15-150 Summer 2016
Homework 09

Out: Wednesday, 15 June 2016
Due: Tuesday, 21 June 2016 at 23:59 EDT

1 Introduction

In this homework, you will get practice with SML's module system. You will implement several structures from scratch, and will be introducted to the idea of representational invariance. Note that this assignment has more code, and less guidance, than previous homeworks.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at `https://autolab.andrew.cmu.edu`.

In preparation for submission, your `hw/09` directory should contain a file named exactly `hw09.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/09` directory (that contains a `code` folder and a file `hw09.pdf`). This should produce a file `hw09.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw09.tar` file via the "Handin your work" link.

This homework will be partially autograded. When you submit your code some *very* basic tests are run. These are the public tests, and you will immediately see your score on these. After the final submission deadline, we will run a more comprehensive suite of private tests on your code. Your final score will be a function of your public score, private score, and any manual grading we perform. Non-compiling code will automatically receive a 0.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

All the code that you want to have graded for this assignment should be contained in `points.sml`, `funMatrix.sml`, `matrixTest.sml`, `combinations.sml`, `search.sml`, and

`shrub.sml`, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, your code will not compile in our environment.

1.3 Due Date

This assignment is due on Tuesday, 21 June 2016 at 23:59 EDT. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester. You may submit as many times as you would like until the due date.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for every function we asked you write in this assignment, as well as any substantial helper functions for those functions. Recall the five step methodology:

1. In the first line of comments, specify the type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments to be passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns when applied to an argument that satisfies the assumptions in `REQUIRES`).
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n! *)
fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)
val 1 = fact 0
val 6 = fact 3
val 720 = fact 6
```

Please test functions which are part of the signature for a structure or functor in a separate testing structure. Test functions which appear in a structure or functor but are not part of the signature directly underneath the function, according to the five-step method.

For example:

```
signature FOO =
sig
  val fact : int -> int
end

structure Foo : FOO =
struct
  (*  fact : int -> int
   *  REQUIRES:  n >= 0
   *  ENSURES: fact(n) ==> n! *)
  fun fact (0 : int) : int = 1
    | fact (n : int) : int = n * fact(n-1)
end

structure FooTests =
struct
  val 1 = Foo.fact 0
  val 6 = Foo.fact 3
  val 720 = Foo.fact 6
end
```

1.4.1 Emphatic Warning

CM will not return cleanly if any of the files listed in the sources have no code in them. Because we want you to learn how to write modules from scratch, we have handed out a few files that are empty except for a few place holder comments. That means that there are a few files in the `sources.cm` we handed out that are commented out, so that when you first get your tarball `CM.make "sources.cm"` will work cleanly.

You must uncomment these lines as you progress through the assignment! If you forget, it will look like your code compiles cleanly even though it almost certainly doesn't.

2 15 Points

Recall two different ways of representing points in a plane. The Cartesian representation of point is a pair of an x coordinate and a y coordinate. The polar representation of a point is a pair of a radius and an angle in radians. (Note that there are multiple radius-angle pairs that represent the same point in a polar coordinate system. Specifically, adding 2π to the angle, or negating the radius and adding π to the angle does not change the point that is represented.)

Your task will be to write some simple functions for computing different properties of points. These tasks are meant to give you practice with writing and using code in the SML module system.

You will implement the following functions for both implementations:

- `eq : point -> point -> bool`. `eq p1 p2` evaluates to true iff `p1` and `p2` represent the same point.
- `ofCoords : (real * real) -> point`. `ofCoords(x,y)` evaluates to the point representing those coordinates.
- `toCoords : point -> (real * real)`. `toCoords(p)` evaluates to the coordinates represented by `p`.
- `abs : point -> real`. `abs p` evaluates to the absolute value of the point, which is defined as its distance from the origin.
- `dist : point -> point -> real`. `dist p1 p2` evaluates to the distance between points `p1` and `p2`.

Recall the following mathematical facts:

- The distance between Cartesian points (a, b) and (x, y) is $\sqrt{(a-x)^2 + (b-y)^2}$.
- The distance between polar points (r_1, θ_1) and (r_2, θ_2) is $\sqrt{r_1^2 + r_2^2 - 2r_1r_2(\cos(\theta_1 - \theta_2))}$.
- The angle (θ) of a polar coordinate is given by $\arctan(y/x)$

Task 2.1 (4 pts). Implement the `Cartesian` structure in `points.sml` such that it ascribes to `POINTS`. Note that values of type `real` cannot be compared with the built-in equals function. Accordingly, you have been provided with an equality function for values of type `real * real` that may be useful to you. You may also find the function `Math.sqrt` useful.

Task 2.2 (4 pts). Implement the `Polar` structure in `points.sml` such that it ascribes to `POINTS`. You may also find the functions `Math.cos`, `Math.sin`, and `Math.atan2` useful.

Task 2.3 (7 pts). Write tests for your two structures in `TestPoints`. In order to refer to any value in a structure, you must prefix the value's name with the name of the structure it is contained in. For example, if you want to test your implementation of `eq` for `Cartesian`, the syntax to use the function would be `Cartesian.eq`. You must test your functions this way. You will receive no credit if you `open` the structures.

3 Matrices

3.1 Matrix Signature

An $n \times m$ matrix is a rectangular array of elements arranged in n rows and m columns. For this problem, assume that all rows and columns are 0-indexed. We will represent matrices using the abstract type `matrix`. The signature we will use here for matrices is as follows[1].

