Homework 1

DUEDATE: February 17, 2020 at 11:59pm

Course Website

Welcome to Programming Languages and Translators

As this course is entirely taught out of OCaml, it's imperative that you become comfortable with functional programming. There is exposition for this assignment writeup, but we also recommend consulting these slides for additional language features.

Setup

To begin install the latest version of OCaml.

To ensure that you installed everything correctly, create a **test.ml** file with the following content:

```
let rec apply_n f n x = if n = 0 then x
        else apply_n f (n - 1) (f x)
in
let plus a b = apply_n ((+) 1) b a in
let mult a b = apply_n ((+) a) b 0 in
let output = plus (mult 2 4) 1 in
(** 2 * 4 + 1 **)
Printf.printf "Output: %d\n" output;
(** Output: 9 **)
```

Now we can test the compilation by running:

```
> ocamlopt -o test.exe test.ml
> ./test.exe
```

There should be no compiler warnings or errors, and the output should be exactly:

```
Output: 9
```

Alternatively you can confirm it with the submission framework described below.

Submission

For submission, we will be using GitHub Classroom. Submit a definition of all the relevant functions collected into a file called *submission_hw1.ml*. To test your submission, we will basically append (using `open`) *submission_hw1.ml* to a suite of unit tests. Although there are many brute force solutions that exists to solve these problems, getting in the habit of writing elegant and maintainable code will pay-off handsomely as the course progresses.

Start by accepting the invitation to the Github Classroom https://classroom.github.com/a/SOKmtOS4 then cloning the template and unit tests from the repository. You should also be able to access it through `githubclassroom` link in Canvas.

DUEDATE: February 17, 2020 at 11:59pm

To test the compilation, run

```
> ocamlopt -o a.out submission_hw1.ml unit_tests_hw1.ml
> ./a.out
```

We have placed dummy implementations to make it compile so you'll have to overwrite those. Please be aware that the `unit_tests_hw1.ml` only indicate a couple of examples and we will be testing more comprehensively to ensure that the submission is correct. In other words, passing the `unit_tests_hw1.ml` doesn't guarantee you a perfect score. You should write your own tests.

You should see the following output:

```
Unit tests:
  Setup:
  Output: 9
Problem 1:
  OOPS
Problem 3.A:
  OOPS
Problem 3.B:
```

```
  OOPS
Problem 4:
  OOPS
Problem 5:
  OOPS
Problem 6:
  OOPS
Problem 7.A:
  OOPS
Problem 7.B:
  OOPS
```

Warm Up: First-class Functions

If you're wondering about good coding standards check out the Jane Street Style Guide. Also, we highly recommend taking a peak at Real World Ocaml. They provide a great introduction to the skills needed for successfully programming in OCaml.

Click for Exposition

Problem 1

Now in a similar fashion to `plus` (using `apply_n`), define `expon` a function that takes the first argument to the power of the second argument.

```
let expon a b = (** YOUR CODE HERE **)
```

Example `expon 4 5 (** => 1024 **)`

Problem 2

How does OCaml know that `=` in `apply_n f n x = ...` is assignment whereas `=` in `n = 0` is comparison?

Write it as a comment in the submission file

Problem 3 Part A

For this problem, let's write a partial sum function called `psum f`. For example, `psum (fun x -> x * x )` equates to the following function of n.

```
f (n) = 1 * 1 + 2 * 2 + ... + n * n
```

There are several twists:

1. We want to always start our index (n) from one, and we allow only positive integers (raise (Failure “Argument Not Positive”).
2. We allow the caller to define what `f` describes the sum.
3. We want to return a function (specifically a function awaiting the value of n)

Sample Unit Test

```
let f x = 3 * x * x + 5 * x + 9 in (** f (x) = 3x^2 + 5x + 9 **)
let partial_sum = psum f in
if partial_sum 1 = 17
&& partial_sum 5 = 285
then print_string "YAY" else raise (Failure "OOPS");
```

NB: Notice that the type of this function is `psum: (int -> int) -> (int -> int)` which is to say a function which takes one arguments a function (which takes a int to int) and returns another function (int to int).

