Contents

1 Types and Polymorphism .......................................................... 3
   Task 1. (1 point) .................................................................. 3
   Task 2. (1 point) .................................................................. 3
   Task 3. (1 point) .................................................................. 3
   Task 4. (1 point) .................................................................. 3
   Task 5. (2 points) ................................................................. 3
   Task 6. (2 points) ................................................................. 3
   Task 7. (2 points) ................................................................. 4
   Task 8. (2 points) ................................................................. 4
1.1 Detypify .............................................................................. 4
   Task 9. (1 point) .................................................................. 4
   Task 10. (1 point) ................................................................ 4
   Task 11. (1 point) ............................................................... 4
   Task 12. (2 points) ............................................................. 4

2 Calculus .................................................................................... 5
   2.1 Representing Polynomials ................................................ 5
   2.2 Differentiation ............................................................... 5
      Task 13. (6 points) .......................................................... 6
   2.3 Integration ................................................................. 6
      Task 14. (6 points) .......................................................... 6

3 No fn fun .................................................................................. 7
   3.1 Reverse Polish Notation ................................................ 7
   3.2 Implementation ........................................................... 7
      Task 15. (6 points) .......................................................... 8
      Task 16. (15 points) ...................................................... 8
      Task 17. (6 points) ........................................................ 9

4 Harrison’s Cool Beans Party ....................................................... 10
   4.1 Requirements ............................................................. 10
   4.2 Implementation .......................................................... 11
      Task 18. (20 points) ...................................................... 11
      Task 19. (2 points) ........................................................ 11

5 o, a map! .................................................................................. 12
      Task 20. (15 points) ...................................................... 12

6 Doop de Doop ......................................................................... 13
6.1 It's complicated
6.2 Seeing Double
6.3 Slow Doop
   Task 21. (2 points)
   Task 22. (1 point)
   Task 23. (2 points)
   Task 24. (2 points)
6.4 Fast Doop
   Task 25. (4 points)
   Task 26. (4 points)

A Appendix
1 Types and Polymorphism

In class we discussed typing rules. In particular:

- A function expression \( \texttt{fn x => e} \) has type \( t \rightarrow t' \) if and only if, by assuming that \( x \) has type \( t \), we can show that \( e \) has type \( t' \).
- An application \( e_1 \ e_2 \) has type \( t' \) if and only if there is a type \( t \) such that \( e_1 \) has type \( t \rightarrow t' \) and \( e_2 \) has type \( t \).
- An expression can have a more specific type that is an instance of its most general type. For instance, an expression of type \( \texttt{`a list} \) can be used in a context where the type is \( \texttt{int list} \).

For each of the following SML declarations, what is the most general type of the value being declared? If one of the declarations is not well-typed, briefly explain why.

\textit{Note:} You might think to just plug the declaration into the SML/NJ REPL and have it do all the work for you. This is fine to check your work, but remember that we might ask you to solve this kind of problem on an exam. You are well-advised to learn how to do it by hand, because you won’t have a REPL in the exam.

\textbf{Task 1.} (1 point)

\begin{verbatim}
fun sml (functions, values) = 
  case functions of 
    [] => values 
  | z::zs => "15150 " ^ sml (zs, values)
\end{verbatim}

\textbf{Task 2.} (1 point)

\begin{verbatim}
val Jacob = fn x => [x] :: []
\end{verbatim}

\textbf{Task 3.} (1 point)

\begin{verbatim}
val c15150 = fn (c0, k) => if c0 then k else nil
\end{verbatim}

\textbf{Task 4.} (1 point)

\begin{verbatim}
val foo = fn (f, x) => not (f x):: nil
\end{verbatim}

\textit{Remember:} \texttt{not : bool \rightarrow bool} is defined by

\begin{verbatim}
fun not true = false 
  | not false = true
\end{verbatim}

\textbf{Task 5.} (2 points)

\begin{verbatim}
fun g (z,[])) = []
  | g (z,x::xs) = g (x,[z]::xs)
\end{verbatim}

