Computer Science 15–212–ML: Final Examination
(Sample Solutions)

Monday, May 11, 1998

Name

Name: ______________________________

Andrew ID: ______________________________

Instructions

- This is an open-book, open-notes examination. No computers.
- There are 25 pages in this examination, including three worksheets.
- This examination consists of 6 questions worth a total of 150 points. The point value of each question is given with the question.
- Read each question completely before attempting to solve any part.
- Write your answers legibly *in the space provided* on the examination sheets. If you use the back of a page, please indicate clearly that you have done so on the front.
- Write your name and Andrew id in the space provided on *each page* of this examination.
- Three worksheets are attached to the end of this examination. *Under no circumstances will anything written on the worksheets count for or against your grade on this exam.*

Grading

Question	1	2	3	4	5	6	Total
Score							
Maximum	20	25	25	25	25	30	150

1 Evaluation and Typing [20 Points]

1. [5 Points] For each of the following expressions, state its *most general* type (using type variables, if necessary), or write "Ill-typed" if it does not have a type.

   (a) `3.14+2.18` `: real`

   (b) `(3=4)=(5=6)` `: bool`

   (c) `[1,2,3] :: 4` ILL-TYPED

   (d) `fn x => print (x ^ "\n")` `: string -> unit`

   (e) `fn x => (fn y => x)` `: 'a -> 'b -> 'a`

2. [5 Points] For each of the following well-typed expressions, determine whether it has a value, and, if not, whether it loops forever or raises an exception. For each item, either give its value, write "diverges" if it runs forever, and write "fails" if it raises an exception.

   (a) `let val x = 3 in x * x end` $\hookrightarrow$ `9 : int`

   (b) `let fun I(x):int = I(x) in I(I) end` DIVERGES

   (c) `let fun f (n, h::t) = f (n-1, t) | f (0, l) = l in f (2, [1,2,3]) end` FAILS

   (d) `let fun f 0 = 1 | f n = 2 * f (n-2) in f 5 end` DIVERGES

The next two questions refer to the following signature:

```
signature SIG = sig
  type t
  val zero : t
  val add : t * t -> t
end
```

3. [5 Points] Which of the following structure bindings is well-formed? (Answer "yes" or "no" for each.)

   (a) YES

   ```
   structure S :> SIG = struct
     type t = int
     val zero = 1
     fun add (x, y) = x*y
   end
   ```

   (b) NO

   ```
   structure S :> SIG = struct
     type t = real
     val zero = 0
     fun add (x, y) = x+y
   end
   ```

   (c) YES

```
structure S :> SIG = struct
  type t = int->int
  val zero = fn x => x
  fun add (f, g) (x) = f(g x)
end
```

4. [5 Points] Consider the following structure binding:

```
structure S :> SIG = struct
  type t = int list
  val zero = nil
  fun add (x, y) = x @ y
end
```

For each of the following expressions, give its type, or write "ill-typed" if it has no type.

(a) `S.zero   : S.t`

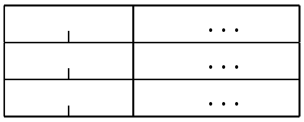
(b) `S.add ([1,2], [3,4])`   ILL-TYPED

(c) `S.add (S.zero, S.zero)   : S.t`

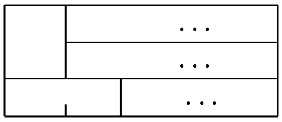
2 Induction and Recursion [25 Points]

In this question we will consider methods for computing the number of tilings T_n of a $3 \times n$ board with tiles of size 1×2. We will derive a recurrence for T_n, then ask that you consider two different methods for programming the recurrence in ML.

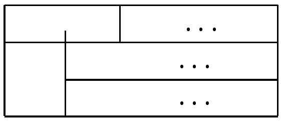
To derive an equation for T_n, we focus on three ways in which we may cover the first column. The choices are these: we may either place three tiles horizontally at the left edge, leaving a $3 \times (n-2)$ board (Figure (a)), or we may place a tile vertically at either the top or bottom edge and place a tile horizontally in the remaining row, leaving a $3 \times (n-1)$ "damaged" board that has one square (top left or bottom left) deleted (Figures (b) and (c)).[1]



(a)



(b)



