CS 3110 Fall 2017
Prelim

October 12, 2017

Solutions

1. Static and Dynamic Semantics [20 pts]

   Give the types of the following expressions. Or, if the expression is not well-typed, say so.

   (a) [1 pt] `3110`

   Answer:

   int

   (b) [2 pts] `List.map (fun x -> x+1)`

   Answer:

   int list -> int list

   (c) [3 pts] `[[None]; [Some []]]`

   Answer:

   'a list option list list

   (d) [3 pts] `let r x y z = y z x in r`

   Answer:

   'a -> ('b -> 'a -> 'c) -> 'b -> 'c

   Fill in the blank to make each expression evaluate to `42`.

   (e) [2 pts]

```
List.fold_left ______________________ [2; 3; 7]
```

   Answer:

   *(*) 1 or (fun acc _ -> acc) 42*

   (f) [3 pts] In this part, fill in the blank to make `v` evaluate to `42`.

```
type 'a stream =
    Cons of 'a * (unit -> 'a stream)

let hd (Cons(h,  _)) = h
let tl (Cons(_, tf)) = tf ()

let rec gen_stream f x =

  ------------------------------------------

let v = gen_stream (fun x -> x + 1) 40 |> tl |> tl |> hd
```

Answer:

Cons (x, fun () -> gen_stream f (f x))

(g) [3 pts] ◆

```
let f x y = ______________________ in
let g x y z = x z (y z) in
let h = g f f in
h 42
```

Answer:

x

(h) [3 pts] ◆

```
let rec camels are so cool =
  match cool with
  | [] -> are
  | you::magnificent -> camels (so are you) so magnificent
in
let bae = [1; 2; 3; 4; 5; 6] in
let is it ok = it + ok in

let ocaml = camels ______________________ in
ocaml is bae
```

Answer:

21

2. Variants and Pattern Matching [25 pts]

Reverse Polish Notation (RPN), also known as *postfix notation*, was invented by Polish logician Jan Łukasiewicz in 1924. Hewlett-Packard adopted it for the iconic HP-35 scientific calculator in the 1970s. Modern calculators continue to use it: on a Mac, launch the built-in Calculator app and press Command-R to enter RPN mode.

The idea of RPN is simple for computer scientists, who have studied stacks. (Indeed, Dijkstra re-invented it in the 1960s.) Instead of operators being written as infix, they are written as postfix.

operator	infix	postfix	value
addition	`2 + 3`	`2 3 +`	`5`
unary negation	`- 1`	`1 -`	`-1`

An advantage of RPN is that it reduces the need for parentheses in calculations, because operations are simply processed from left to right:

infix	postfix	value
`(2 + 3) * 4`	`2 3 + 4 *`	20
`2 + (3 * 4)`	`2 3 4 * +`	14

To understand those postfix calculations, think of the numbers as being pushed on a stack, and the operators manipulating the stack. For example, in `2 3 + 4 *`:

- `2` is pushed on the stack.
- `3` is pushed on the stack.
- `+` causes the top two elements of the stack to be popped, added, and their sum pushed back on the stack. So now only `5` is on the stack.
- `4` is pushed on the stack.
- `*` causes the top two elements of the stack to be popped, multiplied, and their product pushed back on the stack. So now only `20` is on the stack.

The goal of this problem is to simulate a simple calculator that uses RPN.

(a) [2 pts] Express `3 * (4 + (5 * (6 + 7)))` in RPN. (But don't just simplify it to `207`; your answer must still contain all the original numbers and operators.) You're welcome to take advantage of the commutativity of `+` and `*` if you wish.

Answer:
Two possible answers:

- *3 4 5 6 7 + * + **
- *6 7 + 5 * 4 + 3 **

(b) [3 pts] So far our examples have used integers, but now let's switch to floating-point numbers. Define a type `entry` to represent the calculator user's entry of a floating-point number, the plus key, or the unary negation key.

Answer:

```
type entry =
  | Num of float
  | Plus
  | Negate
```

(c) [2 pts] Define a type `stack` to represent the calculator's stack of floating-point numbers.

