CS442
Assignment 1

University of Waterloo

Winter 2025

This assignment aims to help you practice and learn OCaml and GNU Smalltalk. Questions 1 and 2 are adaptations of code you saw in Smalltalk and OCaml, respectively, into the opposite language. Question 3 is the most difficult, and is practice for writing interpreters. Question 4 will help you learn to use classes and blocks in Smalltalk.

In this and all assignments, any behavior which we do not explicitly define is not tested, so you may define it however you wish, or allow your program to fail. Make sure though that it actually *is* undefined; ask on Piazza if you're unsure.

Submit all code via UWaterloo submit, e.g.:

```
submit cs442 a1 .
```

This assignment is due on Friday, January 31st, by 12PM *NOON, NOT MIDNIGHT*, Eastern time.

1 Calculator in OCaml

In the introduction to Smalltalk, we wrote a reverse polish notation calculator in Smalltalk. You will now write a similar tool in OCaml.

Write a file, `a1q1.ml`, which defines at least the following functions:

```
push    : float list -> float -> float list
(* e.g. let push lst v = ... *)

binary  : float list -> (float -> float -> float) -> (float * float list) option
(* e.g. let binary lst op = ... *)

add     : float list -> (float * float list) option
sub     : float list -> (float * float list) option
mul     : float list -> (float * float list) option
div     : float list -> (float * float list) option
(* e.g. let add lst = ...;; etc *)

print   : float list -> unit
(* e.g. let print lst = ... *)
```

The Smalltalk version modified a mutable list to act as a stack, but that would be very un-idiomatic in OCaml, so instead, functions to manipulate the stack will return the modified stack, along with their actual answer if any. In Smalltalk, we added elements to the *end* of the list, but in OCaml, pattern matching makes it best for stack elements to be added to the *beginning* of the list, so that's where they go (i.e., the top of the stack is the beginning of the list).

`push` takes a list and a floating point number, and returns the same list with the number added to the front. For instance, `push [2.0; 1.0] 3.0 -> [3.0; 2.0; 1.0]`.

`binary` is the critical function, but we'll start with its less abstract siblings.

`add` adds two numbers on the list and returns a pair containing the sum, and the list, with the two added elements removed and their sum put in their place. The list may not have enough elements, and to handle this case, `add` uses an `option` type for its return: it will either return `Some (...)` or `None`. If there are fewer than two elements in the list, it should return `None`. Hence the return type, `(float * float list) option`, which is an optional pair of float and float list. For instance, `add [3.0; 2.0; 1.0] -> Some (5.0, [5.0; 1.0])`, and `add [1.0] -> None`.

`sub`, `mul`, and `div` behave similarly, with subtraction, multiplication, and division, respectively. Be careful about the order: if you push 2, then you push 3, then you subtract, then the requested operation was `2-3`, *not* `3-2`. For instance, `sub [3.0; 2.0; 1.0] -> Some (-1.0, [-1.0; 1.0])`.

Remember that all of these functions use `floats`, and `floats` in OCaml have different operators than `ints`. Floating-point division by zero is not an error; `div [0.0; 1.0]` should give the same non-error answer as `1.0 /. 0.0` in its result part of the pair.

All of these operators should be implemented in terms of a generic `binary` function, which takes a list and a function as arguments, and applies the function two the first two elements of the list, returning a pair and list exactly like `add`, etc. To understand how `binary` is supposed to be used, here is the complete implementation of `add`:

```
let add l =
    let adder x y = x +. y in
    binary l adder
```

Finally, `print` prints a list. Print in a similar format to Smalltalk's `OrderedCollections`, but using the word `List` instead. For instance, `print [3.0; 2.0; 1.0]` should print:

```
List (3. 2. 1. )
```

Every integer should be listed with a dot, as that's how `printf`'s `%F` formatter prints floats that happen to be integers. Note the space *after* 1. Not having to worry about whether a space is interior makes printing a lot easier.

2 Running sum in Smalltalk

In the guided tour to OCaml, we wrote a small program to keep a running sum and standard deviation of numbers. You will now write a similar class in GNU Smalltalk.

Write a file, `a1q2.st`, which defines the following class:

```
Object subclass: RunningSum [
    updateWith: aNumber.
    mean.
    stdev.
]
```

Like in the OCaml version, it should keep an internal `sum`, `sumSq`, and `samples` count. You are, of course, free to name these anything you wish, as none of them are exposed. You are also free to add any other methods, and in particular should override `new` and use an `init` method to initialize internal state.

