Homework #3

TAs' email: dsa AT csie DOT ntu DOT edu DOT tw

RELEASE DATE: 04/01/2014
DUE DATE: 04/22/2013 (**Tuesday**!!!), 5:30pm (after class)

As directed below, you need to put your code in the designated repository.

Any form of cheating, lying or plagiarism will not be tolerated. Students can get zero scores and/or get negative scores and/or fail the class and/or be kicked out of school and/or receive other punishments for those kinds of misconducts.

Discussions on course materials and homework solutions are encouraged. But you should write the final solutions alone and understand them fully. Books, notes, and Internet resources can be consulted, but not copied from.

Since everyone needs to write the final solutions alone, there is absolutely no need to lend your homework solutions and/or source codes to your classmates at any time. In order to maximize the level of fairness in this class, lending and borrowing homework solutions are both regarded as dishonest behaviors and will be punished according to the honesty policy.

Both English and Traditional Chinese are allowed for writing any part of your homework (if the compiler recognizes Traditional Chinese, of course). We do not accept any other languages. As for coding, either C or C++ or a mixture of them is allowed.

This homework set comes with 200 points and 20 bonus points. The 200 points include 10 points for *git* usage (see submission guide), 90 points for the hand-written part and 100 points for the programming part. In general, every homework set of ours would come with a full credit of 200 points.

1 More about Analysis Tools

(1) (20%) Do Exercise R-4.22 of the textbook.

(2) (10%) Do Exercise R-4.39 of the textbook.

(3) (10%) Do Exercise C-4.11 of the textbook.

2 More about Arrays

(1) (10%) Use any pseudocode to do Exercise C-3.3 of the textbook. (The faster the better.)

(2) (10%) Given two strictly increasing integer arrays A and B, use any pseudocode to write down an algorithm for finding $A \cap B$. (The faster the better.)

3 List, Stack and Queue

(1) (10%) If a singly linked list is wrongly constructed, there would be a cycle within the linked list. Use any search engine or consult any friend to find an $O(N)$ time and $O(1)$ space algorithm that detects whether there is a cycle in a linked list of size N where N is unknown beforehand (*hint*: usually called Floyd's Cycle-Finding Algorithm). Learn and explain the algorithm clearly to the grading TA in your own words. Also, cite the website (or the person) that you learn the algorithm from.

(2) (10%) Use any pseudocode to write down an algorithm that uses two queues (with `enqueue`, `dequeue` and `isempty` operations but no others) to simulate one stack (for `push` and `pop` operations). What is the total running time after N operations?

(3) (10%) Use any pseudocode to write down an algorithm that uses two stacks (with `push`, `pop` and `isempty` operations but no others) to simulate one deque (for `push/pop front` and `push/pop back` operations). What is the total running time after N operations?

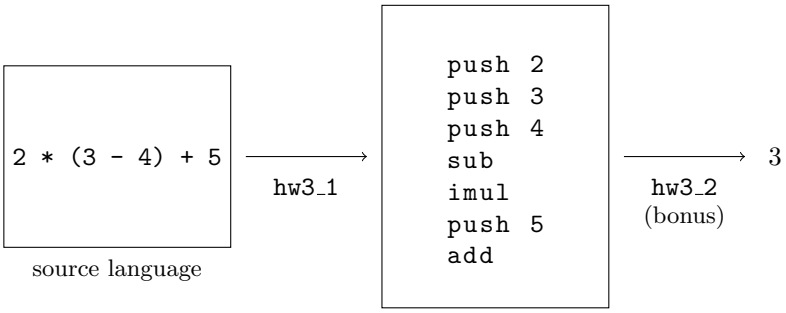
Hint: You can begin your analysis by reasoning how much time it takes to perform ***one operation*** and how much time is required when there are N ***operations***.

4 Programming via SVM

4.1 Introduction

In this homework, we are going to ask for your help in designing a minimal programming language based on the calculator introduced in the class. This will mainly involve the infix-to-postfix algorithm (that is, the "shunting-yard" algorithm invented by Dijkstra) you have learnt.

The work is splitted into two parts. The first part is a toy compiler that converts the input expression into a sequence of low-level instructions. The second part, which is left as a bonus for those who are interested in playing with so, is a stack virtual machine ("SVM" for short) which executes the translated instructions on a stack, as illustrated by the following figure. Note that stack virtual machine is one important part of the famous Java platform. We hope that this minimum programming language helps you think more about how programs can work in different languages/platforms.



(Note: the postfix form of the input expression is `2 3 4 - * 5 +`)

4.2 Core Language

The language is designed to be as small as possible, you need only support several core features. Apart from usual arithmetic operators, the most distinctive feature of our language will be the support for conditional expressions and full mathematical functions.

