15-150 Spring 2012
Homework 08

Out: 3 April 2012
Due: 11 April 2012, 0900 EST

1 Introduction

This homework will introduce you to the SML module system. So far we've largely ignored how code is grouped or organized, instead relying on careful namespace management. The module system is a very powerful tool to associate pieces of code that work on the same types into structures and produce new structures from old.

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

To submit your solutions, place your `hw08.pdf` and modified `bhrefactor.sml`, `fundict.sml`, and `serializable.sml` files in your handin directory on AFS:

```
/afs/andrew.cmu.edu/course/15/150/handin/<yourandrewid>/hw08/
```

Your files must be named exactly: `hw08.pdf`, `bhrefactor.sml`, `fundict.sml`, and `serializable.sml`. After you place your files in this directory, run the check script located at

```
/afs/andrew.cmu.edu/course/15/150/bin/check/08/check.pl
```

then fix any and all errors it reports.

Remember that the check script is *not* a grading script—a timely submission that passes the check script will be graded, but will not necessarily receive full credit.

Also remember that your written solutions must be submitted in PDF format—we do not accept MS Word files.

Your `bhrefactor.sml`, `fundict.sml`, and `serializable.sml` files must contain all the code that you want to have graded for this assignment and 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, it will not be graded. Modules must ascribe to the specified signatures or they will not be graded.

1.3 Methodology

You must use the four step methodology for writing functions for every function you write on this assignment. In particular, you will lose points for omitting the purpose or tests (unless otherwise specified) even if the implementation of the function is correct.

1.4 Style

We will continue grading this assignment based on the style of your submission as well as its correctness. Please consult course staff, your previously graded homeworks, or the published style guide as questions about style arise.

1.5 Due Date

This assignment is due on 11 April 2012, 0900 EST. Remember that this deadline is final and that we do not accept late submissions.

1.6 The SML/NJ Build System

We will be using several SML files in this assignment. In order to avoid tedious and error-prone sequences of `use` commands, the authors of the SML/NJ compiler wrote a program that will load and compile programs whose file names are given in a text file. The structure `CM` has a function

```
val make: string -> unit
```

`make` reads a file usually named `sources.cm` with the following form:

```
Group is

$/basis.sml
file1.sml
file2.sml
file3.sml
...
```

Loading your code using the REPL is simple. Launch SML in the directory containing your work, and then:

```
$ sml
Standard ML of New Jersey v110.69 [built: Wed Apr 29 12:25:34 2009]
- CM.make "sources.cm";
[autoloading]
[library $smlnj/cm/cm.cm is stable]
[library $smlnj/internal/cm-sig-lib.cm is stable]
...
```

Simply call

```
CM.make "sources.cm";
```

at the REPL whenever you change your code instead of a `use` command like in previous assignments. The compilation manager offers a better interface to the command line. There is less typing and less of an issue with name shadowing between iterations of your code. In short, on this assignment, the development cycle will be:

1. Edit your source files.
2. At the REPL, type

   ```
   CM.make "sources.cm";
   ```

3. Fix errors and debug.
4. If done, consider doing 251 homework; else go to 1.

Be warned that `CM.make` will make a directory in the current working directory called `.cm`. This is populated with metadata needed to work out compilation dependencies, but can become quite large. The `.cm` directory can safely be deleted at the completion of this assignment.

It's sometimes the case that the metadata in the `.cm` directory gets in to an inconsistent state—if you run `CM.make` with different versions of SML in the same directory, for example. This often produces bizarre error messages. When that happens, it's also safe to delete the `.cm` directory and compile again from scratch.

1.6.1 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, 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 Functorization

The Homework 7 support code included a signature `SPACE`, with two separate implementations using reals and rationals, in the modules `RealPlane` and `RatPlane` respectively. You may have noticed that these two files contain many lines of duplicated code, due to the many similarities between the two implementations of plane. Now that we know more about the module system, we can use substructures and functors to clean up the `SPACE` implementations by abstracting duplicated code into a functor.

In the Homework 8 code, the modules `RealPlane` (in `realplane.sml`) and `RatPlane` (in `ratplane.sml`) contain lightly modified versions of the code from HW7. The two differences are:

1. We have cut down the signature to a representative sample to make your task easier.
2. We have placed the `scalar` type and all of its associated operations in a `Scalar` substructure abscribing to the signature `SCALAR`.

