

Computer Science 15–212: Final Examination (Sample Solutions)

Monday, May 2, 2011

Name

Name: _____

Andrew ID: _____

Recitation Section: _____

Instructions

- Write your name, Andrew id, and recitation section in the space provided at the top of this page, and write your name at the top of each page in the space provided.
- This is a closed-book, closed-notes, closed-computer examination.
- There are 32 pages in this sample solution set. Solutions for each question follow immediately after the question.
- This examination consists of 6 questions worth a total of 100 points. The point value of each question is given with the question.
- Read each question completely before attempting to solve any part.
- Look at all questions and all parts before starting the exam. Different questions have different levels of difficulty. **Early questions need not be easier than later questions.**
- Write your answers legibly *in the appropriate space provided* on the examination sheets.

Grading

Question	1	2	3	4	5	6	Total
Score							
Maximum	15	20	15	15	20	15	100

In case they are useful to you, here are the types of some common `List` functions:

```
map : ('a -> 'b) -> 'a list -> 'b list
filter : ('a -> bool) -> 'a list -> 'a list
app : ('a -> unit) -> 'a list -> unit
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
exists : ('a -> bool) -> 'a list -> bool
all : ('a -> bool) -> 'a list -> bool
tabulate : int * (int -> 'a) -> 'a list
length : 'a list -> int
nth : 'a list * int -> 'a
hd : 'a list -> 'a
tl : 'a list -> 'a list
```

1 Evaluation and Typing [15 Points]

1. For each of the following expressions, state its *most general* type (using type variables, if necessary), or write “ILL-TYPED” if it does not have a type.

- (a) `3.14+2.18`
- (b) `(3=4)=(5=6)`
- (c) `[1,2,3] :: 4`
- (d) `fn x => print (x ^ "\n")`
- (e) `fn x => (fn y => x)`

2. For each of the following well-typed expressions, determine whether it has a value, and, if not, whether it loops forever or raises an exception. For each item: give its value if it has a value; write “DIVERGES” if it runs forever; or write “FAILS” if it raises an exception.

- (a)

```
let
  val x = let
    val x = 3
    in
      x * x
    end
  in
    x * x
  end
```
- (b)

```
let
  fun I(x):int = I(x)
  in
    I(I)
  end
```
- (c)

```
let
  fun f (n, x::xs) = f (n-1, xs)
    | f (0, xs) = xs
  in
    f (2, [1,2,3])
  end
```
- (d)

```
let
  fun f 0 = 1
    | f n = 2 * f (n-2)
  in
    f 5
  end
```

The next two sets questions refer to the following signature:

```
signature SIG =
sig
  type t
  val zero : t
  val add : t * t -> t
end
```

3. Which of the following structure bindings is well-formed? (Answer “YES” or “NO” for each.)

- (a)

```
structure S :> SIG =
struct
  type t = int
  val zero = 1
  fun add (x, y) = x*y
end
```
- (b)

```
structure S :> SIG =
struct
  type t = real
  val zero = 0
  fun add (x, y) = x+y
end
```
- (c)

```
structure S :> SIG =
struct
  type t = int->int
  val zero = fn x => x
  fun add (f, g) (x) = f(g x)
end
```

4. Consider the following structure binding:

```
structure S :> SIG =
struct
  type t = int list
  val zero = nil
  fun add (x, y) = x @ y
end
```

For each of the following expressions, give its type, or write “ILL-TYPED” if it has no type.

- (a) `S.zero`
- (b) `S.add ([1,2], [3,4])`
- (c) `S.add (S.zero, S.zero)`

Solution to Question 1: Evaluation and Typing

1. (a) `real`
(b) `bool`
(c) `ILL-TYPED`
(d) `string -> unit`
(e) `'a -> 'b -> 'a`
2. (a) `81`
(b) `DIVERGES`
(c) `FAILS`
(d) `DIVERGES`
3. (a) `YES`
(b) `NO`
(c) `YES`
4. (a) `S.t` (**Not** `int list` – that is opaque.)
(b) `ILL-TYPED`
(c) `S.t`

2 Higher-Order Functions [20 Points]

2.1 Coding with Higher-Order Functions [10 Points]

1. Consider the function

```
fun embellish tag name = name ^ tag
```

- (a) What is the type of `embellish`?

```
(* val embellish: _____ *)
```

- (b) Using `embellish`, *without using lambda notation* (i.e., without `fn ... => ...`) define a function `dub` that adds a space and then the title “The Great” after its argument. For example,

```
dub "Alexander" ==> "Alexander The Great"
```

```
val dub = _____
```

2. Imagine applying a *binary infix operator* to only one operand, either left or right, leaving the other operand unspecified. This defines a function of one argument, which we will call a **fone**.

For example, let `(/2.0)` designate the fone formed by supplying the operator `/` with its right argument. Then `(/2.0)` is a function of type `real -> real` that divides its argument by 2.0. Thus `(/2.0)(4.0) = 2.0` and `(/2.0)(0.5) = 0.25`.

