1 Part 1

1. Problem 5.4

   Solution:

   - ```
     fun mapTree f (LEAF x)        = LEAF (f x)
     |   mapTree f (NODE (l, r)) = NODE (mapTree f l, mapTree f r)
     ```

   - The right type is :

     ```
     ('a -> 'b) -> 'a tree -> 'b tree
     ```

     The given type:

     ```
     ('a -> 'a) -> 'a tree -> 'a tree
     ```

     assumes that the passed function returns the same type it takes as its argument, but nothing in the description restricts that to be the case. □

2. Problem 5.5 *Solution:*

   ```
   fun reduce f (LEAF x)        = x
     | reduce f (NODE (l, r)) = f (reduce f l, reduce f r)
   ```

   We treat a tree as instructions for where to apply the combining function. So, a leaf just reduces to itself while a node reduces to the combination of the reduction of its sub trees. □

3. Problem 5.6

   Solution:

   ```
   fun curry f = fn x => fn y => f (x,y)
   fun uncurry f = fn (x,y) => f x y
   ```

   or

   ```
   fun curry f x y = f (x,y)
   fun uncurry f (x,y) = f x y

   curry (uncurry f) x y
   = curry (fn (x,y) => f x y) x y
   = (fn (x,y) => f x y) (x,y)
   = f x y
   ```

   and
   ```

```
uncurry (curry f) (x,y)
= uncurry (fn x => fn y => f (x,y)) (x,y)
= (fn x => fn y => f (x,y)) (x,y)
= f (x,y)
```

□

4. Problem 5.7

   - *Solution:* The program might not work as expected, because, if for example, the branch of the function evaluates to false then x would have the value of a string (that is, a pointer) and so (x.i)+5 would bitwise interpret a pointer to an int and then add 5 which is unlikely to make sense and might explode. The runtime system generally wont detect the problem because C types are gone at runtime. □

   - *Solution:* The bug can not occur. Instead, in that situation the system will generate a runtime error. SML/NJ does not generate a compile time warning because it assumes that if you pattern match in a val declaration you know what you are doing (it would generate a warning in a function definition or case expression), however, a different implementation could generate a warning here as it is obvious from the types that the match is non exhaustive. □

5. Problem 5.8 (a), (b)

   Solution:

```
fun merge (Nil, ys) = ys
|   merge (xs, Nil) = ys
|   merge (Cons (x,xs), Cons (y,ys)) =
Cons (x, fn () => Cons (y, fn () => merge (xs (), ys ())))
```

   or

```
fun merge (Nil, ys) = ys
|   merge (xs, Nil) = ys
|   merge (xs, ys) =
Cons (head x,
      fn () => Cons (head y, fn () => merge (tail xs, tail ys)))
```

   or

```
fun merge (Nil, ys) = ys
|   merge (Cons (x,xs), ys) = Cons (x, fn () => merge (ys, xs ()))

fun compose (f,g) =
let
```

```
val (x,fx) = head f
in
Cons ((x, apply(g,fx)), fn () => compose (tail f, g))
end
```

□

6. Problem 6.1

   - *Solution:*

     ```
     a : int * int -> int
     ```

     □

   - *Solution:*

     ```
     b : real * real -> real
     ```

     □

   - *Solution:*

     ```
     c : ('a -> 'b) -> 'a -> 'b
     ```

     □

   - *Solution:*

     ```
     d : ('a -> 'a) * 'a -> 'a
     ```

     □

   - *Solution:*

     ```
     e : 'a * 'a * ('a -> bool) -> 'a
     ```

     □

7. Problem 6.2 *Solution:* `sort` : $(\alpha \times \alpha \rightarrow$ `bool`$) \times \alpha$ `list` $\rightarrow \alpha$ `list`. The inner function `insert` takes an element and a list and returns a list, so its type must be $\alpha \times \alpha$ `list` $\rightarrow \alpha$ `list`. Then, `sort` takes a comparison function and a list, then returns a list of the same type. The comparison function takes a pair of elements of the list and returns a `bool` (as required by "`if less(a,b) then ...`"), so its type must be the above. □

2 Part 2

You will understand and complete a type checker for a *statically typed language*. We assume function types are provided by the user (this is the approach of languages like C++ and Java).

Here is the syntax of the language we work with:

```
e ::= x | n | true | false | iszero | succ | pred
    | if e then e else e | fn x : t => e | e e | (e)
t ::= 'a | int | bool | t -> t
```

Here 'a is a *type variable*, used for polymorphic types. As in SML, we'll assume that $\rightarrow$ associates to the right.

In the above grammar, x stands for an identifier; n stands for a non-negative integer literal; true and false are the boolean literals; succ and pred are unary functions that add 1 and subtract 1 from their input, respectively; iszero is a unary function that returns true if its argument is 0 and false otherwise; if e1 then e2 else e3 is a conditional expression; $fn\ x : t \Rightarrow e$ is a function with parameter x and body e; e e is a function application; (e) is to control parsing. It should be clear to you that the above grammar is quite ambiguous. For example, should $fn\ f : t \Rightarrow f\ f$ be parsed as $fn\ f : t \Rightarrow (f\ f)$ or as $(fn\ f : t \Rightarrow f)\ f$? We can resolve such ambiguities by adopting the following conventions (which are the same as in SML):

- Function application associates to the left. For example, e f g is (e f) g, not e (f g).
- Function application binds tighter than if, and fn. For example, $fn\ f : t \Rightarrow f\ 0$ is $fn\ f : t \Rightarrow (f\ 0)$, not $(fn\ f : t \Rightarrow f)\ 0$.