\textbf{Task 6.} (2 points)
fun bar (x,y) = [x] :: [y]

Task 7. (2 points)
fun f x = (x, f x)

Task 8. (2 points)
val f = map (fn x => x + 1) o foldr ( op@ ) []

1.1 Detypify

Now, let’s try doing this backwards! Instead of us giving you the value and you telling us the type, let’s have us give you the type and you tell us a value! For each of the following types, provide a value that has that type. Here are the restrictions on your solutions:

**Constraint:**

- Any function you write **must be total**, assuming all function arguments are total.
- **Do not** write any helper functions.

Here is an example. If we ask for a value of type 'a -> 'a, you could give fn x => x.

Task 9. (1 point)
'a * 'b -> 'a

Task 10. (1 point)
'a * ('a -> 'b) -> 'b

Task 11. (1 point)
'a -> (unit -> 'a)

Task 12. (2 points)
('a -> 'b) * ('c -> 'a) -> 'c -> 'b

**Note:** Again, it is fine to check your work by plugging them into the SML/NJ REPL. Just remember that detypify questions may be on an exam as well, where REPLs are not provided.
2 Calculus

2.1 Representing Polynomials

We can represent a polynomial \( c_0 + c_1 x + c_2 x^2 + \ldots \) as a function that maps a natural number, \( i \), to the coefficient \( c_i \) of \( x^i \). For these tasks we will take the coefficients \( i \) to be real numbers (of type \texttt{real}). Therefore, we have the following type definition for polynomials:

\[
\texttt{type poly = int -> real}
\]

For example, we can define a function \( p \) as follows:

\[
\texttt{val p = fn 0 => 1.0 | 1 => 0.5 | 2 => 6.0 | _ => 0.0}
\]

The function \( p \) would represent the polynomial \( 1 + \frac{1}{2}x + 6x^2 \), since \( p \ i \) represents the coefficient on \( x^i \). We’ll adopt the convention that a polynomial just maps the negative integers to 0.0, as \( p \) does.

As an example of how to define operations on polynomials defined in this manner, see the following definition for the addition of polynomials:

\[
\texttt{fun plus (p1 : poly, p2 : poly) : poly = fn e => p1 e + p2 e}
\]

We’re going to ask you to implement two more sophisticated functions which manipulate polynomials: \texttt{differentiate} and \texttt{integrate}.

To help you, we’ve provided a couple utility functions, such as

\[
\texttt{polynomialEqual : poly * poly * int -> bool}
\]

where \texttt{polynomialEqual (p1,p2,n)} compares the first \( n \) coefficients of \( p1 \) and \( p2 \) for equality\(^1\), i.e. it checks \( p1 \ i = p2 \ i \) for \( 0 \leq i < n \). This function is implemented and documented in the \texttt{code/diff/diff.sml}. It should be helpful for writing tests, such as:

\[
\texttt{val true = polynomialEqual (}
\begin{array}{l}
\texttt{fn 1 => 1.0 + 1.0 | _ => 0.0,}
\texttt{fn 1 => 2.0 | _ => 0.0,}
\texttt{3}
\end{array}
\)

2.2 Differentiation

Recall from calculus that differentiation of polynomials is defined as follows:

\[
\frac{d}{dx} \sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} ic_i x^{i-1}
\]

For example, if \( p \) is the same \texttt{poly} from the previous page (the one representing \( 1 + \frac{1}{2}x + 6x^2 \)), then the following equivalence should hold:

\(^1\)We can’t actually check if two real numbers are equal in SML, because of the potential for rounding errors. Under the hood, we use a function which checks if two real numbers are within epsilon of each other for a fixed epsilon.
differentiate $p \equiv \text{fn } 0 \Rightarrow 0.5 \mid 1 \Rightarrow 12.0 \mid \_ \Rightarrow 0.0$

Task 13. (6 points)
Define the function

\[
\text{differentiate} : \text{poly} \rightarrow \text{poly}
\]

