Contents

1 What is CPS? 3
  1.1 Tail Calls .................................................. 3
  1.2 Continuations ............................................. 3
  1.3 CPS .......................................................... 3
  1.4 Guidelines .................................................. 4

2 sepyT tuohA noitseuQ A 5
  Task 1. (1 point) ................................................. 5
  Task 2. (1 point) ............................................... 5
  Task 3. (1 point) ............................................... 5

3 Back to the usual 6
  Task 4. (2 points) .............................................. 6
  Task 5. (2 points) .............................................. 6
  Task 6. (2 points) .............................................. 6

4 Down the Rabbit-hole 7
  Task 7. (2 points) .............................................. 7
  Task 8. (2 points) .............................................. 7
  Task 9. (2 points) .............................................. 7
  Task 10. (2 points) .......................................... 8
  Task 11. (2 points) .......................................... 8
  Task 12. (2 points) .......................................... 8

5 A Shrubbery! 9
  5.1 One or two should do? ..................................... 9
  Task 13. (5 points) ........................................... 9
  Task 14. (10 points) ......................................... 10
  Task 15. (15 points) ......................................... 11

6 I’m the Map! 12
  Task 16. (15 points) ........................................ 12

7 Can’t Prop Won’t Prop 13
  7.1 Evaluation .................................................. 13
  Task 17. (10 points) ........................................ 13
  7.2 Assignments ................................................ 15
  7.3 (I Can’t Get No) Satisfaction ............................ 16
  Task 18. (15 points) ........................................ 16
8 The Curry-Devoured Correspondence
  8.1 Introduction .......................................................... 17
  8.2 Version 1: Not really CPS .............................................. 18
    Task 19. (1 point) .................................................. 18
  8.3 Version 2: A bit of CPS ............................................... 19
    Task 20. (3 points) .................................................. 19
  8.4 Final Version: So much CPS ......................................... 20
    Task 21. (15 points) ............................................... 20

9 Don’t burn the curry!
    Task 22. (17 points) ............................................. 23

10 At least we teach a useful language ................................. 25
  10.1 Cnot’s not C0 ...................................................... 25
  10.2 Semantics of Cnot ................................................. 26
  10.3 Implementing the interpreter .................................... 26
    Task 23. (14 points) ............................................... 27
  10.4 Testing your interpreter .......................................... 28

A Appendix: Is this CPS? ................................................. 30
  A.1 fact ................................................................. 30
  A.2 size ............................................................... 30
  A.3 find ............................................................... 31

B Appendix: Cnot programs ............................................. 31
  B.1 Well-formedness of Cnot programs .............................. 33
  B.2 Cnot environments ............................................... 33
  B.3 Cnot expressions ................................................ 33
1 What is CPS?

1.1 Tail Calls

In functional programming, we very frequently have functions which call other functions. For example, the function foo below makes two calls to other functions: a recursive call to itself and a call to some other function bar:

```ml
fun bar (x,y) = x * y
fun foo 0 = 1
  | foo n = bar (n, foo (n-1))
```

We make the distinction between tail-calls and non-tail-calls, based on whether the result of the recursive call is modified or not. If the evaluation of a function f makes a call to a function g, we say that it is a tail call if f does not modify or examine the result of evaluating g.

In the example above, the recursive call of foo to itself is not a tail-call, because the evaluation of foo n will take the result of calling foo (n-1) and then do something with that value (namely pass it as an argument to the function bar). By contrast, when foo calls bar, it is considered a tail-call, because foo(n) does not modify the value returned by the call to bar.

We say that a function is tail recursive if every recursive call it makes is a tail-call.

1.2 Continuations

It is a useful (and perhaps surprising) fact that every SML function can be written in tail recursive form. In order to achieve this, we make use of continuations. Continuations are a way to more clearly express the control flow of a program. In this class, we implement continuations as additional function arguments. For example,

```ml
(* fact : int -> (int -> 'a) -> 'a
 * REQUIRES: n >= 0
 * ENSURES: fact n k = k(n!)
 *)
fun fact 0 k = k 1
  | fact n k = fact (n-1) (fn res => k (n*res))
```

First of all, convince yourself that this function meets its spec, and additionally is tail recursive. Furthermore, notice that the type forces it to be tail-recursive: suppose the return type of k is t. If fact makes a recursive call to itself, then the result of that recursive call will also have type t. But t could be anything, so there’s no way for us to modify or case on the output of this recursive call.

So, for the purposes of this class (and in particular for this homework), think of continuations as functions which we pass in as arguments and which get applied to the result of our computation. The function k in the above example is a continuation, because it is passed in as an argument to fact, and the value of fact n k is the function k applied to n!. Most of the continuations you’ll see in this class will have polymorphic return type.

1.3 CPS

For the purposes of this class, we’ll say that a function f is in continuation passing style if:
• \( f \) takes (at least one) continuation as an argument

• If \( f \) makes a call to a function \( g \) which itself has continuation(s), then this call is a tail call

• \( f \) only calls its continuation(s) in a tail call (e.g. \( f \) does not call its continuation and then try to do something with the result)

In particular, if \( f \) is recursive, then it must be tail recursive: if \( f \) made a recursive call to itself which was not a tail-call, then \( f \) would violate the second condition above.

**Constraint:** For the purposes of this class, if we say to implement a function in continuation-passing style, you *may not* do all the work in a non-CPS helper then call it from the main CPS function. If we feel that you trivialized a problem by calling a non-CPS helper, we reserve the right to deduct all the points for that problem.

### 1.4 Guidelines

You *may* do the following in a CPS function:

• Case on the value of the input

• Use \texttt{let \ldots in \ldots end} expressions

• Write recursive functions

You *may not* do the following in a CPS function:

• Manipulate, case on, or otherwise use the result of recursive calls. (This would break the tail-call requirement.)

• Manipulate, case on, or otherwise use the result of a call to a CPS helper function.

• Manipulate, case on, or otherwise use the result of a call to the continuation.

• Do all the work in a non-CPS helper.

In the appendix, we provide some examples of code which “appears” to be written in CPS but which actually violates the specification given above.
2 sepyT tuobA noitseuQ A

In code/reverse-types/types.sml, you are given a series of types, and your task is to implement a value with the given type. You may assume that any function you take as an argument is total. Feel free to look at the lecture notes for help - they might come in handy if you’re having trouble. Your answers do not need to be written in continuation-passing style, though you might find it helpful to do so anyway.