```
signature VECTOR =
sig

  type elem
  type vector

  val tabulate : (int -> elem) -> int -> vector

  val dotprod : vector * vector -> elem

  val eq : vector * vector -> bool

  val length : vector -> int

  val nth : vector * int -> elem
end


signature MATRIX =
sig

  structure Vector : VECTOR

  type elem = Vector.elem
  type vector = Vector.vector
  type matrix

  val tabulate : (int * int -> elem) -> int -> int -> matrix
  val update : matrix -> (int * int) -> elem -> matrix

  val transpose : matrix -> matrix
  val identity : int -> matrix
  val size : matrix -> (int * int)
```

[1] This signature is provided in the file `MATRIX.sig`.

```
  val add : matrix * matrix -> matrix
  val subtract : matrix * matrix -> matrix

  val row : matrix -> int -> vector
  val col : matrix -> int -> vector

  val mult : matrix * matrix -> matrix

  val eq : matrix * matrix -> bool

end
```

The components of this signature have the following specifications:

- `Vector` is a structure containing information about vectors.
  - `elem` represents the type of elements in a vector (for purposes of this assignment, `elem` will simply be `int`, though very little would have to change for more general rings).
  - `vector` is an abstract type representing the type of a vector.
  - `tabulate f n` creates a vector of dimension `n` where entry `i` is given by `f i`, with $0 \leq$ `i` $<$ `n`. (The *dimension* of a vector is the number of elements in the vector.)
  - `dotprod (v,w)` evaluates to the scalar dot product of two vectors v and w. If v and w are vectors containing the same number of elements n, the dot product of two vectors is defined as
    $$\sum_{i=0}^{n-1} v_i \cdot w_i$$
    For example:
    $$\texttt{dotprod}\ \left(\begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 5 \\ 0 \end{bmatrix} \right) = (2 \cdot 1) + (1 \cdot 5) + (0 \cdot 0) = 7$$
  - `length v` returns the dimension of the vector `v`, i.e., the number of elements in `v`.
  - `eq (v,w)` returns `true` if the vector `v` is equal to the vector `w` and `false` otherwise.
  - `nth (v, i)` returns the i^{th} element of vector `v`, if it exists. Its behavior is not defined if `i` is out-of-bounds.
- `elem` represents the type of elements in the matrix. Note that this is defined to be the same type as the type of the elements in a vector.

- `vector` is an abstract type representing the type of vector in the matrix, e.g. a row or a column. Note that this is defined to be the same type as the `vector` type in the `Vector` structure.
- `matrix` is an abstract type representing the type of your matrix.
- `tabulate f n m` creates an $n \times m$ matrix M where for each row i and column j, the element at m_{ij} is `f(i,j)`.
- `update M (i,j) v` updates the element in M at row i and column j such that $m_{ij} = v$, assuming valid i and j, and returns the new matrix.
- `transpose M` returns the matrix M^T, the *transpose* of M. The *transpose* of a matrix M is a matrix M^T where the columns of M^T are the rows of M. For example:

$$\begin{bmatrix}1 & 2\end{bmatrix}^{\mathrm{T}} = \begin{bmatrix}1\\2\end{bmatrix}$$

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}^{\mathrm{T}} = \begin{bmatrix}1 & 3\\2 & 4\end{bmatrix}$$

$$\begin{bmatrix}1 & 2\\3 & 4\\5 & 6\end{bmatrix}^{\mathrm{T}} = \begin{bmatrix}1 & 3 & 5\\2 & 4 & 6\end{bmatrix}$$

- `identity n` returns the $n \times n$ identity matrix I_n, in which all the elements on the main diagonal are 1 and all other elements are 0. For example:

$$I_3 = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

- `size M` returns `(n,m)` if `M` is an `n` $\times$ `m` matrix.
