Contents

1 Modules 2
  1.1 print it ............................................................ 2
    Task 1. ............................................................. 2
  1.2 Implementing a Stack ........................................ 3
    1.2.1 A Transparent Attempt ................................ 3
    Task 2. ............................................................. 3
    1.2.2 The Meaning of Opaque Is Unclear .................. 3

2 SML CM 4
  2.1 From the REPL ................................................. 4
  2.2 Shell Flag ....................................................... 4

3 Structure or Signature? 5
  Task 3. ............................................................. 5

4 From my point of view, the Jedi are evil 6
  4.1 A map of the galaxy ........................................... 6
    Task 4. ............................................................. 6
    Task 5. ............................................................. 6
  4.2 Training for the Jedi ORDER ................................. 6
    Task 6. ............................................................. 7
  4.3 Jedi Master ...................................................... 7
    Task 7. ............................................................. 7
    Task 8. ............................................................. 7
    Task 9. ............................................................. 7
    Task 10. ............................................................ 8

5 Representational Independence: Nat 9
  5.1 Motivation ........................................................ 9
    Task 11. ............................................................ 11

6 Representational Independence: List 11
  Task 12. ............................................................. 12
  Task 13. ............................................................. 12

7 Representing Ints 13
  Task 14. ............................................................. 13
  Task 15. ............................................................. 13

8 Representing Sets 14
  Task 16. ............................................................. 14
  Task 17. ............................................................. 14
1 Modules

Modules let us create abstraction boundaries. We specify interfaces via signatures and implementations via structures, which ascribe to signatures.

Values are to types as structures are to signatures.

1.1 print it

signature PRINTABLE =
sig
  type t
  val toString : t -> string
end

structure IntPrint : PRINTABLE =
struct
  type t = int
  val toString = Int.toString
end

structure BoolPrint : PRINTABLE =
struct
  type t = bool
  fun toString true = "true"
  | toString false = "false"
end

functor PrintPair (structure A : PRINTABLE structure B : PRINTABLE) : PRINTABLE =
struct
  type t = A.t * B.t
  val toString = fn (a,b) =>
    "(" ^ A.toString a ^
    "," ^ B.toString b ^
    ")"
end

- val "true" = BoolPrint.toString true
- val "1" = IntPrint.toString 1
- structure IntBoolPair = PrintPair (structure A = IntPrint structure B = BoolPrint)
- val "(1,true)" = IntBoolPair.toString (1, true)

Task 1.

• In transparent ascription MyStructure : MYSIG, the client is allowed to know what the internal types are. Useful for:

• In opaque ascription MyStructure :> MYSIG, the client is not allowed to know anything about the types. Useful for:

In either case, you can only use information from the signature.
1.2 Implementing a Stack

Let’s implement a stack as a structure.

1.2.1 A Transparent Attempt

Task 2.

```
signature STACK =
sig
type 'a t  
  exception Empty  
  val push : 'a -> 'a t -> 'a t  
  val pop : 'a t -> 'a * 'a t  
end
```

```
structure Stack : STACK =
struct
  type 'a t = 'a list  
  exception Empty  
  fun push x s = x::s  
  fun pop [] = raise Empty  
    | pop (x::s) = (x,s)  
  val k = 1 (* huh? *)  
end
```

- val s : int Stack.t = [];
- val s' = Stack.push 2 s;
- val s' = [2] : int Stack.t
- Stack.pop s';
  val it = (2,[]) : int * int Stack.t
- Stack.pop s';
  val it = (* what goes here? *)

1.2.2 The Meaning of Opaque Is Unclear

```
signature STACK =
sig
  type 'a t  
  exception Empty  
  val empty : 'a t  
  val push : 'a -> 'a t -> 'a t  
  val pop : 'a t -> 'a * 'a t  
end
```

```
structure Stack :> STACK =
struct
  type 'a t = 'a list  
  exception Empty  
  val empty = []
  val push x s = x::s  
  fun pop [] = raise Empty  
    | pop (x::s) = (x,s)  
  end
```

- val s : int Stack.t = [];
- val s = [] : int Stack.t
- val s' = Stack.push 2 s;
- val s' = [2] : int Stack.t
- Stack.pop s';
  val it = (2,[]) : int * int Stack.t
- Stack.pop s';
  val it =
Starting from this lab and in your future labs and homeworks, you will be working with multiple SML files that have dependencies on each other (i.e. you may need to load some files first in order for some other files to be compilable). To avoid the hassle of typing `use "<file>.sml"` multiple times each time you try to compile your code, you can make use of a built-in tool for SML called **Compilation Manager (CM)**. More detail on CM can be found via this URL: https://www.smlnj.org/doc/CM/

If you don’t want to go through the horribly long documentation on CM, here is a quick and dirty guide that teaches you how this works.

