Homework 1

Mayer Goldberg

March 18, 2018

Contents

1 Overview

In this assignment you shall write a postfix-based calculator for integer arithmetic, somewhat similar to the `dc` program on Unix/Linux (read over the *man* page for `dc`). The program will be written in a combination of C and x86/64-bit assembly language.

- The C code shall be compiled to an object files using *gcc*.
- The assembly code shall be assembled into an object file using *nasm*.
- The object files shall be linked using *gcc* to produce an executable by the name of `calc`.

This is a complex assignment, and you can do it in one of two versions:

- **Version 1: Simple, maximum grade: 80**. This version works with fixed-size, 1024-bit signed integers.

- **Version 2: Harder, maximum grade: 100**. This version works with arbitrary-precision integers, providing functionality similar to that of *bignums* in Scheme, `int` in Python, or `LargeInteger` in Java. You might want to use a structure similar to the following:

  ```
  typedef struct bignum {
    long number_of_digits;
    char *digit;
  } bignum;
  ```

This is the only difference between the two versions. In terms of functionality and interface, the two versions are identical.

Your calculator will implement the following functionality:

- Addition
- Subtraction
- Multiplication
- Division

2 The format of the input

The calculator will use *postfix* notation for input, and will use a *pushdown stack* in the implementation. **This stack is not the same as the builtin x86 stack, but a separate data structure that is used only for storing numbers during a calculation**. We suggest you implement this stack as an array of some reasonable side, no less than 1024. An expression such as $2*3+4*5$ will be entered as `2 3 * 4 5 * +`. The result (which is 26 in this case), shall be pushed onto the stack.

3 What shall be done in C

- Implement the stack
- Implement the text-based user-interface, supporting the following possibilities:
  - *a signed integer* to be pushed on the stack
    - Positive numbers in *base 10* are entered as usual: `1234`, etc.

    - Negative numbers in *base 10* are entered with the sign denoted by the *underscore* character (so as not to be confused with the operation of *subtraction*: `_1234`, `_1`, etc.
  - `+` for addition
  - `-` for subtraction
  - `*` for multiplication
  - `/` for division
  - `p` for printing the result
  - `c` to clear the stack
  - `q` to quit the program and return to the underlying shell
- The user-interface will then call routines written in assembly code to perform the various arithmetic functions.

4 What shall be done in assembly language

The four operations shall be implemented in x86/64-bit assembly language, using the linux calling conventions. It will be easiest for you to implement these functions to operate directly on the *pushdown stack*.

Example: To compute $2 * 3 + 4 * 5$, print the result, and quit, we input `2 3 * 4 5 * + p q`:

- `2` will be pushed onto the stack
- `3` will be pushed onto the stack
- `*` will pop the top two elements on the stack, and push their product, 6, back onto the stack
- `4` will be pushed onto the stack
- `5` will be pushed onto the stack
- `*` will pop the top two elements on the stack, and push their product, 20, back onto the stack
- `+` will pop the top two elements on the stack, and push their sum, 26, onto the stack
- `p` will print the result in base 10 (decimal)

- `q` will quit the program and return to the underlying shell

Your program should treat all *whitespace* (characters less than or equal to ASCII 32) as delimiters, and use them to separate *tokens*. Space is optional anywhere other than between two numbers. So `2 3 * 4 5 * +` could also have been written as `2 3*4 5*+`.

5 How we shall test the code

Your program reads input from *stdin* and writes output to *stdout*. We shall prepare files with expressions in *postfix* notation, and *redirect* them to your programs, expecting the correct arithmetic result in *stdout*. If the file `foo.input` contains the text `2 3 * 4 5 * + p q`, the following is the kind of interaction we shall expect at the shell:

```
% calc < foo.input
26
```

Your program should come with a *makefile* that runs *gcc* and *nasm* and builds the executable from scratch, using `make calc`. You should also provide for `make clean`, which shall remove the object files and executable. Compilation should generate no warnings, no error messages, and execution should generate only the expected output, without any additional messages or warnings, and certainly no error messages.

Programs that do not compile and run cleanly shall be graded 0.