- `add (A,B)` returns the sum of the matrix A and the matrix B. If `A` and `B` are valid matrices with the same dimensions, the sum is a matrix M such that for each row i and column j, $m_{ij} = a_{ij} + b_{ij}$. For example,

$$\begin{bmatrix}1 & 3\\1 & 0\\1 & 2\end{bmatrix} + \begin{bmatrix}0 & 0\\7 & 5\\2 & 1\end{bmatrix} = \begin{bmatrix}1+0 & 3+0\\1+7 & 0+5\\1+2 & 2+1\end{bmatrix} = \begin{bmatrix}1 & 3\\8 & 5\\3 & 3\end{bmatrix}$$

- `subtract (A,B)` returns the difference of the matrix A and the matrix B. If `A` and `B` are valid matrices with the same dimensions, the difference is a matrix M such that for each row i and column j, $m_{ij} = a_{ij} - b_{ij}$. For example,

$$\begin{bmatrix}1 & 3\\1 & 0\\1 & 2\end{bmatrix} - \begin{bmatrix}0 & 0\\7 & 5\\2 & 1\end{bmatrix} = \begin{bmatrix}1-0 & 3-0\\1-7 & 0-5\\1-2 & 2-1\end{bmatrix} = \begin{bmatrix}1 & 3\\\sim 6 & \sim 5\\\sim 1 & 1\end{bmatrix}$$

- `row M i` returns the i^{th} row of the matrix M in the form of a `vector`.
- `col M j` returns the j^{th} column of the matrix M in the form of a `vector`.
- `mult (A,B)` evaluates to the product of two matrices A and B. If A is an $n \times m$ matrix and B is an $m \times p$ matrix, their product AB is an $n \times p$ matrix whose entries are

$$(AB)_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj}$$

  For example:

$$\begin{bmatrix} 8 & 9 & 9 \\ 5 & -1 & 20 \end{bmatrix} \begin{bmatrix} 10 & 2 \\ 40 & 8 \\ 9 & 0 \end{bmatrix} = \begin{bmatrix} 521 & 88 \\ 190 & 2 \end{bmatrix}$$

  You may find some of the previously implemented functions helpful for implementing this.
- `eq (A,B)` evaluates to `true` if A and B are the same matrix and `false` otherwise, e.g. for each row i and each column j, $a_{ij} = b_{ij}$.

3.2 FunMatrix

This implementation of matrices will represent matrices as functions — `FunMatrix` is conceptually a function that takes a pair (i, j) and returns the element at row i and column j.

However, this doesn't tell you how big the matrix is; you can pass in any integer for (i, j), but we need to know the dimensions since we're working with finite matrices. To do this, we simply include the dimensions of a matrix as part of the `matrix` datatype. In particular, if we have an $n \times m$ matrix where the function `f` represents the matrix function, then we represent it with the value `((n,m),f) : (int * int) * (int -> int -> elem)`.

Note the function `f` may or may not be total, but you should maintain the invariant that it returns a value whenever you pass it indices within the bounds of the matrix.

Task 3.1 (20 pts). Implement the structures `FunVector` and `FunMatrix` in `funMatrix.sml`. The type of `elem` in `FunVector` must be `int` — all your testing functions may assume this to be true.

3.3 Glass-Box Testing

The use of signatures allows us to test our structure implementations via black-box testing, a method of testing where the internal implementation of the code is not visible to the tester. We don't yet have the machinery to do this, so you'll be doing *glass-box testing*: we ask that you write your tests as if you aren't aware of the internal implementation. This is

kind-of, sort-of like black-box testing, except the box is transparent, and we just ask that you don't look inside. Next week, you'll see functors, which can be used for black-box testing.

In `matrixTest.sml`, you will need to test all of the functions you have implemented in `FunMatrix`. In order to refer to a function in a structure, you must prefix the function name with the name of the structure it is contained in; for example, if you want to test your implementation of `mult` for `FunMatrix`, the syntax to call the function would be