This is much better style than using a naming convention (`s_plus`, `s_times`, etc.) to separate out the scalar operations—for example, you can change the naming convention by renaming the module.

If you read through `realplane.sml` and `ratplane.sml`, you will see that much of the code is copy-and-pasted from one file to the other. This is bad—for example, if you fix a bug in one implementation, you have to remember to fix it in the other.

Your task in this problem is to reorganize this code to avoid this duplication. Overall, your solution must have the following form:

- Define a functor `MakePlane` whose argument is all of the code that is *different* between `RealPlane` and `RatPlane`, and whose body is all the code that *the same* in `RealPlane` and `RatPlane`.
- Define arguments `RealPlaneArg` containing all the code specific to `RealPlane`, and similarly for `RatPlaneArg`.

To make this work, you need to define a signature `PLANE_ARGS` that describes the differences, for use as the argument type of `MakePlane`.

Note: most of this problem involves rearranging the existing code in `RealPlane` and `RatPlane` into a different module structure, rather than writing new types and values. As such, there are no methodology (purpose/test) points for this section.

Submit your answer to all of the below questions in the file `bhrefactor.sml`.

Task 2.1 (5%). Define the signature

```
signature PLANE_ARGS
```

which describes the types, functions, and structures that are different between the real and rational implementations of the plane. You should carefully analyze `RealPlane` and `RatPlane` to identify all of the implementation-specific code.

Task 2.2 (7%). Implement the functor

```
functor MakePlane (P : PLANE_ARGS) : SPACE
```

that defines a structure ascribing to `SPACE`. The argument `P` provides all the code that is different between the two implementations; `MakePlane` should contain the code common to both implementations.

Task 2.3 (6%). Implement the structures

```
structure RealPlaneArg : PLANE_ARGS
structure RatPlaneArg : PLANE_ARGS
```

which contain the code specific to the `RealPlane` and the `RatPlane`, respectively.

Task 2.4 (2%). Now, recreate the structures

```
structure RealPlane : SPACE
structure RatPlane : SPACE
```

by applying the functor you wrote.

3 Representation Independence

3.1 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 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 you 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.[1]

Informally, the answer is that you can swap implementations when they behave the same. There is a theorem about SML called *relational parametricity* which justifies the following formalization of this intuition:

> One implementation of a signature can be replaced by another without breaking any client code if and only if there exists a mathematical relation $\mathcal{R}$ between the two implementations of the abstract type that is preserved by all the operations in the signature.

3.1.1 Example: Sequences

For example, consider two modules implementing `SEQUENCE`: `TreeSeq` and `ArraySeq`. A relation $\mathcal{R}$ between the two might relate an array and a tree if and only if the tree flattens to the same list as the array—so the array

```
[0, 1, 2]
```

would be related to the trees

```
Node(Leaf 0, Node (Leaf 1, Leaf 2))
```

and

[1] Many of you experienced this first-hand while working on the tests for Barnes-Hut: the real and rational implementations of the plane ascribe to the same signatures, but have very different behaivours because of precision loss. In this case, the differences were o severe enough that the same client code would often produce different answers, or raise runtime errors and crash, depending on the implementation of the plane.

```
Node(Node(Leaf 0, Leaf 1), Leaf 2)
```

If we show that this relation is preserved by all of the operations in both implementations of `SEQUENCE`—e.g. for `map`, if we show that

$$\text{If } \mathcal{R}(t, a) \text{ then } \mathcal{R}(\texttt{TreeSeq.map}\ f\ t, \texttt{ArraySeq.map}\ f\ a)$$

—then `ArraySeq` can be freely replaced with `TreeSeq` without changing the meaning of, or breaking, any client code. The reason is that every computation performed inside `TreeSeq` is mirrored inside `ArraySeq` in a way formalized by $\mathcal{R}$.

3.2 Queues

Proving such a theorem for the whole `SEQUENCE` signature would be onerous and somewhat outside the scope of this course. Instead, we'll ask you to prove an analagous theorem for a simple signature that formalizes the notion of queues of integers.