Constraint: In order to receive credit:

- your functions must compile and must be valuable when passed all curried inputs (assuming totality of any argument functions).
- The type specified in the task must be the most general type of your answer. For example, 'a -> 'a is a more general type than int -> int.
- You may not use any type constraints <exp>:<type>.

Task 1. (1 point)
val task1 : ('a -> 'b) -> 'a -> 'b

Task 2. (1 point)
val task2 : ('a -> bool) -> 'a -> ('a -> 'b) -> (unit -> 'b) -> 'b

Task 3. (1 point)
val task3 : ('a -> 'b) -> 'a list -> ('b list -> 'c) -> 'c
3 Back to the usual

Now, we're back to the usual types of types questions. For each of the following expressions, state the type of the expression (if it is not well-typed, put “ill-typed”).

Though you may plug these expressions into the SML/NJ REPL and get an answer, please remember that we might ask you similar questions on an exam (so it would be well-advised to learn how to solve these types of problems without the REPL).

Task 4. (2 points)

```sml
fun g (s : bool -> 'a) (0 : int) = s true
  | g s 1 = s false
  | g s n = not (g (fn x => x) (n - 1))
```

Task 5. (2 points)

```sml
fun make 0 k = k 0
  | make 1 k = k 1
  | make n k = make (n - 1) (fn a => make (n - 2) (fn x => x) + a)
```

Task 6. (2 points)

```sml
fun transform f [] k = k []
  | transform f (x::xs) k = f x (fn y => transform f xs (fn ys => k (y :: ys)))
```
4 Down the Rabbit-hole

Exceptions allow us to control what a function can and cannot do, and give meaningful feedback when code does something it shouldn’t. Perhaps more importantly, we can use handle to recover from an expression that raises an exception.

For each of the following expressions:

- If the expression is well-typed, give the type. If it is not well-typed, explain why (in at most one sentence).
- If the expression reduces to a value, state what that value is. If the expression raises an unhandled exception, state what that exception is. If the expression does not reduce to a value and does not raise an unhandled exception, then state so explicitly.

You may assume that Fail, Bind, and Match are the only exn defined at top-level.

As always, you’re welcome to just plug these expressions into the REPL to check your work, but it’s in your best interest to understand how they work and why they exhibit the behavior they do (in particular, you won’t have access to the smlnj REPL on an exam).

Task 7. (2 points)

(fn x =>
  if x > 3 then
    4
  else
    raise Fail "If you don’t know where you’re going"
) 2 handle _ => 5

Task 8. (2 points)

(fn x =>
  if x > 3 then
    4
  else
    raise Fail "Any road can take you there"
) 5 handle _ => "8"

Task 9. (2 points)

let
  exception bike
  exception bars
in
  "I can " ^ "ride my " ^ (raise bike) ^ "with no " handle bars =>
  "but I always wear my helmet"
end
Task 10. (2 points)

```ml
let
  exception bike
in
  "I can " ^ "ride my " ^ (raise bike) ^ "with no " handle bars =>
  "but I always wear my helmet"
end
```

Task 11. (2 points)

```ml
let
  val y :: ys = []
in
  case ys of
    [] => 0
end handle
  Bind => 1
| Match => 2
```

Task 12. (2 points)

```ml
let
  val y::ys = [[]]
in
  case ys of
    [[]] => 0
end handle
  Bind => 1
| Match => 2
```
5 A Shrubbery!

5.1 One or two should do?

Consider the following datatype for shrubs. A “t shrub” is a tree-like data structure that stores values of type t in its Leaves (this is different from a “t tree”, which stores its values in its Nodes).

datatype 'a shrub = Leaf of 'a | Branch of 'a shrub * 'a shrub

Also note that unlike trees there is no empty shrub; every shrub stores at least one value.

Task 13. (5 points)

In code/findn/findn.sml, write a continuation-passing style function

```
findOne : ('a -> bool) -> 'a shrub -> ('a -> 'b) -> (unit -> 'b) -> 'b

REQUIRES: p is total
ENSURES: findOne p T sc fc ⇒

\{ sc v where v is the leftmost value in T such that p v ⇒ true
fc () if no such v exists
```

**Constraint:** You may NOT write any additional helper functions for this task unless they are used as continuations.

When we say “leftmost”, we mean that `findOne` should prioritize the left sub-shrub over the right sub-shrub. For example,

```
val T = Branch (Leaf 1, Leaf 2)
val p = fn x => x > 0
val sc = SOME
val fc = fn () => NONE

val SOME 1 = findOne p T sc fc
```
Task 14. (10 points)

You’ll now write a **continuation-passing style** function `findTwo` which looks through a shrub for two distinct values which satisfy a predicate. So that this function is fully polymorphic, we’ll pass in a function `eq : 'a * 'a -> bool` which will tell us whether two values are equal. You can assume that `eq` is total and represents an equivalence relation ¹.

**Note:** The correctness of your `findTwo` is dependent on the correctness of your `findOne`.

Here’s a formal spec:

```haskell
findTwo : ('a -> bool) -> ('a * 'a -> bool) -> 'a shrub -> ('a * 'a -> 'b) -> (unit -> 'b) -> 'b
```

**REQUIRES:** `p` is total, `eq` is total, `eq` represents an equivalence relation

**ENSURES:** `findTwo p eq T sc fc` =

```text
{ sc (v1, v2) where v1 and v2 are the leftmost two values in T such that
  eq (v1, v2) ⇒ false and p v1 ⇒ true and p v2 ⇒ true
  fc () if no such v1, v2 exist
}
```

**Constraint:**

- Your implementation of `findTwo` should **not** call `findTwo` recursively. `findTwo` may call `findOne`, but – since `findOne` takes continuations – any calls to `findOne` must be tail calls.

- You may **not** write any additional helper functions for this task, unless they are used as continuations. You can use `findOne` if you need. You also cannot instruct another function to call `findTwo` in a continuation!

For example,

```haskell
val T1 = Branch(Leaf 1, Leaf 2)
val T2 = Branch(Leaf 1, Leaf 1)
val p1 = fn x => x > 0
val p2 = fn x => x = 1
val eq = op=
val sc = SOME
val fc = fn () => NONE
val SOME (1, 2) = findTwo p1 eq T1 sc fc
val NONE = findTwo p2 eq T1 sc fc
val NONE = findTwo p1 eq T2 sc fc
```

¹For all `a, b, c`, `eq(a, a) ≡ true`, `eq(a, b) ≡ eq(b, a)`, `eq(a, b) ≡ true` and `eq(b, c) ≡ true` implies `eq(a, c) ≡ true` (i.e. the relation is reflexive, symmetric, and transitive).
Task 15. (15 points)

We can generalize the result of the previous task even further: what if we wanted to find \( n \) distinct values \( v \) of a shrub such that \( p \ v \Rightarrow \text{true} \)? Unlike the previous task, where we could have our success continuation take in a tuple of type \( t \times t \) for some type \( t \), there is no way to define a general \( n \)-ary tuple in SML. Instead, we will have our success continuation take in an \( 'a \text{ list} \).