```
FunMatrix.mult
```

.

Task 3.2 (10 pts). Test your implementations of `MATRIX` in a structure named `MatrixTests` located in `matrixTest.sml`. Be sure to test every function you implemented `FunMatrix`. We have created a structure called `M` in `MatrixTests` that refers to your `FunMatrix` implementation. So when writing tests use `M` to refer the code you wrote. We should be able to change the definition of `M` to a different structure that implements `MATRIX` without breaking your tests.

You should write tests for every function in the `MATRIX` signature We have also provided a function `fromLList` and `vecFromList` to help write your test cases.

4 Representation Independence

4.1 Motivation

One key advantage of abstract types is that they enable *local reasoning about invariants*: if there is an invariant about the values of an abstract type—e.g. "this tree is a red-black tree"—and all of the operations in a particular implementation of the signature specifying that type preserve that invariant—e.g. "insert creates a RBT when given a RBT"—then any client code using that implementation necessarily maintains the invariant. The reason is that clients can only use the abstract type through the operations given in the signature, so if these operations preserve the invariant all client code must as well.

In this problem, we will investigate a related question, allowing us to reason about several different implementations of the same abstract type. Specifically, we want to know:

> When can we replace one implementation of a signature with another without breaking any client code?

The answer is not as immediate as it may seem. Assuming all types in the signature are abstract, swapping implementations will produce a program that still typechecks; it may or may not, however, be correct.

Informally, the answer is that one can swap two implementations when they behave the same. How do we model such similar behavior? By defining a relation $\mathcal{R}$ between the two implementations that describes which values of one implementation we view as equivalent to values of the other implementation. The relation must respect contextual equivalence. We then show that the relation is preserved by all operations in the signature. In other words, if $\mathtt{v}_1$ is a value in one implementation and $\mathtt{v}_2$ is a value in the other implementation, such that $\mathtt{v}_1$ and $\mathtt{v}_2$ represent the *same* instance (or extensionally equivalent instances)[2] of an abstract type τ, then we write $\mathcal{R}(\mathtt{v}_1, \mathtt{v}_2)$. If f is any function in the signature, then we must show that applying that function to related values produces related values, meaning f cannot behave differently in any way that is observable through the signature. For instance, if $f : \tau * \tau \to \tau$, then we must show that $\mathcal{R}(f(\mathtt{v}_1, \mathtt{u}_1), f(\mathtt{v}_2, \mathtt{u}_2))$ whenever $\mathcal{R}(\mathtt{v}_1, \mathtt{v}_2)$ and $\mathcal{R}(\mathtt{u}_1, \mathtt{u}_2)$.

4.2 Queues

The following implementation is different from the one in lecture, but the high level ideas are the same.

[2] In lecture, we mentioned abstraction functions informally. If we wrote abstraction functions more formally as mathematical functions, then we would say that $\mathcal{R}(\mathtt{v}_1, \mathtt{v}_2)$ iff $AF_1(\mathtt{v}_1) \cong AF_2(\mathtt{v}_2)$ where AF_1 maps values of our first implementation to instances of our abstract type (e.g., lists of integers to queues of integers) and AF_2 maps values of our second implementation to instances of the abstract types (e.g., pairs of lists to queues).

4.2.1 Signature

