1 Introduction

This lab will give you some practice with writing some simple structures and functors. In particular, you will use the now familiar dictionary structure to implement a set. You will also write a few structures and functors using the ORDERED signature to build the integers from the naturals.

1.1 Getting Started

Update your clone of the git repository to get the files for this weeks lab as usual by running

```
git pull
```

from the top level directory (probably named 15150).

1.2 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, every function you write should have a purpose and tests.

1.3 Compiling This Lab

As is common with modular code, this lab is distributed across many files and relies on the SML/NJ compilation manager to introduce structures into the environment at the right time. The files that contain relevant code are listed in the file sources.cm, and the compilation manager takes it from there. When you want to run your code for this lab, at the REPL, you will enter

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

Two of the files are empty, since we want you to get used to writing structures from scratch. That means that as you progress through the lab, you’ll have to edit the sources.cm file to uncomment the files you’ve filled in. This process is described in somewhat more detail in the write up for Homework 8.
Make sure you’re comfortable with this process! The current homework is organized in the same way, so ask your TA or a neighbour if you can’t get this to work.

2 Types

2.1 Units

The definition of SML includes a type called unit. By design, there is only one value of type unit; that value is written () and pronounced “unit.”

The following REPL session displays a few simple expressions involving the unit type.

- ();
val it = () : unit
- fn _ => ();
val it = fn : 'a -> unit
- (fn x => ()) "call me ishmael";
val it = () : unit
- (fn () => "call me ishmael");
val it = () : unit
- (fn () => "call me ishmael")();
val it = "call me ishmael" : string
- fun ack (m,n) = case (m,n)
    of (0,_) => n+1
      | (_,0) => ack(m-1,1)
      | _ => ack(m-1, ack(m,n-1));
val ack = fn : int * int -> int
- fn () => fn x => ack(100,x);
val it = fn : unit -> int -> int (* returns immediately *)
- (fn () => fn x => fn () => ack(100,x)) () 100;
val it = fn : unit -> int (* returns immediately *)

You can think of unit as the nullary tuple, or the tuple of zero things. Do not confuse the notation for the value of type unit with the notation from other languages for passing arguments to a function—() is a value and has no particular association with function application.

You will use units in this lab as place holders while implementing sets with dictionaries. They will give you a way to insert something into a dictionary without putting anything interesting in that dictionary. Later in the term, we will have other more interesting uses for unit.

2.2 Types We Provide

The file types.sml contains a structure Types that contains a few data types that you’ll use on this lab.
2.2.1 Types.nat

At the very beginning of the course, we said that every natural number is either 0 or 1 + \( n \), where \( n \) is a natural number. We represent this as a datatype in the module Types:

```haskell
datatype nat = Z | S of nat
```

Types.nat represents the natural numbers. Types.Z represents 0, the first natural number. If \( n \) is any value of type Types.nat, then \( (\text{Types.S } n) \) represents the successor of \( n \)—that is, \( 1 + n \). This datatype representation avoids the problems we had representing natural numbers with int: it is not bounded-precision, and there are no negatives.

2.2.2 ('a, 'b) Types.choice

The type ('a,'b) Types.choice is defined by

```haskell
datatype ('a,'b) choice = A of 'a | B of 'b
```

Intuitively, ('a,'b) Types.choice represents “either an 'a or a 'b”, since a value of type ('a,'b) choice is either (Types.A x) for some x of type 'a, or (Types.B x) for some x of type 'b. For example, an (int,string) Types.choice has values like Types.A 7 and Types.B "hi", and represents a value that is either an integer or a string. This “either 'a or 'b” description isn’t perfect, though, because in English, something which is “either a string or a string” is just a string, but a (string, string) Types.choice is not the same as a string! There is an additional tag bit, A or B.