In code/findn/findn.sml, write the following function in continuation-passing style:

\[
\text{findN} : ('a -> \text{bool}) -> ('a * 'a -> \text{bool}) -> 'a \text{ shrub} -> \text{int} -> ('a \text{ list} -> 'b) -> (\text{unit} -> 'b) -> 'b
\]

REQUIRES: \( n \geq 0 \), \( p \) is total, \( \text{eq} \) is total, \( \text{eq} \) represents an equivalence relation

ENSURES: \( \text{findN} \ p \ \text{eq} \ T \ n \ \text{sc} \ \text{fc} \approx \)

\[
\begin{cases} 
\text{sc} \ L & \text{where the } n \text{ elements of } L \text{ are the leftmost values in } T \text{ such that for all } x \in L, \\
p(x) \Rightarrow \text{true} \text{ and the elements of } L \text{ are } \text{eq}-\text{distinct}, \text{ that is, for all } x, y \in L \\
\text{such that } x \text{ and } y \text{ are not in the same position in } L, \ \text{eq}(x,y) \Rightarrow \text{false} \\
\text{fc} \ () & \text{if no such } L \text{ exists}
\end{cases}
\]

In your code file, fill in the function below so that it meets the spec:

\[
\text{fun} \ \text{findN} \ (p: 'a -> \text{bool}) \ (\text{eq: 'a} * 'a -> \text{bool}) \ (T: 'a \text{ shrub}) \ (n: \text{int}) \ (\text{sc: 'a \text{ list} -> 'b}) \ (\text{fc: unit -> 'b}): 'b =
\]

\[
\begin{cases} 
\text{case } n \ \text{of} \\
0 \Rightarrow (* \text{ Fill in base case } *) \\
| \_ \Rightarrow \\
\text{let} \\
\quad \text{fun} \ \text{success} \ x = (* \text{ Fill in success continuation } *) \\
\quad \text{in} \\
\quad \text{findOne} \ p \ T \ \text{success} \ \text{fc} \\
\text{end}
\end{cases}
\]

Note: The correctness of your \( \text{findN} \) is dependent on the correctness of your \( \text{findOne} \).

Constraint: You may add other declarations to the \text{let} \ block if they help you write the success continuation, but any additional functions you write here may NOT be recursive. You may not otherwise modify the code.

Note that although \( \text{findTwo} \) CANNOT be recursive, \( \text{findN} \) CAN be recursive.
6 I'm the Map!

Continuations are all well and good, but can we prove the correctness of continuation-passing functions in the same way that we prove other functions correct? It is your job to show that we can!

Consider the following code implementing the map function, in CPS and not in CPS:

```ml
fun map f L =
  case L of
  | [] => []
  | x::xs => f x :: map f xs

fun mapCPS f L k =
  case L of
  | [] => k []
  | x::xs => mapCPS f xs (fn xs' => k (f x :: xs'))
```

**Task 16.** (15 points)

Prove that, for all types \( t_1, t_2 \), for all values \( f : t_1 \to t_2 \) and \( L : t_1 \text{ list} \), if \( f \) is total, then

\[
\text{map } f \ L \sim \text{mapCPS } f \ L (\text{fn } x \Rightarrow x)
\]

You may use the following lemma:

If \( f \) is a total function value, then map \( f \) evaluates to a total function value.

**Hints:**

- You may need to prove something stronger than what is asked for.
- Pay attention to how you quantify the variables in your inductive hypothesis.
7 Can’t Prop Won’t Prop

In logic, we’re often concerned with *propositions*, which are like statements that can be either true or false. For example, “it is raining today” is a proposition and “I like turtles” is a proposition. Furthermore, we have several constructors which combine propositions: “it is raining today *and* I like turtles” is a proposition, as is “it is raining today *or* I like turtles”.

Formally, we express this as follows: a *proposition* is either:

1. A constant value, either true or false.
2. A propositional variable, represented by a string.
3. $A \land B$, read “$A$ and $B$”, where $A$ and $B$ are propositions.
4. $A \lor B$, read “$A$ or $B$”, where $A$ and $B$ are propositions.
5. $\neg A$, read “not $A$”, where $A$ is a proposition.

We provide the datatype `prop` to represent propositions in SML.

```sml
datatype prop = Const of bool
  | Var of string
  | And of prop * prop
  | Or of prop * prop
  | Not of prop
```

so, for example, my proposition “it is raining today *and* I like turtles” could be expressed in SML as:

```sml
val myProp : prop = And (Var "it is raining today", Var "I like turtles")
```

7.1 Evaluation

You may notice that we cannot find the truth value of a proposition if the proposition contains propositional variables without knowing their assigned truth values (in order to know whether it’s true that “it is raining today and I like turtles”, you need to know the truth values of the proposition “it is raining today” and the proposition “I like turtles”). However, if a proposition contains no variable, we can find the truth value of an entire proposition by following a set of rules parallel to those above:

1. The constant value true has truth value true, and the constant value false has truth value false.
2. $A \land B$ is true if $A$ is true and $B$ is true; otherwise, $A \land B$ is false.
3. $A \lor B$ is true if $A$ is true or $B$ is true; otherwise, $A \lor B$ is false.
4. $\neg A$ is true if $A$ is false; otherwise, $\neg A$ is false.