```
signature QUEUE=
sig
   type queue
   val emp : queue
   val ins : int * queue -> queue
   val rem : queue -> (int * queue) option
end
```

In this signature[3],

- `emp` represents the empty queue.
- `ins` adds an element to the back of a queue.
- `rem` removes the element at the front of the queue and returns it with the remainder of the queue, or `NONE` if the queue is empty.

Taken together, these three values codify the familiar "first-in-first-out" behavior of a queue.

4.2.2 Implementations

We have given two implementations of this signature in the file `queues.sml`.

`LQ` The first implementation represents a queue with a list where the first element of the list is the front of the queue.

New elements are inserted by being appended to the end of the list. Elements are removed by being pulled off the head of the list. If the list is empty, we know that the queue is empty, so the removal fails.

This implementation is slow in that insertion is always a linear time operation—we have to walk down the whole list each time we add a new element.

Note that we also could have chosen to have front of the queue be the last element of the list, but then removal would be linear time and we'd have the same problem—we can't escape the fact that one of these operations will be constant time and the other will be linear.

`LLQ` The second implementation represents a queue with a pair of lists. One list stores the front of the queue, while the other list stores the back of the queue in reverse order. The split between "front" and "back" here can be anywhere in the queue; it depends on the sequence of operations that have been applied to the queue.

New elements are inserted by being put at the head of the reversed back of the queue. Elements are removed in one of two ways:

[3]Provided in `queues.sml`.

1. If the front list is not empty, the front of the queue is its head, so we peel it off and return it.
2. If the front list is empty, we reverse the reversed back list—now bringing it into order—make that the new front list, take an empty list as the back list, and try remove again on the pair of them.

   If both the front and reversed back are empty, we know that the queue is empty, so the removal fails.

   If we assume that reverse is implemented efficiently, this implementation needs to do a linear time operation on removal sometimes but not every time. Therefore, this represents a substantial speed up in the average case over the one-list implementation.[4]