Answer:

```
type stack = float list
```

(d) [6 pts] Define a function `calc_entry : stack -> entry -> stack`, where `calc_entry st e` is the stack that results from processing entry `e` with stack `st`, as follows:

- Entry of a floating-point number should push that number on the stack.
- Entry of the plus key should pop the top two numbers off the stack, add them, and push the result on the stack.
- Entry of the unary negation key should pop the top number off the stack, negate it, and push the result on the stack.

If there are not enough numbers on the stack to carry out an operation, the operation should leave the stack unchanged and should not produce any kind of error.

Answer:

```
let calc_entry st e =
  match e, st with
  | Num f, _ -> f::st
  | Plus, x::y::rest -> x+.y :: rest
  | Negate, x::rest -> ~-.x :: rest
  | _ -> st
```

(e) [6 pts] Define a function `calc : entry list -> float option`, where `calc es` is `Some x` if `x` is the top number of the stack that results from processing the entries in `es`. If the resulting stack is empty, `calc es` is `None`. For full credit, your solution must not use the `rec` keyword.

Answer:

```
let calc es =
  match List.fold_left calc_entry [] es with
  | [] -> None
  | x::_ -> Some x
```

(f) [6 pts] Here is a type that represents infix expressions as trees:

```
type iexpr =
  | IConst of float
  | IAdd of iexpr * iexpr
  | INeg of iexpr
```

The leaves of the trees are floating-point numbers, and there are two kinds of nodes, corresponding to the two operators. For example, the infix expression `2. +. (3. +. 4.)` would be represented as the tree `t`, where

```
t = IAdd (IConst 2., IAdd(IConst 3., IConst 4.))
```

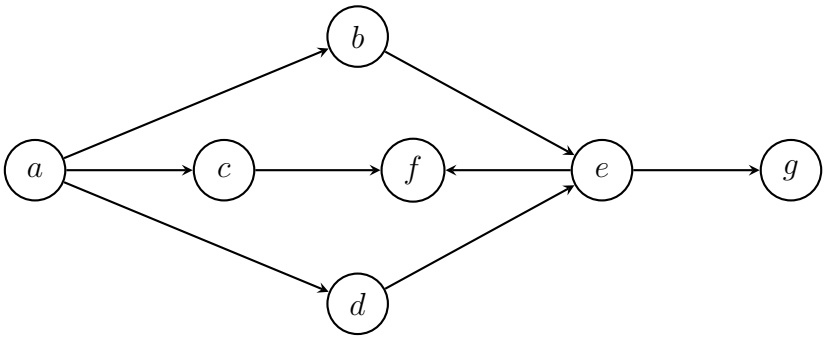
Define a function `rpn_of_iexpr : iexpr -> entry list` that converts an expression tree to RPN. For example, `rpn_of_iexpr t` would result in an `entry list` that represents the RPN calculation `2. 3. 4. +. +.`. The efficiency of your solution is not a concern.

Answer:

```
let rec rpn_of_iexpr = function
  | IConst f -> [Num f]
  | IAdd (e1, e2) -> rpn_of_iexpr e1
                   @ rpn_of_iexpr e2
                   @ [Plus]
  | INeg e -> rpn_of_iexpr e @ [Negate]
```

3. Higher-order Functions [25 pts]

A *directed graph* G is a set V of *vertices* (aka *nodes*) and a set E of *edges* (aka *arcs*), where $E \subseteq V \times V$. For example, here is a directed graph g_1 with $V = \{a, b, c, d, e, f, g\}$ and $E = \{(a, b), (a, c), (a, d), (b, e), (c, f), (d, e), (e, f), (e, g)\}$:



One way to represent a directed graph is to ignore V and simply store E as a list of pairs:

```
type 'a graph = ('a * 'a) list
```

For example, the graph above could be represented as:

```
let g1 =
  [ ("a","b"); ("a","c"); ("a","d"); ("b","e");
    ("c","f"); ("d","e"); ("e","f"); ("e","g")  ]
```

The order of elements in the list is irrelevant, but the order of components in the pair is important, because it indicates the direction of the edge.

Note that the representation we are using does not enable us to model vertices that are completely unconnected by an edge: such a vertex would never appear in the list, hence there is no way to know that it exists.