**Task 17.** (10 points)

In code/sat/sat.sml, write a continuity-passing style function
try_eval : prop -> (bool -> 'a) -> (string -> 'a) -> 'a
REQUIRES: true
ENSURES: try_eval p sc fc ≡

\{ sc b where b is the truth value of p if p contains no variables \\
f( c c where c is the leftmost variable in p \\

For example,

val (SOME false, NONE) =
  try_eval
    (And (Not (Const true), Const true))
    (fn b => (SOME b, NONE))
    (fn c => (NONE, SOME c))

However,

val (NONE, SOME "a") =
  try_eval
    (And (Not (Var "a"), Var "b"))
    (fn b => (SOME b, NONE))
    (fn c => (NONE, SOME c))

Make sure your try_eval works correctly before proceeding. You will not be able to complete the next task without a working implementation of try_eval.
7.2 Assignments

You can think of propositional variables as variables that can be assigned a truth value. Define assignment of a particular propositional variable as

\[
\text{type assignment} = \text{string} \times \text{bool}
\]

So for example, \(("\text{it is raining today}"\), false) would encode the assignment of the value false to the propositional variable "it is raining today". Here we provide you with a useful helper function that replaces a propositional variable with some truth assignment.

\[
\text{subst} : \text{assignment} \times \text{prop} \rightarrow \text{prop}
\]

REQUIRES: true

ENSURES: subst ((c, b), p) => p′ such that

1. p′ does not contain c
2. p′ is equivalent to p except all instances of c have been replaced with Const b

We also provide a function that does many of these substitutions at once, but you should not need this function to write your solution. We only define subst_all to help us write the spec for a later function.

\[
\text{subst_all} : \text{prop} \rightarrow \text{assignment list} \rightarrow \text{prop}
\]

REQUIRES: All strings in assignment list is unique (each propositional variable appears at most once in the list)

ENSURES: subst_all p assignments evaluates to a proposition with all instances of variables in assignments replaced with the Const true or Const false, according to their corresponding booleans in assignments.
7.3 (I Can’t Get No) Satisfaction

We say that a value \( p: \text{prop} \) is **satisfiable** if there exists some **assignments**: \( \text{assignment list} \) such that

\[
\text{try_eval} \ (\text{subst_all} \ p \ \text{assignments}) \ sc \ fc \implies sc \ true
\]

The proposition \( a \land \neg a \) is **unsatisfiable** because no truth value assignment to the propositional variable \( a \) makes the proposition true. However, the proposition \( b \lor (a \land \neg a) \) is satisfiable, because we can assign \( true \) to \( b \) and \( false \) to \( a \), which satisfies the formula.

Your task is to design an SML function that finds a satisfying assignment for a proposition using continuation-passing style. You will choose assignments to each propositional variable in a proposition, building up a truth assignment as you go. If there is no assignment of truth values to the remaining propositional variables such that the proposition is satisfied, you will call the failure continuation. Once you find a truth assignment that works, you will call the success continuation on it.

**Task 18.** (15 points)

Using \( \text{try_eval} \) and \( \text{subst} \), write a function in continuation-passing style

\[
sat : \text{prop} -> (\text{assignment list} -> 'a) -> (\text{unit} -> 'a) -> 'a
\]

**REQUIRES:** true

**ENSURES:** \( sat \ p \ sc \ fc \equiv \begin{cases} 
sc \ \text{asgns} & \text{where } \text{try_eval} \ (\text{subst_all} \ p \ \text{asgns}) \ sc' \ fc' \equiv sc' \ true \\
fc () & \text{if no such asgns exist}
\end{cases} \)

For example,

\[
\text{val NONE } = \text{sat} \ (\text{And} \ (\text{Not} \ (\text{Var} \ "a"), \ \text{Var} \ "a")) \ \text{SOME} \ (\text{fn} () \Rightarrow \text{NONE})
\]

However,

\[
\text{val SOME } [("a", \ \text{true})] = \text{sat} \ (\text{Or} \ (\text{And} \ (\text{Not} \ (\text{Var} \ "a"), \ \text{Var} \ "a"), \ \text{Var} \ "a")) \ \text{SOME} \ (\text{fn} () \Rightarrow \text{NONE})
\]

**Remember:** As per the guidelines at the beginning of this assignment, all calls to CPS functions must be tail calls.
8 The Curry-Devoured Correspondence

8.1 Introduction

We define the notion of two lists \( L \) and \( R \) being a partition of another list \( A \) as:

- \((\[], \[])\) is a partition of \( \[] \).
- If \((L, R)\) is a partition of \( A \) then \((x :: L, R)\) is a partition of \( x :: A \).
- If \((L, R)\) is a partition of \( A \) then \((L, x :: R)\) is a partition of \( x :: A \).

So for example, \(([2], [1, 3])\) is a partition of \([1,2,3]\) but \(([3, 1], [2])\) and \(([1, 2], [])\) are not partitions of \([1, 2, 3]\).

Our goal in this section is to ultimately implement a CPS function \texttt{findPartition} which, given a list \( A \) and two predicates \( P_L \) and \( P_R \), finds a partition \((L, R)\) of \( A \) such that \( L \) satisfies \( P_L \) and \( R \) satisfies \( P_R \). In other words, \texttt{findPartition} must find \( L \) and \( R \) such that \( A \) can be split into \( L \) and \( R \), and both \( P_L(L) \) and \( P_R(R) \) hold. We will later use \texttt{findPartition} to implement another function \texttt{won'tStarve}, which we’ll present in the next section.

The precise type and specification of \texttt{findPartition} can be daunting at first, so rather than presenting it now it we’ll start by work through a few simpler versions of \texttt{findPartition}, gradually working our way up to \texttt{findPartition} itself.

Here are some hints which may be useful to refer back to as you work through the tasks in this section. If you’re having trouble, running through this list might help:

- Each of our reference implementations are fewer than 10 lines long (not including type annotations).
- There may be many combinations of \( L \) and \( R \) for which both \( P_L \) and \( P_R \) hold. It doesn’t matter which combination you choose.
- Let the types guide you! If you don’t know how to write part of the function, think about how you use what you have to get a value of the return type.
- Think about what functions you change in the recursive call.
- Think about where a value \( x \) in \( A \) could “end up”, and what that means for how you recur.
8.2 Version 1: Not really CPS

The properties $P_L$ and $P_R$ can be represented as predicate functions which take in a list and return true if the property holds for the list and return false otherwise. Their types are as follows:

\[
\begin{align*}
p_L & : 'e \text{ list } \to \text{ bool} \\
p_R & : 'e \text{ list } \to \text{ bool}
\end{align*}
\]

Here, $'e$ is the type of an element of this list\(^2\).

Sure, someone can write an implementation of `findPartition` and claim that it only returns true if it really does find a satisfactory partition, but that’s too easy to fudge. If I get a `true` from applying `findPartition` to a list and some predicates, how do I know it really found a satisfactory partition and didn’t, for example, bogusly return true? We can cleverly use SML’s type system to, at least to some degree, keep the implementation behaved \(^3\). Rather than having predicates which return booleans, which are easy to fake, we’ll have predicates which return options. Informally, returning `SOME` of something corresponds to returning `true`, and returning `NONE` corresponds to returning `false`.

