15-150 Assigment 7

Jonathan Li
jlli
Section S
June 10, 2016

Task 2.1

`if 3 < 2 then raise Fail "foo" else raise Fail "bar"!`

This expression raises the exception `Fail`, which carries the value `"foo" : string`.

Task 2.2

`(fn x => if x > 42 then 42 else raise Fail "Don't Panic") 16 handle _ => "42"`

This expression is ill-typed, as the type of the return value of one case in the anonymous function `(fn x => if x > 42 then 42 else raise Fail "Don't Panic")` is `int`, while the return type of the exception handler is `string`.

Task 2.3

```
let
    exception Less
    fun bar(x,y) = case Int.compare(y, x) of LESS => raise Less
                                            | _ => y
    fun foo f [] = []
      | foo f [x] = [x]
      | foo f (x::y::L) =
          f(x,y) :: (foo f (x::L)) handle Less => x::(foo f (y::L))
in
    foo bar [3,2,1,42]
end
```

This expression evaluates to the list `[3, 2, 42, 1]`.

Task 2.4

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

This expression raises the exception `bike`, as defined within the scope of the `let-in-end` expression.

Task 2.5

```
(case 15150 > 15251 of
    true => 15150
  | false => raise Fail "epicfail") handle EpicFail => 15150
```

This expression evaluates to the `int 15150`.

Task 2.6

```
datatype tree = Empty | Node of tree * int * tree
exception EmptyTree

fun mysteryMachine T =
case T of
    Empty => raise EmptyTree
     | (Node (L, n, R)) =>
        ((mysteryMachine L) + n + (mysterMachineR)) handle EmptyTree => n
```

When given a `T : int tree`, the function `mysteryMachine` will raise the exception `EmptyTree` if `T` $\cong$ `Empty`, and otherwise returns the sum of the integers at the nodes of `T`, up to the point where `T` is symmetric. At the branch where `T` becomes asymmetrical, i.e. where one branch is `Empty` and the other is not, `mysteryMachine` does not further compute the non-empty branch and just returns the value at the last symmetric node.

Task 3.1

Theorem 3.1: `findOne p T s k` will always evaluate to `s v`, where `v` is the leftmost element that satisfies `p` if such an element exists, and `k ()` otherwise.
Proof: By structural induction on `T`
<u>**Base Case:**</u> `T` $\cong$ `Leaf v` for some `v : 'a`
To show: `findOne p (Leaf v) s k` $\cong$ `s v`, where `v` is the leftmost value for which `p v` $\cong$ `true`, and $\cong$ `k ()` if such a value doesn't exist.
Proof:

`findOne p (Leaf v) s k`	$\cong$	`case Leaf v of Leaf x => ...`	[step]
	$\cong$	`if p(v) then s v else k ()`	[step]

Here, if `p v` $\cong$ `true`, then the whole expression `findOne p (Leaf v) s k` $\cong$ `s v` in finite steps. In this case, since `T` $\cong$ `Leaf v`, `v` is the leftmost (only) value in `T` for which `p v` $\cong$ `true`, so `findOne` is correct if `p v` $\cong$ `true` in this case.

If `p v` $\cong$ `false`, the expression steps to `k ()`. Again, since `T` $\cong$ `Leaf v`, in this case there are no values `v` in `T` for which `p v` $\cong$ `true`, so `findOne p (Leaf v) s k` is correct in this case as well.

<u>**Inductive Step:**</u> `T` $\cong$ `Branch(L,R)` for some `L : 'a shrub` and some `R : 'a shrub`
<u>Inductive Hypothesis 1:</u> `findOne` is correct for the `'a shrub` `L`. [IH 1]
<u>Inductive Hypthesis 2:</u> `findOne` is correct for the `'a shrub` `R`. [IH 2]
To Show: `findOne p Branch(L,R) s k` $\cong$ `s v`, where `v` is the leftmost value for which `p v` $\cong$ `true`, and $\cong$ `k ()` if such a value doesn't exist.
Proof:

`findOne p (Branch (L,R)) s k`	$\cong$	`case Branch(L,R) of Leaf x => ...`	[step]
	$\cong$	`findOne p L s (fn () => findOne p R s k)`	[step]

Here, let us consider two cases.
<u>Case 1:</u> `findOne p (Branch(L,R)) s k` $\cong$ `s v` for some value `v`
In this case, `v` is either an element in `L` or it is an element in `R`.