To get an intuition for how these implementations work consider the following actions linked together in sequence, stated formally in `queueEx.sml`:

$$\langle \text{ins 1, ins 2, ins 3, rem, ins 4, rem, rem, rem, rem} \rangle$$

Figure 1 shows the internal state of each representation through this sequence.

4.2.3 Relation

The relation that shows these two implementations are interchangeable flattens the two-lists representation into the one-list representation. We define the relation $\mathcal{R}$ between `int list`s (that reduce to values) and pairs of `int list`s (that reduce to values) by:

$$\mathcal{R}(\texttt{L:int list}, \texttt{(f,b):int list * int list}) \quad \text{iff} \quad \texttt{L} \cong \texttt{f@(rev b)}.$$

$\mathcal{R}$ respects extensional equivalence: if `L` $\cong$ `L'`, `(f,b)` $\cong$ `(f',b')`, and $\mathcal{R}$(`L`, `(f,b)`), then $\mathcal{R}$(`L'`, `(f',b')`).

[4] In particular, you can show that if you can reverse a list in linear time, the two-lists implementation has amortized constant time insert and remove, while the one-list implementation will always have at least one operation that's always linear time. We won't cover amortized analysis in this class, but it's based on the idea of "expensive things that don't happen very often can be considered cheap."

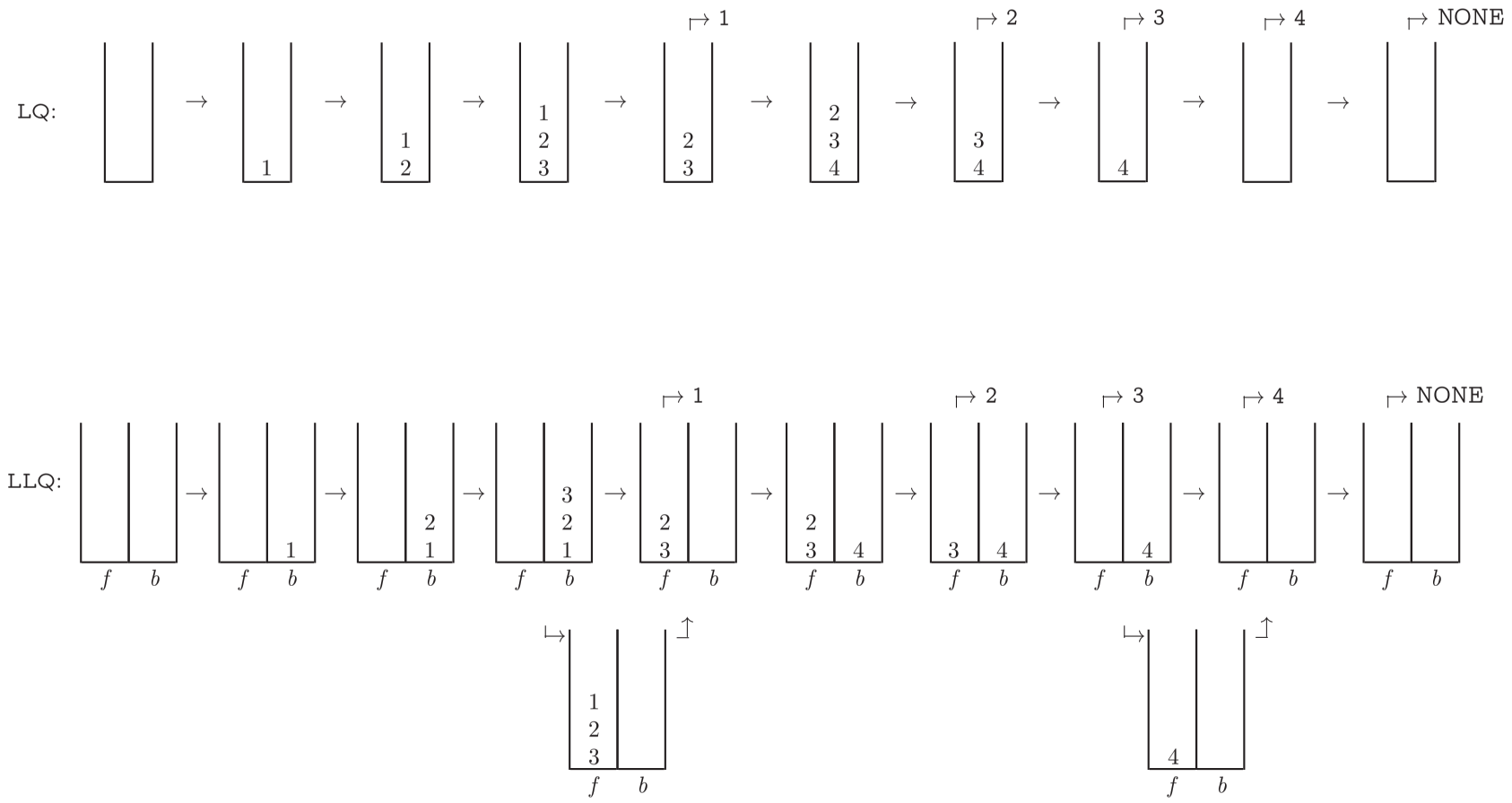


Figure 1: Queue Example

Showing that this relation is respected by both implementations for all the values in `QUEUE` amounts to proving the following theorem:

Theorem 1.

(i.) The empty queues are related:

$$\mathcal{R}(LQ.emp, LLQ.emp)$$

(ii.) Insertion preserves relatedness:

For all `x:int`, `l:int list`, `f:int list`, `b:int list`

$$\textit{If } \mathcal{R}(l,(f,b)), \textit{ then } \mathcal{R}(LQ.ins(x,l), LLQ.ins(x,(f,b)))$$

(iii.) On related queues, removal gives equal integers and related queues:

For all `l:int list`, `f:int list`, `b:int list`, *if* $\mathcal{R}(l,(f,b))$ *then one of the following is true:*

(a) `LQ.rem l` $\cong$ `NONE` *and* `LLQ.rem (f,b)` $\cong$ `NONE`

(b) There exist `x:int`, `y:int`, `l':int list`, `f':int list`, `b':int list`, *such that*

i. `LQ.rem l` $\cong$ `SOME(x,l')`

ii. `LLQ.rem (f,b)` $\cong$ `SOME(y,(f',b'))`

iii. $x \cong y$

iv. $\mathcal{R}(l',(f',b'))$

Task 4.1 (25 pts). Prove Theorem 1. Here are some guidelines, hints, and assumptions:

- Be sure to carefully state your assumptions and goals in each case, especially the two cases where you're proving an implication.
- You may use the following lemmas without proof, but you must carefully cite all uses.

  Lemma 1. *For all* `l1:'a list`, `l2:'a list`, `l3:'a list`,