For the purpose of `findPartitionV1` we’ll say that a predicate function $p$ accepts $x$ with $y$ if and only if $p \ x \cong \text{SOME } y$.

Now, when a predicate function accepts a list, it returns a value $y$ which acts as “evidence”, “witness”, or “verification” that it accepted something. We can require that when `findPartitionV1` claims to have found a partition which satisfies the predicates, it “show us the evidence” by returning the values gotten from satisfying the predicates.

To allow the predicates full freedom, we’ll keep the type of $y$ polymorphic. With this, we’ll say “evidence” has type $'l$ for $P_L$ and type $'r$ for $P_R$.

**Task 19.** (1 point)

In `code/wontStarve/findPartition.sml`, implement `findPartitionV1`. Do not use CPS for `findPartitionV1`.

```sml
findPartitionV1 : 'e \text{ list } \to (\text{ ('e \text{ list } \to 'l \text{ option}) } \to \text{ ('e \text{ list } \to 'r \text{ option}) } \to \text{ ('l } \times 'r \text{) option}

\text{REQUIRES:} \ pL \text{ and } pR \text{ are total.}

\text{ENSURES:} \ \text{findPartitionV1 } A \ pL \ pR \implies

\begin{align*}
\text{SOME } (\text{LL}, \text{RR}) & \text{ if there exist } \text{L and R such that } \\
& \text{(L, R) is a partition of } A, \text{ and} \\
& pL \text{ accepts L with LL, and} \\
& pR \text{ accepts R with RR.} \\
\text{NONE} & \text{ otherwise.}
\end{align*}
```

\(^2\)We usually use $'a$ and $'b$ in order for type variables, but you can actually make your type variables any arbitrary identifier, provided it starts with a $'$. \(^3\)A better chosen type can sometimes exclude incorrect implementations. Yet another reason to use a statically typed language.
8.3 Version 2: A bit of CPS

It’s time to turn this into continuation-passing style! Rather than returning an option, we want `findPartitionV2` to take continuation functions and express its result by applying one of them.

Recall that for a function to be properly written in CPS, all calls it makes to CPS functions must be tail calls.

**Task 20.** (3 points)

In `code/wontStarve/findPartition.sml`, implement `findPartitionV2` in continuation-passing style. It has the type:

`'e list -> ('e list -> 'l option) -> ('e list -> 'r option) -> ('l * 'r -> 'a) -> (unit -> 'a) -> 'a`

**findPartitionV2 :** `'e list -> ('e list -> 'l option) -> ('e list -> 'r option) -> ('l * 'r -> 'a) -> (unit -> 'a) -> 'a`

**REQUIRES:** `pL` and `pR` are total.

**ENSURES:** `findPartitionV2 A pL pR sc fc`:

\[
\begin{cases}
sc (LL, RR) & \text{if there exist L and R such that} \\
& (L, R) \text{ is a partition of } A, \text{ and} \\
& pL \text{ accepts } L \text{ with } LL, \text{ and} \\
& pR \text{ accepts } R \text{ with } RR. \\
fc () & \text{otherwise.}
\end{cases}
\]
8.4 Final Version: So much CPS

The final change we will make, which brings us to the final version of \texttt{findPartition}, is that we now generalize predicate functions to also be CPS functions. This may seem extreme but as you’ll find when implementing \texttt{won’tStarve} in the next section, this generalization affords us the freedom to use \texttt{findPartition} to implement other CPS functions.

A predicate function now ranges over the type:

\[
'e\ list \to ('b \to 'a) \to (unit \to 'a) \to 'a
\]

So, for example, for any particular types \(t_1\) and \(t_2\) we might have

\begin{verbatim}
fun pL (L : t1 list) (sc : t2 -> 'a) (fc : unit -> 'a) : 'a = ...
\end{verbatim}

Here, \(pL\) calls the \textit{success continuation} \(sc : t2 \to 'a\) if the list \(L\) satisfies the property \(pL\) cares about, and \(pL\) calls the \textit{failure continuation} \(fc : unit \to 'a\) otherwise.

We must accordingly update our definition of what it means for a predicate to accept something. For the purpose of the final version of \texttt{findPartition}, we’ll say a predicate \(p\) accepts \(x\) with \(y\) if for any \(sc\) and \(fc\), \(p\ x \ sc \ fc \equiv sc\ y\).

\textbf{Task 21.} (15 points)

In \texttt{code/wontStarve/findPartition.sml}, implement \texttt{findPartition} in continuation-passing style. It has the type:

\[
'e\ list \to
('e\ list \to ('l \to 'a) \to (unit \to 'a) \to 'a) \to
('e\ list \to ('r \to 'a) \to (unit \to 'a) \to 'a) \to
('l \ast \ 'r \to 'a) \to
(unit \to 'a) \to
'a
\]

\begin{verbatim}
findPartition : 'e\ list \to ('e\ list \to ('l \to 'a) \to (unit \to 'a) \\
\to 'a) \to ('e\ list \to ('r \to 'a) \to (unit \to 'a) \to 'a) \to ('l \ast \\
'r \to 'a) \to (unit \to 'a) \to 'a
\end{verbatim}

\textbf{REQUIRES:} \(pL\) and \(pR\) are total CPS functions.

\textbf{ENSURES:} \(findPartition\ A\ pL\ pR\ sc\ fc \Rightarrow
\begin{cases}
sc\ (LL,\ RR) & \text{if there exist } L\ and\ R\ such\ that \\
(L,\ R)\ is\ a\ partition\ of\ A,\ and \\
pL\ accepts\ L\ with\ LL,\ and \\
pR\ accepts\ R\ with\ RR.
\end{cases}
\]
Here is an example of `subsetSum` using `findPartition`:

```ml
val sum : int list -> int = foldr op+ 0

fun subsetSum (n : int) (xs : int list) : int list option =
  let
    fun pL L scL fcL = if sum L = n then scL L else fcL ()
    fun pR R scR fcR = scR ()
    fun sc (LJ, ()) = SOME LJ
    fun fc () = NONE
  in
    findPartition xs pL pR sc fc
  end

val SOME [] = subsetSum 0 []
val NONE = subsetSum 1 []
val SOME [1] = subsetSum 1 [1]
val SOME [1,1] = subsetSum 2 [1,1]
val SOME [1,2] = subsetSum 3 [1,2,1]
val SOME [2,1] = subsetSum 3 [5,2,1]
```
9 Don’t burn the curry!