In each problem directory, we’ve already prepared for you a file called `sources.cm`. This file contains the information of all files and their dependencies needed for compilation. If you are interested, you can take a look at the `sources.cm` file. Otherwise, you don’t have to. There are two ways to use the source file for compilation:

### 2.1 From the REPL

After launching your SML interpreter:

```smlnj```

You can run this in the REPL:

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

### 2.2 Shell Flag

Alternatively, you can type in your shell

```smlnj -m "sources.cm";```

The compiler will read compilation information from the source file and compile your code accordingly.
Task 3.

State whether the following declarations/definitions belong in a structure, belong in a signature, or could be found in both.

1. `val x : int`
2. `val x = 10`
3. `type t`
4. `type t = int`
5. `exception E`
6. `exception I of int`
7. `datatype t = Foo of int | Bar of bool`
8. `fun f x = x + 10`
4 From my point of view, the Jedi are evil

You are Darth Yeetius, little-known apprentice of Darth Sidious. Your master’s been gone for a couple of months, and you’ve been feeling down. One day, you discover a secret box containing Darth Sidious’ genius plans to annihilate the Jedi. The only problem? Darth Sidious hates actually implementing code, so all you have are signatures to work with!

4.1 A map of the galaxy

Darth Sidious needs some kind of map to locate the Jedi hideouts. He had some plans written, but but it seems like he couldn’t decide what type to make it. So it’s up to you to implement his map, but for any possible type.

The first signature you find is

```sml
signature MAPPABLE =

sig
  type 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
end
```

This signature represents an abstract polymorphic datatype t which (somewhat) functions as a “container” of data: if t1, t2 are types, f : t1 -> t2 is some function, and X : t1 t (X is a container storing t1 values), then we can apply f to the data “inside” X by evaluating map f X, yielding a “t2 container”, i.e. (map f X) : t2 t.

You can see an example in code/typeclasses/mappable/ListMap.sml. Note that ListMap is transparently ascribed to the signature MAPPABLE. Before proceeding, try to figure out why we want to transparently ascribe, and not opaquely. The answer is given in this footnote.¹

Task 4.

In code/typeclasses/mappable/OptionMap.sml, implement OptionMap.

Task 5.

In code/typeclasses/mappable/TreeMap.sml, implement TreeMap.

4.2 Training for the Jedi ORDER

The next signature you find is

```sml
signature ORDERED =

sig
  type t
  val compare : t * t -> order
end
```

¹If ListMap :> MAPPABLE, then the SML type system would not allow us to use the fact that a t1 ListMap.t is just a t1 list. So writing [3,4] : int ListMap.t would not typecheck. We would therefore not be able to create any values of type int ListMap.t, so we would not be able to use the ListMap.map function. If we transparently ascribe, then we can construct value of type t1 ListMap.t using [] and ::.
This signature represents a type $t$ endowed with an “ordering”, i.e. a function which puts the values of that type in order. You can see examples of structures ascribing to this signature in `code/typeclasses/ordered/IntOrdered.sml` and `code/typeclasses/ordered/StringOrdered.sml`.

Important to note: we assume as an invariant that the function `compare` behaves like a “reasonable” ordering function. For instance, we want the “less-than” relation given by this function is transitive (if `compare (x,y) \equiv LESS` and `compare (y,z) \equiv LESS`, then `compare (x,z) \equiv LESS`), that the “equality” notion it gives is symmetric (if `compare (x,y) \equiv EQUAL`, then `compare (y,x) \equiv EQUAL`), and so on.

Task 6.

In `code/typeclasses/ordered/IntToIntOrdered.sml`, implement the structure `IntToIntOrdered` ascribing to the `ORDERED` signature with the type $t$ as $\text{int} \rightarrow \text{int}$. You can make `compare` to be anything you want as long as it reasonably gives some notion of ordering.