3.2.1 Signature

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

In this signature,[2]

- `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" behaviour of a queue.

3.2.2 Implementations

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

[2]Provided in `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:

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,
3. 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.[3]

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

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

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

[3] 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."

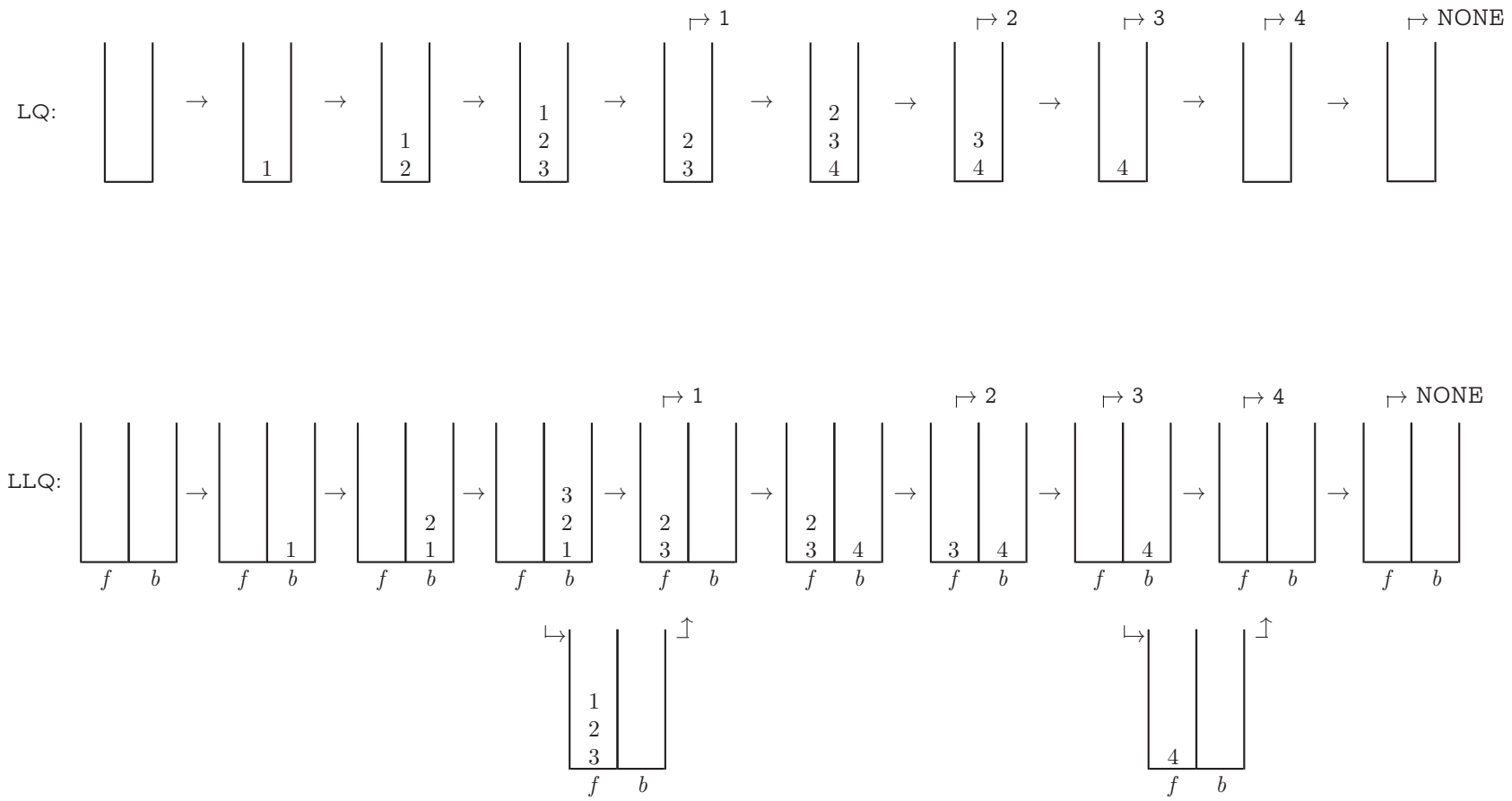


Figure 1: Queue Example

3.2.3 Relation

The relation that shows these two implementations are interchangeable flattens the two-lists representation into the one list representation. Formally, we define a relation between valuable `int lists` and valuable pairs of `int lists` as

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

and $\mathcal{R}$ respects equivalence in that if $l \cong l'$, $(f, b) \cong (f', b')$, and $\mathcal{R}(\texttt{l}, \texttt{(f,b)})$ then $\mathcal{R}(\texttt{l'}, \texttt{(f',b')})$.