Former 15-317 TAs Avery, Cam and Siva (who dream of being professional chefs) are trying to prepare a large batch of food for Harrison’s (the former head TA of 15-150) potluck party. Since the only meal they can prepare is curry, they want to make several different types of curry (and maintain a facade of variety), each of which takes a different duration to cook in a stove. The combined exceptional laziness of the TAs, however, has induced them to procrastinate the cooking until right before the potluck! Consequently, they must figure out if they can still prepare all the types of curry within the little time left or resort to buying pizza off of Avery’s dinex. To potentially cook his curries in time, they realize that they must utilize all the pots they can from Avery’s vast pot collection. Unfortunately, he only has a limited supply of pots.

Your task is to devise a function that when given the number of pots Avery, Cam, and Siva can use, a list of cooking times for the curries they want to make, and a time limit under which the TAs must cook all the curries, decides whether they are able to finish cooking within the time limit or admit defeat and order dominos $6 medium 2-topping pizzas instead. Furthermore, if they are able to finish making all their curries, your function should also generate a way to distribute the cook times among the many pots that allows the curries to all be prepared under the time limit.

We will use an int list to represent the cook time of each curry. To represent the distribution of cook times, you will use an int list list whose inner lists represent the cook times assigned to each pot.

Each pot Avery uses can only cook one curry at a time, and for each pot, he can start cooking the next curry immediately after the current curry finishes. In addition, Avery, Cam, and Siva are able to simultaneously cook curry in all their pots at the same time since they are such synchronized and masterful curry chefs.

We will also guarantee that all cooking times for all curries will be positive, since it would be a tad silly to have negative cooking times. Furthermore, the TAs will always have a nonnegative number of pots to cook with (can you imagine cooking with negative pots?), and your function will never be tasked to decide if the curries can be cooked under a negative time bound (what does that even mean?).

Time is running out!!! The TAs have to quickly decide how to distribute the curries among the pots. Avery picks up one pot and weighs his options...
Task 22. (17 points)

In code/wontStarve/wontStarve.sml, use continuation-passing style and findPartition to implement:

```sml
won'tStarve : int -> int -> int list -> (int list list -> 'a) -> (unit -> 'a) -> 'a

REQUIRES:  pots ≥ 0 and time ≥ 0 and for all x in curries, x > 0
ENSURES:  won'tStarve pots time curries sc fc ⇒

  \begin{cases} 
    sc D & \text{if there exists D such that} \\
    & \text{concat D is a permutation of curries and} \\
    & \text{length D} \cong \text{pots and} \\
    & \text{for all d in D, sum d} \leq \text{time}. \\
    fc () & \text{otherwise.} 
  \end{cases}

where \text{sum} = \text{foldr op+ 0}.
```

Constraint:

- You \textit{must} use findPartition in your implementation of won'tStarve.
- You \textit{may not} use findPartitionV1 or findPartitionV2 in your implementation of won'tStarve.

For example:

```sml
won'tStarve 3 50 [10, 20, 30, 40] sc fc \cong sc [[[10, 40], [20], [30]]]
```

This means that there exists an assignment of curries to pots such that the curries can all be cooked under 50 minutes. It also indicates that an assignment that does so is cooking the 10 minute curry and then the 40 minute curry in one pot (for a total of 50 minutes), the 20 minute curry in another pot, and the 30 minute curry in a 3rd pot.

```sml
won'tStarve 2 40 [20, 20, 30, 40] sc fc = fc ()
```

For the case above, there is no assignment of curries to pots that will have a cook time of less than or equal to 40 minutes per pot.

Some hints:

- Think \textit{very carefully} about what your pL and pR should be.
- What better justification that a list satisfies a property, than using that list itself to check?
- Think about recursion, where would you want your recursive call to be made?
- Our implementation is around 10 lines long with generous newlines and indentation (not including type annotations).

To test your implementation, run the following command:

`smlnj findPartition.sml wontStarve.sml`
This will first load your solution to \texttt{findPartition} before loading \texttt{wontStarve}. When grading, we will use the reference solution for \texttt{findPartition}.
At least we teach a useful language

An innocent 122 TA once exclaimed: “Purely functional programming doesn’t have mutation? Mutation is like, my favorite feature of imperative programming languages. My second favorite feature is loops, of course, which purely functional programming languages also don’t have. I bet without mutation and loops, purely functional programming languages aren’t even as powerful as imperative programming languages. I bet you couldn’t simulate imperative programs in a purely functional language like SML.”

The 150 TA took a deep breath and replied: “

10.1 C_not’s not C_0

In this section we will implement an interpreter for a simple imperative language named C_not with variables, if commands, and loops.

A C_not program is a sequence of commands. In SML we will represent a C_not program as a value of type command list, where the command type is as presented in Figure 10.1. Figure B also describes aexp which corresponds to arithmetic expressions in C_not, and bexp which corresponds to boolean expressions in C_not. You do not have to implement these in your interpreter. Below, we will explain the syntax of C_not and how C_not programs are executed.

Implementation

```plaintext
datatype command =
  Assign of string * aexp
| If of bexp * command list * command list
| Loop of command list
| Break
| Continue
| Return of aexp
```

Syntax

```plaintext
x := aexp;
if (bexp) {cmds} else {cmds}
loop {cmds}
brack;
continue;
return aexp;
```

Figure 1: The implementation and corresponding syntax of C_not commands
10.2 Semantics of C_not

At every point in time during the execution of a C_not program, there is an environment which maps variables to the values they contain. As C_not is an imperative language, the specific values which the environment maps variables to can change over time as the program mutates those variables. Variables in C_not can only contain integers.

Next we’ll define the semantics for the various commands in C_not:

<table>
<thead>
<tr>
<th>Command</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := e; )</td>
<td>Updates the environment to map variable ( x ) to the result of evaluating ( e ) under the current environment.</td>
</tr>
<tr>
<td>if ( (e) { \text{cmds1} } ) else ( { \text{cmds2} } )</td>
<td>Executes ( \text{cmds1} ) if ( e ) evaluates to true; executes ( \text{cmds2} ) if ( e ) evaluates to false.</td>
</tr>
<tr>
<td>loop ( { \text{cmds} } )</td>
<td>Repeatedly executes the loop body ( \text{cmds} ) until ( \text{cmds} ) uses a break command to exit the loop. If a break or continue command is encountered while executing ( \text{cmds} ), handle the behaviors as described below.</td>
</tr>
<tr>
<td>( \text{break;} )</td>
<td>Stops executing the current loop, and does not continue executing iterations of the loop. Execution continues after the loop command.</td>
</tr>
<tr>
<td>( \text{continue;} )</td>
<td>Stops the current iteration of the loop, and continues executing iterations of the loop.</td>
</tr>
<tr>
<td>return ( e; )</td>
<td>Stops execution of the entire program and returns the result of evaluating ( e ).</td>
</tr>
</tbody>
</table>

10.3 Implementing the interpreter

When implementing your interpreter, you can assume all the programs you execute are well-formed. Doing this will allow you to use certain helper values without proof (for the sake of this problem). The notion of well-formedness is further defined in the Appendix.