Problem 3 Part B

In OCaml, arithmetic operators like `+` are only for int; there are another set of them for float: `+.`, so what if we want to use floats instead or do products instead of sums. We can use polymorphism to abstract away the actual operation used to accumulate. So now we define the function `partial` which will require an additional input function for how to combine consecutive terms. Note that the input `f` has to take as input an integer representing the position in the series.

```
let rec partial accum f =
  (** YOUR CODE HERE **)
```

Note that partial has type

```
val partial: ('a -> 'a -> 'a) -> (int -> 'a) -> int -> 'a
```

Just as we wished, we can have a version that handles floats,

Sample Unit Test

```
let f x = 3 * x * x + 5 * x + 9 in (** f (x) = 3x^2 + 5x + 9 **)
let g x = (** g (x) = 3x^2 + 5x + 9 **)
  let x = float_of_int x in
  3. *. x *. x +. 5. *. x +. 9.
in

let partial_sum = partial (+) f in
let fpartial_sum = partial (+.) g in
if fpartial_sum 1 = 17.
&& fpartial_sum 5 = 285.
&& partial_sum 1 = 17
&& partial_sum 5 = 285
then print_string "YAY" else raise (Failure "OOPS");
```

Try it on products to confirm you implemented it correctly.

Lists Type Check: Abstract Data Type and Lists

Just as imperative programs have arrays, the go-to workhorse data structure in OCaml are lists. Take a look at the ocaml manual for a great introduction to the `List` module in the Standard Library .

Click for Exposition

Problem 4 Write `maxrun`, a function which returns the length of the longest sublist of negative or non-negative numbers and a list of absolute values of the original list elements. Ensure that this list is in the same order.

To report multiple outputs, use a record. We'll define this particular output record as `run_output`:

```
type run_output =
{  length : int;
   entries : int list;
}
```

Template:

```
let maxrun (l:int list) : run_output =
  (** YOUR CODE HERE **)
```

Sample Unit Test

```
let out = moveNeg [8; -5; 3; 0; 10; -4] in
if out.length = 3
&& out.entries = [8; 5; 3; 0; 10; 4]
then print_string "YAY" else raise (Failure "OOPS");
```

Problem 5 Given the following typed representation of lists which allows only for ints and bools and if an typed expression (texpr), write a `check`, with the type, `val check: texpr -> bool` which checks this. Note that the `*` indicates it's a tuple, and so we have that texpr is a tuple of a type and an expression. Also, the `and` keyword allows the definition of expr to see texpr and vice versa. This is called mutual recursion.

```
type ty = Int | Bool | List of ty
type texpr = ty * expr
and expr =
  | IntLit of int
  | BoolLit of bool
  | Seq of texpr list

let rec check texpr =
  (** YOUR CODE HERE **)
```

Here's an example of a well-typed texpr:

```
let posExample =
  (List (List Int),
    Seq
     [ (List Int, Seq [(Int, IntLit 4); (Int, IntLit 2); (Int, In
       (List Int, Seq []) ])
```

and an example of an incorrectly typed texpr (return false):

```
let negExample =
  (List Bool,
    Seq
      [ (Bool, IntLit 9);
        (List Bool, Seq [(Bool, BoolLit false); (Bool, BoolLit tr
        (Bool, BoolLit true) ])
```

Note you need to check that

- each tuple is correctly typed
- each list is correctly typed
- nested lists are also correct
- if the Seq has an empty list of texpr, any List type is legal

Problem 6

Note that we can actually infer the type of an texpr. If we see `(x, IntLit (8))`, we know `x` should be an Int. So now write a similar function `infer` which overwrites any incorrect types.

```
val infer: texpr -> texpr
```

Note:

- if an empty list is encountered, `raise (Failure "empty list")`
- if a list has inconsistent types, `raise (Failure "inconsistent")`
- list order is preserved