REQUIRES: $p$ is a valid polynomial (i.e. $p \ n \equiv 0.0$ for $n < 0$ and $p \ n \equiv 0.0$ for all but finitely many $n$)

ENSURES: differentiate $p \equiv p'$, where $p'$ is the derivative of $p$

**Constraint:** differentiate may not be recursive.

### 2.3 Integration

Recall from calculus that integration of polynomials is defined as follows:

\[
\int \sum_{i=0}^{n} c_i x^i \, dx = C + \sum_{i=0}^{n} \frac{c_i}{i+1} x^{i+1}
\]

where $C$ is an arbitrary constant known as the constant of integration. Because $C$ can be any number, the result of integration is a family of polynomials, one for each choice of $C$. We will represent a family of polynomials as a function real $\rightarrow$ poly which takes the value $x : \text{real}$ and returns the polynomial whose constant of integration $C$ is $x$.

So, if $p : \text{poly}$ is some polynomial, then integrate $p$ should be a family of polynomials integrate $p : \text{real} \rightarrow \text{poly}$. Thus, the type of integrate is $\text{poly} \rightarrow (\text{real} \rightarrow \text{poly})$.

For example, if $p$ is the same $\text{poly}$ from the previous page (the one representing $1 + \frac{1}{2}x + 6x^2$), then the following equivalence should hold:

\[
\text{integrate} \ p \equiv \text{fn } C \Rightarrow (\text{fn } 0 \Rightarrow C \mid 1 \Rightarrow 1.0 \\
\mid 2 \Rightarrow 0.25 \mid 3 \Rightarrow 2.0 \mid \_ \Rightarrow 0.0)
\]

Task 14. (6 points)
Define the function

\[
\text{integrate} : \text{poly} \rightarrow (\text{real} \rightarrow \text{poly})
\]

REQUIRES: $p$ is a valid polynomial (i.e. $p \ n \equiv 0.0$ for $n < 0$ and $p \ n \equiv 0.0$ for all but finitely many $n$)

ENSURES: integrate $p \equiv P$, where $P \ c$ is the antiderivative of $p$ with constant of integration $c$.

Alternatively, you could say that differentiate ((integrate $p) \ c) \equiv p$ for all $c : \text{real}$.

**Constraint:** integrate may not be recursive.
Before he was a Jedi Master, Yoda was actually a 150 TA. He spent many years working on his types, coding in SML, and making mathematical expressions. However, upon learning about the wonders of Higher-Order Functions, his powers increased and he started writing even more powerful expressions. However, the evil Sith also learned of this power and wanted it for the forces of evil. Yoda, wanting to keep working on his mathematical expressions but also wary of Sith involvement, started communicating in reverse to send secret messages to the Resistance, even his math!

Before he passed away, Yoda wrote an expression that would reveal the masterminds behind the Sith. However, he passed away before he could write a Higher-Order Function to solve it. You, as a 150 student, are tasked with this important duty.

Here is the expression: \(10 \ 11 \ + \ 6 \ - \ 1000 \ * \ 130 \ + \ 8 \ -\)

### 3.1 Reverse Polish Notation

Most of the time we write mathematical expressions with infix operations like \(3 \ * \ (1 \ + \ 5) \ - \ 15\). This is great, but there is another way to write these expressions called Reverse Polish Notation. In reverse polish notation, we use a “postfix” style, so the operations actually go after the operands, like \(3 \ 1 \ 5 \ + \ * \ 15 \ -\). We can calculate the value of such an expression by using a stack. When we see an integer, we push it onto the stack, and when we see an operator, we pop its arguments off the stack then calculate and push the result onto the stack.

![Diagram of Reverse Polish Notation]

Let’s step through the example above. We push the numbers 3, 1, and 5 onto the stack. Encountering the addition operation, we pop two numbers (1 and 5) off and add them, pushing 6 onto the stack. We then encounter the multiplication operation, so we pop two numbers (3 and 6), calculate the result, and push it. Finally we push 15 to the stack and subtract from 18, leaving us with a stack containing just 3.