The `updateWith:` method takes a number as its argument, which may or may not be a float, so should be converted.

The `mean` method returns the mean of all numbers that have been added with `updateWith:`. The behavior of this method if no samples have been added is not defined.

The `stdev` method returns the standard deviation of all numbers that have been added with `updateWith:`. Consult the OCaml guided tour for the correct algorithm. The behavior of this method if no samples have been added is not defined.

For instance:

```
$ gst floatfix.st a1q2.st -
st> | x |
st> x := RunningSum new.
a RunningSum
st> x updateWith: 1.
a RunningSum
st> x updateWith: 50.
a RunningSum
st> x updateWith: 1000.
a RunningSum
st> x mean.
350.33334
st> x stdev.
459.81906
st>
```

To square a number, use the `squared` method, e.g. `3 squared = 9`. To square root a number, use the `sqrt` method, e.g. `2 sqrt = 1.4142`. The OCaml guided tour implementation uses `**.`, the equivalent of which is the `raisedTo:` method, but the OCaml and Smalltalk implementations have different behaviors for negative numbers, so you're advised to only use `squared` and `sqrt`.

3 BrainBranch in OCaml

Most of what you'll be doing in this course is implementing programming language interpreters. In this question, you'll be implementing an interpreter for a simple programming language.

The programming language you'll be interpreting is a slight modification of an existing language, the name of which is "Brain" followed by a word that rhymes with "luck". A program is provided on the course web page to convert programs in that original language into BrainBranch, but be aware that many programs in that language require input (which we will not implement here), or require particular number formats (we will be using OCaml `int`s).

The modification to the language is simply pre-parsing. The original language has structured loops, which requires parsing for matching brackets. Instead, we will use an OCaml list with branches and jumps like you would find in an assembly language such as MIPS.

A program in BrainBranch is an OCaml `(string * int) array`. That is, it's an array of pairs of strings and ints. The string is a command, and the int is an argument to that command. Because of the nature of OCaml's types, all commands have an argument; only two commands, jump and branch, actually use their arguments. During execution, there is an infinite tape of `int`s, starting as all `0`s. At any given time, there is a "current" value on the tape. Commands in the language can move left or right on that tape, or modify the current value by incrementing or decrementing it. The command to execute is dictated by a program counter. Normally the program counter simply increments by one after running a command, but the `B` (branch) and `J` (jump) commands operate differently.

The BrainBranch commands and their behaviors are as follows:

- `"+"`: Increment the current value on the tape.
- `"-"`: Decrement the current value on the tape.
- `"<"`: Move one space to the left on the tape.
- `">"`: Move one space to the right on the tape.
- `"."`: Output the current value on the tape as an ASCII character.
- `"J"`: Instead of incrementing the program counter by one, add the argument of this command to the program counter. The argument may be negative.

- `"B"`: If the current value is zero, then instead of incrementing the program counter by one, add the argument of this command to the program counter. If the current value is not zero, then do nothing and proceed as normal (i.e., increment the program counter by one).

The program ends when the program counter is beyond the length of the instruction array.

Write a file, `a1q3.ml`, which defines at least the following function:

```
run     : (string * int) array -> unit
(* e.g. let run prog = ... *)
```

The `run` function runs the given BrainBranch program. Note that running a BrainBranch program produces output to standard output, but does not actually return a value. If you implement `run` in terms of some other function or functions (which you should!) and wish to ignore their return and return unit, use the `ignore` function, e.g. `ignore (runSteps ...)`.

Recommendations

You may be scratching your head over exactly how to implement an infinite tape. Well, since the vast majority of the tape will never be visited, you don't actually need to store that space. There is no random access in BrainBranch, so you only need a data structure that can store the value at your current location, as well as all the values you've seen so far in both directions. The easy way to do this is with a triple `(int list * int * int list)`. The elements are everything to the left of the current element, the current element, and everything to the right of the current element, respectively. Here's how you might implement stepping left in this data structure:

```
let stepLeft state =
    match state with
    | (x :: tapeleft, tc, taperight) -> (tapeleft, x, tc :: taperight)
    | ([], tc, taperight) -> ([], 0, tc :: taperight)
```

By matching the case of the left of the tape being non-empty or empty, we can choose to either generate a new 0 or use the existing value.

Note that the above example did not use mutation. Things are generally easier to understand in OCaml if you do *not* use mutation to accomplish this kind of task.