Consider the following higher-order functions, `fone1` and `fone2`, which we will use to define fones in SML:

```
fun fone1 x f y = f(x, y)
```

```
fun fone2 f y x = f(x, y)
```

- (a) What are the types of `fone1` and `fone2`?

```
(* val fone1: _____ *)
```

```
(* val fone2: _____ *)
```

- (b) Redefine `dub` using `fone2` instead of `embellish`. Again, do not use lambda notation. You may use `op` in one place.

```
val dub = _____
```

- (c) What are the type and value of the following expression? (You may give the value in equivalent simple form.)

`(fone1 2.0 op/) o (foner op- 1.0)`

Type: _____

Value: _____

- (d) What is the value of the following expression?

`((foner op- 1.0) o (fone1 2.0 op/)) (4.0)`

3. Consider the definition of a function, `inter`, that takes two sets (represented as lists with no duplicate elements) and returns their intersection:

```
fun inter([], ys) = []
  | inter(x::xs, ys) = if x mem ys then x::inter(xs, ys)
                       else inter(xs, ys)
```

Observe that `inter` uses an infix operator `mem` that determines whether its left operand is a member of its right operand.

- (a) Implement the `mem` operator on lists using `fone1` and a higher-order `List` function (along with other arguments). You may assume that the type of the list elements is an equality type. Do not use lambda notation.

```
infix mem;
```

```
fun x mem xs = _____
```

- (b) Redefine `inter` in one line using `foner` and a higher-order `List` function (along with other arguments). Do not use lambda notation. You may use `mem`.

```
fun inter(xs, ys) = _____
```

2.2 Proof about Higher-Order Functions [10 Points]

Recall the following implementation of the function `rev`, for reversing a list:

```
fun rev [] = []
  | rev (x::xs) = (rev xs) @ [x]
```

Recall also the following definition of the function `map`:

```
fun map f [] = []
  | map f (x::xs) = (f x)::(map f xs)
```

Fill in the missing parts of our proof of the following theorem:

Theorem: For all pure functions f , $(\text{map } f) \circ \text{rev} \cong \text{rev} \circ (\text{map } f)$.

[Notes: “pure” means that f has no side-effects and raises no exceptions. Also, to keep the proof simple, you may assume that $f(v)$ produces a value for all values v of the correct type. Finally, recall that \cong means “operationally equivalent.”]

Proof: By the Pointwise Principle, the claimed equivalence holds if

$$((\text{map } f) \circ \text{rev}) \text{ xlist} \cong (\text{rev} \circ (\text{map } f)) \text{ xlist}$$

for all list values `xlist`. So, our proof will be by structural induction on the list `xlist`.

Base Case: `xlist` is `[]`.

Base Case, Need to Show: (fill it in below):

Proof of Base Case (fill it in below):

[please go to the next page]

Inductive Step: `xlist` is of the form `x::xs` for some list `xs`.

Inductive Hypothesis (IH) (fill it in below):

We need to show for arbitrary `x` and `xs`:

$$((\text{map } f) \circ \text{rev}) (x::xs) \cong (\text{rev} \circ (\text{map } f)) (x::xs)$$

In order to prove this, we will reduce the LHS (left-hand side) and RHS (right-hand side) simultaneously and show that they reduce to the same expression.

The reductions appear in the table below. We give some of the steps; please fill in the rest.

Also, in the brackets to the right are the pairs of justifications for each step (one for the LHS, one for the RHS). The two occurrences of “--” indicate no simplification is made for that half of the desired equivalence from the previous step. Please fill in the three blank reasons.

Be sure to indicate where you use the inductive hypothesis (by writing “IH” as a reason).

The proof uses the following lemma. It says that `(map f)` distributes over `@`:

Lemma: $(\text{map } f) (l1 @ l2) \cong ((\text{map } f) l1) @ ((\text{map } f) l2)$

(We will not prove this lemma. You should use it where indicated.)

Proof of the Inductive Step:

LHS	$\cong?$	RHS	Reasons
$((\text{map } f) \circ \text{rev}) (x::xs)$?	$(\text{rev} \circ (\text{map } f)) (x::xs)$	
$(\text{map } f) (\text{rev } (x::xs))$?	$\text{rev}((\text{map } f) (x::xs))$	[o, o]
_____	?	_____	[rev, map]
_____	?	_____	[Lemma, --]
_____	?	_____	[map, rev]
_____	?	_____	[_____, _____]
$((\text{rev} \circ (\text{map } f)) xs) @ [(f x)]$	\cong	$((\text{rev} \circ (\text{map } f)) xs) @ [(f x)]$	[_____, --]

QED

Solution to Question 2: Higher-Order Functions

2.1

1. (a) `(* val embellish: string -> string -> string *)`
 (b) `val dub = embellish " The Great"`
2. (a) `fonel: 'a -> ('a * 'b -> 'c) -> 'b -> 'c`
`foner: ('a * 'b -> 'c) -> 'b -> 'a -> 'c`
 (b) `val dub = foner op^ " The Great"`
 (c) Type: `real -> real`
 Value: A function operationally equivalent to `fn x => 2.0 / (x - 1.0)`
 (d) `~0.5`
3. (a) `infix mem;`
`fun x mem xs = List.exists (fonel x op=) xs`
 (b) `fun inter(xs, ys) = List.filter (foner (op mem) ys) xs`

2.2

Base Case, Need to Show: $((\text{map } f) \circ \text{rev}) [] \cong (\text{rev} \circ (\text{map } f)) []$.

