COL226: Programming Languages
II semester 2021-22

Assignment: The VMC machine for the ASTs of WHILE

In a previous assignment you have already converted WHILE programs into abstract syntax trees (AST). It is well-known that given the arity of each operator/construct in the programming language, a program may be transformed into a semantically equivalent bracket-free expression in which all the operators are used in **postfix** form.

In this assignment you need to execute the ASTs on a VMC machine by first transforming each program as a sequence of operations (either as tokens or as string representations of the operators or any other data-type that you find it convenient to convert them to) in post-fix form.

The VMC machine

Consider a simple machine with two stacks (**V** and **C**) with a memory **M** where

V is the **V**alue stack used for decision-making and evaluation,

M is the **M**emory – an array of integer locations (use the integers 0 and 1 for boolean values) representing the locations of the variables in a program and

C is the **C**ontrol stack to store the operations to be performed on **V**.

Configurations

The set Γ of configurations of the VMC machine is represented as triples of the form $\langle V, M, C\rangle$ where V, M and C represent the contents of **V**, **M** and **C** respectively.

Initial Configuration: $\langle \varepsilon, M_0, C_0\rangle$ where

- ε denotes an empty stack
- M_0 denotes the initial state of the memory where all variables (both integer and boolean variables) are initialised to 0, and
- C_0 represents the entire program represented as a single command sequence[1].

Final Configuration: $\langle V, M, \varepsilon\rangle$. But there may be no final configuration if the program does not terminate.

The Stacks: Assume the following signature for each of the stacks in the VMC machine.

```
signature STACK =
sig
    type 'a Stack
    exception EmptyStack
    exception Error of string
    val create: -> 'a Stack
    val push : 'a * 'a Stack -> 'a Stack
    val pop : 'a Stack -> 'a Stack
    val top : 'a Stack -> 'a
    val empty: 'a Stack -> bool
```

[1]We don't include the declarations that precede the body of the program and the program header, since they were part of the symbol table construction in the previous assignment on AST construction of WHILE programs.

```
    val poptop : 'a stack -> ('a * 'a stack) option
    val nth : 'a stack * int -> 'a
    val drop : 'a stack * int -> 'a stack
    val depth : 'a stack -> int
    val app : ('a -> unit) -> 'a stack -> unit
    val map : ('a -> 'b) -> 'a stack -> 'b stack
    val mapPartial : ('a -> 'b option) -> 'a stack -> 'b stack
    val find : ('a -> bool) -> 'a stack -> 'a option
    val filter : ('a -> bool) -> 'a stack -> 'a stack
    val foldr : ('a * 'b -> 'b) -> 'b -> 'a stack -> 'b
    val foldl : ('a * 'b -> 'b) -> 'b -> 'a stack -> 'b
    val exists : ('a -> bool) -> 'a stack -> bool
    val all : ('a -> bool) -> 'a stack -> bool
    val list2stack : 'a list -> 'a stack  (* Convert a list into a stack *)
    val stack2list: 'a stack -> 'a list (* Convert a stack into a list *)
    val toString: ('a -> string) -> 'a stack -> string
end
```

For an explanation of the functions `nth` to `all` look up the definitions of these functions in list signature of SML. Notice that that `toString` function requires a function parameter `a2s:  'a -> string`. All other functions are self-explanatory

Notational Conventions used in this document

- V , M, and C denote respectively the current contents of the **V**alue stack, the **M**emory and the **C**ontrol stack respectively.
- m, n, p denote an integer or boolean value.
- x denotes an integer or boolean variable and $\#x$ denotes the value of the variable as stored in the appropriate memory location (which requires a look-up from the symbol table)
- t, u and v denote truth-values (0 or 1). For any truth value t, $\bar{t}$ denotes its logical complement (1 or 0 resp.).
- b denotes a boolean expression. More generally e denotes an (integer or boolean) expression that needs to be evaluated.
- $\circ$ denotes any of the binary operations on integers and booleans and any binary relational operation.
- $M\{x \leftarrow m\}$ denotes the storing of the value m in the memory location of the variable x.
- c and d denote commands.
- ε denotes the value of an empty stack.
- For any stack value S, $a.S$ denotes the result of pushing an element a on to the stack. In other words, any, stack value of the form $m.C$ denotes that the value m is on the top of the control stack. Similarly $0.c_0.c_1.V$ denotes that 0, c_0 " and c_1 are respectively the top three elements on the Value stack.
- The exception `Error` is raised (along with a suitable message) if there is a run-time error.

Semantics

The operational semantics of the WHILE language for the VMC machine is expressed in the form of transition rules between configurations. Each of the transition rules has been named on the left (which may or may not be useful except as a reference to the rule). The last column is either a condition or a comment