Case: `v` is an element of `L`

`findOne p L s (fn () => findOne p R s k)`	$\cong$	`s v`	[step]
`v` is the leftmost element in `L` for which `p v` $\cong$ `true`.			[IH 1]
`L` is to the left of `R` in `Branch(L,R)`			
$\therefore$ `v` is the leftmost element in `Branch(L,R)` for which `p v` $\cong$ `true`.			

Case: `v` is an element of `R`

`findOne p L s (fn () => findOne p R s k)`	$\cong$	`findOne p R s k`	[step, IH 1]

Task 3.1 (cont.)

This is justified if no element exists in `L` for which `p` applied to that element evaluates to `true`.

`findOne p R s k)`	$\cong$	`s v`	[step]
`v` is the leftmost element in `R` for which `p v` $\cong$ `true`.			[IH 2]
No elements in `L` for which `p` applied to that element evaluates to `true`.			
$\therefore$ `v` is the leftmost element in `Branch(L,R)` for which `p v` $\cong$ `true`.			

Thus, if the whole expression `findOne p (Branch(L,R)) s k` $\cong$ `s v` for some value `v`, then that value must be the leftmost value for which `p v` $\cong$ `true`.

<u>Case 2:</u> `findOne p (Branch(L,R)) s k` $\cong$ `k ()`

`findOne p L s (fn () => findOne p R s k)`	$\cong$	`(fn () => findOne p R s k) ()`	[IH 1]
	$\cong$	`findOne p R s k`	[step]

This is justified, since there exist no values `v` in `L` for which `p v` $\cong$ `true`. By IH 1, if no such values exist, then `findOne p L s k`, for any continuation `k`, will evaluate to `k ()`

`findOne p r s k`	$\cong$	`k ()`	[IH 2]

This is justified, since there exist no values `v` in `R` for which `p v` $\cong$ `true`. By IH 2, if no such values exist, then `findOne p L s k`, for any continuation `k`, will evaluate to `k ()`

Both cases satisfy the "To Show" statement.
$\therefore$ Since the Base Case and the Inductive Step hold, Theorem 3.1 must be true.

Task 3.2

<u>Theorem 3.2:</u> ∀ total functions `p : 'a -> bool` and ∀ `S : 'a shrub`,

`findOne p S (fn x => x) (fn () => raise NotFound)` ≅ `search p S`

Proof: By structural induction on `S`
<u>Base Case:</u> `S` ≅ `Leaf v` for some value `v : 'a`
To Show: `findOne p (Leaf v) (fn x => x) (fn () => raise NotFound)` ≅ `search p (Leaf v)`
Proof:

`findOne p (Leaf v)` `(fn x => x) (fn () => raise NotFound)`	≅ `case (Leaf v) of ...`	[step]
	≅ `if p(v) then ...`	[step]

Since `p` is a total function, ∀ values `v : 'a`, there are two cases: `p v` ≅ `true` or `p v` ≅ `false`.

<u>Case 1:</u> `p v` ≅ `true`

`findOne p (Leaf v)` `(fn x => x) (fn () => raise NotFound)`	≅ `if p(v) then...`	[step]
	≅ `if true then...`	[p is total, Referential Transparency]
	≅ `(fn x => x) v`	[step]
	≅ `v`	[step]
	≅ `case true of` `true => v` `|false => raise NotFound`	[step, symmetry]
	≅ `case (Leaf v) of ...`	[step, symmetry]
	≅ `search p (Leaf v)`	[step, symmetry]

Thus, for the case `S` ≅ `Leaf v` for some value `v : 'a` and `p v` ≅ `true`, Theorem 3.2 holds.