Proof of Base Case:

$$\begin{aligned} ((\text{map } f) \circ \text{rev}) [] &\implies (\text{map } f) (\text{rev } []) \implies (\text{map } f) [] \implies [] \\ (\text{rev} \circ (\text{map } f)) [] &\implies \text{rev} ((\text{map } f) []) \implies \text{rev } [] \implies [] \end{aligned}$$

Inductive Hypothesis: $((\text{map } f) \circ \text{rev}) xs \cong (\text{rev} \circ (\text{map } f)) xs$.

Proof of the Inductive Step:

LHS	$\cong?$	RHS	Reasons
$((\text{map } f) \circ \text{rev}) (x::xs)$?	$(\text{rev} \circ (\text{map } f)) (x::xs)$	
$(\text{map } f) (\text{rev } (x::xs))$?	$\text{rev}((\text{map } f) (x::xs))$	[o, o]
$(\text{map } f) ((\text{rev } xs) @ [x])$?	$\text{rev}((f x)::(\text{map } f) xs)$	[rev, map]
$((\text{map } f) (\text{rev } xs)) @ ((\text{map } f) [x])$?	$\text{rev}((f x)::(\text{map } f) xs)$	[Lemma, --]
$((\text{map } f) (\text{rev } xs)) @ [(f x)]$?	$\text{rev}((\text{map } f) xs) @ [(f x)]$	[map, rev]
$((\text{map } f) \circ \text{rev}) xs @ [(f x)]$?	$((\text{rev} \circ (\text{map } f)) xs) @ [(f x)]$	[o, o]
$((\text{rev} \circ (\text{map } f)) xs) @ [(f x)]$	\cong	$((\text{rev} \circ (\text{map } f)) xs) @ [(f x)]$	[IH, --]

3 Modules [15 Points]

Style counts in this problem. Your answers should be short, taking advantage of higher order function properties and/or simplifying SML syntactic sugar, whenever possible.

3.1 Scalars [5 Points]

Let us model a *scalar* as a type that has two operations defined on it, which we will call `add` and `mult`, as shown in the following signature:

```
signature SCALAR =
sig
  type scalar          (* parameter *)

  val zero : scalar

  val add : scalar * scalar -> scalar
  val mult : scalar * scalar -> scalar
  val equal : scalar * scalar -> bool
end
```

Two additional specifications:

- The function `equal` decides whether its two arguments are the same scalar.
- The value `zero` is a special scalar, with the following abstract invariants: For any scalar `s : scalar` the following four conditions must hold:
 1. `equal(add(s, zero), s) ⇔ true`
 2. `equal(add(zero, s), s) ⇔ true`
 3. `equal(mult(s, zero), zero) ⇔ true`
 4. `equal(mult(zero, s), zero) ⇔ true`

Examples of scalars include the integers, the real numbers, and the rational numbers. The following structure implements integer scalars. Please fill in the blanks.

```
structure IntScalar :> SCALAR where _____
= struct

  type scalar = int

  val zero = _____

  val add = _____

  val mult = _____

  val equal = _____

end
```

3.2 Vector Spaces [10 Points]

Remark: This question is about mathematical vectors not about SML's built-in vectors.

One simple mathematical definition of vectors is as points in n -dimensional space, with coordinates given by some particular type of scalar. For instance, if we consider integer scalars, then the following is a 4-dimensional vector: $[4, 3, 2, 6]$. In addition, vectors have two important operations defined on them, namely vector addition and scalar multiplication:

- Adding two vectors happens coordinate-wise.
For instance, $[4, 3, 2, 6] + [\sim 4, 0, 1, 100] = [0, 3, 3, 106]$.
- Similarly, scalar multiplication happens coordinate-wise.
For instance, $7 * [4, 3, 2, 6] = [28, 21, 14, 42]$.

The following signature captures these basic ideas:

```
signature VECTORSPACE =
sig
  structure Scalar : SCALAR    (* parameter *)

  type vector                  (* abstract *)

  val dim : int                (* strictly positive integer *)
  val origin : vector

  val vadd : vector * vector -> vector
  val smult : Scalar.scalar -> vector -> vector

  val eq : vector * vector -> bool

  exception WrongDim
  val proj : vector * int -> Scalar.scalar  (* may raise WrongDim *)

  val fromList : Scalar.scalar list -> vector  (* may raise WrongDim *)
  val toList : vector -> Scalar.scalar list
end
```

The meanings of the individual components of the signature are:

- `Scalar` is a parameter component of the signature that implements the underlying scalars. These scalars comprise the coordinates of the vectors.
- `vector` is an abstract type representing the vectors.
- `dim` is the dimension of the vector space, i.e., the number of coordinates each vectors has. **Note** that this number can be an arbitrary strictly positive integer, but it is fixed in any particular implementation.
- `origin` is a vector whose coordinates should all be `Scalar.zero`.
- `vadd` should add two vectors coordinate-wise. Conceptually,

$$\text{vadd}([v_1, \dots, v_{\text{dim}}], [w_1, \dots, w_{\text{dim}}]) \leftrightarrow [v_1 + w_1, \dots, v_{\text{dim}} + w_{\text{dim}}].$$

- `smult` should left-multiply each coordinate of the given vector by the given scalar. Conceptually,

$$\text{smult } s [v_1, \dots, v_{\text{dim}}] \mapsto [s * v_1, \dots, s * v_{\text{dim}}].$$

Note the curried form of `smult`.

- `eq` decides whether two vectors are equal. This is the case if and only if corresponding coordinates are equal as scalars.
- `WrongDim` is an exception.
- `proj` is a projection function. `proj(v, i)` should return the *i*th coordinate of vector *v*, or raise the exception `WrongDim` if *i* lies outside the range $1, \dots, \text{dim}$. Conceptually,

$$\text{proj}([v_1, v_2, \dots, v_{\text{dim}}], i) \mapsto v_i.$$

Note that list functions usually count starting from 0, whereas the vector functions in this question count starting from 1.

- `fromList` should take a list of scalars and create a vector. The list should have length `dim`. The order of elements in the list determines the order of vector coordinates. In other words, the first scalar in the list should be the first coordinate of the vector, and so forth. If the list is too long or too short, the function should raise the exception `WrongDim`.
- `toList` should take a vector and return a corresponding list of scalars, again preserving order.

Note: You may find the structures `List` and `ListPair` from the SML Basis Library useful. See page 2 of the exam for some important `List` functions. Here is a reminder with further specs:

```
(* val List.nth : 'a list * int -> 'a
   List.nth(ls, i) returns the ith element of the list ls, counting from 0. *)

(* List.tabulate : int * (int -> 'a) -> 'a list
   List.tabulate(n, f) returns a list of length n equal to
   [f(0), f(1), ..., f(n-1)], created from left to right. *)
```

And here is a quick reminder for `ListPair`:

```
signature LIST_PAIR =
sig
  <...>
  val zip : ('a list * 'b list) -> ('a * 'b) list
  val unzip : ('a * 'b) list -> ('a list * 'b list)
  val map : ('a * 'b -> 'c) -> ('a list * 'b list) -> 'c list
  val app : ('a * 'b -> unit) -> ('a list * 'b list) -> unit
  val foldl : (('a * 'b * 'c) -> 'c) -> 'c -> ('a list * 'b list) -> 'c
  val foldr : (('a * 'b * 'c) -> 'c) -> 'c -> ('a list * 'b list) -> 'c
  val all : ('a * 'b -> bool) -> ('a list * 'b list) -> bool
  val exists : ('a * 'b -> bool) -> ('a list * 'b list) -> bool
end

structure ListPair :> LIST_PAIR = ...
```

The following functor implements vectors using lists. Please fill in the blanks. You should not need more space than that provided.

Caution: The first blank involves a “where type”. Do not say that one structure is the same as some other structure. You must explicitly equate two types.

```

functor VectorSpace(structure S : SCALAR
                    val d : int)
                    :> VECTORSPACE where type _____

= struct

  structure Scalar = _____

  type vector = _____ list

  val dim = d

  val origin = List.tabulate _____

  (* In your definition of vadd, DO NOT USE ANY LAMBDA NOTATION. *)

  val vadd = _____

  val smult = fn s => _____

  val eq = ListPair._____ Scalar._____

  exception WrongDim

  fun proj (v, i) =
    if (i > 0) andalso (i <= dim) then _____
    else raise WrongDim

  fun fromList l =
    if _____ then _____
    else raise WrongDim

  fun toList v = _____

end

```

Solution to Question 3: Modules

3.1

```
structure IntScalar :> SCALAR where type scalar = int
= struct
  type scalar = int
  val zero = 0
  val add = op+
  val mult = op*
  val equal = op=
end
```

3.2

```
functor VectorSpace(structure S : SCALAR
                    val d : int)
  :> VECTORSPACE where type Scalar.scalar = S.scalar
= struct
  structure Scalar = S
  type vector = Scalar.scalar list
  val dim = d
  val origin = List.tabulate (dim, fn _ => Scalar.zero)
  val vadd = ListPair.map Scalar.add
  val smult = fn s => List.map (fn v => Scalar.mult(s, v))
  val eq = ListPair.all Scalar.equal
  exception WrongDim
  fun proj (v, i) =
    if (i > 0) andalso (i <= dim) then List.nth(v, i-1)
    else raise WrongDim
  fun fromList l =
    if (length l) = dim then l
    else raise WrongDim
  fun toList v = v
end
```

4 Searching a Maze [15 points]

4.1 The Problem [0 points]

Consider the classic problem of a maze. We define a maze as a finite set of locations, a start location and a goal location (both in the given set), and a set of transitions between locations. As a simplification, for this question, locations are unit squares within a finite subspace of a two-dimensional grid indexed by integer coordinates. Moreover, transitions can occur at most between non-diagonally adjacent squares.