### 3.2 Implementation

Your task will be to implement a function \(\texttt{rpn}\) that evaluates reverse polish notation:

\[
\texttt{rpn : string \rightarrow int list}
\]
To help you out, we have created a type that captures all kinds of “tokens” that might appear in a reverse polish notation calculator:

```
datatype token =
  Integer of int
| Multiply
| Add
| Subtract
| Divide
| SumAll
```

We have also provided a function `tokenize : string -> token` that will convert the string representations of these tokens to this type. For example, `tokenize "+"` evaluates to `Add`.

Let \( x \) be the top of the stack and \( y \) be the second item on the stack. The following table describes each operation:

<table>
<thead>
<tr>
<th>String</th>
<th>Token</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiply</td>
<td>( x \times y )</td>
</tr>
<tr>
<td>+</td>
<td>Add</td>
<td>( x + y )</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
<td>( y - x )</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
<td>( y \div x )</td>
</tr>
<tr>
<td>sum</td>
<td>SumAll</td>
<td>Sum of all integers on the stack (0 if the stack is empty)</td>
</tr>
</tbody>
</table>

For all of the following three tasks, **you may only use/define functions as specified in each task**. If you violate this requirement, **you will lose all the points** on that task.

As a reminder, you can declare functions using `val` just fine. After all, functions are values!

Also, the composition operator has type `op o: ('b -> 'c) * ('a -> 'b) -> 'a -> 'c`. You will find higher order functions like `map`, `filter`, `foldl`, and `foldr` indispensable.

**Task 15.** (6 points)

Implement `parse`, which, when applied to an RPN string, outputs a list of tokens in the same order as they appeared in the string.

```
parse : string -> token list
```

**REQUIRES:** `s` is a space-separated string of RPN tokens

**ENSURES:** `parse s` ⇒ an ordered list of the tokens that appear in `s`

For example, `parse "3 2 +"` should evaluate to `[Integer 3, Integer 2, Add]`.

You may find the `String.tokens : (char -> bool) -> string -> string list` function useful. It splits a string where the predicate is true. Character literals are written with a `#` symbol. For example the literal for ‘a’ is `#"a"`.

**Constraint:** You may write up to one `fn` expression for this task. You may not write any recursive function declarations or `fun` declarations. You may not use any functions from the `Char` library.

**Task 16.** (15 points)
Implement the \texttt{eval} : \texttt{token list} \rightarrow \texttt{int list} function, which evaluates a token list to the resulting stack after applying the operations as defined above. You may find it useful to have a helper function.

If the \texttt{token list} is invalid (for example, if we tried to evaluate \texttt{Add} on an empty stack), then raise an \texttt{InvalidExpression} exception. To raise an exception, in any clause you would like to raise the exception use the \texttt{raise} syntax and type \texttt{raise InvalidExpression}, or any other expression you’d like to raise!

\begin{verbatim}
  eval : token list -> int list
  REQUIRES: true
  ENSURES: eval T =⇒ the resulting stack after evaluating all of the tokens in T on an empty stack if T is a valid list of tokens. If not, eval T raises an InvalidExpression exception
\end{verbatim}

\textbf{Constraint:} You may write up to one non-recursive \texttt{fun} declaration. You may not write any \texttt{fn} expressions or recursive function declarations.

\textbf{Task 17.} (6 points)

Implement the \texttt{rpn} : \texttt{string} \rightarrow \texttt{int list} function that when applied to an RPN expression returns the stack that results from evaluating it.

\begin{verbatim}
  rpn : string -> int list
  REQUIRES: s is a space-separated string of RPN tokens
  ENSURES: rpn s =⇒ the resulting stack after evaluating RPN expression s. If s is not a valid RPN expression, rpn s raises an InvalidExpression exception
\end{verbatim}

\textbf{Constraint:} You may not write any \texttt{fun} declarations or \texttt{fn} expressions for this task.