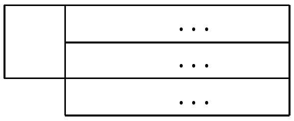
(c)

To complete the tiling after move (a) requires that we complete the tiling of the remaining $3 \times (n-2)$ board by the same procedure. To complete the tiling after move (b) or (c) requires that we complete the tiling of the remaining $3 \times (n-1)$ "damaged" board. This leads to the following recursion equation for T_n:

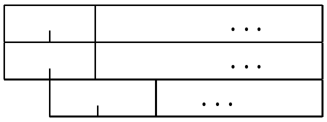
$$T_n = T_{n-2} + 2 \times U_{n-1}$$

where U_n is the number of ways of tiling a $3 \times n$ "damaged board".

To derive an equation for U_n we again consider the possible ways to tile the leftmost squares. The choices are these: we may either place a tile vertically in the first column, leaving a $3 \times (n-1)$ undamaged board (Figure (d)), or we may place three tiles horizontally as far to the left as possible, leaving a $3 \times (n-2)$ "damaged board" (Figure (e)).



(d)



(e)

This leads to the following recurrence for U_n:

$$U_n = T_{n-1} + U_{n-2}$$

To complete the recurrence we must provide the base cases. By convention the 3×0 board has one tiling, the null tiling; by inspection, we see that the 3×1 board has no tilings.

$$\begin{aligned} T_0 &= 1 \\ T_1 &= 0 \end{aligned}$$

Since there can be no "damaged" board of size 3×0, there are no tilings of the 3×0 "damaged" board, and by inspection we see that the 3×1 "damaged" board has one tiling.

$$\begin{aligned} U_0 &= 0 \\ U_1 &= 1 \end{aligned}$$

[1] In the diagrams the vertical and horizontal tiles are intended to be the same shape.

This completes the derivation of the recurrence for tiling a $3 \times n$ board with 1×2 tiles. In what follows it will be convenient to define T_n and U_n to be 0 for all $n < 0$.

1. [5 Points] Give a direct implementation of this recurrence in ML using two mutually-recursive functions `T` and `U` of type `int->int`.

```
fun T 0 = 1
  | T 1 = 0
  | T n = T (n-2) + 2 * U (n-1)
and U 0 = 0
  | U 1 = 1
  | U n = T (n-1) + U (n-2)
```

2. [8 Points] The straightforward implementation you just gave is quite inefficient because of redundancy in the pattern of the recursive calls. To see this, observe that to compute T_n (for $n \geq 2$) we must compute T_{n-2} and U_{n-1}; to compute the latter, we must compute T_{n-2} a second time, and also compute U_{n-3}.

   We can avoid redundant recomputation by keeping around "old" results just long enough to avoid recomputing them using a technique resembling a shift register. You are to define a function `F` of type

   ```
   int -> int * int * int * int
   ```

   mapping natural numbers to four-tuples of natural numbers with the following property:

   for all $n \geq 0$, `F` n evaluates to (U_{n-1}, U_n, T_{n-1}, T_n).

   On the following page, give a definition of `F` by defining it for 0, 1, and $n \geq 2$ by filling in the given template.

```
fun F 0 =  (0, 0, 0, 1)

  | F 1 =  (0, 1, 1, 0)

  | F n =
    let
        val (a, b, c, d) = F (n-1)
    in
        (b, d+a, d, 2*b+c)
    end
```

3. [2 Points] Now re-define `T` and `U` in terms of `F`:

```
fun T n =  let (_, _, _, t) = F n in t end

fun U n =  let (_, u, _, _) = F n in u end
```

4. [10 Points] The correctness of `F` may be established by induction on the argument, $n \geq 0$. Complete the following outline of a proof of the correctness of your implementation of `F`.

