Contents

1  Hype for Types 3
   1.1  Is This Your Signature? ........................................ 3
       1.1.1  Corn ................................................ 3
       Task 1.  (1 point) ......................................... 3
       Task 2.  (1 point) ......................................... 3
       1.1.2  Potatoes ............................................. 4
       Task 3.  (1 point) ......................................... 4
       1.1.3  Turkey ............................................... 5
       Task 4.  (1 point) ......................................... 5
       1.1.4  Pie ................................................ 6
       Task 5.  (1 point) ......................................... 6
       Task 6.  (1 point) ......................................... 6
   1.2  Sign Here .................................................. 7
       Task 7.  (2 points) ........................................ 7
   1.3  Ascription Decryption ....................................... 8
       Task 8.  (1 point) ......................................... 8
       Task 9.  (2 points) ....................................... 9

2  Signatures 10
   2.1  The Forests and the Trees ................................. 10
       Task 10.  (5 points) ...................................... 10
       Task 11.  (5 points) ...................................... 11

3  Storytime 12
   3.1  Books ................................................... 12
       3.1.1  The Book Signature .................................. 12
       3.1.2  A Slow Implementation, and a Faster One ............ 14
       Task 12.  (10 points) ..................................... 14
       3.1.3  Testing ............................................. 14
       Task 13.  (5 points) ..................................... 15
   3.2  Students .................................................. 16
       3.2.1  The Student Signature ................................ 16
       3.2.2  Some Useful Students ............................... 17
       Task 14.  (3 points) ..................................... 17
       Task 15.  (3 points) ..................................... 17
       Task 16.  (3 points) ..................................... 18
       Task 17.  (1 point) ....................................... 18
       3.2.3  Reading a Whole Book ................................ 19
       Task 18.  (5 points) ..................................... 19
3.3 More Complicated Students

3.3.1 Making a Nemesis

Task 19. (3 points)

3.3.2 Composition

Task 20. (7 points)

Task 21. (3 points)

3.3.3 Putting It All Together

Task 22. (5 points)

4 Representation Independence

4.1 Motivation

4.2 Queues

4.3 The Code

4.4 The Proof

Task 23. (15 points)

5 In Simpler Terms

5.1 The Approach

5.1.1 Substitutions

5.1.2 Naïve Implementation

5.1.3 A Clever Idea

5.2 FToStringable

5.3 BoolMatchable

Task 24. (4 points)

5.4 Functions - This Time, With Arguments!

5.4.1 Either Way

Task 25. (3 points)

5.5 Working our Way Back Up

Task 26. (3 points)

5.6 Simplification Nation

5.6.1 Differentiation

Task 27. (5 points)

A Functor Syntax

A.1 Syntactic Sugar

A.1.1 Tricky: One Input Structure

A.2 “Multiple” Arguments
1 Hype for Types

The code for all following tasks is included in code/types/. Feel free to use the REPL to check
your answers. However, note that if an explanation is requested, REPL output is not sufficient!
You should explain all reasoning using the principles about modules we’ve discussed in class (and
in your own words).

1.1 Is This Your Signature?

You’ve just been recruited to the LambdaLegs delivery service. Congratulations! Your job is
straightforward, but important: in order to make sure the next round of funding is secured,
you must verify that each customer signs for an appropriate delivery.

In other words, you must check that their signature matches the structure you’ve brought.

1.1.1 Corn

Consider the following structure:

```ml
structure Corn =
  struct
    datatype kernel = White | Yellow | Red (* "flint corn" *)
    type t = kernel * int (* dominant color, number *)
    val color = fn
      (White, _) => "White"
    | (Yellow, _) => "Yellow"
    | (Red, _) => "Red"
    val eat = fn
      (_,0) => NONE
    | (k,i) => SOME (k,i-1)
  end
```

Task 1. (1 point)

Does Corn : CORN1 typecheck? If it does, say so. Otherwise, explain why it doesn’t.

```ml
signature CORN1 =
  sig
    datatype kernel = White | Yellow | Red
    type t
    val color : t -> string
    val eat : kernel * int -> t option
  end
```

Task 2. (1 point)

Does Corn : CORN2 typecheck? If it does, say so. Otherwise, explain why it doesn’t.

```ml
signature CORN2 =
  sig
```
1.1.2 Potatoes

Consider the following functor:

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

functor MkPotatoes (val prefix : string
                      structure Topping : PRINTABLE)
  =
  struct
    type t = bool * Topping.t
    val show = fn (hot, top) =>
      (if hot then "hot" else "cold") ^ " "
      ^ prefix ^ " potatoes with " ^ Topping.toString top
  end
```

**Task 3. (1 point)**

Does the following code typecheck? If it does, say what value is bound to x. Otherwise, explain why it doesn’t.

```ocaml
structure Marshmallow : PRINTABLE =
  struct
    type t = bool
    val toString = fn
      false => "burnt marshmallows"
      | true => "toasted marshmallows"
  end

structure MashedPotatoes = MkPotatoes (val prefix = "mashed"
                                        structure Topping = Marshmallow)

val x = MashedPotatoes.show (true, false)
```
1.1.3 Turkey

Task 4. (1 point)

Does the following code typecheck? If it does, say what value is bound to \( x \). Otherwise, explain why it doesn’t.

```ml
signature TURKEY =
  sig
    type t
    val Boil : t
    val Bake : t
    val show : t -> string
  end

structure Turkey : TURKEY =
  struct
    datatype t = Boil | Bake | Stuff
    val show = fn Boil => "Boiled Turkey"
    | Bake => "Baked Turkey"
    | Stuff => "Stuffed Turkey"
  end

val t1 = Turkey.Bake
val x = case t1 of
  Turkey.Boil => Turkey.show t1
  | _ => "Not Boiled Turkey"
```
1.1.4 Pie

Consider the following structure:

```ml
structure Pie =
  struct
   exception AlreadyEmpty
   type 'a t = 'a list (* list of slices *)
   val mkPie = fn
     nil => raise AlreadyEmpty
     | l => l

   val eat = fn
     nil => NONE
     | _ :: slices => SOME slices
  end
```

Task 5. (1 point)

Does Pie : PIE1 typecheck? If it does, say so. Otherwise, explain why it doesn’t.

```ml
signature PIE1 =
  sig
   type t
   val mkPie : 'a list -> t
  end
```

Task 6. (1 point)

Does Pie : PIE2 typecheck? If it does, say so. Otherwise, explain why it doesn’t.