\textbf{Hint:} Follow the types!
4 Harrison’s Cool Beans Party

Harrison, a.k.a. HRSN, one of the head TAs of 15-150, is planning to hold a potluck for all the cool beans on the 150 course staff. He wants to know what the best time to meet is, so he makes everyone send him their schedule. However, Harrison is rather lazy, so he doesn’t want to have to pick through all the data to figure out when everyone’s free. Therefore, he’s enlisted your help to help him comb through all the data and figure out a good time!

4.1 Requirements

Everyone has (thankfully) represented their schedule in the same way: they have sent you a list of intervals where they are busy. For example, they might send in the list

\[ [(3, 6), (9, 10), (11, 12)] \]

to indicate that they are busy from 3 to 6, from 9 to 10, and from 11 to 12. You will be given a list of such lists, and your task is to find a time when no one is busy.

We’ll also define the following type alias, for ease of explanation:

```
type interval = int * int
```

You may assume the following information for your implementation:

- Each person’s obligations are disjoint (i.e. no single schedule will have overlaps)
- All “schedules” are well-formatted (i.e. the first element of each tuple will be strictly less than the second element of the tuple)
- Assume time (measured in some arbitrary unit) starts at 0 and ends at the largest number given in the input
- Assume that people can instantly transport themselves to and from commitments (i.e. if someone is busy until 3, they are free immediately at 3)

Unfortunately, Harrison’s computer is ancient, so any implementation that runs in greater than \(O(n \log n)\) asymptotic time will cause his computer to explode.
4.2 Implementation

Task 18. (20 points)

Write the function

```plaintext
all_available : interval list list -> interval list
```

that returns a single list of all intervals (no degenerate intervals) in any order for which everyone is available.

**NOTE:** The `msort` function can be found in the `code/intervals/lib.sml` file. In order to use this function, please type `smlnj lib.sml intervals.sml` into the command line.

**Constraint:**
- `all_available` *may not* be recursive. You may write additional helper functions for this task, but they *may not* be recursive either.
- This must have $O(n \log n)$ running time or less.

Here are some examples of what we’re looking for:

```plaintext
val [] = all_available []
val [(0,1),(3,4)] = all_available [[[1,2),(4,6),(8,12)],[[1,3),(5,9)]]
val [(1,2),(3,5)] = all_available [[[0,1),(2,3)],[[5,7]]]
val [(1,2)] = all_available [[[0,1),(2,3)]]
```

To help you out, here are some hints!

- We have provided you a polymorphic sorting function with $O(n \log n)$ work in `code/intervals/lib.sml`. You will probably find this helpful.

- You may use the built-in `List.concat` function, which "flattens" a nested list by one level. For example,

  ```plaintext
  List.concat [[1], [2,3]] => [1,2,3]
  List.concat [[]], [1,2,3]] => [1,2,3]
  ```

  This function has $O(n)$ work, where $n$ is the sum of the lengths of the sublists.

Task 19. (2 points)

In at most 3-4 sentences, informally justify why your function’s work is in $O(n \log n)$. In our function `all_available` let $n$ be the total number of interval tuples inputted. If our function takes in some interval list list, $L$, $n$ would be the length of `List.concat` $L$. 

5 o, a map!

Task 20. (15 points)

Given the following definitions:

```
infix 3 o
fun f o g = fn x => f (g x)

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

Prove that: If
- for all \( a, b, c, f, g \),
- where \( a, b, c \) are types,
- and \( f, g \) are valuable,
- and \( f : b \to c \),
- and \( g : a \to b \),
- and \( f \) total,
- and \( g \) total,

then

\[(\text{map } f) \circ (\text{map } g) \cong \text{map } (f \circ g)\]

You may assume the following lemmas:
1. \( o \) total.
2. \( \text{map} \) total.
3. For all functions \( h \), if \( h \) total, then \( \text{map } h \) total.

Recall: the definition of extensional equivalence for functions is as follows:

For some types \( t, u, f : t \to u \) and \( g : t \to u \) are extensionally equivalent \( (f \equiv g) \) if for all values \( x : t \), \( f x \cong g x \).
6 Doop de Doop

6.1 It’s cmp-licated

Recall the built-in order type:

```haskell
datatype order = GREATER | LESS | EQUAL
```

It can be useful to define an order function to impose an ordering on the elements of a type: that is, to take a pair of elements \(x, y\) and tell whether \(x\) is less than, equal to, or greater than \(y\) for any arbitrary notion of less than, equal to, or greater than, using the order datatype to do so. However, for this ordering to be useful the function must be consistent: if it tells us that \(x\) is less than \(y\), then it better not also claim that \(y\) is less than \(x\). Therefore, we define an order function as follows:

**Definition** (Order Function). For any type \(t\), an order function on \(t\) is a function \(\text{cmp} : t \times t \to \text{order}\) such that

1. \(\text{cmp}\) is total
2. For all \(x \in t\), \(\text{cmp} (x, x) \equiv \text{EQUAL}\)
3. For all \(x, y \in t\), if \(\text{cmp} (x, y) \equiv \text{EQUAL}\), then \(\text{cmp} (y, x) \equiv \text{EQUAL}\)
4. For all \(x, y \in t\) and all \(x, y \in t\), \(\text{cmp} (x, y) \equiv \text{LESS}\) if \(\text{cmp} (y, x) \equiv \text{GREATER}\)
5. For all \(x, y, z \in t\), if \(\text{cmp} (x, y) \equiv \text{cmp} (y, z) \equiv \text{EQUAL}\), then \(\text{cmp} (x, z) \equiv \text{EQUAL}\)
6. For all \(x, y, z \in t\), if \(\text{cmp} (x, y) \equiv \text{cmp} (y, z) \equiv \text{LESS}\), then \(\text{cmp} (x, z) \equiv \text{LESS}\)
7. For all \(x, y, z \in t\), if \(\text{cmp} (x, y) \equiv \text{EQUAL}\) and \(\text{cmp} (y, z) \equiv \text{LESS}\), then \(\text{cmp} (x, z) \equiv \text{LESS}\)
8. For all \(x, y, z \in t\), if \(\text{cmp} (y, z) \equiv \text{EQUAL}\) and \(\text{cmp} (x, y) \equiv \text{LESS}\), then \(\text{cmp} (x, z) \equiv \text{LESS}\)

This definition formalizes our notion of an order function, an arbitrary ordering on a type \(t\). All order functions also define a notion of equality on their type, so it may be useful to use that notion to generalize what it means for an element \(x \in t\) to be an element of a list \(L : t\ list\) with respect to an order function \(\text{cmp}\), that is, for \(x\) to be \(\text{cmp}\)-in \(L\), as follows:

**Definition** (cmp-in). For all types \(t\), all order functions \(\text{cmp} : t \times t \to \text{order}\), and all values \(x : t\)

1. If \(L : t\ list = []\), then \(x\) is not \(\text{cmp}\)-in \(L\).
2. For all \(L : t\ list\) and all \(y : t\), \(x\) is \(\text{cmp}\)-in \(y::L\) if \(\text{cmp} (x, y) \equiv \text{EQUAL}\).
3. If \(x\) is \(\text{cmp}\)-in \(L\), then for all values \(y : t\), \(x\) is \(\text{cmp}\)-in \(y::L\).

6.2 Seeing Double

We want to write a function that takes in a \(t\) list and removes all the duplicates.

**Definition** (No Duplicates). For all types \(t\), we say that a list \(L : t\ list\) contains no duplicates if

- \(L \equiv []\), or
- \(L \equiv x::xs\) where \(x\) is not \(\text{cmp}\)-in \(xs\) and \(xs\) contains no duplicates.
6.3 Slow Doop

We’ll implement a function to remove the duplicates from a `list` in the naïve way, which will turn out to be somewhat slow.

**Task 21.** (2 points)
Implement the function `cmpIsIn` in code/removeDups/slowDoop.sml.