   The base cases, $n = 0$ and $n = 1$, are proved by showing that the specification holds for the result (a_0, b_0, c_0, d_0) of evaluating `F(0)` and the result (a_1, b_1, c_1, d_1) of evaluating `F(1)`. Complete the following template by referring to your code and the defining equations for T and U:

Result		*Formula*		*Justification*
a_0	$=$	0	$=$	U_{-1}
b_0	$=$	0	$=$	U_0
c_0	$=$	0	$=$	T_{-1}
d_0	$=$	1	$=$	T_0

Result		*Formula*		*Justification*
a_1	$=$	0	$=$	U_0
b_1	$=$	1	$=$	U_1
c_1	$=$	1	$=$	T_0
d_1	$=$	0	$=$	T_1

   For the inductive step, assume that $n > 1$ and that `F`$(n-1)$ behaves according to specification. We are to show that `F`(n) behaves according to specification. Let (a, b, c, d) be the result of the recursive call to `F`$(n-1)$. The inductive hypothesis states that the following equations hold:

$$\begin{aligned} a &= U_{n-2} \\ b &= U_{n-1} \\ c &= T_{n-2} \\ d &= T_{n-1} \end{aligned}$$

   Let (a', b', c', d') be the result of evaluating `F`(n). Show that the implementation is correct by filling in the missing equations for a', b', c', and d' and providing a justification for each by consulting your code and the definitions of T and U:

$$
\begin{array}{llllll}
\textit{Result} & & \textit{Formula} & & \textit{Justification} \\
a' & = & b & = & U_{n-1} \\
b' & = & d + a & = & T_{n-1} + U_{n-2} \\
c' & = & d & = & T_{n-1} \\
d' & = & 2 * b + c & = & 2 * U_{n-1} + T_{n-2}
\end{array}
$$

3 Infinite Series [25 Points]

An *infinite series* is a polynomial with infinitely many terms:

$$a_0 + a_1\,x + a_2\,x^2 + a_3\,x^3 + \cdots,$$

where the coefficients a_n are real numbers.
Often an infinite series is written compactly as:

$$\sum_{n=0}^{\infty} a_n\,x^n$$

Many functions may be represented by infinite series. For instance:

- $e^x = 1 + x + x^2/2 + x^3/3! + x^4/4! + \cdots$
- $\sin x = x - x^3/3! + x^5/5! - x^7/7! + \cdots$

The coefficient sequence $a_0, a_1, a_2, \ldots$ completely defines an infinite series. For the exponential and sine functions these sequences are:

- e^x: 1.0, 1.0, 0.5, 0.1666666, 0.04166666, ...
- $\sin x$: 0.0, 1.0, 0.0, -0.1666666, 0.0, ...

Observe that any standard polynomial is its own infinite series, with almost all the coefficients being zero. For instance, the polynomial $p(x) = x^2 + 3$ has the coefficient sequence 3.0, 0.0, 1.0, 0.0, 0.0, 0.0, ...

In SML we will represent infinite series by functions from the natural numbers to the reals, just as we represented sequences of type `real seq` in lecture:

```
type nat = int                  (* natural numbers only *)
type series =  nat -> real
```

(Recall that the natural numbers are the nonnegative integers.)
For instance, we would represent the exponential function's infinite series by:

```
(* val exp : series *)
(* exp(n) should evaluate to the nth coefficient of the exponential
          series, i.e, to the real number 1/n!  *)

fun exp 0 = 1.0
  | exp n = exp(n-1)/real(n)
```

And the polynomial $p(x)$ described earlier would have the series

```
(* val p : series *)
(* p(n) = nth coefficient of the polynomial p(x) *)

fun p 0 = 3.0
  | p 2 = 1.0
  | p _ = 0.0
```

1. [5 Points] Two infinite series may be added term-by-term Specifically, the sum of two series $a_0, a_1, a_2, \ldots$ and $b_0, b_1, b_2, \ldots$ is a new series $c_0, c_1, c_2, \ldots$, with $c_n = a_n + b_n$, for $n = 0, 1, \ldots$. This is because:

$$\sum_{n=0}^{\infty} a_n\, x^n + \sum_{n=0}^{\infty} b_n\, x^n \;=\; \sum_{n=0}^{\infty} (a_n + b_n)\, x^n$$

Write a function `ADD : series * series -> series` that adds two series: (Note that `ADD` is a combinator of series.)

```
(* val ADD : series * series -> series *)
fun ADD (a:series, b:series) n = a(n) + b(n)
```

2. [10 Points] We can also define the product of two infinite series. The computation is slightly more complicated. Observe that if we multiply two infinite series, and then collect terms with like powers of x we get:

$$\left(\sum_{n=0}^{\infty} a_n\, x^n\right) * \left(\sum_{n=0}^{\infty} b_n\, x^n\right) \;=\; \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} a_k b_{n-k}\right)\, x^n$$

In other words, the product of two series $a_0, a_1, a_2, \ldots$ and $b_0, b_1, b_2, \ldots$ is a new series $c_0, c_1, c_2, \ldots$, with $c_n = a_0 b_n + a_1 b_{n-1} + \cdots + a_n b_0$.