Task 3.2 (cont.)

Case 2: `p v` ≅ `false`

```
findOne p (Leaf v)
(fn x => x) (fn () => raise NotFound)  ≅ if p(v) then...                     [step]
                                       ≅ if false then...                    [p is total,
                                                                              Referential
                                                                              Transparency]
                                       ≅ (fn () => raise NotFound) ()        [step]
                                       ≅ raise NotFound                      [step]
                                       ≅ (case false of
                                           true => v
                                          | false => raise NotFound)         [step, symmetry]
                                       ≅case (Leaf v) of
                                           Leaf x => ...
                                          |Branch(L,R) => ...                [step, symmetry]
                                       ≅ search p (Leaf v)                   [step, symmetry]
```

Thus, for the case `S` ≅ `Leaf v` for some value `v : 'a` and `p v` ≅ `false`, Theorem 3.2 holds as well.
Inductive Step: `S` ≅ `Branch(L,R)` for some values `L : 'a shrub` and `R : 'a shrub`
Inductive Hypothesis 1:
`findOne p L (fn x => x) (fn () => raise NotFound)` ≅ `search p L` [IH 1]
Inductive Hypothesis 2:
`findOne p R (fn x => x) (fn () => raise NotFound)` ≅ `search p R` [IH 2]
To Show: ∀ total functions `p : 'a -> bool`,

`findOne p (Branch(L,R)) (fn x => x) (fn () => raise NotFound)` ≅
`search p Branch(L,R)`

Proof:

```
findOne p (Branch (L,R)
(fn x => x) (fn () => raise NotFound)    ≅ case Branch(L,R) of ...        [step]
                                         ≅ findOne p L s
                                           (fn () => findOne p R s k)    [step]
```

Here, there are four cases: ∃ `v`in `L` and not in `R` such that `p v` ≅ `true`, ∃ such a `v` in `L` and in `R`, ∃ `v` in `R` but not in `L`, and such a `v` does not exist in either `L` nor `R`.

Task 3.2 (cont.)

Case 1: ∃ `v : 'a` in `L` such that `p v` ≅ `true`, and such a value does not exist in `R`

```
findOne p (Branch (L,R)
(fn x => x) (fn () => raise NotFound)  ≅ case (Branch(L,R) of
                                           Leaf(x) =>...
                                           |Branch(L,R) => findOne p L (fn () =>
                                           findOne p R s k)                    [step]
                                       ≅ v                                     [Theorem 3.1]
```

By Theorem 3.1, since we have assumed that there exists a value `v` in `L` for which `p v` ≅ `true`, the function call `findOne p L s k` ≅ `s v` for all total functions `s`,`k`, and the anonymous functions `(fn x => x)` (identity) and `(fn () => raise NotFound)` are total.

To prove the rest of this case, let us take a detour and prove the correctness of `search`.

Theorem 3.2.1: ∀ total functions `p : 'a -> bool` and ∀ `S : 'a shrub`, `search p S` ≅ the leftmost `v` if a value `v : 'a` exists in `S` for which `p v` ≅ `true`, and ≅ `raise NotFound` if such a value does not exist.
Proof: By structural induction on `S`
Base Case: `T` ≅ `Leaf v` for some value `v : 'a`
To Show: `search p (Leaf v)` ≅ `v` if `p v` ≅ `true`, and ≅ `raise NotFound` otherwise
Proof:

```
search p (Leaf v)       ≅ case (Leaf v) of
                          Leaf x => (case p(x) of ...        [step]
                        ≅ case p(v) of
                          true => v
                          |false => raise NotFound)          [step]
```

Again, since `p` is assumed to be a total function, either `p v` ≅ `true`, or `p v` ≅ `false`.
Case 1: `p v` ≅ `true`

```
search p (Leaf v)  ≅ case p(v) of
                     true => v
                     |false => raise NotFound)
                   ≅ case true of
                     true => v
                     |false => raise NotFound)  [p is total, Referential Transparency]
                   ≅ v                          [step]
```

Thus, in this case, Theorem 3.2.1 holds, since there exists a value `v` in `Leaf v` for which `p v` ≅ `true`, and this `v` is the leftmost (only) such value.