Given an expression e, let $eval(e)$ denote its value. So $eval(1+2) = 3$, $eval(2 \times (3-4)+5) = 3$.

- Conditional Expression, `if` $expr_1$ `then` $expr_2$ `else` $expr_3$

  The conditional expression is similar to the ternary operator `_?_:_` in C++. We shall first evaluate $expr_1$ to a number (say, n), continue with $expr_2$ if $n \neq 0$ and $expr_3$ otherwise.

$$eval(\texttt{if } expr_1 \texttt{ then } expr_2 \texttt{ else } expr_3) = \begin{cases} eval(expr_2) & \text{if } eval(expr_1) \neq 0 \\ eval(expr_3) & \text{if } eval(expr_1) = 0 \end{cases}$$

- Functions, \ *argument* `->` *body*

  - **(Creating Functions)** Our mathematical function will be anonymous. In our syntax, an expression `\x -> e` denotes a function that takes an argument `x` and yields `e`, where `e` is some expression (which may involve `x`).

    * Example 1. `\x -> x * x` denotes the function f such that $f(x) = x \times x$.
    * Example 2. The function g where $g(y) = 2 \times y + 1$ can be written as `\y -> 2 * y + 1`.

Note that in these examples, the names of the function f, g are completely dropped. Also, since it doesn't matter which name we choose for the formal paramter; `\x -> x * x` will behave exactly the same as `\t -> t * t`.

Our functions will always be univariate.

- **(Function Application)** We denote function application of f on x by an `@` operator (read "ap"). That is, the usual "math" notation $f(x)$ becomes `f @ x` in this language. Moreover, `@` associates to the left, so `g @ x @ y` is parsed as `(g @ x) @ y`, i.e. $g(x)(y)$; $g(h(x))$ should be written as `g @ (h @ x)` – here `(`, `)` is just the usual notation specifying that `h @ x` will be calculated first.

  Function applications work by substituting the formal parameter by the applied value. In the following example, the function on the left-hand side is applied to 5. We thus replace all the occurences of `t` by 5.
- **(Example)** the expression `(\t -> t * 3 + t * t) @ 5` means $f(5)$ where f is the function with $f(t) = t \times 3 + t^2$. Hence

$$\begin{aligned} eval(\texttt{(\t -> t * 3 + t * t) @ 5}) &= eval(\texttt{t * 3 + t * t}\,[\text{where } \texttt{t} = 5]) \text{ (substitute 5 for t)} \\ &= eval(\texttt{5 * 3 + 5 * 5}) \\ &= 40 \end{aligned}$$

To ease your load, we will provide some code for tokenizing the input and simplifying the expression to the core language (called "desugaring") for `hw3_1`. Thus you can focus on the main algorithm. If you are interested in what syntactic extensions are available, please check the provided code.

4.3 Toy Compiler (`hw3_1`)

There will be two kinds of expressions, called *topexpr* and *expr*, respectively. *topexpr* represents any valid expression in our language, and *expr* denotes the part without conditionals and functions. We provide code for *topexpr*, and your job is to help with the *expr* part.

- A *topexpr* can be either
  - An *expr*
  - A conditional expression, `"if"` *expr* `"then"` *expr* `"else"` *expr*
  - An anonymous function, `"\"` *id* `"->"` *topexpr*

  where *id* is a valid variable name (specified below)
- An *expr* can be
  - A non-negative integer n or a variable
  - `"("` *topexpr* `")"`
  - *expr op expr* where *op* is either `"@"`, `"^"`, `"*"`, `"/"`, `"+"`, `"-"`, or `"<="`

Please implement a program `hw3_1` that converts an *expr* expression into a sequence of instructions. The conversion algorithm is the same as the one that converts an infix expression into postfix notation. You should print the instructions to the screen, one per line. We have provided the code for handling *topexpr*.

The following operators, ordered from the highest precedence to the lowest, may be in the expression.

1. `@`: The function application operator.
   - "`f @ x`": f applies to x.
   - "`(\x -> 5) @ (1 + 2)`": `(\x -> ...)` applies to `(1 + 2)`.
   - "`g @ x @ y`": `(g @ x)` applies to `y` where `g @ x` means `g` applies to `x`.
   - "`g @ (h @ x)`": `g` applies to `(h @ x)` where `h @ x` means `h` applies to `x`.
2. `^`: The (integer) exponential operator. This operator **associates to the right**.

3. `*`, `/`: The usual arithmetic operators that **associates to the left**. Please note that `/` is **integer** division, hence `10 / 3` $\mapsto 3$

4. `+`, `-`: The usual arithmetic operators that **associates to the left**.

5. `<=`: A comparison operator that **associates to the left**.

