Programming Assignment: COOL Example

This assignment asks you to write a short Cool program. The purpose is to acquaint you with the Cool language and to give you experience with some of the tools used in the course. This assignment will *not* be done with a partner; you should turn in your own individual work. All future programming assignments will be done in teams of either one or two.

A machine with only a single stack for storage is a *stack machine*. Consider the following very primitive language for programming a stack machine:

Command	*Meaning*
int	push the integer *int* on the stack
+	push a '+' on the stack
s	push an 's' on the stack
e	evaluate the top of the stack (see below)
d	display contents of the stack
x	stop

The 'd' command simply prints out the contents of the stack, one element per line, beginning with the top of the stack. The behavior of the 'e' command depends on the contents of the stack when 'e' is issued:

- If '+' is on the top of the stack, then the '+' is popped off the stack, the following two integers are popped and added, and the result is pushed back on the stack.
- If 's' is on top of the stack, then the 's' is popped and the following two items are swapped on the stack.
- If an integer is on top of the stack or the stack is empty, the stack is left unchanged.

The following examples show the effect of the 'e' command in various situations; the top of the stack is on the left:

stack before	*stack after*
+ 1 2 5 *s* ...	3 5 *s* ...
s 1 + + 99 ...	+ 1 + 99
1 + 3 ...	1 + 3 ...

You are to implement an interpreter for this language in Cool. Input to the program is a series of commands, one command per line. Your interpreter should prompt for commands with `>`. Your program need not do any error checking: you may assume that all commands are valid and that the appropriate number and type of arguments are on the stack for evaluation. You may also assume that the input integers are unsigned. Your interpreter should exit gracefully; do not call `abort()` after receiving an `x`.

You are free to implement this program in any style you choose. However, in preparation for building a Cool compiler, we recommend that you try to develop an object-oriented solution. One approach is to define a class `StackCommand` with a number of generic operations, and then to define subclasses of `StackCommand`, one for each kind of command in the language. These subclasses define operations specific to each command, such as how to evaluate that command, display that command, etc. If you wish, you may use the classes defined in `atoi.cl` in the `~cs164/examples` directory to perform string to integer

conversion. If you find any other code in `~cs164/examples` that you think would be useful, you are free to use it as well.

We wrote a solution in approximately 200 lines of Cool source code. This information is provided to you as a rough measure of the amount of work involved in the assignment—your solution may be either substantially shorter or longer.

Sample session

The following is a sample compile and run of our solution.

```
%coolc stack.cl atoi.cl
%spim -file stack.s
SPIM Version 5.6 of January 18, 1995
Copyright 1990-1994 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README a full copyright notice.
Loaded: /home/ff/cs164/lib/trap.handler
>1
>+
>2
>s
>d
s
2
+
1
>e
>e
>d
3
>x
COOL program successfully executed
```

Getting and turning in the assignment

Create a working directory called PA1 and `cd` into it. From there, type

```
% gmake -f ~cs164/assignments/PA1/Makefile
```

This command creates several files you will need in the directory. Follow the directions in the README file.

To turn the assignment in:

1. Make sure your code is in stack.cl and that it compiles and works :-). A copy of `Makefile` and `atoi.cl` will be present when we test your submission, so all we need from you is `stack.cl` and `README`.

2. Answer the 3 required questions in the `README file`, and include any other relevant comments here.

3. Make sure everything is in a directory called `PA1`.

4. Run the command `submit PA1` (case is important) from your PA1 directory.

Please note: Your stack machine will be tested by comparing its output to that of our reference implementation. Therefore, your stack machine should not produce any output aside from whitespace (which our testing harness will ignore), '`>`' prompts, and the output of a 'd' command. Prior to submitting, please remove any output commands that you used for debugging.

An optional tool that may be of use to you

Prof. Susan L. Graham's Harmonia project provides Harmonia-mode, an XEmacs extension that can assist you when writing Cool programs. Harmonia-mode offers a number of useful features, including syntax highlighting and proper indentation. Harmonia-mode will also highlight syntactic and semantic errors as you type.

Note: Harmonia-mode is research technology and may crash on you; however, it has been engineered to recover from any type of crash, preserving your work. *Its use in this class is completely optional*. We encourage you to try it out at `http://harmonia.cs.berkeley.edu/harmonia/cs164`. If you like it (or don't), tell us in your README. Also, please feel free to email the Harmonia group at `harmonia-feedback@sequoia.cs.berkeley.edu`.

If you used Harmonia-mode for all or part of your project, please take a minute to answer the following questions in your README:

1. Rate Harmonia-mode's usefulness on a scale of 0 to 6, where 0 means that it kept you from doing any work, 3 means that it neither helped nor hindered you, and 6 means that you'll never again dare to edit a Cool program without it.

2. What aspects of Harmonia-mode particularly helped you?

3. What aspects were particularly detrimental?

4. Is there anything else about Harmonia-mode that you would like to mention?

Extra Credit.

There is a chance that you will discover a bug in our Cool compiler. We will award extra credit for legitimate bug reports; to get credit, send a bug report to `cs143-aut0506-staff@lists.stanford.edu`. Your report must include all of the needed Cool source and a transcript of a terminal session showing how to reproduce the bug (use the `script` command). There are a number of ways the compiler can potentially fail: the compiler may dump core, the generated code may be incorrect, the compiler may refuse to accept a legal program, it may accept an illegal program, etc. *Please be sure you have found a bug before submitting a report!* The course staff are the final arbiters of what is a "bug" and what is a "feature". Credit usually will be awarded only to the first person to report a bug.