`negExample` from Problem 5 should be inconsistent.

```
let infExample =
  (List Int,
    Seq
      [ (Int, Seq [(Bool, BoolLit false); (Bool, BoolLit true)]);
        (Bool, Seq [(Int, BoolLit true)]) ])
in
infer infExample
```

evaluates to

```
(List (List Bool),
 Seq
   [ (List Bool, Seq [(Bool, BoolLit false); (Bool, BoolLit tru
     (List Bool, Seq [(Bool, BoolLit true)]) ])
```

Basic Calculator

Old HP Calculators, programming languages like Forth and Postscript, and abstract machines like the Java Virtual Machine all evaluate arithmetic expressions using a *stack*. For instance, the expression

```
(2*3)+(3*(4-2))
```

would be written as

```
2 3 * 3 4 2 - * +
```

and evaluated like this (where we show the program being evaluated on the right and the contents of the stack on the left):

```
[ ]           |    2 3 * 3 4 2 - * +
[2]           |    3 * 3 4 2 - * +
[3, 2]        |    * 3 4 2 - * +
[6]           |    3 4 2 - * +
[3, 6]        |    4 2 - * +
[4, 3, 6]     |    2 - * +
[2, 4, 3, 6]  |    - * +
[2, 3, 6]     |    * +
[6, 6]        |    +
[12]          |
```

The goal of the following problems is to write a small interpreter that translates [aexp]s into stack machine instructions.

The instruction set for our stack language will consist of the following instructions:

- SLit n : Push the number [n] on the stack.

- SVar x : Load the identifier [x] from the store and push it on the stack
- SPlus : Pop the two top numbers from the stack, add them, and push the result onto the stack.
- SMinus : Similar, but subtract.
- SMult : Similar, but multiply.
- SDiv : Similar, but with integer division and `raise (Failure "Divide by Zero")`

In OCaml we define the type as:

```
type sinstr =
| SLit of int
| SVar of string
| SPlus
| SMinus
| SMult
| SDiv
```

Click for Exposition

Problem 7

In the following problem, we write a function that can execute a single program represented by a list of stack instructions.

Part A

First we write a function, `prog_exec_inst` that can execute a single program instructions updating the stack. It should take as input a symbol table, a stack represented as a list of numbers (top stack item is the head of the list), and a program represented as a list of instructions, and it should return the stack after executing the program. A symbol table is a map which returns the variable value given a variable name.

The option type defined: `type 'a option = Some of 'a | None` is a standard library utility that allows us to report if an operation fails to compute. Note that the specification leaves unspecified what to do when encountering an [SPlus], [SMinus], or [SMult] instruction if the stack contains less than two elements. Use the option type to handle these cases by returning None for these unspecified cases. But our interpreter will never emit such a malformed program.

```
let prog_exec_instr (symbol_table: int StringMap.t) (inst : sinst
  : (int list) option =
  (** YOUR CODE HERE **)
```

Note that we raise an error when we see a divide by zero but returns `None` if there is unspecified behavior

Sample Unit Test

```
module StringMap = Map.Make(String)
let sym_tbl = StringMap.add "X" 3 StringMap.empty in
let stack = prog_exec_instr sym_tbl (SVar "X") [4] in
let stack' = prog_exec_instr sym_tbl (SMult) [3;4] in
if stack = Some [3;4] && stack' = Some [12]
then print_string "YAY" else raise (Failure "OOPS");
```

Part B

Now use `prog_exec_instr` to write `prog_exec` which evaluates a list of `sinstr` with a symbol table and returns the int option. Note that if the inputs don't correspond to an evaluable expression `prog_exec` returns `None`.

```
let prog_exec (symbol_table: int StringMap.t) (prog : sinstr list
  (** YOUR CODE HERE **)
```

Sample Unit Test

```
module StringMap = Map.Make(String)
let sym_tbl = StringMap.add "X" 3 StringMap.empty in
let prog = [SLit 4; SVar "X"; SMult] in
if (prog_exec sym_tbl prog) = Some (12)
then print_string "YAY" else raise (Failure "OOPS");
```

We will expand on this to this problem on the next assignment.

Acknowledgment

The assignment and reference implementation are designed and implemented by Justin Wong TA for COMS W4115 Programming Languages and Translators, Spring 2020, Columbia University.