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:

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

(ii.) Insertion preserves relatedness:

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

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

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

For all `x:int`, `l:int list`, `f:int list`, `b:int list`, *if* $\mathcal{R}(\texttt{l}, \texttt{(f,b)})$ *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. $\mathcal{R}(\texttt{l'}, \texttt{(f',b')})$

Task 3.1 (25%). 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`,

$$\textit{if } \texttt{x::p} \cong \texttt{y::q}\textit{, then } \texttt{x} \cong \texttt{y} \textit{ and } \texttt{p} \cong \texttt{q}$$

- You may 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[4]

```
infix @
fun (l1 : 'a list) @ (l2 : 'a list) : 'a list =
    case l1
     of [] => l2
      | x::xs => x::(xs @ l2)

fun rev (l : 'a list) : 'a list=
    case l
     of [] => []
      | x::xs => (rev xs) @ [x]
```

- 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).
- When proving an existentially quantified statement, remember to explicitly instantiate each existentially quantified variable.

[4]This implementation of `rev` is not the fast reverse given by

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

or `revTwoPiles` from Lecture, but it is contextually 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.

4 Dictionaries, Third Edition

The following signature slightly extends the notion of dictionaries we've seen in lecture and in lab.[5]

```
signature DICT =
sig
  structure Key : ORDERED
  type 'v dict

  val empty  : 'v dict

  val insert : 'v dict -> (Key.t * 'v) -> 'v dict
  val lookup : 'v dict -> Key.t -> 'v option
  val remove : 'v dict -> Key.t -> 'v dict
  val map    : ('u -> 'v) -> 'u dict -> 'v dict
  val filter : ('v -> bool) -> 'v dict -> 'v dict
end
```

The components of the signature have the following specifications:

- `Key : ORDERED` defines the ordering of the keys in the dictionary.
- `'v dict` is an abstract type representing the type of the dictionary mapping keys of type `Key.t` to values of type `'v`.
- `empty` $\cong ()$
- `insert` $(\mathtt{k}_1 \sim \mathtt{v}_1, \ldots, \mathtt{k}_n \sim \mathtt{v}_n)$ $(\mathtt{k}, \mathtt{v})$

  $$\cong \begin{cases} (k_1 \sim v_1, \ldots, k_i \sim v, \ldots, k_n \sim v_n) & \text{if } \texttt{Key.compare(k,}k_i\texttt{)} \cong \texttt{EQUAL for some } i \\ (k_1 \sim v_1, \ldots, k_n \sim v_n, k \sim v) & \text{otherwise} \end{cases}$$

- `lookup` $(\mathtt{k}_1 \sim \mathtt{v}_1, \ldots, \mathtt{k}_n \sim \mathtt{v}_n)$ `k`

  $$\cong \begin{cases} \texttt{SOME } v_i & \text{if } \texttt{Key.compare(k,}k_i\texttt{)} \cong \texttt{EQUAL for some } i \\ \texttt{NONE} & \text{otherwise} \end{cases}$$

- `remove` $(\ldots, k_{i-1} \sim v_{i-1}, k_i \sim v_i, k_{i+1} \sim v_{i+1}, \ldots)$ k

  $$\cong \begin{cases} (\ldots, k_{i-1} \sim v_{i-1}, k_{i+1} \sim v_{i+1}, \ldots) & \text{if } \texttt{Key.compare(k,}k_i\texttt{)} \cong \texttt{EQUAL for some } i \\ (\ldots, k_{i-1} \sim v_{i-1}, k_i \sim v_i, k_{i+1} \sim v_{i+1}, \ldots) & \text{otherwise} \end{cases}$$

- `map f` $(k_1 \sim v_1, \ldots, k_n \sim v_n) \cong (k_1 \sim (f\ v_1), \ldots, k_n \sim (f\ v_n))$

[5] This signature is provided in the file `dict.sig`.