(a) [4 pts] Define a function `vertices : 'a graph -> 'a list`, such that `vertices g` is the list of all vertices in `g`. The order of elements in the output of `vertices` is irrelevant. All elements in the output list must be unique; there cannot be any duplicates. *Hint: there are* `List` *module functions in the appendix that can make this solution very short.*

Answer:

```
let uniq lst =
  List.sort_uniq Pervasives.compare lst

let vertices g =
  let vs1, vs2 = List.split g in
  uniq (vs1 @ vs2)

(* another solution *)
let vertices g =
  uniq (List.fold_left (fun acc (v1,v2) -> v1::v2::acc) []

(* yet another solution *)
let vertices g =
  let fold_func acc (v1,v2) =
    let new_acc = if List.mem v1 acc then acc else v1::acc
    if List.mem v2 new_acc then new_acc else v2::new_acc
  in
    List.fold_left fold_func [] g
```

Another way to represent a graph is with functions:

```
type 'a graf = 'a -> 'a list
```

(The name `graf` alludes to a graph implemented with functions.) A graph is now a function that, on input v_1, returns a list ℓ of vertices, such that v_2 is an element of ℓ if and only if $(v_1, v_2) \in E$. In other words, the graph *is* the immediate successor function. If a vertex is not in the graph, the function raises the exception `Not_found`. That enables us to model graphs with unconnected vertices, unlike our previous representation `'a graph`.

The following function `succ : 'a -> 'a graf -> 'a list` computes the immediate successors of a vertex:

```
(* [succ v g] is the list of vertices that are successors
 * of vertex [v] in graph [g].  The order of elements
 * in the list is irrelevant, and it contains no duplicates.
 * raises: [Not_found] if [v] is not a vertex in [g]. *)
let succ v g =
  g v
```

The empty graph is the function that always raises `Not_found`, since there are no vertices in it:

```
let empty =
  fun v -> raise Not_found
```

(b) [4 pts] Define a function `is_vertex : 'a -> 'a graf -> bool`, such that `is_vertex v g` is `true` if `v` is a vertex in `g` and `false` otherwise.

Answer:

```
let is_vertex v g =
  try (let _ = succ v g in true) with Not_found -> false

(* another solution *)
let is_vertex v g =
  match succ v g with
  | exception Not_found -> false
  | _ -> true
```

(c) [6 pts] Define a function `add_vertex : 'a -> 'a graf -> 'a graf`, such that `add_vertex v g` adds `v` as a vertex with no successors to `g`. If `v` was already a vertex, the graph is unchanged.

Answer:

```
let add_vertex v g =
  if is_vertex v g then g
  else (fun ver -> if ver=v then [] else succ ver g)

(* another solution *)
let add_vertex v g =
  fun v' ->
    if v = v' then
      (if is_vertex v' g then succ v' g else [])
    else succ v' g
```

(d) [6 pts] Define a function `add_edge : ('a*'a) -> 'a graf -> 'a graf`, such that `add_edge (v1,v2) g` adds an edge from `v1` to `v2` in `g`. If there was already such an edge, the graph is unchanged. As a precondition, you may assume `v1` and `v2` are already vertices in `g`.

Answer:

```
let add_edge (v1,v2) g =
  if List.mem v2 (succ v1 g) then g
  else (fun ver ->
    if ver=v1
    then v2::(succ v1 g) else succ ver g)

(* another solution *)
let add_edge (v1, v2) g =
  fun v ->
    let succs = succ v g in
    if v=v1 && not (List.mem v2 succs)
    then v2::succs
    else succs
```

(e) [5 pts] ◆◆ Define a function `reachable : 'a -> 'a graf -> 'a list`, such that `reachable v g` is the list of all vertices that are reachable from `v` in `g` by following zero or more edges. The order of elements in the output list is irrelevant, and there may not be any duplicates in the list. For example, the vertices reachable from e in graph g_1 at the beginning of this problem are e, f, and g. As a precondition, you may assume that `v` and all vertices you encounter starting from it are in the graph.