Task 3.2 (cont.)

Case 2: `p v` $\cong$ `false`

```
search p (Leaf v)  ≅ case p(v) of
                      true => v
                      |false => raise NotFound)
                   ≅ case false of
                      true => v
                      |false => raise NotFound)   [p is total, Referential Transparency]
                   ≅ raise NotFound               [step]
```

Again, Theorem 3.2.1 holds, since there are no values `v` in `Leaf v` for which `p v` $\cong$ `true`. As such, Theorem 3.2.1 holds for the case that `S` $\cong$ `Leaf v` for some value `v : 'a`.

Inductive Step: `S` $\cong$ `Branch(L,R)` for some values `L : 'a shrub` and `R : 'a shrub`
Inductive Hypothesis 1:
`search p L` $\cong$ the leftmost `v : 'a` if $\exists$ `v` in `L` such that `p v` $\cong$ `true`, and $\cong$ `raise NotFound` otherwise. [IH 1]
Inductive Hypothesis 2:
`search p R` $\cong$ the leftmost `v : 'a` if $\exists$ `v` in `R` such that `p v` $\cong$ `true`, and $\cong$ `raise NotFound` otherwise. [IH 2]
To Show: `search p (Branch(L,R))` $\cong$ `v` for a value `v : 'a` in `(Branch(L,R))` such that `p v` $\cong$ `true`, and $\cong$ `raise NotFound` otherwise.

```
search p (Branch(L,R))   ≅ case (Branch(L,R)) of
                            Leaf(x) =>...
                            |Branch(L,R) =>...                           [step]
                         ≅ search p L handle NotFound => search p R      [step]
```

Once again, there are 4 cases here: either a value `v` for which `p v` $\cong$ `true` exists in `L` only; it exists in `R` only; such a value exists in both `L` and `R`; and such a value does not exists in either `L` nor `R`.

However, inspecting the code above, we see that `search p L` is carried out first, and `search p R` is only carried out if the call to `search p L` raises the `NotFound` exception, so the case where a value `v` exists in both `L` and `R` results in the same result as if such a value existed only in `L`. Therefore, we will only examine three cases.

Task 3.2 (cont.)

Case 1: ∃ `v` in `L` such that `p v` ≅ `true`

```
search p (Branch(L,R))   ≅ search p L handle NotFound => search p R
                         ≅ v handle NotFound => search p R            [IH 1]
```

This is justified, as according to IH 1, if such a value exists in `L`, then the call to `search p L` must evaluate to `v`.

```
search p (Branch(L,R))   ≅ v handle NotFound => search p R
                         ≅ v                                          [handle-value]
```

By IH 1, `v` is the leftmost value in `L` for which `p v` ≅ `true`, and `L` is in the leftmost branch of `Branch(L,R)`.
∴ `v` is the leftmost value in `Branch(L,R)` for which `p v` ≅ `true`, and Theorem 3.2.1 holds.

Case 2: ∃ `v` in `R` such that `p v` ≅ `true`, and such a `v` does not exist in `L`

```
search p (Branch(L,R))   ≅ search p L handle NotFound => search p R
                         ≅ raise NotFound handle NotFound => search p R   [IH 1]
```

This is justified, as by IH 1, since no value `v` exists in `L` for which `p v` ≅ `true`, the expression must raise the `NotFound` exception.

```
search p (Branch(L,R))   ≅ raise NotFound handle NotFound => search p R
                         ≅search p R                                      [handle-raise]
                         ≅ v                                              [IH 2]
```

Since a value `v` exists in `R` for which `p v` ≅ `true`, by IH 2 `search p R` ≅ the leftmost such `v`. As no such value exists in `L`, this `v` is thus the leftmost value in `Branch(L,R)` for which `p v` ≅ `true`, and Theorem 3.2.1 holds.