- `filter p` $(k_1 \sim v_1, \ldots, k_n \sim v_n)$ is equivalent to the dictionary containing all and only those $k_i \sim v_i$ such that (`p` v_i') $\cong$ `true`.

Task 4.1 (20%). Write a functor `FunDict` in `fundict.sml` that takes a structure ascribing to the `ORDERED` signature and yields a structure ascribing to the above `DICT` signature using the following definition for the type `'v dict`:

```
datatype 'v func = Func of (Key.t -> 'v option)

type 'v dict = 'v func
```

That is, we choose to represent a dictionary as a function from keys to value options which, when applied to a key `k`, evaluates to `SOME v` iff `k` maps to `v` in the dictionary it represents.

The file `dictclient.sml` has some tests for the components of the `FunDict` functor. You can type `use "dictclient.sml"` at the REPL after you call `CM.make` to run these tests. Therefore, you do not need to write tests or examples for this section.

5 Serialization

5.1 Introduction

One common programming task is *serialization*: transforming data into a form where it can be written out and later read back in. This is useful if you want data to persist across different runs of your program by storing it in a file, or if you want to send data across a network.[6] It's easy to write a string to a file or to send a string over the network, so we won't take this problem any farther than producing a string from data and vica-versa.

In this problem, you will define a small serialization library, focused on capturing the notion of data that can be serialized with the typeclass

```
signature SERIALIZABLE =
sig
    type t

    (* invariant: For all v, s: read (write v ^ s) == SOME (v , s) *)
    val write : t -> string
    val read : string -> (t * string) option
end
```

That is: a type `t` is serializable only if it supports `read` and `write` operations that convert it to and from a `string`, which interact appropriately. "Appropriately" means that if you read from a string whose prefix was constructed by `write`, then `read` returns the value that was written, along with any suffix. More formally:

> For all `v:t` and `s:string`, `read (write v ^ s)` $\cong$ `SOME (v, s)`

For example, given `S : SERIALIZABLE`, if a web server sends a string produced by `S.write v` to a browser, and then the browser calls `S.read` on that string, the spec says that the client will recover the value the server intended. Note that this spec allows `read` to have aribtrary behavior when applied to a string that does not have a prefix produced by `write`—e.g., we assume that the string is not corrupted during transmission.

5.2 Utility Structure

To minimize the amount of tedious parsing code you need to write, we've provided an implementation of the following signature in the handout code. Be sure to understand these functions and use them when you can: they should make your code a lot cleaner.

[6] Check out http://en.wikipedia.org/wiki/Marshalling_(computer_science), http://en.wikipedia.org/wiki/Serialization for a lot more information.

This signature lives in `util.sig`. We have provided an implementation for you in `util.sml`. You should understand the signature and freely use the functions the structure ascribing to it provides, but you do not need to understand their implementation.[7]

```
signature UTIL =
sig
  (* peelOff (s1,s2) == SOME s' if s2 == s1 ^ s'
   *                 == NONE otherwise
   *
   * Ex:
   *   peelOff ("a","a")  == SOME("")
   *   peelOff ("a","ab") == SOME("b")
   *   peelOff ("a","c")  == NONE
   *)
  val peelOff : string * string -> string option

  (* peelInt s == SOME (i,s') if the longest non-empty prefix of s
   *                comprised only of digits and #"~" parses as the
   *                integer i
   *           == NONE otherwise
   *
   * Ex:
   *   peelInt "55hello"  == SOME(55,"hello")
   *   peelInt "~55hello" == SOME(~55,"hello")
   *   peelInt "-55hello" == NONE
   *   peelInt "hello55"  == NONE
   *)
  val peelInt : string -> (int * string) option
end
```

5.3 Serializing Booleans

Here is an example structure that demonstrates one of many possible ways to serialize the type `bool`.[8]

```
structure SerializeBool : SERIALIZABLE =
struct
    type t = bool
```

[7] This signature is provided in `util.sig` and implemented in a module `Util` ascribing to `UTIL` provided in `util.sml`.