As we said earlier, executing a C_not program involves an environment which maps variables to the values they currently contain. To deal with this environment in our code, we provide the type environment. This type is very similar to a dictionary data structure. We’ve provided some helper values

\[
\text{set: \ string \rightarrow \ int \rightarrow \ environment \rightarrow \ environment} \\
\text{get: \ environment \rightarrow \ string \rightarrow \ int \ option}
\]

that allows you to interact with the type environment.

We’ve also provided helper values

\[
\text{evalAexp: \ environment \rightarrow \ aexp \rightarrow \ int} \\
\text{evalBexp: \ environment \rightarrow \ bexp \rightarrow \ bool}
\]

which evaluate C_not arithmetic expressions and boolean expressions.

If needed, refer to the Appendix for robust definitions / example usages of \( \text{set}, \text{get}, \text{evalAexp}, \) and \( \text{evalBexp} \).

When using \( \text{evalAexp} \) and \( \text{evalBexp} \), you can assume that the provided environment contains mappings for all variables that appear in the provided expression.
We will use exceptions to implement the control-flow for C\textsubscript{not} programs. In particular, we define the following three exceptions and use them as follows:

- **exception BreakExn of environment**
  Raise this upon encountering a \texttt{break} command. It carries the environment as it was upon encountering the \texttt{break} command.

- **exception ContinueExn of environment**
  Raise this upon encountering a \texttt{continue} command. It carries the environment as it was upon encountering the \texttt{continue} command.

- **exception ReturnExn of int**
  Raise this upon encountering a \texttt{return} command. As the program will not continue executing after it returns, this exception does not need to carry the environment. It carries the value returned.

It will be your responsibility to implement the interpreter for C\textsubscript{not} commands. In particular, you’ll implement the function \texttt{runCommand : environment -> command -> environment} which, given a command and the current environment, updates the environment according to the command or raises an exception as described next.

As commands can contain lists of commands (for example in the case of loop commands), \texttt{runCommand} will be mutually recursive\textsuperscript{9} with \texttt{runCommands : environment -> command list -> environment} which we’ve implemented for you as

\[
\texttt{runCommands} \equiv \texttt{foldl (fn (c, e) => runCommand e c)}
\]

You must use \texttt{runCommands} to process lists of commands. You may not define a recursive helper which recurs through lists of commands. You may not alter the definition of \texttt{runCommands}.

Additionally, we’ve provided a wrapper function \texttt{runProgram : command list -> int}, which runs an entire program, starting from an empty environment. It expects as input a well-formed C\textsubscript{not} program.

**Task 23.** (14 points)

Implement the following function:

\[
\texttt{runCommand : environment -> command -> environment}
\]

**REQUIRES:** \texttt{runCommand} will only be invoked during the interpretation of a well-formed C\textsubscript{not} program.

**ENSURES:** \texttt{runCommand env cmd} either

1. Evaluates to \texttt{env'}, where \texttt{env'} is the result of updating \texttt{env} according to \texttt{cmd}
2. Raises one of the exceptions described above.

You may assume as a precondition that \texttt{runCommand} will only be invoked during the interpretation of a well-formed C\textsubscript{not} program.

\textsuperscript{9}and allows us to define functions to be mutually recursive.
10.4 Testing your interpreter

To help with understanding the semantics of C\text{not} and to help with debugging your interpreter, we've included example C\text{not} programs in the code/interpreter/tests directory. For each of these examples, the first line is a comment containing the value the program is expected to return.

We've also included three helper functions in the file code/interpreter/utils.sml:

\begin{verbatim}
parseString : string -> command list
parseFile : string -> command list
testAll : unit -> unit
\end{verbatim}

To use these helpers, simply open the utils.sml file into SML/NJ after opening your interpreter.sml file. For example, to first load your interpreter.sml file into SML/NJ, inside code/interpreter/, run

\begin{verbatim}
smlnj interpreter.sml.
\end{verbatim}

If it does not compile properly at this point, fix your code before proceeding to use code/interpreter/utils.sml. If code/interpreter/interpreter.sml compiles successfully, you should now be in the SML/NJ REPL. In the REPL, run

\begin{verbatim}
use "utils.sml";
\end{verbatim}

After this, you should be able to use the three helper functions we provided.

In parseString and parseFile we've implemented a parser which parses C\text{not} programs into values of type command list. The parseString function expects to be given actual C\text{not} programs; for example

\begin{verbatim}
parseString "x := 1; return x;"
\end{verbatim}

On the other hand, the parseFile function expects to be given the path to a file containing a C\text{not} program; for example

\begin{verbatim}
parseFile "tests/add.cnot"
\end{verbatim}

After parsing a C\text{not} program (or constructing one directly using the datatype constructors) you can then run it through the runProgram function mentioned earlier to see how your interpreter behaves on this particular program; for example

\begin{verbatim}
runProgram (parseFile "tests/add.cnot")
\end{verbatim}

should evaluate to 12.

The testAll function is a test harness which runs every file in tests through your interpreter and compares the result to the expected result (where the expected result is commented at the top of each test file). You can use testAll to check that your interpreter has the intended behavior on all the programs in the tests directory. Furthermore, you can add new test programs to the tests directory. As long as the first line of the file is a comment with the expected return value, testAll will test your interpreter against it along with all the other files in tests. To use testAll, simply evaluate testAll ()

In case the interpreter infinitely loops, testAll limits the execution of each test case to one second by default. \footnote{unless you're on Windows} In case you'd like to make test cases which legitimately take a long time to run,
we’ve also provided a fourth function

\[
\text{setTimeoutSeconds} : \text{int} \to \text{unit}
\]

, which you can use to change the time limit. You probably won’t ever need to do this.

".

The 122 TA, astonished and amazed, replied, “Oh, I guess I was wrong; you can totally simulate imperative programs in purely functional languages like SML. I concede that purely functional programming is just as powerful as imperative programming.” The 122 TA then applied to be a 150 TA.
A Appendix: Is this CPS?