Case 3: A value `v` exists in neither `L` nor `R` for which `p v` ≅ `true`

```
search p (Branch(L,R))   ≅ search p L handle NotFound => search p R
                         ≅ raise NotFound handle NotFound => search p R   [IH 1]
                         ≅ search p R                                     [handle-raise
                         ≅ raise NotFound                                 [IH 2]
```

In this case, since a value `v` that fulfills the conditions does not exist in either `L` nor `R`, it does not exist in `Branch(L,R)`. Since the call `search p Branch(L,R)` evaluates to `raise NotFound` in this case, Theorem 3.2.1 holds.

Since Theorem 3.2.1 holds in all three cases, the Inductive Step holds.
As the Base Case and the Inductive Step hold, Theorem 3.2.1 must be true, and `search p S` is correct for all total functions `p : 'a -> bool` and all `S : 'a shrub`.

Task 3.2 (cont.)

Having proved that `search p S` is correct, let us now return to Case 1 of the Inductive Step in the proof for Theorem 3.2. If we consider the expression `search p (Branch(L,R))`,

`search p (Branch(L,R))`	$\cong$ `case (Branch(L,R)) of ...`	[step]
	$\cong$ `search p L handle NotFound => search p R`	[step]
	$\cong$ `v handle NotFound => search p R`	[Theorem 3.2.1]

Remember that in Case 1, we are assuming that $\exists$ `v` in `L` such that `p v` $\cong$ `true`.

`search p (Branch(L,R))`	$\cong$ `v handle NotFound => search p R`	
	$\cong$ `v`	[handle-value]

Since `findOne p (Branch(L,R)) (fn x => x) (fn () => raise NotFound)` $\cong$ `v`, and `search p (Branch(L,R))` $\cong$ `v`, by extensional equivalence,

`findOne p (Branch(L,R)) (fn x => x) (fn () => raise NotFound)` $\cong$
`search p (Branch(L,R))`

<u>Case 2:</u> $\exists$ `v : 'a` in `R` such that `p v` $\cong$ `true`, and such a value does not exist in `L`

`findOne p (Branch (L,R)`		
`(fn x => x) (fn () => raise NotFound)`	$\cong$ `case (Branch(L,R) of`	
	`Leaf(x) =>...`	
	`|Branch(L,R) => findOne p L (fn () =>`	
	`findOne p R s k)`	[step]
	$\cong$`(fn () => findOne p R s k) ()`	[Theorem 3.1]

This is justified, as since we are assuming that no value `v` exists in `L` for which `p v` $\cong$ `true`, by Theorem 3.1 the call `findOne p L s k` $\cong$ `k ()` for all total `p`,`s`, and `k`.

`findOne p (Branch (L,R)`		
`(fn x => x) (fn () => raise NotFound)`	$\cong$ `(fn () => findOne p R s k) ()`	
	$\cong$ `findOne p R s k`	[step]
	$\cong$ `v`	[Theorem 3.1]

Following similar logic to Case 1, we can see that if $\exists$ `v` fulfilling the condition `p v` $\cong$ `true` in `R`, then by Theorem 3.2.1 `search (Branch(L,R))` $\cong$ `v` as well, and by extensional equivalence the call to `findOne p (Branch(L,R))...` $\cong$ `search p (Branch(L,R))`.

Task 3.2 (cont.)

Case 3: ∃ `v` in both `R` and `L` such that `p v` ≅ `true`
This case, in a similar manner to the analagous case in the proof for Theorem 3.2.1, is trivial, in that in both `findOne` and `search`, the left branch is evaluated first, so if a value exists in both the left and right branches of the shrub, the failure continuation in `findOne` and the exception handler in `search` will not come into play, and the proof will go exactly like Case 1.