Name	**Source**		**Target**	**Comment**
$E.m$	$\langle V, M, m.C\rangle$	$\longrightarrow$	$\langle m.V, M, C\rangle$	
$E.x$	$\langle V, M, x.C\rangle$	$\longrightarrow$	$\langle \#xV, M, C\rangle$	
$E.\circ 00$	$\langle V, M, m.n.\circ.C\rangle$	$\longrightarrow$	$\langle n.m.V, M, \circ.C\rangle$	
$E.\circ 01$	$\langle V, M, m.x.\circ.C\rangle$	$\longrightarrow$	$\langle \#x.m.V, M, x.\circ.C\rangle$	
$E.\circ 10$	$\langle V, M, x.m.\circ.C\rangle$	$\longrightarrow$	$\langle m.\#x.V, M, \circ.C\rangle$	
$E.\circ 11$	$\langle V, M, x.y.\circ.C\rangle$	$\longrightarrow$	$\langle \#y.\#x.V, M, \circ.C\rangle$	
$E.\circ$	$\langle n.m.V, M, \circ.C\rangle$	$\longrightarrow$	$\langle p.V, M, C\rangle$	$p = m \circ n$
$C.\{\}$	$\langle V, M, \{\}.C\rangle$	$\longrightarrow$	$\langle V, M, C\rangle$	Empty command
$C.:=0$	$\langle V, M, x.e.\texttt{SET}.C\rangle$	$\longrightarrow$	$\langle x.V, M, e.\texttt{SET}.C\rangle$	Order of x and e is reversed
$C.:=1$	$\langle m.x.V, M, \texttt{SET}.C\rangle$	$\longrightarrow$	$\langle V, M\{x \leftarrow m\}, C\rangle$	
$C.;$	$\langle V, M, c.d.\texttt{SEQ}.C\rangle$	$\longrightarrow$	$\langle V, M, c.d.C\rangle$	
$C.ite?$	$\langle V, M, b.c.d.\texttt{ITE}.C\rangle$	$\longrightarrow$	$\langle b.V, M, c.d.\texttt{ITE}.C\rangle$	
$C.ite0$	$\langle 0.V, M, c.d.\texttt{ITE}.C\rangle$	$\longrightarrow$	$\langle V, M, d.C\rangle$	
$C.ite1$	$\langle 1.V, M, c.d.\texttt{ITE}.C\rangle$	$\longrightarrow$	$\langle V, M, c.C\rangle$	
$C.wh?$	$\langle V, M, b.c.\texttt{WH}.C\rangle$	$\longrightarrow$	$\langle c.b.V, M, b.\texttt{WH}.C\rangle$	
$C.wh0$	$\langle 0.c.b.V, M, \texttt{WH}.C\rangle$	$\longrightarrow$	$\langle V, M, C\rangle$	
$C.wh1$	$\langle 1.c.b.V, M, C\rangle$	$\longrightarrow$	$\langle V, M, c.b.c.\texttt{WH}.C\rangle$	

There are two more commands viz. `read` and `write` which access only the memory and are only for user interaction and do not have any effect on the configuration of the stack machine.

What you need to do

1. You need to implement a general-purpose *functional stack* structure called

```
structure FunStack :> STACK =
struct
      ...
end
```

2. Allocate space for the variables in a WHILE program in the **M**emory array of the VMC machine (exactly one index for each integer or boolean variable in a contiguous sequence) and store the address of each variable as an index into the **M**emory.

3. Implement the semantic rules of the VMC machine using the stack operations defined in the signature of the stacks of the VMC machine. Call the main function of this implementation `rules`. In particular define a signature called `VMC` and a structure called `Vmc` which contains the implementation of `rules`. Also define a function `toString` which can output a configuration of the VMC machine as 3-tuple of lists representing the values of **V**, **M** and **C**.

4. For any program expressed as an AST from the previous assignment, translate the Block of commands into post-fix form. Call this function `postfix`.

5. Define a function `execute` which executes the post-fix command sequence in terms of the semantic rules of the VMC machine and outputs the final configuration of the VMC machine.

6. These main functions must have the names mentioned against them. You are free to name auxiliary functions as you please. But since your code may have to be read and understood by humans, please give them names that are suggestive of what those functions do.

Note for all assignments in general

1. Some instructions here may be overridden explicitly in the assignment for specific assignments

2. Upload and submit a single zip file called `<entryno>.zip` where `entryno` is your entry number.
3. You are *not* allowed to change any of the names or types given in the specification/signature. You are not even allowed to change upper-case letters to lower-case letters or vice-versa.
4. The evaluator may use automatic scripts to evaluate the assignments (especially when the number of submissions is large) and penalise you for deviating from the instructions.
5. You may define any new auxiliary functions/predicates if you like in your code besides those mentioned in the specification.
6. Your program should implement the given specifications/signature.
7. You need to think of the *most efficient way* of implementing the various functions given in the specification/signature so that the function results satisfy their definitions and properties.
8. In a large class or in a large assignment, it is not always possible to specify every single design detail and clear each and every doubt. So you are encouraged to also include a README.txt or README.md file containing
    - all the decisions (they could be design decisions or resolution of ambiguities present in the assignment) that you have taken in order to solve the problem. Whether your decision is "reasonable" will be evaluated by the evaluator of the assignment.
    - a description of which code you have "borrowed" from various sources, along with the identity of the source.
    - all sources that you have consulted in solving the assignment.
    - name changes or signature changes if any, along with a full justification of why that was necessary.
9. The evaluator may look at your source code before evaluating it, you must explain your algorithms in the form of comments, so that the evaluator can understand what you have implemented.
10. Do *not* add any more decorations or functions or user-interfaces in order to impress the evaluator of the program. Nobody is going to be impressed by it.
11. There is a serious penalty for code similarity (similarity goes much deeper than variable names, indentation and line numbering). If it is felt that there is too much similarity in the code between any two persons, then both are going to be penalized equally. So please set permissions on your directories, so that others have no access to your programs.
12. To reduce penalties, a clear section called "**Acknowledgements**" giving a detailed list of what you copied from where or whom may be be included.