Below, we implement some examples in both a “pseudo-CPS” style and in CPS. Note that all of the “bad” examples would earn 0 points on this homework (because they are not correct CPS). Please understand the distinction and ask us if you have any questions.

A.1 fact

We want to implement the factorial function in CPS. The following is not CPS:

```haskell
fun fact_pseudoCPS 0 k = k 1
| fact_pseudoCPS n k =
  let
    val res = fact_pseudoCPS (n-1) (fn x => x)
  in
    k (n*res)
  end
```

This is not CPS because it makes a recursive call to itself, and then does arithmetic with the result. Therefore, the recursive call is not a tail call, so this is not CPS. Instead, we should do something like the following.

```haskell
fun factCPS 0 k = k 1
| factCPS n k = factCPS (n-1) (fn res => k(n*res))
```

A.2 size

We want a function which calculates the size (number of nodes) of a tree. The following is not CPS:

```haskell
fun size_pseudoCPS Empty k = k 0
| size_pseudoCPS (Node(L,x,R)) k = 1 + (size_pseudoCPS L k) + (size_pseudoCPS R k)
```

This would not typecheck if the return type of k is not `int`, and also it is not CPS because we perform arithmetic on the result of the recursive call. Instead, we should do something like:

```haskell
fun sizeCPS Empty k = k 0
| sizeCPS (Node(L,x,R)) k =
  sizeCPS L (fn sizeL => sizeCPS R (fn sizeR => k(1+sizeL+sizeR))

Note that we could also write `sizeCPS` as follows:
fun sizeCPS Empty k = k 0
| sizeCPS (Node(L,x,R)) k =
  let
  fun cont sizeL = sizeCPS R (fn sizeR => k (1 + sizeL + sizeR))
  in
  sizeCPS L cont
  end

This is still CPS because the call to sizeCPS is a tail-call, and there are no calls to cont which are not tail calls.

A.3 find

We want a function which checks if an int list contains an element which is 0. The following is not CPS:

fun find [] sc fc =
  let
    fun find' [] = false
    | find' (0::_) = true
    | find' (_::xs) = find' xs
  in
    if find' L then sc () else fc ()
  end

This is not CPS because it delegates all the work to a non-CPS helper, trivializing the CPS function. Instead, we would write:

fun find [] sc fc = fc ()
| find (0::_) sc fc = sc ()
| find (_::xs) sc fc = find xs sc fc

B Appendix: C_not programs
**Implementation**

```ocaml
datatype aexp =
  AConst of int
| Variable of string
| Add of aexp * aexp
| Sub of aexp * aexp
| Mult of aexp * aexp

datatype bexp =
  BConst of bool true
| false
| Equals of aexp * aexp
| NotEquals of aexp * aexp
| LessThan of aexp * aexp
| LessThanEqual of aexp * aexp
| GreaterThan of aexp * aexp
| GreaterThanEqual of aexp * aexp
| Not of bexp
| Or of bexp * bexp
| And of bexp * bexp

datatype command =
  Assign of string * aexp
| If of bexp * command list * command list if (bexp) {cmds} else {cmds}
| Loop of command list loop {cmds}
| Continue
| Break
| Return of aexp
```

**Syntax**

```
datatype aexp =
  n
| x
| aexp + aexp
| aexp - aexp
| aexp * aexp

datatype bexp =
  true | false
| aexp == aexp
| aexp != aexp
| aexp < aexp
| aexp <= aexp
| aexp > aexp
| aexp >= aexp
| !bexp
| bexp || bexp
| bexp && bexp

datatype command =
  x := aexp;
| if (bexp) {cmds} else {cmds}
| loop {cmds}
| break;
| continue;
| return aexp;
```

Figure 2: The implementation and corresponding syntax of $C_{\text{not}}$
B.1 Well-formedness of $C_{\text{not}}$ programs

Every $C_{\text{not}}$ program is either well-formed or ill-formed. We define this notion as follows:

- A $C_{\text{not}}$ program is ill-formed if it uses a variable without first assigning to it.
- A $C_{\text{not}}$ program is ill-formed if a `break` command is encountered not within a loop.
- A $C_{\text{not}}$ program is ill-formed if a `continue` command is encountered not within a loop.
- A $C_{\text{not}}$ program is ill-formed if the end is reached without encountering a `return`.
- All other $C_{\text{not}}$ programs are well-formed.

**Important:** Your implementation does not need to deal with ill-formed $C_{\text{not}}$ programs. You may assume all the programs you execute are well-formed.

B.2 $C_{\text{not}}$ environments

As we said earlier, executing a $C_{\text{not}}$ program involves an environment mapping variables to the values they currently contain. To help keep track of this environment we’ve provided a type `environment`, along with the helper values:

```sml
emptyEnvironment : environment
set : string -> int -> environment -> environment
get : environment -> string -> int option
```

Given arbitrary values $x : \text{string}$, $v : \text{int}$, and $e : \text{environment}$, `set $x \ v \ e$` evaluates to a new environment (let’s call it $e'$) which now contains the mapping from $x$ to $v$. Then `get $e'$ $x$` would evaluate to `SOME v`. Note that the original environment $e$ remains unchanged by the call to `set` (SML is still purely functional, after all), so $e$ still does not contain the mapping from $x$ to $v$ (unless it already happened to contain that mapping). If $e$ doesn’t have any mapping for $x$, then `get $e \ x$` evaluates to `NONE`.

B.3 $C_{\text{not}}$ expressions

Recall the helper values

```sml
evalAexp : environment -> aexp -> int
evalBexp : environment -> bexp -> bool
```

which evaluate $C_{\text{not}}$ arithmetic expressions and boolean expressions. They can be used in the following way. If we let

```sml
val e = set "x" 1 (set "y" 2 empty_environment)
```

then

```sml
evalAexp e (Add (Variable "x", Variable "y")) \rightarrow 3
```

and

```sml
evalBexp e (LessThan (Variable "x", Variable "y")) \rightarrow \text{true}
```

evalAexp and evalBexp require that the provided environment contain mappings for all variables that appear in the provided expression. But since we are ignoring ill-formed $C_{\text{not}}$ programs,
you shouldn’t need to worry about failing this precondition. This is because we can assume that a provided environment contains mappings for all the variables in a provided expression when a $C_{\text{not}}$ program is well-formed. Note that this is also assuming you’ve been maintaining the environment correctly.

This assignment has a total of 141 points.