   `e_1 <= e_2` is 1 if $e_1 \leq e_2$ and is 0 if $e_1 > e_2$.

In addition to the operators, the expressions contain non-negative integers and variables. A variable name will begin with a letter or an underscore, followed by an arbitrary number of letters, digits or underscores.

We ask you to implement an algorithm of the following form, where ParseTopExpr is in the provided code. As you can see, parentheses connecting *topexpr* with *expr* should be handled specially. Tokens like "`then`", "`else`" should also be handled specially.

```
function ParseExpr()
    while there are input tokens do
        tok ← PeekToken()          ▷ PeekToken will not remove the token from the buffer
        if tok ∈ { ")", "then", "else" } then
            break
        else if tok = "(" then
            insts ← the output of ParseTopExpr()
            tok ← NextToken()                          ▷ here tok should equal ")"
            output insts
        else
            ...                                        ▷ the expression translation code
        end if
    end while
end function
```

For your information, the provided function ParseTopExpr() roughly works by translating

The input "`if` $expr_{con}$ `then` $expr_{th}$ `else` $expr_{el}$" to the following instructions

k insts. {
- ParseExpr($expr_{con}$)
- "`jz` m k"
- ParseExpr($expr_{th}$)
- "`ret`"

} m insts.

- ParseExpr($expr_{el}$)
- "`ret`"

The input "\ id `->` $topexpr_{body}$" to the following instructions

- "`closure` id n"
- ParseTopExpr($topexpr_{body}$)
- "`ret`"

} n insts.

Otherwise, recursively call ParseExpr().

So another responsibility of yours is to make sure that the ParseExpr() that you implements can be properly called by ParseTopExpr().

For your main job "the expression translation code", Please convert the input expression into postfix form and output according to the following table.

Output Token	Instruction
n (positive integer)	`push` n
"`x`" (variable)	`access x`
`@`	`apply`
`^`	`pow`
`*`	`imul`
`/`	`idiv`
`+`	`add`
`-`	`sub`
`<=`	`setle`

(Example) If your ParseExpr() is working properly with ParseTopExpr(), you should be able to get that the postfix form of the expression

```
g @ (x - 3) / 4
```

is

```
g x 3 - @ 4 /,
```

This is because function application is of the highest precedence, and hence the expression `g @ (x - 3) / 4` means `(g @ (x - 3)) / 4`.
Then, your program should output

```
access g
access x
push 3
sub
apply
push 4
idiv
```

4.4 Grading Details

- Implementation
  - Please note that you are **required** to use the provided code (in `hw3_1`). Otherwise, it is your responsibility to make your program be completely compatible with the provided code.
- Input limits
  - All input [output] should be read from [printed to] the screen. All input expressions are **guaranteed** to be correct in sense that there will be no syntax error
  - There will be at most 500 tokens in the input of `hw3_1`. Variable names are at most 31 characters long.
- Testing (100%)
  - There will be a total of 10 checkpoints of the homework, each requires some (or all) of the features defined above. It is up to your choice to (selectively) skip some features.

No.	Required Feature(s)
1, 2, 3	`<=`, `+`, `-`, `*`, `/`
4, 5, 6	all operators (no parentheses)
7, 8	all except *topexpr* (So the expression inside `(,)` can only be *expr*)
9, 10	all

Submission File (Program) and Written Copy

Please push your program to your repository `<user_name>/dsa14hw3` (on GitHub) before the deadline at 5:30pm on Tuesday (04/22/2014). We will use the latest time that you pushed to the repository as your submission time.

10 of the total points will also depend on how you use *git*, such as whether the commit message is meaningful, whether each commit involves reasonable logical units of the source, etc. Please **DO NOT PUT BINARY FILES** in your repository.

Your repository should contain the following items:

- all the source code for your program.
- a Makefile to compile your code ~~and run your program~~. The TAs will type `make` to compile your source files to one (or two) programs. Then they will use `./hw3_1 < input > output` and (if you choose to do the bonus) `./hw3_2 < input > output` to test your program.
- an optional `README`, anything you want the TAs to read before grading your code

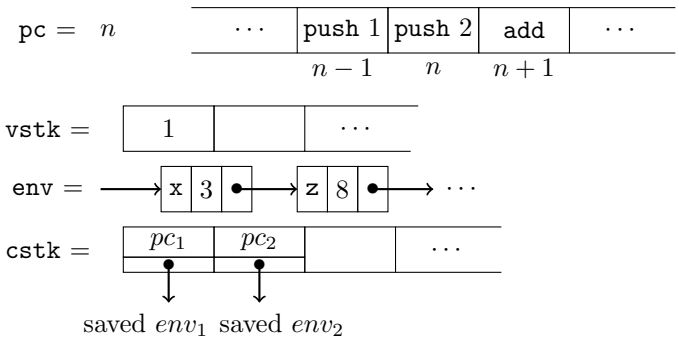
For all the problems that require illustrations, please submit a written (or printed) copy in class or to CSIE R217 before the deadline.