Answer:

```
let reachable v g =
  let rec helper lst acc =
  match lst with
  | [] -> acc
  | h::t -> begin
    if List.mem h acc then helper t acc
    else helper t (helper (succ h g) (h::acc))
    end
  in
helper (succ v g) [v]

(* another solution -- DFS *)

let reachable v g =
  let rec dfs g visited frontier =
    match frontier with
    | [] -> visited
    | hd::tl ->
      if List.mem hd visited then dfs g visited tl
      else dfs g (hd::visited) (g hd @ tl)
  in dfs g [] [v]

(* yet another solution -- fixpoint *)

let succ_all vs g =
  let rec loop = function
  | [] -> []
  | v::vs -> succ v g @ loop vs
  in
  uniq (loop vs)

let rec fix f x =
  let x' = f x in
  if x=x' then x' else fix f x'

let reachable v g =
  fix (fun vs -> uniq(vs @ succ_all vs g)) [v]
```

4. Modules [15 pts]

   A pseudorandom number generator (PRNG) can be understood as a functional data abstraction with two operations:

   - **seed**, which takes an integer and produces an abstract *state*; and

- **next**, which takes a state and produces a pseudorandom integer as well as a new state. This operation is deterministic: given the same input, it will always produce the same output.

(The term *pseudorandom* means that the values produced by the generator are not truly random.)

(a) [5 pts] Declare a module type `Prng` for pseudorandom generators.

Answer:

```
module type Prng = sig
  type t
  val seed : int -> t
  val next : t -> t * int
end
```

(b) [5 pts] A *linear congruential generator* (LCG) is a (rather poor) kind of pseudorandom generator. The state of an LCG is an integer. The next value x_n it generates is computed according to the formula

$$x_n = (a \cdot x_{n-1} + c) \bmod m,$$

where a, c, and m are constants, and x_{n-1} is the previous value that was generated. The seed is treated as x_0. For example, if $a = 2$, $c = 3$, $m = 10$, and $x_0 = 5$, then the sequence of pseudorandom values would be $5, 3, 9, 1, 5, 3, 9, 1, 5, \ldots$

(Continued on next page)

Implement a module `Lcg` that matches the `Prng` module type. You may assume the following are already defined and in scope:

```
let a = 1103515245
let c = 12345
let m = 2147483647
```

(These are the constants used by glibc, the GNU standard C library, for its `rand()` function.)

Answer:

```
module Lcg : Prng = struct
  type t = int

  let seed s = s

  let next x =
    let y = (a * x + c) mod m
    in y, y
end
```

(c) [5 pts] The previous part assumed that the constants a, c, and m were already defined. Suppose they were not. Instead, implement a functor `MakeLcg` that takes as input a structure providing those definitions and uses it to produce as output a structure matching `Prng` that implements a linear congruential generator. You may assume that all your code from previous parts of this problem is in scope.

Answer:

```
module type LcgParams = sig
  val a : int
  val c : int
  val m : int
end

module MakeLcg (Params:LcgParams) : Prng = struct
  open Params

  (* have to copy the code from Lcg.
     include doesn't work *)
  type t = int

  let seed s = s

  let next x =
    let y = (a * x + c) mod m
    in y, y
end
```

5. Specifications [15 pts]

The *Borda count* voting system, named after Jean-Charles de Borda (1733-1799), is useful for organizational decision making when building consensus is deemed important. Each voter *ranks* all of the candidates in order from most preferred to least preferred. Based on the ranking, a number of *points* is awarded to each candidate. If there are n candidates in the election, then the candidate ranked first on the ballot receives n points, the candidate ranked second receives $n-1$ points, ..., and the candidate ranked last receives 1 point. The *tally* of the election is the total number of points each candidate receives from all ballots. The *winner* is the candidate who receives the highest number of points. If there is a tie for the highest number of points, then there is no winner.

A programmer has written the following (imperfect) specifications for Borda count voting:

```
type candidate = string
type ballot = candidate list
type tally = (candidate * int) list
```

```
(**
 * [tally bs] is the number of points each candidate received
 * in [bs].
 *)
val tally : ballot list -> tally

(**
 * [winner t] is the candidate who got the most points in [t].
 *)
val winner : tally -> candidate
```

(a) [4 pts] The specification of `winner` overlooks an important possibility mentioned in the description of Borda count, above. What is that possibility? Improve the specification (either the comment or the code) to account for it.