[8] This particular implementation is provided in `serializebool.sml`; you should feel free to experiment with it to use it for testing.

```
    fun write b =
        case b
         of true => "(TRUE)"
          | false => "(FALSE)"

    fun read s =
        case Util.peelOff("(TRUE)",s)
         of SOME s => SOME (true,s)
          | NONE =>
            (case Util.peelOff("(FALSE)",s)
              of SOME s => SOME (false,s)
               | NONE => NONE)
end
```

In `write`, we choose to serialize the value `true` as the string `"(TRUE)"` and the value `false` as the string `"(FALSE)"`.

To try to read back one of these values from a string `s`, we first try to peel off `"(TRUE)"` from `s`. If that succeeds, we return the value `true` and whatever is left over; if it fails, we try to peel off `"(FALSE)"`. If this succeeds, return the value `false` and any leftovers. If this fails, having now failed overall, we return `NONE`.

5.4 Integers, Pairs, and Lists—Oh, my!

Your task is to write serializers for integers, pairs, and lists.

5.4.1 One Possible Strategy

To serialize values of a type, consider how many constructors that type has and how many arguments each of them takes. To keep track of the tree-structure of an expression, you will need to be able to distinguish between constructors and between different instances of the same constructor.

One way to do this is to represent a value constructed with the constructor `con` and arguments `A1` through `An` as

```
(con A1 A2 ...  An)
```

If you consider the type `bool` to be defined as

```
datatype bool = true | false
```

then this is exactly what we did above: the type is given by two nullary constructors and nothing else.

5.4.2 Tasks

Submit your solutions for the following tasks in `serializable.sml`.

Task 5.1 (10%). Implement a structure `SerializeInt` that ascribes to `SERIALIZABLE` and defines serialization for the type `int`.

Task 5.2 (10%). The signature

```
signature SERIALIZABLEPAIR =
sig
  structure S1 : SERIALIZABLE
  structure S2 : SERIALIZABLE
end
```

packages together two serializable modules.[9] Implement a functor

```
functor SerializePair (P : SERIALIZABLEPAIR) : SERIALIZABLE
```

that implements serialization for the type `P.S1.t * P.S2.t`. You may assume that `P.S1` and `P.S2` both obey the above invariant, but you may not make any other assumptions about them.

Task 5.3 (10%). Implement a functor

```
functor SerializeList (S : SERIALIZABLE) : SERIALIZABLE
```

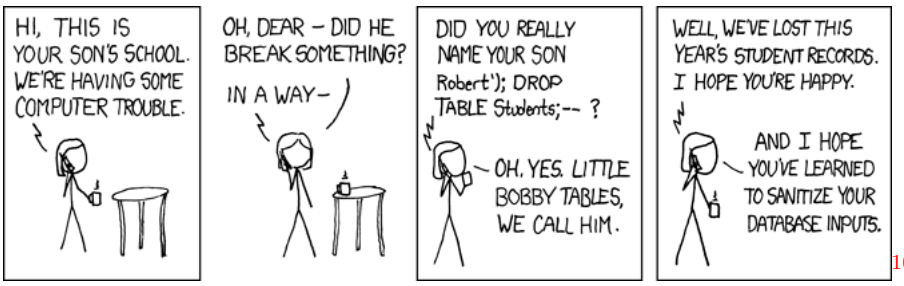
that defines serialization for the type `S.t list`. You may assume that `S` obeys the serialization invariant above, but you may not assume anything else about `S`.

Task 5.4 (5%). Define serialization instances for the following types using only the modules above.

1. `int list list` with a structure named `ILL` ascribing to `SERIALIZABLE`.
2. `(int list) * bool` with a structure named `ILSB` ascribing to `SERIALIZABLE`.
3. `(int * bool) list` with a structure named `ISBL` ascribing to `SERIALIZABLE`.
4. `(int * (bool list)) list` with a structure named `ISBLL` ascribing to `SERIALIZABLE`.

Task 5.5 (Extra Credit).Write a structure `SerializeString` ascribing to `SERIALIZABLE` that defines serialization for the type `string`. Hint:

[9] This signature is provided in `serializablepair.sig`.



[10]

[10] http://xkcd.com/327/