The types will be represented by this SML type:

```
datatype typ = VAR of string | INT | BOOL | ARROW of typ * typ
```

The abstract syntax trees (ASTs) representing this language will have the following SML type:

```
datatype term = AST_ID of string | AST_NUM of int
   | AST_BOOL of bool
   | AST_FUN of (string * typ * term)
   | AST_APP of (term * term)
   | AST_SUCC | AST_PRED | AST_ISZERO
   | AST_IF of (term * term * term)
```

As with most ASTs, this datatype does not represent parentheses as they appear in the source language; in other words, e and (e) have the same representation.

Environments

As it goes, the type checker must remember the types of the variables that have been declared. It does so by storing the types in a *type environment*, which is a mapping from names to types. This environment must be extendable to allow new variables to be bound, and it must be searchable, to allow the types of bound variables to be later retrieved. For example, consider this term:

```
fn x : int => fn y : bool => x
```

We would start with an empty environment (). When we come to the outer function, we would add the relation (x,int) to our environment, and evaluate the body of the function in that context. Similarly, when we come to the inner function, we add (y,bool) to our environment, and evaluate its body in the further extended environment. Thus, when we come to the expression x, we check it in the environment ((x,int), (y,bool)). In order to determine the type of x, we merely look it up.

1. Write down ML expressions of type typ corresponding to the abstract syntax trees for each of the following type expressions:

   (a) `int`

   Solution:

   ```
   INT
   ```

   □

   (b) `int -> bool`

   Solution:

   ```
   ARROW(INT, BOOL)
   ```

   □

   (c) `('a -> 'b) -> ('a -> 'b)`

   Solution:

   ```
   ARROW(ARROW(VAR "a", VAR "B"), ARROW(VAR "a", VAR "b"))
   ```

   □

2. Write down ML expressions of type term corresponding to the abstract syntax trees for each of the following expressions

   (a) `succ (pred 5)`

   Solution:

   ```
   AST_APP(AST_SUCC, AST_APP(AST_PRED, AST_NUM 5))
   ```

   □

   (b) `if 7 then true else 5`

   Solution:

   ```
   AST_IF(AST_NUM 7, AST_BOOL true, AST_NUM 5)
   ```

   □

   (c) `fn a : int => f a a`

   Solution:

   ```
   AST_FUN("a", INT, AST_APP(AST_ID "f", AST_ID "a"), AST_ID "a"))
   ```

   □

3. Typing is done with respect to an environment E, which is a mapping from identifiers to types; the environment gives the types of any free identifiers in the expression. Think of these as the variables that are defined somewhere higher up in the program.

   We express this using the following notation

   ```
   E |- e : t
   ```

   which can be read "from environment E, it follows that expression e has type t".

   Next we describe the actual typing rules.

   (a) A variable has the type the environment has stored for it.

   ```
                E(x) = t
   (ID)        ----------
               E |- x : t
   ```

   (b) An integer literal has type int and a Boolean literal has type bool.

```
(NUM)       E |- n : int

(TRUE)      E |- true : bool

(FALSE)     E |- false : bool
```

(c) The built-in succ and pred functions have type $int \rightarrow int$ and the built-in *iszero* function has type $int \rightarrow bool$.

```
(SUCC)       E |- succ : int  ->  int

(PRED)       E |- pred : int -> int

(ISZERO)     E |- iszero :  int -> bool
```

(d) In an if-then-else, the condition must have type bool and the branches have to have the same type (but this can be any type).

```
            E |- e1 : bool      E |- e2 : t       E |- e3 : t
(IF)    ---------------------------------------------------
                    E |- if e1 then e2 else e3 : t
```

(e) For a function $fn\ x : t1 \Rightarrow e$, if e has the type t2 when we extend the environment by mapping x to t1, then the function has type $t1 \rightarrow t2$.

```
                     E[x : t1] |- e : t2
(-> INTRO) --------------------------------
              E |- fn x : t1 => e : t1 -> t2
```

(f) The ($\rightarrow$ INTRO) rule uses the notation E[x : t1] to denote an *updated environment* that is the same as E except that it maps x to t1.

When we apply a function e1 to an argument e2, the function must have type $t1 \rightarrow t2$ for some t1 and t2; then the argument must have type t1 and the application as a whole has type t2.

```
             E |- e1 : t1 -> t2       E |- e2 : t1
(-> ELIM) ---------------------------------------
                       E |- e1 e2 : t2
```

Given these rules, we can write a function $typeOf\ : term * env \rightarrow typ$ which takes an environment and a term, and returns its type (if the term is well typed) or an error (if it isn't). Most of the implementation is in: *typechecker.sml* . It uses a type datatype and environment implementation defined in *type.sml* . Fill in the "*raise Unimplimented*" slots in the code. (Both of these files can be found under files on canvas)

Solution:

```
fun typeOf env (AST_ID s)          = env s
  | typeOf env (AST_NUM n)         = INT
  | typeOf env (AST_BOOL b)        = BOOL
  | typeOf env (AST_FUN (i,t,e))   = ARROW (t,typeOf (update env i t) e)
  | typeOf env (AST_APP (e1,e2))   =
    (case typeOf env e1 of
         ARROW (t1,t2) =>
         (case t1 = typeOf env e2 of
              true => t2
            | false => raise TypeError)
       | _ => raise TypeError)
  | typeOf env AST_SUCC            = ARROW (INT,INT)
  | typeOf env AST_PRED            = ARROW (INT,INT)
  | typeOf env AST_ISZERO          = ARROW (INT,BOOL)
  | typeOf env (AST_IF (e1,e2,e3)) =
    let val t1 = typeOf env e1
        val t2 = typeOf env e2
        val t3 = typeOf env e3
    in
        case (t1 = BOOL) andalso (t2 = t3) of
            true => t3
          | false => raise TypeError
    end
```

□