Answer:
The spec does not account for the possibility that there is no winner: perhaps because of a tie, or perhaps because the tally is the empty list. There are at least three mutually-exclusive ways to improve the spec:

- *Add a precondition:*

```
(* requires: there is a unique candidate who
 * receives the highest number of points
 * in [t]. *)
```

- *Add a raises clause:*

```
(* raises: [Failure "no winner"] if there
 * is no winner. *)
```

- *Change the type to* `winner : tally -> candidate option`*, and the post-condition to*

```
(* [winner t] is [Some c] if [c] is the
 * candidate who got the most points in [t],
 * or [None] if there is no such candidate. *)
```

(This wouldn't be acceptable on a homework assignment, because it changes the type, but we'll allow it on this exam.)

(b) [6 pts] The specification of `tally` is missing some information. As a result, at least two implementations of it would be possible, and those implementations would produce different tallies. Briefly describe two such implementations. (You may use natural language and/or pseudocode, rather than OCaml.) Also give an example input that would distinguish the two implementations, as well as the two different outputs. (Note that there are many possible answers to this question; you do not need to find them all.)

Answer:
There are many possible answers. Here are three. The example input and outputs for each naturally depends on how the ambiguity reflected in the other two is resolved.

A possible answer. *The spec doesn't say what order candidates are ranked in a* `ballot`*.*

- ***Implementation 1.*** *Assume that the candidates are ordered most preferred to least preferred.*
- ***Implementation 2.*** *Assume that the candidates are ordered least preferred to most preferred.*

On input `[ ["Alice"; "Bob"] ]`*, implementation 1 would cause an output of* `[ "Alice", 2; "Bob", 1 ]`*, whereas implementation 2 would cause an output of* `[ "Bob", 2; "Alice", 1 ]`*.*

Another possible answer. *The spec doesn't say whether there is any kind of* well-formedness condition*: specifically, whether all ballots have to contain the same set of candidates or even have to have the same length.*

- ***Implementation 1.*** *Raise an exception if there is a ballot whose candidate set or length is different than the rest.*
- ***Implementation 2.*** *Process each ballot without regard for any of the other ballots.*

On input `[ ["Alice"; "Bob"]; ["Alice"] ]`*, implementation 1 would raise an exception, whereas implementation 2 would cause an output of* `[ "Alice", 3; "Bob", 1 ]`*.*

A third possible answer. *The spec doesn't say whether candidate names are case sensitive. (Analogues of this could include whether initials can be used in place of first or middle names, or whether postnominals can be omitted, etc.)*

- ***Implementation 1.*** *Case sensitive.*
- ***Implementation 2.*** *Case insensitive.*

On input `[ ["ALICE"]; ["Alice"] ]`*, implementation 1 would cause an output of* `[ "Alice", 1; "ALICE", 1 ]`*, whereas implementation 2 would cause an output of* `[ "Alice", 2 ]`*.*

(c) [5 pts] ◆ Revise the specification of `tally` to make it unambiguous what the correct output is for any input. This task will potentially require you to address more issues with the specification than whichever you chose to address in the previous part of this problem.

Answer:
The revision needs to address all three issues in the answer above: ordering in the list, whether ballots must be well-formed and what that means, and whether candidate names are case sensitive. Here is one possible answer:

```
(* [tally bs] is an association list representing
 *   a map from candidates to the number of points
 *   they receive in [bs]. Every distinct string
 *   represents a different candidate, e.g.,
 *   "Alice", "alice", and "A." are all different
 *   candidates.  Ballots are interpreted as the
 *   most preferred candidate being at the head
 *   and the least preferred at the end of the list.
 * requires: Every ballot in [bs] has the same
 *   length, contains the same set of candidates,
 *   and there are no duplicates.
 *)
```

This page intentionally left nearly blank. Please do not detach it. Rather, use it as overflow space for any answers that did not fit on a previous page.