```ml
signature PIE2 =
  sig
   val eat : 'a list -> 'a list option
  end
```
1.2 Sign Here

Consider the following signature:

```ml
signature PATH =
  sig
    type path
    datatype arc = Up | Down of string
  exception InvalidPath

  val root : path
  val push : arc * path -> path
end
```

Task 7. (2 points)

Does the following code typecheck? If it does, say so. Otherwise, explain why it doesn’t.

```ml
structure Path :> PATH =
  struct
    type path = string list
    datatype arc = Up | Down of string
    exception InvalidPath

    val root = nil
    val push = fn
      (Up , nil ) => raise InvalidPath
      | (Up , p :: ps) => ps
      | (Down p, ps ) => p :: ps
  end

  val "modules" :: _ = List.foldl Path.push Path.root [ Down "private",
              Down "15150",
              Down "modules"
            ]
```
1.3 Ascription Decryption

Task 8. (1 point)

What is the type of x in the following code? If it is not well-typed, explain why not.

```haskell
signature FOO =
  sig
    type t
    val into : int -> t
    val out : t -> int
  end

structure Bar :> FOO =
  struct
    type t = int
    fun into x = x
    fun out x = x
  end

val x = 5 + (Bar.into 10)
```
Task 9. (2 points)

Does this piece of code typecheck? If so, give the type of y. If not, explain why not.

```
signature FOO =  
  sig  
    type t  
    val into : int -> t  
    val out : t -> int  
  end  

structure Bar :> FOO =  
  struct  
    type t = int  
    fun into x = x  
    fun out x = x  
  end  

signature BAZ =  
  sig  
    type t  
    val create : int -> t  
    val destroy : t -> int  
  end  

functor Qux (Waldo : FOO) :> BAZ =  
  struct  
    type t = Waldo.t  
    val create = Waldo.into  
    val destroy = Waldo.out  
  end  

structure S = Qux (Bar)  
structure S' = Qux (Bar)  

fun f x = S.destroy x  
val y = f (S'.create 10)
```
2 Signatures

In this section, you’ll write signatures corresponding to some structures and functors.

2.1 The Forests and the Trees

In code/signature/BinaryTree.sml and code/signature/MkForest.fun, you will find an implementation of a binary tree and a forest, which is essentially a group of trees.

The structure BinaryTree ascribes to the TREE signature, while the functor MkForest takes in a structure ascribing to TREE and gives a structure which ascribes to the FOREST signature.

Below is some code which uses the implementations from the BinaryTree structure and MkForest functor. You must use this code to help you determine some of the necessary values and types to include in your signatures. This code can also be found in code/signature/Test.sml.

```
structure Test =
  struct
  structure Forest = MkForest (structure Tree = BinaryTree
    val lim = 4
  )
  val cmp = Int.compare
  val t0 = foldl (fn (x, t) => Forest.T.insert cmp t x) (Forest.T.empty ()) [4, 1, 65, 3, 8, 7, 2]
  val t1 = foldl (fn (x, t) => Forest.T.insert cmp t x) (Forest.T.empty ()) [10, 9, 11, 12, 13]
  val 4 = Forest.limit
  val fe : int Forest.forest = Forest.empty ()
  val f1 = Forest.addToForest fe t0
  val f2 = Forest.addToForest f1 t1
  val f3 = Forest.addToTree cmp f2 3 15 handle Forest.Full => f2
  val f4 = Forest.addToTree cmp f3 9 20
  val _ = SOME (Forest.findRoot cmp f4 20) handle Forest.NotInForest => NONE
  end
```

Task 10. (5 points)

Write the signature for TREE in code/signature/TREE.sig. Your signature should contain all declarations necessary for the BinaryTree structure, MkFunctor functor, and above code to typecheck.

You can run the following to ensure that your signature works with BinaryTree.sml:

```
smlnj TREE.sig BinaryTree.sml
```

However, to ensure that your signature works with Test.sml you will also need to complete the next task.
**Constraint:** The signature you write must not only allow the code to typecheck, but must also contain the *fewest* things possible, i.e. your signature should *not* contain more values than are necessary for the given code to typecheck.

**Task 11.** (5 points)

Write the signature for FOREST in code/signature/FOREST.sig. Your signature should contain all declarations necessary for the MkForest functor and given usage code to typecheck.

**Constraint:** The signature you write must be the *minimal* signature, so it should include all necessary types and values for the provided code to typecheck but should *not* contain more types or values than are necessary for the code to typecheck.

To compile your code you can run the following:

```
smlnj -m sources.cm
```

If everything successfully compiles, then your signatures certainly contain all of the necessary types and values. However, successful compilation will not tell you whether you have the *minimal* signatures.
3 Storytime

3.1 Books

As you wander through Gates on your way to 150 lab with your favorite TAs, you see countless students poring over textbooks and Diderot course notes, flipping pages here and there. You hear groans of frustration and sighs of hopelessness. Naturally, ever the consummate Reasonable Person™, you begin to wonder: what if there’s a way to represent these books to make it easier for students to figure out whether they’re useful, before spending hours reading, only to realize that they’re looking at something that isn’t even related to the topic they’re looking for?

In this problem, you’ll try to model a book and some students using the SML module system.

3.1.1 The Book Signature

There are many ways we could implement a book, as long as it satisfies some fundamental properties. Whatever we do with our books shouldn’t depend on how exactly the book is implemented, so we’ll take a single BOOK signature and implement a structure to create a book with the properties we want.

We want our books to have three key features:

- The ability to create a book from a list of pages and turn a book back into the pages it was made from. We’ll call these functions bind and unbind, respectively.
- The ability to identify which page of the book we’re currently at, if we’re in the middle of reading it. We’ll need this function, called currentPage, to tell us both where we are in the book (i.e., if we’re at the beginning or not) and what page we’re currently on, if there is one.

Note that when a book has just been bound, the currentPage should be set to the beginning of the book.

- The ability to flip either forward or backward in a book. This functionality will be implemented in a function called flip. flip should raise the exception OutOfBounds if you attempt to flip forward from the end or back from the beginning.

We’ve encoded these features in the signature in code/books/BOOK.sig, included below for your convenience. The only types we want to enforce are those of a page, which will be a string list where each string is one word, and of a direction, which will represent which way we flip in the book.

The key difference between a book and a page list is that a value of type book carries around an extra piece of information: the page that it’s been opened to. A bound page list should at first be opened to the first page of the book, if one exists. A student’s position in the resulting book can then be updated using the flip function.

Apart from these fundamental types, the details of our books will depend on their implementation.

---

1There might not always be a current page; think about what happens if we’re at the end of the book!
signature BOOK =

sig
  type page = string list
  datatype direction = FORWARD | BACKWARD
  type book

  exception OutOfBounds

  (* REQUIRES: true
   * ENSURES: bind l = b, such that
   * b encodes all pages in l
   * b is "opened" to the first page in l, if one exists
   * b preserves the order of the pages in l
   *)
  val bind : page list -> book

  (* REQUIRES: true
   * ENSURES:
   * unbind b = l, where l consists of all pages in b
   * unbind (bind l) = l
   *)
  val unbind : book -> page list

  (* REQUIRES: true
   * ENSURES:
   * currentPage b = (true, NONE) iff b is empty
   * currentPage b = (true, SOME p) iff
     * b is "opened" to page p and p is the first page in b
   * currentPage b = (false, NONE) iff
     * b is non-empty and encodes the end of the book
   * currentPage b = (false, SOME p) iff
     * b is "opened" to page p and p is not the first page in b
   *)
  val currentPage : book -> bool * page option

  (* REQUIRES: true
   * ENSURES:
   * flip FORWARD b raises OutOfBounds iff
     * currentPage b encodes the end of b
   * flip FORWARD b = b' otherwise,
     * where currentPage b' encodes the next page in b
   * flip BACKWARD b raises OutOfBounds iff
     * currentPage b encodes the beginning of b
   * flip BACKWARD b = b' otherwise,
     * where currentPage b' encodes the previous page in b
   *)
  val flip : direction -> book -> book
end
3.1.2 A Slow Implementation, and a Faster One

In code/books/SlowBook.sml, you’ll find an implementation of the BOOK signature that works as desired. However, it’s too slow for the average CMU student’s purposes – in order to get to your current page, you have to flip through every single one of the pages first.

Your first task is to improve this structure to make it efficient enough for CMU students.

Task 12. (10 points)

In code/books/Book.sml, write a Book structure opaquely ascribing to the BOOK signature that allows you to get the currentPage and flip a page in constant time.

Constraint: Your currentPage and flip functions must operate in constant time. bind and unbind should be no more than \(O(n)\), where \(n\) is the length of the page list.

Note that while we normally compile and test modules with the command smlnj -m sources .cm, this won’t work until you’ve finished the whole problem. Instead, we recommend manually including files and their dependencies as you write them. For instance, in this task, you can use the following command:

smlnj BOOK.sig Book.sml

As you work through this problem, feel free to add files and their dependencies to this command when you write them.

3.1.3 Testing

Now that you have a much more efficient implementation of books, you want to write some tests that convince your peers of its correctness. However, you don’t want to expose any underlying types or helper functions to others, so you decide to write tests using a different functor, MkTest.

You might wonder, since you do not have access to the underlying types of your Book implementation, how are you supposed to test it? The idea here is called black box testing. For example, when you test Book.bind l, you don’t actually care about the particular value or underlying type of l, but you want to make sure that the value created by Book.bind l behaves the way you expect when you use it with the other Book functions. The following code snippet should give you a sense of how to start:

```json
exception Error

val test = fn () =>
  let
    val B0 = Book.bind []
    val () = case (Book.currentPage B0) of
      (true, NONE) => ()
    | _ => raise Error
    val () = case (Book.unbind B0) of
      [] => ()
    | _ => raise Error
    (* Add more tests here *)
  in
    ()
  end
```

Task 13. (5 points)

In code/books/MkTest.fun, write test cases that perform *black box testing* for the implementation in the Book structure. All your tests should be run in the test : unit -> unit function, and if there is an error in the implementation, your code should raise an Error exception.

With *black box testing*, we ask you to test all the aspects of each function. For example, for Book.flip, you should have at least one test case for the forward direction and at least one test case for the backward direction. One advantage of *black box testing* is that your MkTest structure should be able to test all structures ascribing to the BOOK signature, no matter what their underlying types are. For grading, we will run your MkTest structure against some alternative implementations we come up for Book, and your code should fail on the buggy ones.

Note that this MkTest functor ascribes to the TEST signature, which is included in code/books/TEST.sig. To run your tests locally, open a repl with

```
smlnj BOOK.sig Book.sml TEST.sig MkTest.fun
```

Then type the following command, which should raise Error if any of your tests failed or evaluates to () if all the test passes.

```
structure T = MkTest (Book);
T.test ();
```
3.2 Students

Now that we have an efficient implementation of a book, we can start modeling students! A student can, in theory, look through a book, flipping back and forth as desired, and somehow identify whether they want to use it for their classes or not.

Note, however, that our STUDENT signature only allows a student to look at one page of a book at a time, updating their opinions accordingly.

3.2.1 The Student Signature

We’ll use the STUDENT signature provided in code/books/STUDENT.sig, outlined below:

- The student’s thoughts represent any information they want to store or keep track of as they read.
- The status represents what progress the student has made. It is defined with the following datatype declaration:
  
  ```
  datatype status
  = YES
  | NO
  | MAYBE of thoughts * Book.direction
  ```

  A status of YES means that the student wants to use the book, while NO means that the student has found the book unsatisfactory and wishes to discard it.

  If a student’s status is MAYBE, they are still in the process of reading the book and have not yet come to any conclusions. In this case, the status also stores the student’s current thoughts and the next direction to move in.

- start represents the student’s initial thoughts as they begin to look through the book.

- This kind of student can only read and analyze one page of a book at a time, and will only do so when told. Thus, they need a think function that allows them to update their status given some thoughts and information about the current page (encoded by a bool * Book.page option).

  Just like in the currentPage function of the BOOK signature, the bool in this last piece of information indicates whether the student is at the beginning of the book, while the Book .page option represents the page the student is currently reading, if one exists.

For reference, the STUDENT .sig file is included below.
signature STUDENT =

  sig
    type thoughts
    datatype status = YES | NO | MAYBE of thoughts * Book.direction
  end

val start : thoughts (* the student’s starting thoughts *)

(* REQUIRES: true
* ENSURES:
* Given current thoughts t and a book state p (containing
* information about the current page) to consider:
* think (t,p) = YES iff the student "likes" the book
* think (t,p) = NO iff the student "dislikes" the book
* think (t,p) = MAYBE (t’,d) iff the student remains
* undecided, with updated thoughts t’ and a direction
* d in which to flip to change the state of the book
*)
val think : thoughts * (bool * Book.page option) -> status

3.2.2 Some Useful Students

Now, you’ll implement a few students ascribing to this STUDENT signature.

Task 14. (3 points)

In code/books/Polly.sml, implement a structure, Polly, that opaquely ascribes to the STUDENT signature described above. Polly flips forward through a given book from its current position, liking it if any page contains the word “lambda” and disliking it otherwise.

To compile all files up to this point, run

smlnj BOOK.sig Book.sml TEST.sig MkTest.fun STUDENT.sig Polly.sml

Task 15. (3 points)

What if we want to find more than one instance of “lambda”? In code/books/PollyThree.sml, write a structure called PollyThree, opaquely ascribing to STUDENT. PollyThree is an extreme 150 fanatic, and only likes books that contain at least three instances of the word “lambda” from the current page onward.

To compile all files up to this point, run

smlnj BOOK.sig Book.sml TEST.sig MkTest.fun STUDENT.sig Polly.sml PollyThree.sml

Notice that there isn’t really anything special about the word “lambda,” nor about the number 3 – we might want to do something similar for students of other classes, but we don’t want to have to
rewrite the same structure over and over again. Luckily enough, we can use functors to make our lives easier!

**Task 16.** (3 points)

In `code/books/MkStudent.fun`, write a functor to create a STUDENT who likes only books that contain a certain word a certain number of times (from the current page onward).

Your functor should opaquely ascribe to STUDENT, and take in the following two values:

```plaintext
val count : int
val favoriteWord : string
```

`count` represents the minimum number of times a STUDENT must read the `favoriteWord` in order to respond with `YES` to the book.

To compile all files up to this point, run

```
smlnj BOOK.sig Book.sml TEST.sig MkTest.fun STUDENT.sig Polly.sml PollyThree.sml MkStudent.fun
```

**Task 17.** (1 point)

Now that you’ve written the functor, let’s try it out! In `code/books/Honk.sml`, instantiate your `MkStudent` functor to create a STUDENT called `Honk` that only likes books containing at least one instance of the word “pointer,” from the current page onward.

**Constraint:** Your implementation must use the functor you defined in the previous task.

To compile all files up to this point, run

```
smlnj BOOK.sig Book.sml TEST.sig MkTest.fun STUDENT.sig Polly.sml PollyThree.sml MkStudent.fun Honk.sml
```
3.2.3 Reading a Whole Book

So far, our `think` functions have only been able to look at a single page of a book at a time. What if a student wants to read through an entire book, though, to figure out whether they like it or not?

Your next task is to implement this functionality, based on the signature in `code/books/READER.sig`, shown below.

```sml
signature READER =
sig
  structure Student : STUDENT

  (* REQUIRES: true
   * ENSURES:
   * read b = true iff Student's status is YES
   * upon thinking repeatedly about the pages of b
   * read b = false iff Student's status is NO
   * upon thinking repeatedly about the pages of b
   *)
  val read : Book.book -> bool
end
```

**Task 18. (5 points)**

In `code/books/MkReader.fun`, implement a functor `MkReader` that takes in a `STUDENT` structure and produces a structure opaquely ascribing to the `READER` signature. The `read` function should evaluate to `true` if the student likes the input book, and `false` if they do not.

To compile all files up to this point, run

```bash
smlnj BOOK.sig Book.sml TEST.sig MkTest.fun STUDENT.sig Polly.sml PollyThree.sml MkStudent.fun Honk.sml READER.sig MkReader.fun
```

With these new tools, the students of Gates can navigate their course materials much more easily, thanks to you! But you’re not quite done yet.

Before you move on, test this `MkReader` functor out on some of your students to get a better sense of what’s going on.
3.3 More Complicated Students

Polly and Honk, as the representatives for 15-150 and 15-122, each guard several secret guidebooks for their classes. These guidebooks are of the highest priority, since they contain everything from notes for TAs to homework reference solutions to midterm and final exams. Somehow, Polly and Honk have gotten their books all mixed up, and it’s your job to return the correct books to the correct mascot before any confidential materials get out.

To do so, you’ll need to create a student who can identify books that contain the word “lambda” but not the word “pointer” – a 150 guidebook would never have a word like “pointer,” but would always include “lambda” somewhere.

3.3.1 Making a Nemesis

Task 19. (3 points)
To start off, implement a MkNemesis functor in code/books/MkNemesis.fun that takes in a STUDENT and produces another structure opaquely ascribing to STUDENT. This output student likes all books that the original student disliked and dislikes all books that the original student liked.

To compile all files up to this point, run

```
smlnj BOOK.sig Book.sml TEST.sig MkTest.fun STUDENT.sig Polly.sml PollyThree.sml MkStudent.fun Honk.sml READER.sig MkReader.fun MkNemesis.fun
```

The MkNemesis functor will allow us to create a student that checks for the lack of a word, rather than the presence of one.

3.3.2 Composition

Although we’ve made some progress, we still need a way to consider the opinions of not just one, but two students. To do so, you’ll want something that allows you to sequentially compose the actions of two students. In other words, you want one student to be able to look through the book, and then pass it on to the other, to check whether both students like it or not.

Task 20. (7 points)
In code/books/MkCompose.fun, write a MkCompose functor that implements sequential composition. This functor should take in two STUDENTS, called StudentA and StudentB, and produce another structure opaquely ascribing to STUDENT.

The resulting student should only like a given book if the first student likes it and passes it along (without closing the book or restarting from the beginning) to the second student, who also likes it. If either student dislikes the book, the resulting student must also dislike it.

Note that StudentB should start reading the book from wherever StudentA left off – the currentPage should not change between when the first student finishes looking at the book and when the second student starts looking at it.

Hints:

- Think carefully about the types!
• If Gradescope is giving back errors, make sure you have correctly named the structures passed into this functor; they should be `StudentA` and `StudentB`. See the Appendix on functor syntax for more details.

To compile all files up to this point, run