Write a function `MULT : series * series -> series` that multiplies two series.

Hint: Internal to `MULT` the helper function `sum` has type `int * real -> real`. The role of `sum` is to compute c_n. Specifically,

$$\texttt{sum(k, acc)} \hookrightarrow \texttt{acc} + \sum_{i=k}^{n} \texttt{a(i)*b(n-i)}$$

and thus

$$\texttt{sum(0, 0.0)} \hookrightarrow \sum_{i=0}^{n} \texttt{a(i)*b(n-i)}$$

```
(* val MULT : series * series -> series *)
fun MULT (a:series, b:series) n =
    let
       fun sum (k, acc) =
            if k > n then acc
            else sum(k+1, acc + a(k)*b(n-k))
    in
       sum (0, 0.0)
    end
```

3. [5 Points] Now let us differentiate an infinite series term-by-term, then simplify. We get:

$$
\begin{aligned}
\frac{d}{dx}\left(\sum_{n=0}^{\infty} a_n\, x^n\right) &= \sum_{n=0}^{\infty} n * a_n\, x^{n-1} \\
&= \sum_{n=1}^{\infty} n * a_n\, x^{n-1} \\
&= \sum_{n=0}^{\infty} (n+1) * a_{n+1}\, x^n
\end{aligned}
$$

In other words, the derivative of a series $a_0, a_1, a_2, \ldots$ is the series $c_0, c_1, c_2, \ldots$, with $c_n = (n+1) * a_{n+1}$.

Write a function `DERIV : series -> series` that computes the derivative of an infinite series:

```
(* val DERIV : series -> series *)
fun DERIV (a:series) (n:nat) = real(n+1)*a(n+1)
```

4. [5 Points] Write a function `nthOrder : series * nat -> real list` that expects a series and returns a list of the first $n+1$ coefficients $a_0, \ldots, a_n$. (This list describes the coefficients of an *nth*-order polynomial that approximates the full series.)

   Specifically, `nthOrder(a, n)` should evaluate to `[a(0), a(1), ..., a(n)]`. In your code for `nthOrder` you should use the function `shift` given below. The effect of `shift` is to drop the first element of a series, i.e., $a_0, a_1, a_2, \ldots$ becomes $a_1, a_2, a_3, \ldots$.

```
(* val shift : series -> series *)
fun shift a n = a(n+1)

(* val nthOrder: series * nat -> real list *)
fun nthOrder (a:series, 0:nat) = [a 0]
  | nthOrder (a:series, n:nat) = (a 0)::nthOrder(shift a, n-1)
```

4 Coding Trees [25 Points]

In this question, you will build up a representation of a code and implement a decoder that takes an encoded message and returns a list of possible decoded messages.

A *code* is a mapping from *code-words* from one alphabet into another alphabet. Decoding means taking a list of characters from the input alphabet, and attempting to break up the list into chunks of contiguous code-words, and replacing each code-word with its mapping.

For example, suppose the input alphabet is $\{0, 1\}$ and the output alphabet is character strings. Then the following is a sample code:

```
00 -> "walk"
10 -> "i"
11 -> "money"
100 -> "tree"
110 -> "the"
```

The message

```
100011000
```

can be decoded only one way. The decoded message is

```
 10   00     110    00
["i","walk","the","walk"]
```

Not all codes are uniquely decipherable. For example:

```
01 -> "don't"
0  -> "go"
10 -> "now"
```

is not uniquely decipherable. The message

```
010
```

can be decoded in two different ways:

```
   01     0
["don't","go"]
  0     10
["go","now"]
```

Clearly, codes that are not uniquely decipherable can result in vastly different messages being extracted from an input message!