You will write a function to solve mazes. A solution to a maze is a list of locations from start to goal, in reverse order. Specifically, the maze's goal location is at the head of the list, each location in the list can be reached in one transition from the next location in the list, and the maze's start location is at the end of the list.

You will use the following signature:

```
signature MAZE =
sig
  type pos = int * int    (* (x, y) location in a 2D grid *)
  type map              (* abstract, representing locations in the maze *)

  (* A maze is of the form (map, start, goal). *)
  type maze = map * pos * pos

  (* newmaze(str) generates a new maze, based somehow on the string str.
     We do not care how; just assume it makes reasonable mazes. *)
  val newmaze : string -> maze

  (* (canmove map p1 p2) returns true if both p1 and p2 are valid locations
     on the map and if there is a valid transition from p1 to p2;
     returns false otherwise. *)
  val canmove : map -> pos -> pos -> bool

  (* seek(maze), with maze=(map,start,goal), returns a valid path in
     the maze, if any exists, describing valid transitions from the
     start to the goal. The path is a list of locations, including the
     start and the goal, in reverse traversal order.
     seek(maze) returns nil if no valid path exists.
     Invariants:
     > The start and goal are valid locations on the map.
     > For any two adjacent locations in the path returned, such as
        [..., to, from, ...] ,   canmove map from to ==> true .
     > The returned path has no circularity and no duplicates.
     *)
  val seek : maze -> pos list
end
```

4.2 The Solution [10 points]

Please fill in the blanks in the following structure. You should assume that the functions `newmaze` and `canmove` have been written for you; their specific internal details are irrelevant. Be sure that `seek` finds a path from `start` to `goal` if any exists (returning that path in reverse order), and returns `nil` otherwise.

```

structure Maze :> MAZE =
struct
  type pos = int * int    (* (x, y) location in a 2D grid *)
  type map = (* it would be filled in; you do not need to know the details *)

  (* A maze is of the form (map, start, goal). *)
  type maze = map * pos * pos

  fun newmaze str = (* assume this has been written for you *)
  fun canmove map from to = (* assume this has been written for you *)

  (* val inpath : pos -> pos list -> bool
     (inpath pos path) returns true if pos is already in path, false otherwise. *)

  fun inpath pos = List.exists _____

  (* val distance : pos -> pos -> int
     (distance p1 p2) returns the Manhattan distance between locations p1 and p2. *)
  fun distance (x1,y1) (x2,y2) = Int.abs(x1-x2) + Int.abs(y1-y2)

  (* val sort : (pos -> int) -> pos list -> pos list
     sorts a list of locations ordered by the given function, by insertion sort. *)
  fun sort (eval : pos->int) (poslist : pos list) =
    let
      (* Invariant: the second argument is already sorted *)
      fun ins (x, []) = [x]
        | ins (x, y::ys) = if eval x <= eval y
                          then x::y::ys else y::ins(x,ys)
    in
      List.foldr _____ _____ _____
    end

  (* val neighbors : pos * pos -> pos list
     neighbors is a helper function that lists the four nondiagonal
     neighbors of (x,y) in the underlying 2D grid (these may or may
     not be locations in the map; that doesn't matter here).
     The returned list should be sorted by Manhattan distance from the goal. *)
  fun neighbors ((x,y), goal) =
    sort _____ [(x+1,y), (x-1,y), (x,y-1), (x,y+1)]

```

```
(* val seek : maze -> pos list

seek(maze), with maze=(map,start,goal), returns a valid path in
the maze, if any exists, describing valid transitions from the
start to the goal. The path is a list of locations, including the
start and the goal, in reverse traversal order
(start is at the end of the list, goal is at the head).

seek(maze) returns nil if no valid path exists.
```

Invariants:

- > The start and goal are valid locations on the map.
- > For any two adjacent locations in the path returned (or in pathacc),
such as [..., to, from, ...] , canmove map from to ==> true .
- > The path returned has no circularity and no duplicates.
- > pathacc = here::_ (in helper function sk).