```
smlnj BOOK.sig Book.sml TEST.sig MkTest.fun STUDENT.sig Polly.sml  
PollyThree.sml MkStudent.fun Honk.sml READER.sig MkReader.fun  
MkNemesis.fun MkCompose.fun
```

We’ve nearly achieved what we wanted, but notice that sequential composition isn’t quite the same as checking whether two different properties hold over an entire book. The key difference is that the second student in sequential composition will start from where the first student left off, rather than from the beginning of the book.

To get around this, we’ll want to somehow reset the book before the second student gets it. We can do so by flipping all the way to the beginning of the book – in fact, we can even make a student do this!

**Task 21.** (3 points)

In `code/books/Reset.sml`, implement a `Reset` structure ascribing to the `STUDENT` signature from earlier.

This student should do nothing but flip to the beginning of the book, and then act as an identity for `MkCompose`. In other words, composing this `Reset` student with any other student should not affect the output of the other student, but should force the other student to start from the beginning of a book.

To compile all files up to this point, run

```
smlnj BOOK.sig Book.sml TEST.sig MkTest.fun STUDENT.sig Polly.sml  
PollyThree.sml MkStudent.fun Honk.sml READER.sig MkReader.fun  
MkNemesis.fun MkCompose.fun Reset.sml
```

### 3.3.3 Putting It All Together

Now that we have all the pieces, let’s get back to helping Polly and Honk.

**Task 22.** (5 points)

In `code/books/Find150.sml`, implement a `Find150` structure that opaquely ascribes to the following signature:

```sml
sig
  val is150Book : Book.page list -> bool
end
```

Here, `is150Book b` \(\Rightarrow\) `true` iff `b` contains the word “lambda” but no instances of the word “pointer,” and `false` otherwise.

**Constraint:** You must use only structures and functors you defined in previous tasks; you may not write any new structure expressions, i.e. you may not use the `struct` keyword outside of what is already included.
Since this is the last task, you can compile all your files with `smlnj -m sources.cm`!

Congratulations! You’ve helped Polly and Honk get their books back, and restored order at CMU.
4 Representation Independence

4.1 Motivation

Throughout the course, we’ve heavily emphasized the notion of 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 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. If, say, \( e_2 \) is more efficient to evaluate than \( e_1 \), then replacing every instance of \( e_1 \) in my code with \( e_2 \) will improve the efficiency of the code without having any impact on its behavior/correctness. 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 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 representation independence proof.

4.2 Queues

As discussed in lecture, a queue is a first-in, first-out sequential data structure. That is, it is a structure which stores some number of elements of the same type in order, and allows you to insert new elements and remove elements from the queue. The “first-in, first-out” part refers to the fact that new elements are inserted at the end of the queue and elements are removed from the front of the queue, so the first elements to be added will be the first ones removed. This is often contrasted with stacks, which are last-in, first-out.

For instance, if I had an empty queue of integers, and inserted 4, 5, 2, and 7 (in that order), I’d have a queue like

\[
4 \quad 5 \quad 2 \quad 7
\]

If I removed one element of the queue, it’d be 4 (because 4 was first in), and the resulting queue would be

\[
5 \quad 2 \quad 7
\]

If I then added a 11 to the queue, I’d have

\[
5 \quad 2 \quad 7 \quad 11
\]

Easy enough, right?
4.3 The Code

We represent queues with the following signature:\textsuperscript{2}:

\begin{verbatim}
signature QUEUE =
sig
  type 'a queue
  val emp : 'a queue
  val ins : 'a * 'a queue -> 'a queue
  val rem : 'a queue -> ('a * 'a queue) option
end
\end{verbatim}

In this signature,

- \texttt{'a queue} is the abstract type of an \texttt{'a queue}
- \texttt{emp} represents the empty queue
- \texttt{ins} adds an element to the end of the queue
- \texttt{rem} removes the element at the front of the queue and returns it with the remainder of the queue. When called on the empty queue, \texttt{rem} should return \texttt{NONE}

As discussed in lecture, there are two standard ways to implement the queue signature: the 1-stack and 2-stack implementations. We call these \texttt{LQ} and \texttt{LLQ}, respectively. Their implementations are given on the next page.

- \texttt{LQ} implements an \texttt{'a queue} as a single \texttt{'a list}. The empty list represents the empty queue, and the head of the list represents the first element of the queue. New elements must therefore be appended (using \texttt{@}) to the back of the list, whereas removing an element consists of removing the head of the list by pattern-matching.

  This implementation has constant-time removal, but linear (in the length of the queue) insertion time (because we must perform the linear-time \texttt{@} every time we add a new element to the queue). This implementation also has the advantage of being easy to understand.

- \texttt{LLQ} implements queues as a pair of lists, which we’ll call \texttt{front} and \texttt{back}. \texttt{front} represents the front of the queue, so, if \texttt{front} is nonempty, then its first element is the element at the front of the queue (and so \texttt{rem} takes the head from \texttt{front}). \texttt{back} stores the back of the queue in reverse order, so the last element of the queue is the first element of \texttt{back}. So adding a new element consists of cons-ing the element to the front of \texttt{back}, which takes only constant time.

  In order for this to work, we occasionally need to transfer the elements from \texttt{back} to \texttt{front}. If we call \texttt{rem} on a queue of the form \texttt{([], back)}, then – assuming \texttt{back} is nonempty – the first element of the queue is actually the last element of \texttt{back}. So we move \texttt{back} to \texttt{front}, reversing the order (because, remember, \texttt{front} is stored in order whereas \texttt{back} is stored in reverse order), and then take the head from the reversed \texttt{front}. This operation takes linear time\textsuperscript{3}, but is done infrequently enough that the cost of \texttt{rem} is essentially constant\textsuperscript{4}.

\textsuperscript{2}Which might be slightly different from what you saw in lecture, although the idea is the same
\textsuperscript{3}Assuming \texttt{List.rev} is implemented in the fast, tail-recursive way: \texttt{List.foldl (op ::) []}.
\textsuperscript{4}We can make this precise with amortized analysis, which we don’t cover in 150
4.4 The Proof

Recall from lab that a representation independence proof consists of defining a relation between the two implementations (which describes which values of one implementation we view as equivalent) and then proving that all the operations in the signature preserve the relation. More formally, we define a relation \( R \) between valuable expressions of type \( \text{int list} \) (the implementation of the abstract type \( \text{int queue} \) in LQ) and \( \text{int list} \times \text{int list} \) (the implementation of \( \text{int queue} \) in LLQ).

We define the relation for you. For concreteness, we consider only \( \text{int queues} \), but this technique will work generally.

**Definition 1.** \( R \) is the relation between valuable expressions of type \( \text{int list} \) and valuable expressions of type \( \text{(int list \times int list)} \) given by

\[
R(L, (f, b)) \text{ if and only if } L \equiv f @ (\text{rev } b)
\]

Convince yourself that \( R(L, (f, b)) \) means that \( L \) and \( (f, b) \) represent the same queue. Furthermore, convince yourself that \( R \) respects extensional equivalence: if \( L \equiv L' \) and \( (f, b) \equiv (f', b') \) and \( R(L, (f, b)) \), then \( R(L', (f', b')) \).

Now on to your task: we prove that all operations in the signature respect this relation. If \( L \) and \( (f, b) \) represent the same queue, then calling \text{ins} or \text{rem} should have the same result. This is made formal in Theorem 1.
Task 23. (15 points)
Prove the following theorem:

Theorem 1.

(i) The empty queues are related:

\[ R(L.Q.\text{emp}, LL.Q.\text{emp}) \]

(ii) Insertion preserves \( R \): for all valuable \( x: \text{int} \), \( L: \text{int list} \), \( f: \text{int list} \), \( b: \text{int list} \),

\[ \text{If } R(L, (f, b)), \text{ then } R(L.Q.\text{ins}(x, L), LL.Q.\text{ins}(x, (f, b))) \]

(iii) On related queues, removal gives equal integers and related queues: for all valuable \( L: \text{int list} \), \( f: \text{int list} \), \( b: \text{int list} \), if \( R(L, (f, b)) \) then one of the following is true.

(a) \( L.Q.\text{rem} L \cong \text{NONE} \) and \( LL.Q.\text{rem}(f, b) \cong \text{NONE} \)

(b) \( L.Q.\text{rem} L \cong \text{SOME}(x1, L') \) and \( LL.Q.\text{rem}(f, b) \cong \text{SOME}(x2, (f', b')) \) such that \( x1 \cong x2 \) and \( R(L', (f', b')) \)

You may use the following lemmas without proof (though clearly cite them whenever you use them).

Lemma 1. For all valuable \( A: \text{int list} \), \( B: \text{int list} \), \( C: \text{int list} \),

\[(A @ B) @ C \cong A @ (B @ C)\]

Lemma 2. For any \( A, B \) of type \( \text{int list} \), \( A @ B \cong [] \) if and only if \( A \cong [] \) and \( B \cong [] \).

Lemma 3. For all valuable \( x: \text{int} \), \( y: \text{int} \), \( A: \text{int list} \), \( B: \text{int list} \),

\[x::A \cong y::B \text{ if and only if } x \cong y \text{ and } A \cong B\]

Lemma 4. For all \( A: \text{int list} \),

\[A @ [] \cong A \cong [] @ A\]

You may assume without proof that \( @ \), \( \text{rev} \), and all the functions in the structures are total. You may assume that \( @ \) and \( \text{rev} \) are implemented as follows.