  `(l1 @ l2) @ l3` $\cong$ `l1 @ (l2 @ l3)`

  Lemma 2. *For all* `l:'a list`, `[] @ l` $\cong$ `l`

  Lemma 3. *For all* `l:'a list`, `l @ []` $\cong$ `l`

  Lemma 4. *For all* `x:int`, `y:int`, `p:int list`, `q:int list`,

  if `x::p` $\cong$ `y::q`, *then* `x` $\cong$ `y` *and* `p` $\cong$ `q`

- You may assume without proof that `@`, `rev`, and all of the functions in both structures are total.

- When you need to step through code, assume that `@` and `rev` are given by the code in the structures. [5]
- We can prove that any call to `LLQ.rem` results in at most one recursive call to `LLQ.rem`, so you do *not* need induction to prove case (iii).

[5] This implementation of `rev` is not the fast tail recursive reverse implementation

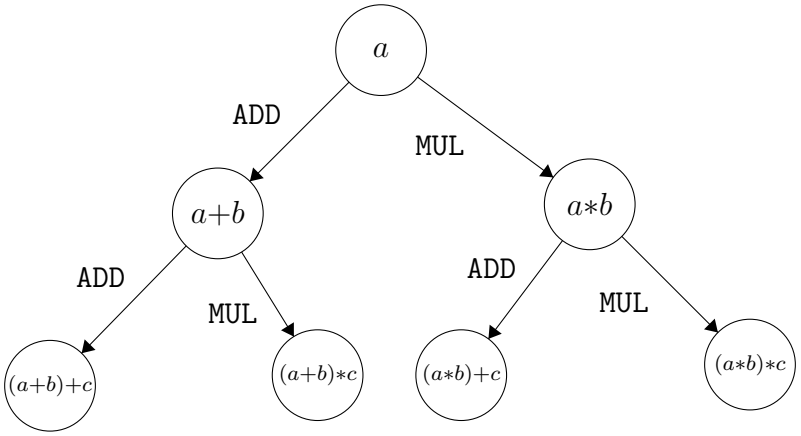
```
foldl op::  []
```

but it is extensionally equivalent to it. All you would need to go from a proof of Theorem 1 for `LLQ` with this slow reverse to a proof for `LLQ` with fast reverse is a proof of their equivalence, so we don't really lose anything. The proof of Theorem 1 is substantially more straight-forward this way, so it's a nice assumption to make.

5 Decision Trees

There are many problems in computer science that can be represented by a decision tree. You have already seen several examples, such as subset sum and matrix combination. In the example of tree pathfinding, the decision tree follows from the `tree` datatype, whereas in subset sum or matrix combination, the decision tree is not as obvious.

For example, one search problem you could do is searching for a way to combine a set of matrices into a target matrix using multiplication and addition. Here is a decision tree for combining three matrices a, b, c, from left to right using addition and multiplication, while keeping track of partial results:



We would say that the tree contains the matrices a, $a + b$, $a * b$, $(a + b) + c$, $(a + b) * c$, $(a * b) + c$, and $(a * b) * c$, since these 7 combinations appear as we make decisions.

You can see that the tree to search for a matrix combination is very similar to the tree to search for the path to a specific tree element, even though these problems may seem like they have nothing in common In particular, if we imagine combining the matrices

At various times in the semester you've had to write or use special-purpose search trees. With structures, we can define and solve an abstract version of this problem, and thus only have to write the code once.

The following signature defines a searchable problem.