4.3 Jedi Master

Now, you’ll be implementing some functors which transform `ORDERED` structures in more complex ways.

Task 7.

In `code/typeclasses/ordered/MkFlippedOrder.fun`, implement a functor `MkFlippedOrder` which takes in a structure `Ord : ORDERED` and returns a structure `MkFlippedOrder (Ord) : ORDERED` which “flips” the order. Formally, given `structure Flipped = MkFlippedOrder (Ord)` and for all values $x, y$ of type `Ord.t`:

- `Ord.compare (x,y) \equiv LESS` if and only if `Flipped.compare (x,y) \equiv GREATER`
- `Ord.compare (x,y) \equiv EQUAL` if and only if `Flipped.compare (x,y) \equiv EQUAL`
- `Ord.compare (x,y) \equiv GREATER` if and only if `Flipped.compare (x,y) \equiv LESS`

Task 8.

In `code/typeclasses/ordered/MkOrderedPair.fun`, implement a functor `MkOrderedPair` which takes in two `ORDERED` structures (call them `A` and `B`) and returns a structure ascribing to `ORDERED` representing the lexicographical order on $A.t \times B.t$.

Recall that if $A$ is a set and $\leq_A$ an order relation on $A$, and $B$ is a set with $\leq_B$ a order relation on $B$, we define the lexicographical order on their Cartesian product $A \times B$ to be the relation $\leq$ given by

$$(a,b) \leq (a',b') \iff a <_A a' \text{ or } (a =_A a' \text{ and } b \leq_B b')$$

Task 9.

In `code/typeclasses/ordered/MkFunctionOrder.fun`, implement a functor `MkFunctionOrder` which takes in three pieces of data:

- a type `input`
- a value `x : input`
• a structure \texttt{Output : ORDERED} and returns the structure ascribing to \texttt{ORDERED} whose type \texttt{t is input \to Output.t}, where the comparison between two values of this type is done by applying both to \texttt{x}, and then using \texttt{Output.compare} to compare the resulting outputs.

**Task 10.**

In \texttt{code/typeclasses/ordered/IntStarBoolToStringOrdered.sml}, implement a structure \texttt{IntStarBoolToStringOrdered : ORDERED} whose underlying type \texttt{(IntStarBoolToStringOrdered.t) is int \times (bool \to string)}.

\textbf{Constraint:} You should not implement \texttt{IntStarBoolToStringOrdered.compare} directly. Instead, you should implement this structure by applying the functors you’ve already implemented to structures you’ve already implemented.
5 Representational Independence: Nat

5.1 Motivation

Throughout the course, we’ve heavily emphasized the notion of \textit{extensional equivalence} as a relationship between SML expressions of the same type. We’ve been interested in extensional equivalence primarily because extensionally-equivalent pieces of code are \textit{interchangeable}: if \( e_1 : t \) and \( e_2 : t \) are such that \( e_1 \equiv e_2 \), then the two have the exact same behavior, and I can use them in place of each other. This would often take the form of first writing a “naïve” implementation of a function which is easy to write, analyze, and reason about, and then writing a “complex” version which is more efficient. If we can prove the two implementations are extensionally equivalent, then anywhere in our code where we used the naïve implementation we could replace it with the complex implementation (thereby taking advantage of its superior efficiency) without any impact on the correctness of the surrounding code. This kind of procedure significantly streamlines the development of complex pieces of software, as we can upgrade various components and be sure our code remains correct.

We’d now like to do this with modules: we’d like a way to \textit{prove} that two implementations of the same signature are “equivalent” (for the proper notion of equivalence), and thereby guarantee that we can freely replace one implementation with the other (say, replace a simple but slow implementation with a more complicated but faster one) without affecting the correctness of any code that uses that module. This kind of proof is called a \textit{representation independence proof}.