4.1 Data types

We will use the following data types:

```
type letter = bool

type plainWord = string

type codeWord = letter list

type message = letter list

datatype tree =
    Empty
  | Node of plainWord option * tree * tree
```

The meanings of these data types are:

- `letter` is a letter in the code alphabet. There are only two letters, represented pictorially in Figure 1 as `0` and `1`. In ML we will use `false` and `true`.
- `plainWord` is a string representing a decoded word. An example plain word in Figure 1 is `"the"`.
- `codeWord` is a list of code letters. For simplicity we use lists of code letters, rather than strings of code letters. As an example, the plain word `"the"` above is encoded by `110` which we represent internally by the list `[true true false]`.
- `message` is another list of code letters (booleans in our case). Such a list may represent an encoded message of several words.
- `tree` is just a binary tree. Each node in the tree consists of a `plainWord option` and two sub`trees`. Internal nodes may or may not have a plain word stored in them. If there is no plain word, then that field is simply `NONE`. If there is a plain word, it is stored, for example as `SOME("the")`.

  In contrast, leaf nodes have empty subtrees but *must* contain a plain word, stored again as above.

  Traversing the tree based on a code word entails going left or right depending on whether a code letter is `false` or `true`, respectively. In other words, visit the left subtree when seeing `false` and visit the right subtree when seeing `true`.

The code tree that you will need to use is intended to store sets of code-words in which many of the code-words share a common prefix. We minimize redundancy of storage by representing each code-word as a node element at the end of a path from the root determined by each successive letter of the code-word. For example, “10” and “100” share the common prefix “10”. See Figure 1 for an example of a code tree created by adding to an empty tree the words (one-by-one) of the code

```
00 -> "walk"
10 -> "i"
11 -> "money"
100 -> "tree"
110 -> "the"
```

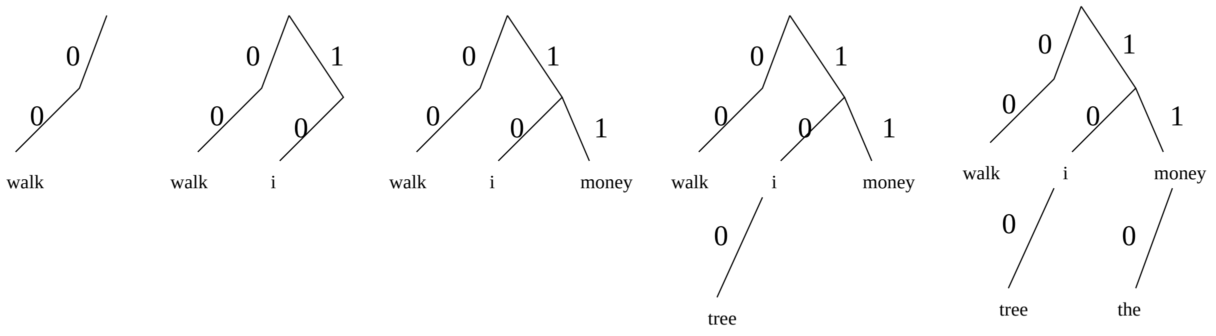


Figure 1: Example Code Tree (`Empty` subtrees are omitted)

4.2 Implementation

In this question, you will write functions that operate on a tree that has already been created.

4.2.1 find [10 Points]

The function `find` should decide whether a given `tree` contains a specified `codeWord`. We have provided an outline of the body of this function. Please fill in the remaining blanks.

```
(*
 * val find : tree * codeWord -> bool
 *
 * Determine whether a code-word is present in the tree.
 *)

fun find (Empty, _) = false
  | find (Node (NONE, _, _), []) = false
  | find (Node (SOME _, _, _), []) = true
  | find (Node (_, t1, _), false::ls) =
    find (t1, ls)
  | find (Node (_, _, t2), true::ls) =
    find (t2, ls)
```

4.2.2 decode [15 Points]

The function `decode` expects a code `tree` and a `message`. Recall that a `message` is just a list of booleans. The function `decode` should return a list of all possible decodings of that message. A single decoding is a list of plain words determined by traversing the tree repeatedly until all coding letters in the message have been exhausted. Multiple decodings are possible when such a traversal encounters a plain word in an internal node of the tree.

A valid decoding has been achieved if all the letters of a message have been consumed, and the tree traversal has stopped at a node indicating the last code-word recognized.

Note that if all the letters of a message have been consumed, but there is no valid code-word at the current location in the tree, the decoding has failed (but there may be other decodings that succeed by choosing different code-word boundaries while moving down the tree).

