      # push 2 on the stack
    popq  %r10                    # . multiply the top two numbers
    popq  %r11                    # |
    imulq %r10, %r11              # |
    pushq %r11                    # push the result
    pushq $4                      # put 4 on the stack
    popq  %r10                    # . subtract the top two numbers
    popq  %r11                    # |
    subq  %r10, %r11              # |
    pushq %r11                    # push the result
    popq  %rax                    # pop final answer into the return register
    retq                          # return from the function
```

However, this is inefficient for a few reasons. First, numbers are pushed on the stack and immediately popped off into registers. Second, numbers are being popped off the stack into registers before performing arithmetic operations on them. It's true that in x86-64, we can't write arithmetic instructions where both the source and destination are in memory, however sometimes it is possible to operate on a value that is on the stack when using an integer constant. This can be done with some instructions (like `add`), but not others (like `imul`).

Here is a second attempt, with some optimizations added:

```
    .globl compute
  compute:
    pushq %rdi                    # push x on the stack
    addq  $4, (%rsp)              # add 4 (directly in memory)
    popq  %r8                     # . multiply by 2 (using a temporary
    imulq $2, %r8                 # | register)
    pushq %r8                     # push the result
    subq  $4, (%rsp)              # subtract 4 (directly in memory)
    popq  %rax                    # pop final answer into the return register
    retq                          # return from the function
```

Notice that `add` and `sub` can target a memory location, but `imul` must target a register. Experiment with (or search the web for information about) different instructions to see which ones can target a memory location and which ones have to use registers.

You can use the `as` command to call the GNU assembler on your assembly code to check if it is valid x86-64 assembly. Use the `--64` flag to ensure you're assembling as x86-64, as opposed to 32-bit x86 or any other kind of assembly code. More on this beow.

Now, there are even more optimizations we might be able to do. Notice how in the previous assembly code, we never put more than one thing on the stack at a time. So why don't we just use a register for that one location on the stack we keep using? Then we don't need to access memory at all. In fact, we can just use the `%rax` register, which we eventually return.

Here's a third attempt:

```
   .globl compute
compute:
  movq  %rdi, %rax              # put x in %rax
  addq  $4, %rax                # add 4 to %rax
  imulq $2, %rax                # multiply %rax by 2
  subq  $4, %rax                # subtract 4 from %rax
  retq                          # return from the function
```

Note that we've started doing optimizations that are specific to the exact expression we started out with, `x 4 + 2 t 4 -`. These particular optimizations will not work for all expressions. Therefore, if you want to implement these optimizations, the C code which generates this assembly will have to be smart and know when it can and can't do certain optimizations.

For your compilation, do not use callee saved registers. In other words, *only use the following registers*:

- `%rax`: for the return value and possibly for intermediate storage
- `%rsp`: the stack pointer
- `%rdi`, `%rsi`, and `%rdx`: the registers used for passing in the three arguments
- `%rcx`, `%r8`, and `%r9`: the three unused argument-passing registers
- `%r10`, and `%r11`: the other two caller-saved registers

You can reuse registers if it's safe to do so, and you can push things onto the stack if you run out of registers at any point.

Your code should be able to handle arbitrary RPN expressions that follow our constraints on variables.

5 Testing your code

You don't want to submit your code without testing it first. In this section various ways in which you can make sure your code is working correctly are covered.

5.1 Your C code should compile

Make sure your C code doesn't have any syntax errors, etc., and that your `Makefile` is in order. If you go to the directory where your compiler code is and type `make`, an executable file called `compiler` should be created in the same directory. Test this on `skipper.cs.utexas.edu`—this machine has an environment similar to what we'll use for grading.

5.2 Your C program should produce valid x86-64 assembly code

Make sure that when you run the `compiler` executable which you created with a valid RPN expression (passed in as a series of command line arguments), valid assembly code is printed to the screen.

Beyond just eyeballing that your assembly code is correct, make sure that the output is valid assembly code by using the special shell directive "`>`" (as mentioned earlier) to make your program direct its output into a `.s` file such as `test.s`, and then run the assembler on it to see if there are any syntax errors or other problems. This will tell you if the assembly code in `test.s` is valid x86-64 or not:

```
$ as --64 test.s
```

(As mentioned before, the `--64` is to ensure that the assembly code is treated as x86-64 assembly code.)

5.3 The assembly code should provide the right function

Your program is required to produce assembly code for a function called `compute` which takes three 64-bit signed integer arguments and returns a 64-bit signed integer argument. In C, that means the function would have looked something like this:

```
#include <stdint.h> // this needs to be at the top of the file

int64_t compute (int64_t x, int64_t y, int64_t z) {
  // do stuff
  return something;
}
```

Since you're writing the function in assembly, if you wanted to use it in some C code, you must "import" it. This is how to do it in C: first, at the top of your C code, make an `extern` declaration of the function `compute`, like so:

```
#include <stdint.h> // this needs to be at the top of the file

extern int64_t compute(int64_t x, int64_t y, int64_t z);
```

Notice that no function body is provided, only the argument types, return type, and name of the function, with the word `extern` (for "externally defined"). Then, when you compile your C code, specify the name of the assembly file which contains the function `compute`, and gcc will hook it up for you. For example, assuming the assembly code for `compute` is in `compute_function.s` and that `test_program.c` uses the `compute` function, this is how to compile `test_program.c`.