You can do the actual stepping state by state either in an imperative style or in a functional style. To do it in a functional style, you will probably want a simple `step` function that takes an input state and returns the state after running one step. To indicate the program is over, you can make the return an option. Then, running a full program looks something like this:

```
let rec runSteps ins inState =
    match step ins inState with
    | Some outState -> runSteps ins outState
    | None -> inState
```

Note that you will probably want to add more to the `state` than what is given in the `stepLeft` example; both it and `runSteps` are starting points, but may need to be changed to fit your program, depending on how you choose to implement it.

How you output an integer as an ASCII character depends on whether you're using `Base` or not. If you're using `Base`:

```
let () = match Char.of_int value with | Some c -> printf "%c" c | None -> () in
```

If you're not using `Base`:

```
printf "%c" (Char.chr value);
```

Several example programs are provided on the course web page.

4 Sorted tree in Smalltalk

You will write a simple sorted tree type in GNU Smalltalk. In this case, what we mean by a sorted tree is a binary tree in which all elements to the left of a node have a lower value than that node, and all elements to the right of a node have a higher value than that node. The tree will be kept sorted simply by inserting elements at the correct location in the first place; you do not need to perform an actual sort (except for insertion sort).

Write a file, `a1q4.st`, which defines the following class:

```
Object subclass: SortedTree [
    SortedTree class >> new: aNumber.
    add: aNumber.
    displayString.
    printString.
    do: block.
    reduce: block.
]
```

A new `SortedTree` is created with `SortedTree class >> new:`, which takes the root value as its argument. Note that `SortedTree` is the type for a *node* in the sorted tree, so its left and right children should also be `SortedTree`s. A new sorted tree has no left or right child. The likely way to implement this is that `left` and `right` are `nil`, but as instance variables are not exposed, you may implement this in any way you wish, so long as the exposed methods behave correctly.

The `add:` method adds a number to the sorted tree, creating a new node in the appropriate place. The existing tree is never rearranged; instead, you must walk down the tree, choosing left or right based on whether the number is less than or equal to the value in a given node, until you find an empty slot (`left` or `right`, as appropriate, is `nil`), and insert the new node there. If the number is equal to the value of an existing node, nothing is added to the tree. `add:` should return the node you called it on, not the added node. Note that this is the default behavior if you don't include a return statement.

The `displayString` method converts the tree into a string. The format is (*left*) *value* (*right*). *left* is the value returned by `displayString` on the left subtree, and *right* is the value returned by `displayString` on the right subtree. *value* is the value of the current node. If there is no left or right subtree, then that part is excluded from the returned string, along with the parentheses and the space before or after the current node's value. The `printString` method is just an alias to `displayString`, i.e., it should simply return the same result as `displayString` (but do not actually implement it twice). Note that the `printString` method is used by the REPL, so its result will be shown when you interact with a `SortedTree`. For instance:

```
$ gst floatfix.st a1q4.st -
st> | x |
st> x := SortedTree new: 100.
100
st> x add: 50.
(50) 100
st> x add: 101.
(50) 100 (101)
st> x add: 125.
(50) 100 (101 (125))
st> x add: 100.25.
(50) 100 ((100.25) 101 (125))
st> x add: 55.
(50 (55)) 100 ((100.25) 101 (125))
st> x add: -20.
((-20) 50 (55)) 100 ((100.25) 101 (125))
st> x add: -30.
(((-30) -20) 50 (55)) 100 ((100.25) 101 (125))
st> x add: -40.
((((-40) -30) -20) 50 (55)) 100 ((100.25) 101 (125))
st>
```

The `do:` method should run the block for every element in the tree, in ascending order from lowest to greatest value, and return the current tree. For instance, continuing from the tree we built in the above example:

```
st> x do: [:v| (0-v) displayNl.].
40
30
20
-50
-55
-100
-100.25
-101
-125
((((-40) -30) -20) 50 (55)) 100 ((100.25) 101 (125))
st>
```

The `reduce:` method takes a two-argument block, and reduces the tree to a single value. The block should be called in pairs in numeric order. For instance:

```
$ gst floatfix.st a1q4.st -
st> | x |
st> x := SortedTree new: 3.
3
st> x add: 1.
(1) 3
st> x add: 2.
(1 (2)) 3
st> x add: 4.
(1 (2)) 3 (4)
st> x reduce: [:a :b| a-b].
-8
st>
```

Note that $((1 - 2) - 3) - 4 = -8$. Be careful with the first value: the lowest value in the tree is used as the first argument to the block, but is never used as the second argument to the block. This is different from, for instance, Racket's `fold` functions, which take an initial value. You should probably use `do:` to help implementing `reduce:`.

Rights