*)

CAUTION: Be sure to use the continuation k somewhere and do so correctly.

```
fun seek (map, start, goal) =
  let
    fun sk (here, pathacc, k) =
      let
        fun tryneigh [] = _____
          | tryneigh (n::ns) =
              if inpath _____ orelse _____
              then _____
              else sk (_____, _____, fn () => _____)
      in
        if here = goal then _____ (* done! *)
        else tryneigh _____
      end
  in
    sk (start, [start], _____)
  end

end (* structure Maze *)
```


Solution to Question 4: Searching a Maze

4.2

```

structure Maze :> MAZE =
struct

  type pos = int * int
  type map = ...
  type maze = map * pos * pos
  fun newmaze str = ...
  fun canmove map from to = ...

  fun inpath pos = List.exists (fn p => p=pos)

  fun distance (x1,y1) (x2,y2) = Int.abs(x1-x2) + Int.abs(y1-y2)

  fun sort (eval : pos->int) (poslist : pos list) =
    let
      fun ins (x, []) = [x]
        | ins (x, y::ys) = if eval x <= eval y
                          then x::y::ys else y::ins(x,ys)
    in
      List.foldr ins [] poslist
    end

  fun neighbors ((x,y), goal) =
    sort (distance goal) [(x+1,y),(x-1,y),(x,y-1),(x,y+1)]

  fun seek (map, start, goal) =
    let
      fun sk (here, pathacc, k) =
        let
          fun tryneigh [] = k ()
            | tryneigh (n::ns) =
              if inpath n pathacc orelse not (canmove map here n)
              then tryneigh ns
              else sk (n, n::pathacc, fn () => tryneigh ns)
        in
          if here = goal then pathacc
          else tryneigh (neighbors (here, goal))
        end
    in
      sk (start, [start], fn () => [])
    end

end (* structure Maze *)

```

4.3

1. `pos * pos list * (unit -> pos list) -> pos list`
2. `pos list -> pos list`
3. It is possible. We need to make any variables that `tryneigh` needs from `sk`'s environment be explicit arguments of `tryneigh`. The additional variables in this case are simply the arguments passed to `sk`, namely `here`, `pathacc`, and `k`. So a reasonable type for `tryneigh` would be

```
pos list * pos * pos list * (unit -> pos list) -> pos list
```

(Any permutation or currying of the components of the tuple above is fine too.)

We *do not* need to make `map`, `start`, or `goal` be additional arguments, since they will still be available to `tryneigh` within `seek` itself.

4. (a) We require either that `pos` be an equality type or that there be some explicit test for location equality. We also need to be able to compute all the neighbors of a location, so would require a graph type that supports this computation.
- (b) The code would not change except perhaps for the test `here = goal` , which might be replaced by a call to an explicit equality function.

5 Recursive Descent Parsing [20 Points]

In this question, you will write a recursive descent parser for a context-free grammar representing regular expressions based on terminal symbols 0 and 1. First, you will consider a context-free grammar without left-recursion that represents regular expressions without Kleene stars. Since the grammar has no left-recursion, one can readily write a recursive descent parser. Subsequently, we augment the grammar with a rule for Kleene stars. The resulting grammar has a left-recursive rule, so instead of implementing a recursive descent parser blindly, you will implement the rule for Kleene star carefully.

5.1 Regular Expressions without Kleene Stars [12 Points]

The grammar representing regular expressions without Kleene stars is as follows:

$$\begin{aligned} S &\rightarrow E ; \\ E &\rightarrow P + E \mid P \\ P &\rightarrow T \cdot P \mid T \\ T &\rightarrow 0 \mid 1 \mid (E) \end{aligned}$$

Non-terminals are S , E , P , and T . Terminal symbols are $;$, $+$, \cdot , 0 , 1 , $($, and $)$. The starting non-terminal is S . Expressions will be lexed into a list of tokens of type `token` shown below. The goal is to implement a function `parse : token list -> exp`, whose job is to translate a list of tokens into an equivalent abstract syntax tree, given by type `exp`, also shown below.

```
datatype token =
  SEMICOLON | PLUS | DOT | ZERO | ONE | LPAREN | RPAREN

datatype exp =
  Sum of exp * exp      (* + *)
  | Concat of exp * exp (* . *)
  | Zero                (* 0 *)
  | One                 (* 1 *)
```

Example:

The regular expression `0.1+(0+1);` should parse as follows:

```
parse [ZERO, DOT, ONE, PLUS, LPAREN, ZERO, PLUS, ONE, RPAREN, SEMICOLON]
  ↪ Sum (Concat (Zero, One), Sum (Zero, One))
```

The parser you are about to implement is an *exception-based recursive descent parser*. An exception-based recursive descent parser tries to expand a non-terminal by one particular grammar rule for the non-terminal. If the expansion fails, the parser should raise an exception. If there are other grammar rules for the non-terminal, an exception handler within the non-terminal's parsing function should handle the exception, and try the next grammar rule. If none of the grammar rules for that non-terminal apply, the final exception raised should percolate outward to the calling function.

Please fill in the blanks in the parser given on the next page. Here is some additional description for some of the functions:

- `parse : token list -> exp`
`parse(toklist)` parses the list `toklist`, returning a single abstract syntax tree. If that is not possible, or if some tokens remain after parsing, `parse` should raise the exception `ParseError`.
- The helper function `consumeToken : token -> token list -> token list` tests whether the first token in the given token list is the specified token:
`(consumeToken t tokens)` returns `List.tl(tokens)` if `List.hd(tokens)` is equal to `t`. Otherwise it raises the exception `ParseError`.
- Functions `S`, `E`, `P`, and `T` correspond to non-terminals S , E , P , and T , respectively. Their types are all `token list -> exp * token list`; they return an abstract syntax tree and a list of tokens (representing the tokens still left after the current parse). They raise the exception `ParseError` if parsing cannot be done. Since `P` is completely analogous to `E`, we omit its implementation.

