CS 330: Homework 1
Savage: Assembly Programming

Overview

Computer science is a field full of tension. As developers, we want to express our algorithms clearly and succinctly. As users, we want our programs to run efficiently. The clearest and most succinct programs are written in domain-specific languages, whose primitives and constructs reflect the problems we're trying to solve. The most efficient programs are written in machine language, whose primitives and constructs reflect the underlying hardware. Rarely do we find one language that suits both developers and users.

Instead, most problems we solve we involve at least four languages: the input protocol, the output protocol, the high-level general purpose language that we write our program in, and the low-level machine language into which our high-level program is translated. In this homework, the input language is a simple shape description language. The output language is HTML5's scalable vector graphics (SVG) language. The high-level language is—well, that's the gotcha. Let's throw out the high-level language. You will directly solve the problem of translating from one geometric language to another by writing assembly code, using the Intel x86 32-bit instruction set.

In completing this painful exercise, you will grow neurons that hold knowledge on the following topics: how scalar and array variables reside in memory, the call-by-value calling convention, the operation of the program stack, the terrors of typelessness, and a universal method of controlling flow within a program.

Figure 1: An example output image from this homework.

The input language supports the placement of rectangles, ellipses, and text in a scene. Consider the following representative utterance, which places an orange rectangle beneath a semi-opaque yellow ellipse and a line of semi-opaque black text:

```
color 255 128 0 1.0
rectangle 0 0 470 120
color 255 255 0 0.5
ellipse 235 60 225 50
color 0 0 0 0.5
text 100 75 48 n is never big
```

Translating to HTML5's SVG language yields:

```
<svg xmlns="http://www.w3.org/2000/svg">
<rect x="0" y="0" width="470" height="120" fill="rgb(255,128,0)" opacity="1.00" />
<ellipse cx="235" cy="60" rx="225" ry="50" fill="rgb(255,255,0)" opacity="0.50" />
```

```
<text x="100" y="75" font-size="48" fill="rgb(0,0,0)" opacity="0.50">n is never big</text>
</svg>
```

Deduce what the numbers in the input mean by examining the corresponding output and reading about SVG online. (An example reference is Scott Murray's at `http://alignedleft.com/tutorials/d3/an-svg-primer`.) When the SVG is viewed in a browser, we see figure 1.

Your task is to write a tool—we'll call it Savage—that translates input consisting of `color`, `rectangle`, `ellipse`, and `text` commands to SVG.

Requirements

Specifications do not tell you how to solve a problem—just what pieces may be used. The requirements described below will need to be thought about and pieced together using your own good mind. You will likely need to read this document many times, ask questions, and afford yourself plenty of time to finish this homework.

Your solution is to meet the following specification:

1. Place all files in directory `<YOUR-REPOSITORY>/savage`.
2. All code must run on `dplsubmit`.
3. Your output must exactly match the expected output. Each line of output must end with `\n`.
4. The assembly code must be hand written in file `savage.s`—not generated from a high-level language.
5. Write a makefile that compiles `savage.s` into an executable named `savage`. Compile with `gcc`. Add flags `-m32` to stick to 32-bits and `-g` to include debugging information in the executable. Have a `clean` rule that removes the executable.
6. Read input from STDIN. You may assume it's well-formed and that no `text` entity is longer than 100 characters.
7. Write output to STDOUT.
8. Freely use C functions. The reference implementation uses `scanf`, `printf`, `strcmp`, and `exit`.
9. All three shapes have attributes `fill` and `opacity` in their SVG form. Factor out the code that prints these two attributes into a helper function labeled `print_color`. This function must preserve and restore the caller's `%ebp`.
10. Make the default color black, completely opaque.
11. Include for each function you write—including `main`—a map of the stack. That is, describe each local variable or parameter and list its offset relative to `%ebp`.
12. Make local memory references ultimately relative to `%ebp`, not `%esp`.
13. Create a scene in the input language. Share it and its corresponding image on Piazza in a note tagged `savage`.

Hints

Format strings are extremely useful for solving this problem. There are a couple of occasions where they must be used with care.

First, when printing `double`s with `%lf`, we can't push them onto the stack before we call `printf` like we do with other types. Instead, we have to push using some special instructions that manipulate the floating-point unit on the CPU:

```
myformat:
  .asciz "mydouble: %lf\n"
mydouble:
  .double 0.333333

myprinter:
  subl $8, %esp    # "Declare" 8 bytes on the program stack.
  fldl mydouble    # Load a "long float" into the float stack.
  fstpl (%esp)     # Pop the float stack onto the program stack.
  pushl $myformat  # Above was param1. Now for param0.
  call printf      # Work your magic, bud!
  addl $12, %esp   # Parameters can go bye-bye.
```

Second, function `scanf` uses format strings to parse input. One can use `%s` to grab the next white-space delimited token. The `text` entity in the input, however, may contain whitespace, which means we can't use `%s` to read the text. Instead, we can use `%[^\n]`, which reads a string of non-newline characters.

Submission

This homework is graded by hand and with help from the grading script, which can run from your homework directly with `../specs/grade`. Your assignment is expected to fully meet the requirements above, those checked by the grading script, and the following:

1. Variable names should be meaningful and accurate.
2. Non-obvious parts of your code should be commented.
3. Code should be cleanly formatted and indented.

Your work will also be inspected for plagiarism. Please do your own work. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. (If you find violators of this rule, please let me know.) Write your own code.

The grading script sends your instructor an email when it successfully completes. All that remains is for you to push your code to Bitbucket.