We have provided an exhaustive pattern matching for the function `decode`. Please fill in the remaining blanks.

The helper function `loop` takes the following arguments:

- `t' : tree`
- `m' : message`
- `ps : plainWord list`
- `acc : plainWord list list`

The spec: `ps` accumulates (in reverse order) the `plainWord`s decoded successively from `m'`, the portion of the original message `m` not yet consumed, while traversing `t'`, the portion of the tree `t` reached by consuming all the letters of `m` before `m'`.
`acc` accumulates a list of possible decodings. The return value of `loop` is the result of consing to the front of `acc` all the possible decodings of `m'` using `t'`, assuming that `ps` holds (in reverse order) the current attempted decoding.

```
(* val decode : tree * message -> plainWord list list
 *
 * Return a list of all possible decodings of the message, by
 * means of the code tree.
 *
 * There is no preferred order for the possible decodings
 * (each plainWord list is one decoding). *)

fun decode (t : tree, m : message) : plainWord list list =
    let
        (* val loop : tree
         *            * message
         *            * plainWord list
         *            * plainWord list list
         *            -> plainWord list list
         *)

        fun loop (Empty, _, _, acc) =
            acc
          | loop (Node (NONE, _, _), [], _, acc) =
            acc
          | loop (Node (NONE, t1, _), false::ls, ps, acc) =
            loop (t1, ls, ps, acc)
          | loop (Node (NONE, _, t2), true::ls, ps, acc) =
            loop (t2, ls, ps, acc)
          | loop (Node (SOME s, _, _), [], ps, acc) =
            (rev (s::ps))::acc
          | loop (Node (SOME s, t1, _), ll as false::ls, ps, acc) =
            loop (t1, ls, ps,
                  loop (t, ll, s::ps, acc))
          | loop (Node (SOME s, _, t2), ll as true::ls, ps, acc) =
            loop (t2, ls, ps,
                  loop (t, ll, s::ps, acc))

    in
        loop (t, m, [], [])
    end
```

5 Modules [25 Points]

Let us model sets by the following signature:

```
signature SET =
sig
    type elem              (* parameter *)
    type set               (* abstract  *)

    val emptyset : set
    val void : set -> bool

    val insert : elem * set -> set
    val isin : elem * set -> bool

    val union : set * set -> set
    val diff : set * set -> set
end;
```

The meanings of the individual components of the signature are:

- `elem` is a parameter type describing the type of elements in the sets.
- `set` is an abstract type used to implement sets.
- `emptyset` is the empty set.
- `void` decides whether a set is the empty set.
- `insert(`x`,`A`)` returns a new set consisting of x and all elements in A.
- `isin(`x`,`A`)` returns `true` if the element x is in the set A and `false` otherwise.
- `union(`A`, `B`)` is the union $A \bigcup B$ of the two sets A and B. The union consists of all elements that are either in set A or in set B or in both sets.
- `diff(`A`, `B`)` is the difference $A - B$ of the two sets A and B. The difference consists of all elements that are in set A but not in set B.

Suppose we have a structures `X : SET` representing sets of elements x. Let us now consider sets whose elements are themselves sets. For instance, $\{1, 2, 6\}$ and $\{0, 6, 10, 14\}$ are each sets of integers, while $\{\{1, 2, 6\}\}$ and $\{\{1, 2, 6\}, \{0, 6, 10, 14\}\}$ are each sets of sets of integers.

Your task is to write a functor `POWERSET` that takes the structure `X` as an argument. `POWERSET` should have signature `SET` and should represent sets of sets of elements x.

We have provided most of the code for one possible implementation of `POWERSET` below. Please fill in the remaining blanks.

NOTE: We use list implementations of sets, with possible duplicates. Don't worry about removing duplicates, such as in the function `union`.