We’ll first do a (somewhat silly) example to get the general idea. Consider the following signature which represents the natural numbers, and the two structures opaque ascribing to it.

```ml
signature NAT =
  sig
    type nat
    val Zero : nat
    val Succ : nat -> nat
    val recur : ('a -> 'a) -> 'a -> nat -> 'a
  end

structure IntNat :> NAT =
  struct
    type nat = int
    val Zero = 0
    val Succ = fn x => 1+x
    fun recur g z 0 = z
                  | recur g z n = g(recur g z (n-1))
  end

structure ListNat :> NAT =
  struct
    type nat = unit list
    val Zero = []
    val Succ = fn N => () ::N
```
```ml
fun recur g z [] = z
  | recur g z (_::xs) = g(recur g z xs)
end
```

Take a moment to understand these two implementations. The \texttt{IntNat} structure represents natural numbers as the corresponding integers, and the \texttt{recur} function (which automates natural number recursion) works by the same kind of recursion that we’ve been doing for the entire course. The \texttt{ListNat} structure implements natural numbers as lists of (), where the length of the list corresponds to the natural number being encoded.

Here’s some code utilizing these structures.

```ml
val toInt = recur (fn x=>x+1) 0
fun fromInt 0 = Zero | fromInt n = Succ(fromInt (n-1))

val One : nat = Succ Zero
val Two : nat = Succ One
val add : nat -> nat -> nat = fn n => recur Succ n
val mult : nat -> nat -> nat = fn n => recur (add n) Zero
val exp2 : nat -> nat = recur (mult Two) One
```

Now, we claim that it doesn’t matter which implementation of \texttt{NAT} you use: from the outside, they behave the same. To prove this, we’ll define a relation \( R \) between the values of type \texttt{IntNat.nat} and \texttt{ListNat.nat} expressing the property of “representing the same natural number”. We then prove that if a value of type \texttt{IntNat.nat} and a value of type \texttt{ListNat.nat} are related by this relation, then they will have the exact same behavior when using anything from the \texttt{NAT} structure.

Here’s the relation:

**Definition 1.** Define a relation \( R \) between values of type \texttt{IntNat.nat} and \texttt{ListNat.nat}:

\[
R(n: \texttt{IntNat.nat}, L: \texttt{ListNat.nat}) \iff n \equiv \text{length } L
\]

Remember: \( n : \texttt{IntNat.nat} \) represents the natural number \( N \) if \( n \) literally is \( N \). Whereas a value \( L : \texttt{ListNat.nat} \) represents \( N \) if the length of \( L \) is \( N \). That is why we chose this definition of \( R \).

Now, the theorem. This says that the operations in the \texttt{IntNat} and \texttt{ListNat} structures respect this relation.

**Theorem 1.**

1. \( R(\texttt{IntNat.Zero}, \texttt{ListNat.Zero}) \)
2. If \( R(n, L) \), then \( R(\texttt{IntNat.Succ } n, \texttt{ListNat.Succ } L) \)
3. For all values \( n : \texttt{IntNat.nat} \), all values \( L : \texttt{ListNat.nat} \), all types \( t \), all values \( z : t \), and all total functions \( g : t \to t \), if \( R(n, L) \),

\[
\texttt{IntNat.recur } g \ z \ n \equiv \texttt{ListNat.recur } g \ z \ L
\]
Task 11.

Prove the above theorem. You may assume `length` is implemented as

```ml
fun length [] = 0
  | length (_::xs) = 1 + length(xs)
```

and may assume the following lemmas:

**Lemma 1.** `length` is total

**Lemma 2.** For any value `L : unit list`, `length L ≡ 0` if and only if `L = []`.

**Lemma 3.** If `n : int` is negative, then there does not exist an `L : ListNat.nat` such that \( R(n,L) \)

### 6 Representational Independence: List

Now that we’ve been introduced to the basic idea, let’s do a more useful example.

```ml
signature LIST =
  sig
    type 'a t
    val nil : 'a t
    val cons : 'a * 'a t -> 'a t
    val filter : ('a -> bool) -> 'a t -> 'a t
    val length : 'a t -> int
  end

structure List :> LIST =
  struct
    type 'a t = 'a list
    val nil = []
    val cons = op :::
    fun filter p [] = []
      | filter p (x::xs) =
        let
          val res = filter p xs
        in
          if p x then x::res else res
        end
    fun length [] = 0
      | length (x::xs) = 1 + length(xs)
  end
```
What the LenList structure does is implement the abstract 'a t type as an 'a list, but it also keeps track of the length of the list. To an outside user (we claim), these two implementations behave identically. The only difference is that List.length has linear-time work (and span), whereas LenList.length has constant work. So – once we’ve proved the invariance result – we can use LenList in place of List (and enjoy the faster speed), and not see any difference in the code’s behavior.