```ml
infix @
fun [] @ L = L
  | (x::xs) @ L = x::(xs @ L)
fun rev [] = []
  | rev (x::xs) = (rev xs)@[x]
```
5 In Simpler Terms

In this problem, we’re going to build a tool that can simplify mathematical terms.

5.1 The Approach

The easiest approach to simplifying is repeatedly applying rewrite rules, which show how you could turn some “left-hand side” into some “right-hand side”. For example, we’ll probably want the rules (given variables x and y):

\[ x + \text{zero} \rightarrow x \]  
\[ x \times \text{zero} \rightarrow \text{zero} \]  
\[ x \times \text{succ}(\text{zero}) \rightarrow x \]  
\[ \text{abs}(-x) \rightarrow \text{abs}(x) \]  
\[ \text{abs}(x \times y) \rightarrow \text{abs}(x) \times \text{abs}(y) \]

How do we apply such rules? It’s simple! Once we have a term, you attempt to match it to the left-hand side of each rule. If it works, you’ll get some substitution, which binds the variables in the left-hand side. From here, it’s easy to perform the rewrite: just apply that substitution to the right-hand side!

For example, suppose we’re given term \(\text{abs}(\sin(\theta) \times \pi)\). We would match it against the left-hand side of Rule 5, getting the substitution \(\sigma = \left\{ \sin(\theta) / x, \pi / y \right\} \). Then, we can rewrite by applying this substitution \(\sigma\) to the right-hand side, plugging in \(x\) and \(y\) to get \(\text{abs}(\sin(\theta)) \times \text{abs}(\pi)\).

**Constraint:** Make sure you fully understand this before proceeding.

5.1.1 Substitutions

We store substitutions as dictionaries using red-black trees, as seen in lecture. In code/algebra/core/Subst.sml:

```sml
structure Subst = MkRedBlackDict (  
  struct  
    type t = string  
    val compare = String.compare  
  end  
)
```

We instantiate a dictionary with string keys, which will be our variables (such as \(x\) and \(y\) above). For example, we would represent \(\sigma = \left\{ \sin(\theta) / x, \pi / y \right\} \) as a dictionary with key \(x\) mapping to term \(\sin(\theta)\) and key \(y\) mapping to \(\pi\).

Observe that all other code for the duration of this problem need not (and, in fact, may not!) consider the implementation of dictionaries used, due to modular abstraction.

Note that we extend the DICTIONARY signature with a union function, which combines two dictionaries given an equality function, failing if the values associated with any two equal keys are not the same. We make use of this in some of the code we provide you with.
5.1.2 Naïve Implementation

The easiest implementation would use a datatype similar to rose trees.\footnote{a tree where each node has a list of other nodes, and is thus not limited to exactly two children.}

datatype term = Variable of Subst.key | Node of string * term list

Here, a term represents a mathematical term where each variable is stored using a Subst.key identifier (which, recall, is string in our case) and each node consists of a function symbol (as a string) and a list of “arguments”.

- The Variable constructor allows you to construct variables which are matched against, such as x and y from above.
- The Node constructor allows you to construct any other node in the term tree. For example, abs(\text{argument}) would be Node ("abs", [\text{argument}]), and pi would be Node ("pi", []).

Matching should be simple: given pattern : term and a subject : term to match, run through both trees. If we reach a variable in pattern, assign it to the entirety of subject. Otherwise, assuming we’re at a node in both pattern and subject, check the strings for equality (failing if they’re different) and union together the substitutions from recursive calls on the child terms.

5.1.3 A Clever Idea

At first, it seems like this approach should work—and it does! However, the design isn’t particularly extensible.

Suppose we wanted to simplify zero * sin(theta) to zero. If we use the rules from the previous page, this seems tricky—the argument order is flipped! However, since multiplication is commutative, we should be able to succeed anyway.

One simple approach would be adding a special case to the matching function itself, looking for string "*". However, we’ll likely have more special cases to handle later (e.g. "+"), so this seems unappealing.

Additionally, we may want to represent various nodes within the tree in different ways. For example, our lives would be made easier if we could store commutative functions as a tuple, rather than as a list of length 2. Then, the types guide our thinking more naturally, rather than having undesired cases (e.g. what does Node ("*", []) mean?).

With this in mind, we can augment our datatype! Let’s use the following signature to define which functions a term-like data structure must implement, for our purposes:
Some “matchable” must be able to represent a term with arguments: its abstract type should be a data structure storing the data at a node, given that its child nodes have type ‘a. (Often, this data structure will not be recursive.)

We require a handful of other functions, as well; the most interesting ones are:

- The `hide` function turns a `string * 'a list` (a function name and a list of arguments) into our custom node representation (or raises an exception if this is not possible).

- The `match` function matches two nodes against each other, given that we know how to match children of the pattern.

**Constraint:** Carefully understand this signature! It will be used heavily throughout the rest of the problem.

Now, we can upgrade our definition of `term` from before, while still using the same general strategy for matching as previously:

```
datatype term = Variable of Subst.key | Node of term Matchable.t
```

Instead of storing a `string * term list` at the nodes, we store a `term Matchable.t`, where `Matchable.t` is some abstract container type (for a given `structure Matchable : MATCHABLE`).

We can encode this in a `TERM` signature:
Most of the functions are simple generalizations of their counterparts from signature MATCHABLE, handling the Variable case and otherwise calling the corresponding function from Matchable. There are useful utility functions, as well, such as substitute and toString.

So in our updated model, we have terms which represent trees—ones which have the ability to encode more information at the type level than those in our initial approach.

We provide you with functor MkTerm, which turns a MATCHABLE into a TERM.

### 5.2 Fn0Matchable

Let’s take a look at one of the simplest matchables imaginable: data structures which don’t even store any arguments! We can accomplish this by setting type ‘a t = unit. While the resulting “tree” won’t be particularly interesting (no children are allowed!), this lets us make constants, like zero and pi.
**Fn0Matchable** is a functor which takes in a symbol name. It represents `a t as unit` (wow!), since we need no information (such as arguments)—we weed out all other cases in `hide`.

Check out the following REPL session—it combines everything we’ve seen so far to solve a simple task (matching subject `pi` against pattern `x`).

You can run it yourself with:

```
  smlnj -m sources.cm
```

```
(* create Term structure for matchable containing one symbol, pi *)
  - structure Term = MkTerm (Fn0Matchable (val name = "pi"));
structure Term : TERM

(* create value pi by hiding "pi" with no arguments *)
  - val pi = Term.hide ("pi",[]);
val pi = Node - : Term.term

(* match pi against variable x, successfully producing a substitution *)
  - val SOME subst = Term.match (Term.Variable "x", pi);
val subst = - : Term.term Subst.t

(* find the value bound to x *)
  - val SOME xBoundTo = Subst.lookup subst "x";
val xBoundTo = Node - : Term.term
(* hmm, not sure what that is... *)

(* convert it to a string *)
  - Term.toString xBoundTo;
val it = "pi" : string
```

Notably, the type `term`, given `structure Matchable = Fn0Matchable (val name = "pi")`, now looks like:

```
datatype term
  = Variable of Subst.key
  | Node of unit
```

In other words, our terms can be either a variable (e.g. `Variable "x" : string term`) or the one non-variable node we know about, `Node ()`, representing `pi`.

This is, obviously, quite boring—doing math with only one total symbol doesn’t seem sufficient. Onward!

### 5.3 **BoolMatchable**

Why have one symbol when you could have two?

**Task 24.** (4 points)

In `code/algebra/matchables/BoolMatchable.sml`, declare a structure `BoolMatchable` whichopaquely ascribes to `MATCHABLE`, such that:
• `BoolMatchable.hide("false",[])` and `BoolMatchable.hide("true",[])` are both accepted by `hide`, and any other inputs raise `BoolMatchable.Invalid s`, for any error message string `s` you choose.

• `BoolMatchable.show(BoolMatchable.hide(str,[])) \cong (str,[])`, given `str = "false"` or `str = "true"`.

• Matching the two `BoolMatchable.t` instances succeeds with the empty substitution if they are the same (e.g. `BoolMatchable.hide("false",[])` against `BoolMatchable.hide("false",[])`) and produces `NONE` otherwise.

• `BoolMatchable.map f` should be the identity function, since booleans have no arguments to map over.

**Constraint:** You must use opaque ascription, so your representation cannot be known from outside of the structure.

For example:

```haskell
(* make Term structure using BoolMatchable *)
- structure Term = MkTerm (BoolMatchable);
structure Term : TERM

(* matching symbol false against symbol false *)
- Term.match (Term.hide("false",[]), Term.hide("false",[]));
  val it = SOME : Term.term Subst.t option

(* matching symbol true against symbol false *)
- Term.match (Term.hide("false",[]), Term.hide("true",[]));
  val it = NONE : Term.term Subst.t option

(* matching symbol true against variable x *)
- Term.match (Term.Variable "x", Term.hide("true",[]));
  val it = SOME : Term.term Subst.t option

(* matching symbol true against variable (!) false *)
- Term.match (Term.Variable "false", Term.hide("true",[]));
  val it = SOME : Term.term Subst.t option
```

Aha! Math just got more interesting - we gained the ability to *case*, since some of our matches fail. On the other hand, we still only have two constants to work with. More work to do...

### 5.4 Functions - This Time, With Arguments!

We’ve already handled the function case for constants - `pi` is really a node with zero children. Now, we can quickly handle unary and binary functions in `code/algebra/matchables/Fn1Matchable.fun` and `code/algebra/matchables/Fn2Matchable.fun`.

---

6If we wanted to use notation which is less pretty but more explicit, we might consider writing `pi[]` and `sin[theta][]` instead of `pi` and `sin[theta]`. 32
5.4.1 Either Way

It would be great if we could test this code. However, there's an issue - how would we even construct such a term if we're only allowed to use one matchable at a time? Suppose we define:

```
structure Term = MkTerm (Fn1Matchable (val name = "not"))
```

It would be awesome if we could represent terms like `not(not(false))`, using `BoolMatchable` and `Fn1Matchable (val name = "not")` at the same time. However, the given structure declaration says nothing about `BoolMatchable` - oh no!

If only we had a functor which could glue two matchables together...

**Task 25.** (3 points)

In `code/algebra/matchables/MPLikeMatchable.fun`, write a functor `MkEitherMatchable` which glues two matchables together.

Don't worry - we've started it off! The functor takes in two structures ascribing to `MATCHABLE`, named `A` and `B`.

The type we chose is:

```
datatype 'a t
  = FromA of 'a A.t
  | FromB of 'a B.t
```

In other words, the data structure to use simply tags the data structure from `A` or tags the data structure from `B`.

We implemented `hide`, which may give you some intuition regarding how to proceed for the remaining functions. Feel free to make design decisions as you wish, as long as they are consistent with the signature types and specifications.

Here are some examples of how it should work:

```
(* create structure T, representing terms which can use booleans and
  the not function *)
- structure T = MkTerm (MkEitherMatchable (structure A = BoolMatchable
  structure B = Fn1Matchable (val name = "not")));
structure T : TERM

(* helper function *)
- fun not (x : T.term) = T.hide ("not",[x]);
val not = fn : T.term -> T.term

(* not(not(true)) *)
- val term = not (not (T.hide ("true",[]))) : T.term;
val term = Node - : T.term
```

7Remember, the client has no way to tell what choices you make inside the structure!
(* not(x) *)
val pattern = not (T.Variable "x") : T. term;
val pattern = Node - : T. term

(* attempt matching *)
val SOME subst = T. match (pattern, term);
val subst = - : T. term Subst.t

(* find the value bound to x *)
val SOME xBoundTo = Subst.lookup subst "x";
val xBoundTo = Node - : Term.term

(* show the term bound to x *)
val it = "not[true]" : string

5.5 Working our Way Back Up

We’ve built up our matching arsenal quite a bit - we can now work with constants, unary functions, and binary functions, all at once via the MkEitherMatchable functor! Revisiting one of our original goals, though: how do we deal with commutative functions?

Intuitively, we could use code similar to that in Fn2Matchable, but change the matching procedure to attempt both permutations.

Task 26. (3 points)
Implement CommMatchable.match in code/algebra/matchables/CommMatchable.fun such that it matches both permutations of the arguments, failing only if neither works.

Now, we can incorporate whatever commutative function symbols we want. Nifty!

5.6 Simplification Nation

In code/algebra/simplify/MkSimplifier.fun, we have provided you with the original simplification algorithm we described:

- Given a list of rules (pairs of left-hand side and right-hand side), try to find a rule which matches.
- If one matches, perform the rewrite.
- Otherwise, we’re finished.

Simplification just involves applying this algorithm repeatedly, and at every node in the term tree.

In the code/algebra/simplify/STerm.sml, we set up the combination of a handful of matchables.

In code/algebra/simplify/Basic.sml, we give some simple rules for simplifying addition, multiplication, and sine on constants.
5.6.1 Differentiation

Your job: write rules for symbolic differentiation!

**Task 27.** (5 points)

In `code/algebra/simplify/Diff.sml`, define `Diff.rules : Simp.rule list` which implements symbolic differentiation.

If you let `structure Test = MkTest (Diff)` and run `Test.run ()`, all tests should pass.

**Constraint:** You must use no more than six rules. In other words:

```ml
val true = List.length Diff.rules <= 6
```

**Hint:** Consider which cases are relevant. For example, \( \frac{\partial}{\partial t} (t) = 1 \) and \( \frac{\partial}{\partial t} (x + y) = \frac{\partial}{\partial t} (x) + \frac{\partial}{\partial t} (y) \).

And that's it - via the power of modular abstraction, we were able to implement a customizable mathematical simplifier!\(^8\)

---

\(^8\)Obviously, we could consider this problem in substantially more detail, dealing with associative function symbols, attempting to store numbers more efficiently, and such; if you're interested, feel free to look up “term rewriting”.

A  Functor Syntax

Just like how functions must take in exactly one argument, functors must take in exactly one structure:

```sml
functor MkName (StructName : SIGNAME) :> OTHERSIGNAME =
  struct ... end
```

However, explicitly wrapping inputs using the `sig` and `struct` keywords can get syntactically verbose. To help us, SML provides syntactic sugar that allows us to omit this wrapping in the argument to a functor.

A.1  Syntactic Sugar

We can put anything that can go between `sig ... end` in the argument to a functor.

For example,

```sml
functor MkBar (val x : int structure B : BOOL) = struct end
```

is sugar for

```sml
functor MkBar (UnknowableName : sig val x : int structure B : BOOL end) = let open UnknowableName in struct end end
```

A.1.1  Tricky: One Input Structure

The following two functors have different types:

```sml
functor MkFoo (A : SIGNAME) = ...
functor MkFoo' (structure A : SIGNAME) = ...
```

While `MkFoo` takes in a structure that ascribes to the signature `SIGNAME`

`MkFoo'` takes in a structure that ascribes to the signature `sig structure A : SIGNAME end`

Calling the former looks like

```
MkFoo (Arg)
```

while calling the latter looks like

```
MkFoo' (structure A = Arg)
```

Since these functors take in structures ascribing to different signatures, Gradescope will tell you that your code doesn’t typecheck when you use the latter approach instead of the former.

Try putting the following two lines into the REPL to see the differences yourself.

```sml
functor Example1 (A : BOOL) = struct end;
functor Example2 (structure A : BOOL) = struct end;
```
A.2 “Multiple” Arguments

We can allow a functor to take in multiple structures by packaging them up into one structure that contains multiple structures, similarly to how we can effectively take in multiple arguments by taking in a tuple.

Sugared:

```haskell
functor MkPair (
  structure A : SIGA
  structure B : SIGB
) = (* uses A and B *)
```

Desugared:

```haskell
functor MkPair (S :
  sig
  structure A : SIGA
  structure B : SIGB
end
) = (* uses S.A and S.B *)
```

This assignment has a total of 99 points.