Hint: In the function `E`, the `let ... in ... end` corresponds to the grammar rule $E \rightarrow P + E$ and the exception handler corresponds to the grammar rule $E \rightarrow P$.

[Please go to the next page]

```

exception ParseError
fun parse tokens =
  let
    fun consumeToken t nil = raise ParseError
      | consumeToken t (t'::tokens') =
        if t = t' then tokens' else raise ParseError

    fun S tokens =
      let
        val (e, tokens') = E tokens
        val tokens'' = consumeToken SEMICOLON tokens'
      in
        _____
      end

    and E tokens =
      let
        val _____ = _____
        val _____ = _____
        val _____ = _____
      in
        _____
      end

    handle ParseError => _____

    and P tokens = (* analogous to E *)

    and T tokens = (Zero, consumeToken ZERO tokens)

      handle ParseError => _____

        handle ParseError =>
          let
            val _____ = _____
            val _____ = _____
            val _____ = _____
          in
            _____
          end

        val (e, tokens') = S tokens
      in
        if _____ then _____ else _____
      end
end

```

5.2 Regular Expressions with Kleene Stars [4 Points]

The new grammar representing all regular expressions is as follows:

$$\begin{aligned} S &\rightarrow E ; \\ E &\rightarrow P + E \mid P \\ P &\rightarrow T \cdot P \mid T \\ T &\rightarrow 0 \mid 1 \mid (E) \mid T^* \end{aligned}$$

The new terminal symbol is *, which denotes Kleene stars. The new definitions of `token` and `exp` are shown below:

```
datatype token =
  SEMICOLON | PLUS | DOT | ZERO | ONE | LPAREN | RPAREN | STAR

datatype exp =
  Sum of exp * exp      (* + *)
| Concat of exp * exp  (* . *)
| Zero                 (* 0 *)
| One                  (* 1 *)
| Star of exp          (* * *)
```

Since the new grammar contains a left-recursive rule, $T \rightarrow T^*$, you will modify your recursive descent parser slightly, in order to implement the function `T`. Please fill in the blanks in the following code for `T`: (the rest of the code is as before)

```
and T' tokens =
  <identical to the code you wrote as a solution for T in Question 5.1>

and T tokens =
  let
    fun consumeSTAR (e, _____) = _____
      | consumeSTAR (e, tokens)    = _____
  in
    _____
  end
```

5.3 Thought Questions [4 Points]

The questions below refer to the first grammar, from Question 5.1. Justify your answers.

1. Which binds more tightly, alternation (+) or concatenation (.) ?

2. Is the grammar right associative or left associative?

(Hint: Think about which of the following parse trees results from parsing $1+1+1$;

$\text{Sum}(\text{Sum}(\text{One}, \text{One}), \text{One})$

$\text{Sum}(\text{One}, \text{Sum}(\text{One}, \text{One}))$

)

Solution to Question 5: Recursive Descent Parsing

5.1

```

fun parse tokens =
  let
    fun consumeToken t nil = raise ParseError
      | consumeToken t (t'::tokens') =
          if t = t' then tokens' else raise ParseError

    fun S tokens =
        let
          val (e, tokens') = E tokens
          val tokens'' = consumeToken SEMICOLON tokens'
        in
          (e, tokens'')
        end

    and E tokens =
        let
          val (p, tokens') = P tokens
          val tokens'' = consumeToken PLUS tokens'
          val (e, tokens''') = E tokens''
        in
          (Sum(p, e), tokens''')
        end
        handle ParseError => P tokens

    and P tokens = (* analogous to E *)

    and T tokens = (Zero, consumeToken ZERO tokens)
      handle ParseError => (One, consumeToken ONE tokens)
      handle ParseError =>
        let
          let
            val tokens' = consumeToken LPAREN tokens
            val (e'', tokens'') = E tokens'
            val tokens''' = consumeToken RPAREN tokens''
          in
            (e'', tokens''')
          end
        end

    val (e, tokens') = S tokens
  in
    if List.null tokens' then e else raise ParseError
  end

```

5.2

```
and T tokens =
  let
    fun consumeSTAR (e, STAR::tokens) = consumeSTAR(Star(e), tokens)
      | consumeSTAR (e, tokens)       = (e, tokens)
  in
    consumeSTAR (T' tokens)
  end
```

5.3

1. Concatenation binds more tightly than alternation since derivation steps involving alternation must occur before derivation steps involving concatenation, in the absence of parentheses, as the grammar shows.
2. The grammar is right-associative. For instance, consider the recursive grammar rule for alternation: $E \rightarrow P + E$. The recursive subexpression E appears to the right of $+$, effectively putting parentheses around right subexpressions. Similarly for concatenation.