Case 4: There does not exist a value `v` in either `L` nor `R` that fulfills `p`.
By Theorem 3.1, `findOne p L (fn x => x) (fn () => raise NotFound` ≅ `(fn () => raise NotFound) ()`, and `findOne p R (fn x => x) (fn () => raise NotFound` ≅ `(fn () => raise NotFound) ()`.
If we step the code for `search p (Branch(L,R))`,

`search p (Branch(L,R))`	≅`case Branch(L,R) of ...`	[step]
	≅ `search p L handle NotFound => search p R`	[step]
	≅ `(fn () => raise NotFound) () handle NotFound => search p R`	[IH 1, Referential Transparency]
	≅ `raise NotFound handle NotFound => search p R`	[step]
	≅ `search p R`	[handle-raise]
	≅ `(fn () -> raise NotFound) ()`	[IH 2, Referential Transparency]
	≅ `raise NotFound ()`	[step]

Again, using Theorem 3.1, `findOne p (Branch(L,R)) (fn x => x) (fn () => raise NotFound)` ≅ `(fn () => raise NotFound) ()` ≅ `raise NotFound`, since a value `v` that fulfills `p` does not exist in either `L` nor `R` ∴ by extensional equivalence, `findOne p (Branch(L,R))`. . . ≅ `search p (Branch(L,R))`.

Since all four cases of the Inductive Step hold, and the Base Case holds, Theorem 3.2 must be true.

Task 5.1

In the specification for set intersection, a string s is in $L(r_1 \cap r_2)$ if s is in $L(r_1)$ and s is in $L(r_2)$. Translating this specification to the implementation of `match`, `match Both(r1,r2) cs k` should only evaluate to `true` if $\exists p, s$ such that `p@s` $\cong$ `cs`, $p \in L(r_1)$ and $p \in L(r_2)$. The important thing to note here is that the *same* p must be in both $L(r_1)$ and $L(r_2)$. With the implementation of `badBoth`, it simply calls `match` two separate times on the same input character list `cs`, and evaluates the logical AND of these two using the infix `andalso`. The issue is that the two calls to match might evaluate to `true` for *different prefixes* `p`, which is not what is required by the specification.

Say for example we call `badBoth Times(Char #"a", Char #"b") Star(Char #"a") [#"a", #"b"] (fn l : char list => true)`. If we look at the first call to `match` that `badBoth` makes:

```
match (Times(Char #"a", Char #"b")) [#"a", #"b"] (fn l => true)
```

...we can see that this call will eventually evaluate to `true`, since $\exists$ a prefix p in the char list, namely $p \cong$ `"ab"`, such that $p \in L($`Times(Char #"a", Char#"b"`$)$. We can also examine the second call that `badBoth` makes:

```
match (Star(Char #"a")) [#"a", #"b"] (fn l => true)
```

...we can see that this call will also evaluate to true, if we let the prefix $p \cong$ `"a"`. However, notice that in the two cases, the prefixes that fulfill the conditions are not the same:

$$\texttt{"ab"} \ncong \texttt{"a"} \tag{1}$$

Yet, since the two calls to `match` above evaluate to true, `badBoth` will evaluate to `true`. So it can be seen how for these inputs, badBoth will not meet its spec.

Task 5.3

<u>Theorem 3:</u> $\forall$ values `cs : char list` and `k : char list -> bool`, if `match (Both(r1,r2)) cs k` $\cong$ `true`, then $\exists$ values p, s such that `p@s` $\cong$ `cs`, $p \in L($`Both(r1,r2)`$)$, and `k s` $\cong$ `true`.

<u>Inductive Hypothesis 1:</u> `match` is *sound* for `r1`, i.e.
$\forall$ values `cs : char list` and `k : char list -> bool`, if
`match r1 cs k` $\cong$ `true`, then $\exists$ values p, s such that `p@s` $\cong$ `cs`, $p \in L($`r1`$)$, and `k s` $\cong$ `true`. [IH 1]

<u>Inductive Hypothesis 2:</u> `match` is *sound* for `r2`, i.e.
$\forall$ values `cs : char list` and `k : char list -> bool`, if
`match r2 cs k` $\cong$ `true`, then $\exists$ values p, s such that `p@s` $\cong$ `cs`, $p \in L($`r2`$)$, and `k s` $\cong$ `true`. [IH 2]

To Show: Assuming values `cs` and `k` such that `match (Both(r1,r2)) cs k` $\cong$ `true`, show that $\exists$ values p, s such that `p@s` $\cong$ `cs`, $p \in L($`Both(r1,r2)`$)$, and `k s` $\cong$ `true`.