```
$ gcc -o test_program test_program.c compute_function.s
```

So, to test that your assembly code provides the correct function, write a C program which uses the `compute` function you defined (i.e., it calls `compute` on some arguments and prints the result). Make sure to put in the `extern` definition shown above.

Then, compile the program, telling gcc the name of the file where you saved the assembly code output from your `compiler`. If there are no errors, then gcc was able to read your assembly code as a definition of a `compute` function, as desired. (If you follow the format shown in the assembly code examples above, you shouldn't have any problems.)

Here is a practical walk-through of the steps listed above:

5.3.1 Step 1: Compile your compiler

```
$ gcc compiler.c -o compiler
```

5.3.2 Step 2: Use your compiler to produce assembly code for any RPN expression

Below, we are putting the assembly code in the file `compute_function.s`.

```
$ ./compiler x y z + + > compute_function.s
```

5.3.3 Step 3: Create a C program to use the assembly code

Put the below code in a file named `test_program.c`. This code will print the result of the `compute` function when `x`, `y`, and `z` are given values 1, 2, and 3, respectively.

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

extern int64_t compute(int64_t x, int64_t y, int64_t z);

int main()
{
  int64_t x = 1;
  int64_t y = 2;
  int64_t z = 3;

  int64_t computeResult = compute(x, y, z);

  printf("Result: %" PRId64 "\n", computeResult);

  return;
}
```

5.3.4 Step 4: Compile `test_program.c`

Here, we feed `gcc` the file reference to where the `compute` function lives.

```
$ gcc -o test_program test_program.c compute_function.s
```

5.3.5 Step 5: Run the test program to check your result

```
$ ./test_program
```

By the way, if you're interested in how this linking process works, check out Chapter 7 in the textbook, which we may get to later in the semester.

5.4 The generated functions should work correctly

Try compiling a few RPN expressions of your own choosing. For each one, your program will produce some assembly code. Create a C program that will link with your assembly code (as described in the previous section) and test it by calling it with some particular values of `x`, `y`, and `z` and printing the result. Compile the test program together with the assembly code and run the resulting executable to see if it prints the number you expected.

Try various different kinds of RPN expressions to make sure you did everything correctly. Try some with many operators, some with many numbers in a row, some with numbers that are

more than a single digit, some with negative numbers, etc. **Try out the biggest positive and negative numbers**, as well as very long expressions that may cause infinite loops, segmentation faults, or other undesireable behavior in your compiler.

If you're feeling really ambitious, you might try to write a randomizing test harness that automatically tests your compiler on many different randomly generated RPN expressions, and substitutes in many different randomly generated values of `x`, `y`, and `z` for each RPN expression.

5.5 Your submission should be in the correct format

Once you've finished programming, it's time to submit your work. Please read section 7 thoroughly to comply with the handin requirements. Make sure that the file names are exactly as specified and the tarball is created with the given steps.

6 Grading

Your assignment is to write a compiler program as described above.

The basic grading policy for this lab is that you'll get **100 points (full normal points)** if you create a program that correctly does what it is supposed to do—that is, accepts RPN expressions on the command line and dumps assembly code to `stdout` which defines a function called `compute` that takes arguments `x`, `y`, and `z` and returns the value of the RPN expression when those values are substituted in for the variables `x`, `y`, and `z` in the expression.

If you produce straightforward (but inefficient) assembly code such as that shown in the first example, **you will still earn the 100 points (full points) if the code behaves correctly**.

Up to **10 extra credit points** are possible if you optimize the assembly code you generate in some significant way. Two such ways are as follows.

1. Avoid using the stack when you can help it, and instead use the various registers you have at your disposal, which are much faster than accessing memory.

2. Use *constant folding* to reduce the number of actual computations that would take place at runtime.

   For example, if your compiler was given an RPN expression that was constant (i.e. didn't involve any variables), then your compiler could just figure out the integer value of the expression and create an extremely fast, two-instruction function that just wrote the correct answer into `%rax` and returned, without doing any work!

   Even for RPN expressions that do involve variables, maybe you could do partial constant folding. For example, simplify `x 1 2 3 4 + + + +` to `x 10 +`.

   If you are feeling really ambitious, there are even more sophisticated forms of partial constant folding that you might try to explore, such as exploiting commutativity, factoring polynomial-like expressions, etc.

Beyond the 100 + 10 extra points breakdown given above, determination of further partial credit is up to the discretion of the grader.

7 Submission

Create a `.tar.gz` file submission named `<lastName>_<firstName>_<UT-eid>_lab3.tar.gz` to submit using the instructions shown below. For example, with a name of `Mike Feilbach` and a UT-eid of `mf1234`, the proper submission is `feilbach_mike_mf1234_lab3.tar.gz`. Note the underscores in the submission file name.

This `.tar.gz` file should untar to a single folder (of some practical name without spaces) containing exactly three files named as follows:

1. `compiler.c` (the C source code for your compiler)
2. `Makefile` (a makefile which creates the `compiler` execuatble when `make` is called)
3. `README.txt` (a `.txt` file containing a description of optimizations implemented, otherwise empty)

To create the `.tar.gz` file, use the following commands. In this example, the folder `lab3` contains the files `compiler.c`, `Makefile`, and `README.txt` and the current directory contains the `lab3` folder.

```
$ tar -cvzf feilbach_mike_mf1234_lab3.tar.gz ./lab3
```

This should be done on `skipper`. To ensure that the tarball submssion was created properly, try untarring it with the following command:

```
$ tar -xvzf feilbach_mike_mf1234_lab3.tar.gz
```

This should create a folder `lab3` in the current directory. Within `lab3` should be three files: `compiler.c`, `Makefile`, and `README.txt`.

Once untarred, `cd` into `lab3` and execute the following command:

```
$ make
```

This should build your program and generate a C executable named `compiler`.