**Task 12.**

Define an appropriate relation \( R \) between these two implementations of LIST

**Task 13.**

State the representational invariance theorem between these two structures (in the same format as Theorem 1)
7 Representing Ints

As you know, integers can be represented in many bases. All representations should, however, allow for addition and multiplication (and some other) operations on them. A good way to implement this would be to use a signature that provides an interface for the common behavior that all integer representations should have, and then implementing structures ascribing to this signature for the various bases.

In this task, you will implement integers in two different bases, and allow for addition and multiplication on them, following the ARITHMETIC signature in code/arith/ARITHMETIC.sig.

The components of the signature have the following specifications:

- **integer** is the type used to represent integers
- **fromInt** given an int, converts it to an integer type
- **toInt** takes an integer and converts it to an int.
- **toString** displays the given integer as a string.
- **add** takes in two integers and returns the result of their addition.
- **mult** takes in two integers and returns the result of their multiplication.

In each of the following tasks, it might be challenging to write add and multiply without helper functions. For example, it might be useful to have use a helper function to keep track of the carry digits when implementing add. For mult, you may want to first consider solving the problem of multiplying a single digit with a given integer.

**Task 14.**

Implement the structure Binary, ascribing to ARITHMETIC found in code/arith/Binary.sml that implements the specified integer operations for integers in base 2. As an example, the fromInt function should convert the int 4 to an integer that represents the binary number “100” (which is the binary form of 4). For mult and add, try not to simply call toInt and fromInt.

**Hint:** the type of integer should be digit list. With this type, it is possible to represent ints in two ways - one in which the least significant digit is stored as the first element in the digit list and one in which the least significant digit is stored as the last element in the digit list. One of these ways is easier to implement than the others. In addition, while it is possible to represent 0 by either the empty list or the list [0], for simplicity, let 0 be represented by the empty list.

**Task 15.**

Implement the structure Decimal, ascribing to ARITHMETIC found in code/arith/Decimal.sml that implements the specified integer operations for integers in base 10. For mult and add, try not to simply call toInt and fromInt.
8 Representing Sets

Recall from lecture, a *signature* is an interface specification that usually lists some types and values which might use those types. In this task, you will write two implementations of the INTSET signature that can be found in code/sets/INTSET.sig, and is given below:

The components of the signature have the following specifications:

- **set** is the type of the set of elements of type `int`.
- **empty ()** is a set that contains no elements.
- **find** 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.
- **insert** is a function that takes a set and an element and returns the set with the element added.
- **delete** is a function that takes a set and an element and returns the set with the element removed.
- **union** is a function that takes two sets `X` and `Y` and evaluates to the set `X ∪ Y`, i.e. a set that contains all the elements in `X` and `Y` that results from performing a mathematical union of the two sets.
- **intersection** is a function that takes two sets `X` and `Y` and evaluates to the set `X ∩ Y`, i.e. a set that contains the elements in both `X` and `Y` that results from performing a mathematical intersection of the two sets.
- **difference** is a function that takes two sets `X` and `Y` as input and evaluates to the set `X \ Y`, i.e. a set that contains all elements in `X` and not in `Y`, that results from performing the mathematical difference of the two sets.

There are many ways to implement the functionality of sets. We will implement sets in two different ways in this lab. The first should allow for duplicates to be inserted into the internal representation of the set, but care should be taken that (externally) the implementation still behaves like a set (recall that a set does not contain duplicate values for any given element). For example, deleting an element that occurs multiple times internally should externally delete that element completely. The second should not allow for any duplicates to be stored in the internal representation at all.

You should think about what invariants you want to have on your internal representation before starting to program; there are a few ways to do this.

**Task 16.**

Implement the structure `SetKeepDuplicates` ascribing to `INTSET` found in:

code/sets/SetKeepDuplicates.sml

that *allows* for duplicate insertion into the set. Keep in mind that all functions implemented must account for the fact that the internal representation *may* contain duplicate values. This will be particularly important when removing elements from the set.

**Task 17.**

Implement the structure `SetNoDuplicates` ascribing to `INTSET` found in:

code/sets/SetNoDuplicates.sml
that *does not allow* for duplicate insertion into the set. Keep in mind that all functions implemented must account for the fact that the internal representation *must not* contain duplicate values. This will be particularly important when adding elements to the set.