MEDAL USAGE: If you want to use the gold medals for this homework, please visit `http://main.learner.csie.ntu.edu.tw/php/dsa14spring/login_medal.php` and submit the request **before the deadline + 4 days** (4/26).

Bonus: Stack Vector Machine (Not Support Vector Machine) (`hw3_2`)

(Bonus 20%) Please implement a program `hw3_2` that reads the instructions from the screen and runs the compiled instructions. When there are no instructions left, print the value at top of `vstk` (which is guarenteed to be an integer) to the screen.

Our machine consists of 5 components:



A snapshot of the machine

register	name	description
	code	An array storing instructions
`pc`	program counter	Index of the **code** array, denoting the position of the next instruction
`vstk`	value stack	A stack storing value for postfix expression evaluation. A value can either be • An integer n • A triple ("`x`", pc', env') representing a function where `x` is its formal parameter, pc' is the address of the function code and env' is the pointer to the environment saved upon the creation of the function.
`env`	environment	Pointer to the head of the linked-list storing variable names and their values
`cstk`	call stack	A stack of (pc', env') pairs storing the return addresses of function calls

The components of SVM

The instructions of SVM are:

- `add`, `sub`, `imul`, `idiv`, `setle`, `pow`
  - These instructions will pop 2 values from `vstk` and push the result back.
    - `setle` evaluates to 1 if $v_1 \leq v_2$ and 0 otherwise.
    - `pow` evaluates to ${v_1}^{v_2}$. We guarantee that v_2 is non-negative and that ${v_1}^2 + {v_2}^2 \neq 0$.
  - (**Example**) `sub`

	before	after
`pc`	n	$n+1$
`vstk`	$v_2, v_1, u_1, u_2, \ldots$	$(v_1 - v_2), u_1, u_2, \ldots$

- `push` m: Push the integer m to `vstk`
  - (**Example**) `push 8`

	before	after
`pc`	n	$n+1$
`vstk`	$u_1, u_2, \ldots$	$8, u_1, u_2, \ldots$

- `access` <`variable`>: Get the value of <`variable`> from `env` and push it to `vstk`. If there are multiple entries of <`variable`>, push the one that occurs first.
  - (**Example**) `access x`

	before	after
`pc`	n	$n+1$
`vstk`	$u_1, u_2, \ldots$	$13, u_1, u_2, \ldots$
`env`	$(\mathtt{y}, 3) \to (\mathtt{x}, 13) \to \cdots \to (\mathtt{x}, 5) \to \cdots$	(unchanged)

- `closure` <`variable`> m: Create a function, put it on `vstk`, skip the following m instructions.
  - (**Example**) `closure t 4`

	before	after
`pc`	n	$n+1+4$
`vstk`	$u_1, u_2, \ldots$	("`t`", $n+1, p$), $u_1, u_2, \ldots$
`env`	p (pointer)	(unchanged)

    - Note: ("`t`", $n+1, p$) can be viewed as a struct containing a string `"t"`, an integer $n+1$ and a pointer p.

- `apply`: Pop a function and its argument from `vstk`. Call the function. To call a function, we push the current `pc` and `env` to `cstk` and extend the environment with the argument.
  - (**Example**) `apply`

	before	after
`pc`	n	m
`vstk`	$v, (\text{“}\texttt{w}\text{”}, m, q), u_1, u_2, \ldots$	$u_1, u_2, \ldots$
`env`	p (pointer)	$(\texttt{w}, v) \to q$ (q is a pointer)
`cstk`	$addr_1, addr_2, \ldots$	$(n+1, p), addr_1, addr_2, \ldots$

- `ret`: Get the return address and environment from `cstk` and return
  - (**Example**) `ret`

	before	after
`pc`	n	n'
`env`	–	env'
`cstk`	$(n', env'), addr_1, addr_2, \ldots$	$addr_1, addr_2, \ldots$

- `jz` m k: Push the return address $(pc+1)+k$ and the environment to `cstk`. Pop a value v from `vstk` and test if it is zero. If so, skip the following m instructions.
  - (**Example**) `jz 3 6`

	before	after
`pc`	n	if $v \neq 0$ then $n+1$ else $(n+1)+3$
`vstk`	$v, u_1, u_2, \ldots$	$u_1, u_2, \ldots$
`env`	p (pointer)	(unchanged)
`cstk`	$addr_1, addr_2, \ldots$	$(n+1+6, p), addr_1, addr_2, \ldots$