Proof:

`match Both(r1,r2) cs k`	$\cong$	`true`	
`case Both(r1,r2) of...`	$\cong$	`true`	[step]
`match r1 cs (fn cs' =>`			
`match r2 cs (fn cs'' =>`			
`charlisteq(cs',cs'') andalso k cs'))`	$\cong$	`true`	[step]

By soundness of `match r1 cs k` $\forall$ values `cs` and `k` [IH 1], then there must exist values p_1, s_1 such that p_1`@`s_1 $\cong$ `cs`, $p_1 \in L($`r1`$)$, and for the given `k`, `k` s_1 $\cong$ `true`. In this case, `k` is the continuation above, so `k` s_1 $\cong$ `true` really means:
`(fn cs' => match r2 cs (fn cs'' => charlisteq(cs', cs'') andalso k cs'))` s_1
$\cong$ `true`

Now, by the soundness of `match r2 cs k` $\forall$ values `cs` and `k` [IH 2], then there must exist values p_2, s_2 such that p_2`@`s_2 $\cong$ `cs`, $p_2 \in L($`r2`$)$, and for the given `k`, `k` s_2 $\cong$ `true`. In a similar manner to the above steps, in this case, `k` s_2 $\cong$ `true` means:
`(fn cs'' => charlisteq(cs', cs'') andalso k cs')` s_2
$\cong$ `true`
By the application of the first anonymous function, we know that in the expression above, `cs'` $\cong s_1$.

By the application of the second anonymous function, we know that `cs''` $\cong s_2$. This means:

`charlisteq(`s_1, s_2`) andalso k` s_1`)`	$\cong$	`true`	[Referential Transparency]

Task 5.3 (cont.)

If we examine the first section of the expression above, we can see that:

$$\begin{array}{rcrl} \texttt{charlisteq}(s_1, s_2) & \cong & \texttt{true} & \text{[Lemma 1]} \\ s_1 & \cong & s_2 & \text{[Lemma 3]} \\ s_1 & = & s_2 & \text{[Lemma 5]} \end{array}$$

Since $p_1\texttt{@}s_1 \cong \texttt{cs}$ and $p_2\texttt{@}s_2 \cong \texttt{cs}$,

$$p_1\texttt{@}s_1 \cong p_2\texttt{@}s_2 \quad \text{[Extensional Equivalence]}$$

By the equivalence shown above, since $s_1 \cong s_2$, then:

$$\begin{array}{rcrl} p_1 & \cong & p_2 & \text{[Lemma 4]} \\ p_1 & = & p_2 & \text{[Lemma 5]} \end{array}$$

Now, let $p = p_1 = p_2$, and $s = s_1 = s_2$. By extensional equivalence, $p\texttt{@}s \cong \texttt{cs}$. Since $p = p_1 = p_2$,

$$p \in L(\texttt{r1}) \text{ and } p \in L(\texttt{r2})$$

By our definition of Set Intersection, we can then say that:

$$p \in L(\texttt{r1} \cap \texttt{r2})$$

Furthermore, since we defined above that, in the context of the last continuation, `cs'` $\cong s_1$, and $s = s_1$,

$$\begin{array}{rcrl} \texttt{k cs'} & \cong & \texttt{true} & \text{[Lemma 1]} \\ \texttt{k } s_1 & \cong & \texttt{true} & \text{[Referential Transparency]} \\ \texttt{k } s & \cong & \texttt{true} & \text{[Referential Transparency]} \end{array}$$

We have thus proved that, assuming `match (Both(r1,r2)) cs k` $\cong$ `true` for some `cs : char list` and `k : char list -> bool`, $\exists p, s$ such that $p\texttt{@}s \cong \texttt{cs}$, $p \in L(\texttt{Both(r1,r2)})$, and `k` $s \cong$ `true`. Thus, Theorem 3 must be true, and `match` must be sound for `Both`.