```
signature SEARCHABLE =
sig
   type stree
   type elem
   type decision
   type result = decision list
   val branch : stree -> (decision * stree) list
   val atRoot: (elem * stree) -> bool
end
```

- The abstract type `stree` represents the data we are searching through.

- The abstract type `elem` represents an element we are searching for.
- The abstract type `decision` represents the different decisions we could make.
- `result` represents a list of decisions (e.g., go straight, then left, then right; or Add, then Multiply, then Add).
- The function `branch` takes in an `stree` and gives us all possible decisions that we could make from that search tree.
- `atRoot` takes an element and a search tree and tells us if we have found that element at the root of the search tree.

If you are confused as to why `branch` returns a list and not a pair of decisions, remember that we could have more than two decisions at any given point. In the matrix example above, we only considered addition and multiplication. However, if we added subtraction and division to the mix, the decision tree would no longer be binary. In that case, we would need a quaternary decision tree. Since decision problems vary in the number of possible decisions at a given node in a decision tree, it is nice to have a general way of determining all of the decisions and searching through them.

Here is an example of a searchable problem:

```
structure Tree =
struct
  datatype 'a tree = Empty
                   | Node of 'a tree * 'a * 'a tree
  datatype decision = Left | Right
  type elem = int
  type stree = int tree
  type result = decision list
  fun branch Empty = []
    | branch (Node (l,x,r)) = [(Left,l),(Right,r)]
  fun atRoot (j,Node (_,x,_)) = (j = x)
    | atRoot _ = false
end
```

Here, you can see that the `decision` is to go left or right and when we branch, we remember if we turned left or right. Also note that we do not ascribe the `Tree` structure to the `SEARCHABLE` signature. If we did, we would not be able to generate any values of type `tree` since the datatype would not be visible outside of the `Tree` structure.

Your task in this section will be to write a structure searches a decision problem. The way this this will work is you will implement the `Search` structure which contains a structure `P` that implements the `SEARCHABLE` signature. The structure `P` represents the decision problem

that the `Search` structure will decide. Even though you will know the specific implementation of `P`, you must implement the functions in `Search` only using the information given by the `SEARCHABLE` signature. This way, we can change the implementation of the searchable structure that is in `Search` and still have a working implementation of `Search`.

Task 5.1 (15 pts). Implement the function `find_help` in `decision/search.sml`. Remember to use the information given by the `SEARCHABLE` signature when using the structure `P`.

Task 5.2 (7 pts). In `decision/shrub.sml` implement the structure `Shrub` which represents searching for a path in ternary shrubs.

Task 5.3 (8 pts). In `decision/combinations.sml` implement the structure `Combinations` which represents searching for a way to combine integers with addition, subtraction, multiplication, and division. Remember, division can raise the `Div` exception. (Think of the integers as given in a list; the search combines them from left to right in different ways, keeping track of partial results.)

A note on testing: You are still required to write methodology and testing for all of your functions. Once you have completed tasks 5.1, 5.2, and 5.3, try testing `Search` with different implementations of `P`. For example, try: `P = Shrub` or `P = Combinations`. You should not have to change your implementation of `Search` for this to work.