1 Type checking

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 | let x = e in 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. As already pointed out in the previous assignment, 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_LET of (string*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`

   (b) `int -> bool`

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

   (b) `if 7 then true else 5`

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

3. Typing is done with respect to an environment `env`, 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

```
env |- e : t
```

which can be read "from environment env, 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.

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

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

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

(TRUE)      env |- true : bool

(FALSE)     env |- 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)       env |- succ : int  ->  int

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

(ISZERO)     env |- 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).

```
            env |- e1 : bool      env |- e2 : t       env |- e3 : t
(IF)    ---------------------------------------------------------
                    env |- 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 $t1 \rightarrow t2$.

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

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

(f) 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.

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

(g) Even though let expressions are syntactic sugar for function application, we treated them directly using the rule below

```
           env |- e1 : t1       env[x : t1] |- e2 : t2
  (-> LET) ----------------------------------------------
                 env |- let x=e1 in 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). Complete the implementation in: typechecker.sml. It uses a datatype and environment implementation defined in type.sml.

2 Type inference

Derive the types for the following expressions:

1. `fun f (g,h) = g (h 0)`
2. `fun apply (f,x) = f x`
3. ```
   fun reverse nil       = nil
   |     reverse (x::xs) =  reverse xs
   ```
4. `fun f(g,h) = g h + 2`
5. `fun f g = g(g) + 2`
6. `fun ff f x y = if (f x y) then (f 3 y) else (f x "zero")`
7. `fun gg f x y = if (f x y) then (f 3 y) else (f y "zero")`
8. `fun hh f x y = if (f x y) then (f x y) else (f x "zero")`
   ```