Task 5.4

In the specification for set difference, a string s is in $L(r_1 \setminus r_2)$ if s is in $L(r_1)$ and s is not in $L(r_2)$. Translating this specification to the implementation of `match`, `match Diff(r1,r2) cs k` should only evaluate to `true` if $\exists p, s$ such that `p@s` $\cong$ `cs`, $p \in L(r_1)$ and $p \notin L(r_2)$. The important thing to note here is that the *same* p must be in $L(r_1)$ and cannot be in $L(r_2)$. With the implementation of `badDiff`, in a similar manner to `badBoth`, it simply calls `(match r1 cs k) andalso not (match r2 cs k)`. The issue is that the specification requires that *the same* prefix `p` $\in L($`r1`$)$ and $\notin L($`r2`$)$, while `badDiff` might evaluate to `false` due to evaluating to `true` in the call to `match r2 cs k` for a *different prefix* `p` than in the first call to `match r1 cs k`.

Say for example we call `badDiff (Char #"a") (Times(Char #"a", Char #"b")) [#"a", #"b"] (fn l : char list => true)`. From the definition of set difference, we know that $\exists$ a prefix $p \cong$ `"a"`, where `"a"` $\in L($`Char #"a"`$)$, but `"a"` $\notin L($`Times(Char #"a", Char#"b")`$)$, so this call should evaluate to `true` according to the spec.

However, if we look at the first call to `match` that `badDiff` makes:

```
match (Char #"a") [#"a", #"b"] (fn l => true)
```

...we can see that this call will eventually evaluate to `true`, since `"a"` is a prefix in the char list such that `"a"` $\in L($`Char #"a"`$)$.

If examine the second call that `badDiff` makes:

```
match (Times(Char #"a", Char #"b")) [#"a", #"b"] (fn l => true)
```

...we can see that this call will also evaluate to `true`, if we let the prefix $p \cong$ `"ab"`. Since the two calls to `match` are arranged like so:

```
badDiff (Char "a") (Times(Char "a", Char "b")) ["a", "b"] (fn l => true)   ≅
                (match (Char "a") ["a", "b"] (fn l => true)) andalso not
           (match (Times(Char "a", Char "b")) ["a", "b"] (fn l => true))
```

We can see that this call will evaluate to:

```
(true) andalso not (true)
     ≅ true andalso false[step]
                ≅ (false)[step]
```

Which is not what we expected from this call, given the specification for set difference. So it can be seen how `badDiff` does not meet its spec for these inputs.

Task 5.6

Damn, this is pretty cool.
So after I've typed in `findMessage messageKey startlist`, I'm presented with a huge block of text telling me that I am at the entrance to a large CAVE (all caps). Sounds like a choose-your-own-adventure sort of deal. I like caves, so I'll go with the first option given to me and decode `cavelist` using `findMessage` and `messageKey`.

Next, I'm in the cave, with a PEDESTAL and a FOUNTAIN (both all caps. I sense a theme...). I like pedestals, so I'll examine the `pedestallist` first.

Examining the pedestal, I find a small box sitting in the center. At this point, I'm given the option to return to the CAVE, but I'm no wuss. I'll inspect the box by decoding `boxlist`.

...only to find that I've failed, and I need to call the magic failure continuation. Alright, I'll play your game. Let's go back to the cave and inspect the fountain, then (screw going to the field).

So inspecting the fountain, I smell curry?! Are you kidding? I'm a poor hungry college student, and now you taunt me with food. Screw you guys. I wanna grab the shiny object at the bottom of the fountain.

```
findMessage messageKey objectlist
```

!! So this is the treasure! The function I found was:

```
(fn x => (fn y => x))
```

Awesome. Thanks for the game. Peace out yo.