3 Sets As Dictionaries

To get used to functors, you will implement a set using a dictionary. In the set.sig file we give a signature of sets that you’ll implement:

```haskell
signature SET =
sig

structure Element : ORDERED

type set

val empty : set

val insert : set -> Element.t -> set
val remove : set -> Element.t -> set
val member : set -> Element.t -> bool

end
```
The components of the signature have the following specifications:

- **Element**: ORDERED defines the ordering of the elements in the set.
- **set** is the type of the set of elements of type Element.t.
- **empty** is a set that contains no elements.
- **insert** is a function that takes a set and an element and returns the set with the element added.
- **remove** is a function that takes a set and an element and returns the set with the element removed.
- **member** is a function that takes a set and an element, and returns true if that element is in the set, or false if the element is not in the set.

In the dict.sig file we give the now-familiar signature of dictionaries:

```ml
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
end
```

**Task 3.1**

Write a functor DictSet in dictset.sml that takes a structure D : DICT and yields a structure ascribing to the above SET signature using the following definitions for Element and set:

```ml
structure Element = D.Key

type set = unit D.dict
```

**Task 3.2** In dictset.sml write a structure TestSet that includes some tests for the DictSet functor. We have included the TreeDict functor and IntLt : ORDERED structure in the support code to help you instantiate TreeSet.

Have the TAs check your implementation before proceeding!
4 Ordered Types

Recall the signature ORDERED from lecture, defined in src/ordered/ordered.sig, which packages a type with a comparison function for pairs of values of that type:

Task 4.1 Write a structure NatOrder in order.sml ascribing to the ORDERED signature, which orders the values of the type Types.nat by <. Note that, because the constructors are in the module Types, you will need to write Types.Z and Types.S in pattern-matching.

Task 4.2 In order.sml, write a functor FlipOrder that takes some structure O ascribing to ORDERED and produces a structure ascribing to ORDERED by reversing the ordering of O.

For example, let O be some ORDERED structure and O’ be the result of applying FlipOrder to O. If one value of type O.t is less than another according to O.compare, then it will be greater according to O’.compare.

Task 4.3 The signature TWOORDERS defined in src/ordered/twoorders.sig packages two structures ascribing to ORDERED into one module:

In order.sml, implement a functor ChoiceOrder that takes a structure ascribing to TWOORDERS and returns a structure that ascribes to ORDERED where the type of the returned structure is the Types.choice of the two types of the argument structures.

The ordering in the structure you return should consider any value of type O1.t to be less than any value of type O2.t, but otherwise order the values using the argument comparison functions.

Have the TAs check your implementation before proceeding!

5 From Naturals to Integers

You can represent integers as the union of two distinct “copies” of the naturals: one copy represents the negative integers and the other the nonnegative integers. In this representation, every integer is represented as a choice between these two copies: it’s either a particular negative integer or a particular nonnegative integer. This notion is captured by the type (Types.nat, Types.nat) Types.choice. Note that the negatives are shifted by one—e.g. 2 is represented by Types.A (Types.S Types.Z)—but the positives are not—2 is represented by Types.B (Types.S (Types.S Types.Z))

Since you already wrote a structure that represents an ordering of the naturals, and a functor to order the choice of two ordered types, you can now use these structures and functors to create an ordered representation of the integers from the naturals. Your answer for the next two tasks should involve only module-level code.

Task 5.1 Use your FlipOrder functor and your NatOrder structure to create a structure Ints ascribing to TWOORDERS in order.sml. Ints should represent the orderings of the negative and nonnegative “halves” of the integers. Note that it does not represent the integers—the next functor will combine these two halves to get our representation of integers.
Task 5.2 Use your `ChoiceOrder` functor and `Ints` structure to define a structure `IntsOrder`. `IntsOrder` should ascribe to `ORDERED` and represent the integers under their usual ordering.

Task 5.3 Write `toInt : IntsOrder.t -> int` and `fromInt : int -> IntsOrder.t` to convert between this representation of the integers and the usual one.