HINT: Two sets A and B are equal precisely when $(A - B) \bigcup (B - A)$ is the empty set. You should use this hint in order to implement the internal helper function `equal`.

```
functor POWERSET(structure X : SET)
             :> SET where type elem = X.set
=
struct
    type elem = X.set
    type set = elem list
    val emptyset = []
    val void = List.null
    val insert = (op ::)
    (* val equal : elem * elem -> bool *)
    (* equal(a, b) returns true iff a and b represent the same set. *)
    fun equal(a, b) = X.void(X.union(X.diff(a, b), X.diff(b, a)))
    fun isin (_ , []) = false
      | isin (a, b::rest) =
          if equal(a, b) then true else isin(a, rest)
    val union = (op @)
    fun diff([], B) = []
      | diff(a::rest, B) = if isin(a, B) then diff(rest, B)
                           else a::(diff(rest, B))
end;
```

6 Parsing [30 Points]

Consider the following grammar:

```
S -> E;

E -> if B then E else E | n

B -> (E = E) | not B
```

Here are some typical expressions in the language of this grammar:

```
if (5=4) then 7 else 8;

if not (if (5=4) then 7 else 8 = 8) then 1 else 2;
```

The lexer will produce a token stream with tokens of the form:

```
IF, THEN, ELSE, EQUALS, NOT, LPAREN, RPAREN, NUM(n), SEMI
```

For example, the second expression above would produce the following token stream:

```
IF, NOT, LPAREN, IF, LPAREN, NUM(5), EQUALS, NUM(4), RPAREN,
THEN, NUM(7), ELSE, NUM(8), EQUALS, NUM(8), RPAREN, THEN,
NUM(1), ELSE, NUM(2), SEMI
```

The parser will parse the token stream and produce an expression stream. Expressions are specified by the following datatype:

```
(* Invariant:  Respect the constraints specified by the grammar *)

datatype exp = If of exp * exp * exp
             | Num of int
             | Equals of exp * exp
             | Not of exp
```

We have written a recursive-descent parser below, leaving some of the code blank. Please fill in the missing code.

```
(*
   val parse : token stream -> exp stream
   val parseStart : token front -> (exp * token stream) option
   val checkToken : token * (token front) -> token stream
   val parseExp : token front -> exp * token stream
   val parseBool : token front -> exp * token stream
*)
```

```
structure S = Stream;

exception ParseError of string;

(* The function parse expects a token stream.
   It parses the next full sentence S of the grammar, then
   conses the resulting exp onto the outgoing exp stream.
*)

fun parse s =
     S.delay (fn () => (case (parseStart (S.expose s)) of
                             NONE => S.Empty
                           | SOME(exp, s) => S.Cons(exp, parse s)))


(* parseStart parses one full sentence S of the grammar.
   It returns the exp produced by that parse, along with
   the rest of the token stream, all as an option.  *)

and parseStart (S.Empty) = NONE   (* entire token stream has been parsed *)
  | parseStart f = let
                      val (e, s) = parseExp f
                   in
                      SOME(e, checkToken(SEMI, (S.expose s)))
                   end


(* checkToken(t, f) checks whether the token at the front of the
   stream f is token t.
   If so, it returns the rest of the stream.
   If not, it raises an exception.  *)

and checkToken (t, S.Cons(t1, s)) =
       if t=t1 then s else raise ParseError "Unexpected token"
  | checkToken (t, _ )  = raise ParseError "Missing token"
```

[code continues on next page]

```
(* parseExp parses an expression E of the grammar.
   It returns the exp produced by that parse,
   along with the rest of the token stream. *)

and parseExp (S.Cons(NUM(n), s)) = (Num(n), s)
  | parseExp (S.Cons(IF, s)) =
      let
         val (b, s0) = parseBool (S.expose s)
         val (e1, s1) = parseExp (S.expose (checkToken(THEN, S.expose s0)))
         val (e2, s2) = parseExp (S.expose (checkToken(ELSE, S.expose s1)))
      in
         (If(b, e1, e2), s2)
      end
  | parseExp _  = raise ParseError "Improper expression syntax"

(* parseBool parses an expression B of the grammar.
   It returns the exp produced by that parse,
   along with the rest of the token stream. *)

and parseBool (S.Cons(NOT, s)) =
      let
         val (b, s1) = parseBool (S.expose s)
      in
         (Not(b), s1)
      end
  | parseBool (S.Cons(LPAREN, s)) =
      let
         val (e1, s1) = parseExp (S.expose s)
         val (e2, s2) = parseExp (S.expose (checkToken(EQUALS, S.expose s1)))
      in
         (Equals(e1, e2), checkToken(RPAREN, (S.expose s2)))
      end
  | parseBool _  = raise ParseError "Improper boolean expression syntax"
```

Worksheet

Worksheet

Worksheet