6 Computability [15 Points]

A set S is said to be *recursively enumerable*, abbreviated r.e., if there is a computable function which lists its elements. For the purposes of this question, a set is r.e. if its elements can be listed in a (potentially infinite) stream.

Recall that $D_P : t \rightarrow \text{bool}$ is a *decision procedure* for a property P for elements of type t if, for all values v of type t :

1. $D_P(v)$ always has a value;
2. $D_P(v) \leftrightarrow \text{true}$, if property P holds for v ;
3. $D_P(v) \leftrightarrow \text{false}$, if property P does not hold for v .

If such a D_P exists, the property P is said to be *decidable*.

Recall also that $S_P : t \rightarrow \text{bool}$ is a *semi-decision procedure* for a property P for elements of type t if, for all values v of type t :

1. $S_P(v) \leftrightarrow \text{true}$, if property P holds for v ;
2. $S_P(v) \leftrightarrow \text{false}$ or $S_P(v)$ does not halt, if property P does not hold for v .

If such a S_P exists, the property P is said to be *semi-decidable*.

For reference we provide a signature for a `stream` datatype.

```
signature STREAM =
sig
  type 'a stream (* abstract *)
  datatype 'a front = Empty | Cons of 'a * 'a stream

  val delay : (unit -> 'a front) -> 'a stream
  val expose : 'a stream -> 'a front
end
```

6.1 Semi-decidability [3 Points]

Suppose S is an r.e. set whose elements are listed in a stream $\mathbf{s} : \tau$ **stream** defined at top-level. Fill in the blanks to write a semi-decision procedure for membership in the set S . You may assume that τ is an equality type.

Since we can write a semi-decision procedure for membership in set S , S is said to be *semi-decidable*.

```
(* memberS x  $\leftrightarrow$  true, if  $x \in S$ ;
   memberS x  $\leftrightarrow$  false or diverges, if  $x \notin S$ . *)

fun memberS (x :  $\tau$ ) : bool =
  let
    fun memb Empty = _____
      | memb (Cons(z,rest) :  $\tau$  front) =
          _____ orelse _____
  in
    memb _____
  end
```

6.2 Decidability [6 Points]

Now suppose that T is an r.e. set whose complement, denoted T^c , is also r.e. (recall, by definition, $x \in T$ if and only if $x \notin T^c$).

Given top-level streams $\mathbf{t} : \tau$ **stream** and $\mathbf{tcomp} : \tau$ **stream** that list the elements of T and T^c , respectively, write a decision procedure for determining membership in T . Again, you may assume that τ is an equality type.

Since we can write a decision procedure for membership in T , T is said to be *decidable*.

```
(* memberT x  $\leftrightarrow$  true, if  $x \in T$ ;
   memberT x  $\leftrightarrow$  false, if  $x \notin T$ ;
   memberT x halts for all  $x$  of type  $\tau$ . *)

fun memberT (x :  $\tau$ ) : bool =
  let
    fun memb Empty _ = _____
      | memb _ Empty = _____
      | memb (Cons(z,rest)) (Cons(z',rest')) =
          if _____ then true
            else if _____ then false
              else _____
  in
    memb _____
  end
```

6.3 Deciding decidability [6 Points]

As before, let S be an r.e. set and let T be an r.e. set with r.e. complement T^c . Decide whether the following sets are decidable, semi-decidable, or neither. (Give the most precise answer possible. Answer “neither” if it is possible that the set in question might not be decidable or semi-decidable.)

You need only state your conclusion, you need not provide a decision procedure or proof.

Hint: How might the following `interleave` function be useful in writing semi-decision or decision procedures?

```
fun interleave (a : 'a stream) (b : 'a stream) =
  delay(fn () => interleave' (expose a) b)

and interleave' (Empty) (b) = expose b
  | interleave' (Cons(x, ax)) (b) = Cons(x, interleave b ax);
```

1. _____ T^c
2. _____ $S \cup T$
3. _____ $S^c \cup T$
4. _____ The set of all valid ML programs shorter than 100 characters
5. _____ The set of all valid ML programs that halt in finitely many steps
6. _____ The set of all valid ML programs that do not halt in 100 or fewer steps

Solution to Question 6: Computability

6.1

```

fun memberS (x :  $\tau$ ) : bool =
  let
    fun memb Empty = false
      | memb (Cons(z,rest) :  $\tau$  front) =
          x = z orelse memb(expose rest)
  in
    memb (expose s)
  end

```

6.2

```

fun memberT (x :  $\tau$ ) : bool =
  let
    fun memb Empty _ = false
      | memb _ Empty = true
      | memb (Cons(z,rest)) (Cons(z',rest')) =
          if z = x then true
          else if z' = x then false
          else memb (expose rest) (expose rest')
  in
    memb (expose t) (expose tcomp)
  end

```

6.3

1. decidable T^c
2. semi-decidable $S \cup T$
3. neither $S^c \cup T$
4. decidable The set of all valid ML programs shorter than 100 characters
5. semi-decidable The set of all valid ML programs that halt in finitely many steps
6. decidable The set of all valid ML programs that do not halt in 100 or fewer steps