```sml
val L = [4, ~2, 6, 4, ~3, 2, 6, 7]
val cmp = Int.compare
val true = cmpIsIn (cmp, 6, L)
val false = cmpIsIn (cmp, 3, L)
val true = cmpIsIn (cmp, ~3, L)
```

**Task 22.** (1 point)
Write and solve recurrences for the work and span of your implementation of `cmpIsIn` in terms of the length `n` of the input list, assuming that `cmp (x,y)` has constant work and span.

**Task 23.** (2 points)
Using `cmpIsIn`, in code/removeDups/slowDoop.sml, define the function that removes the duplicates from a list:

```sml
slowDoop : 'a ord * 'a list -> 'a list
REQUIRES: cmp is an order function
ENSURES: (slowDoop (cmp, L) ⇒ L’ where L’ has no duplicates and x is cmp-in L’ iff x is cmp-in L.

**Constraint:** You may not use any helper functions besides `cmpIsIn`.

**Task 24.** (2 points)
Write and solve recurrences for the work and span of your implementation of `slowDoop` in terms of the length `n` of the input list, assuming that `cmp (x,y)` has constant work and span.

6.4 Fast Doop

We’ll now try to make a version of doop that runs faster than our original `slowDoop`. For this problem, it’s important to think about how we can more efficiently remove duplicates from our
list, rather than brute-forcing our way through the list. Right now, the fastest way we can remove
the duplicates is $O(n^2)$. However, there is a faster way to do this that involves sorting. We can
try sorting the list and, once we have a sorted list, recursively check if any consecutive elements
are equal to each other. If two consecutive elements are equal, then remove one of them and keep
checking!

**Task 25.** (4 points)

Implement the following function in `code/removeDups/fastDoop.sml`, which removes the dupli-
cates from a list:

```sml
fastDoop : 'a ord * 'a list -> 'a list
REQUIRES: cmp is an order function
ENSURES: fastDoop (cmp, L) \implies L’ where L’ has no duplicates and x is cmp-in L’
iff x is cmp-in L.
```

**Constraint:** Your implementation of `fastDoop` must have work that lists in a strictly better
(faster) complexity class than $O(n^2)$.

You should use the `msort` function provided in the code; you may assume, without proof, that
given a list of length $n$, the work of `msort` is $O(n \log n)$ and the span of `msort` is $O(n)$. Note
that you may also implement helper functions.

**Task 26.** (4 points)

Write and solve recurrences for the work and span of your implementation of `fastDoop` in terms
of the length $n$ of the input list, assuming that `cmp (x,y)` has constant work and span. Make
sure your work is less than $O(n^2)$!
A Appendix

(* o: ('b -> 'c) * ('a -> 'b) -> ('a -> 'c) *)
infix 3 o
fun (f o g) x = f (g x)

(* map: ('a -> 'b) -> 'a list -> 'b list *)
fun map _ [] = []
| map f (x::xs) = f x::map f xs

(* filter: ('a -> bool) -> 'a list -> 'a list *)
fun filter _ [] = []
| filter p (x::xs) = if p x then x::filter p xs else filter p xs

(* foldr: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldr _ z [] = z
| foldr g z (x::xs) = g (x, foldr g z xs)

(* foldl: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldl _ z [] = z
| foldl g z (x::xs) = foldl g (g (x, z)) xs

(* zip: 'a list * 'b list -> ('a * 'b) list *)
fun zip (_, []) = []
| zip ([], _) = []
| zip (x::xs, y::ys) = (x, y)::zip (xs, ys)

NOTE: When you try using these functions, you will notice that map, foldr, and foldl work, whereas the others raise an "Unbound variable or constructor" error. This is essentially because these specific functions are in the top-level environment, and even though they are in the List structure, they can be called as map, foldr, and foldl for convenience and other reasons. However, in order to use the other functions, we must reference the structures they are in (List.filter, List.concat, ListPair.zip).

This assignment has a total of 108 points.