Computer Science 131, Fall 2000
Assignment 3: Manipulating NQSML Syntax

Out: Friday, September 22

Due: Friday, September 29, 5:00pm

To complete this assignment, retrieve the files `assign3.sml` and `assign3-tests.sml` from the Assignments web page. You must then run SML/NJ on turing using the command `/cs/cs131/bin/sml-cs131-a3`. This is a version of the compiler which has some important code predefined.

Your task is to complete the functions in the `assign3.sml` file and turn this in. As always, your file should contain no syntax or type errors. The submission process is the same as for Assignment 1; when you are ready, run the command

```
cs131submit assign3.sml
```

Several of the problems ask you to write code that may raise the `Error` exception. To throw this exception, the SML code is simply "`raise Error`". You will not need to worry about catching exceptions in this assignment.

The extended NQSML abstract syntax is given in Figure 1. It is the language as in the NQSML handout extended with pairs (exactly as in the previous assignment) and lists (new for this assignment). As in SML, for any type t we have the type t `list` which is the type of lists whose elements all have type t. The expression e_1 :: e_2 represents the list whose head (first element) is e_1 and whose tail (the list of remaining elements) is e_2. Since this language contains no polymorphism, the empty list `nil[`t`]` is marked with the type of the elements that this list would contain if it weren't empty. For example, `nil[Int]` is the empty list of integers, while `nil[Bool->Bool]` is the empty list of functions which map booleans to booleans. (The type annotation is necessary in order to ensure that every expression in the language has a unique type, which makes typechecking much easier.) Finally we have a case-analysis for list values, `case` e_1 `of nil =>` e_2 `|` x::y `=>` e_3. Note that unlike SML this case statement must have *exactly* two cases, and that the non-nil case *must* be a pattern of the form x :: y where x and y are variables. (More complicated patterns can be encoded using this construct, however.) The variables x and y are then bound variables with e_3 as their scope.

All of your programs will require using the following representation for NQSML programs. (These types have been pre-defined for you in the `sml-cs131-a3` binary. You will *not* need to type this in as part of your `assign3.sml` file.)

$$
\begin{array}{rcl}
v & ::= & n \\
& | & \overline{\texttt{tt}} \\
& | & \overline{\texttt{ff}} \\
& | & \texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2 \\
& | & \langle v_1, v_2 \rangle \\
& | & \texttt{nil[}t\texttt{]} \\
& | & v_1\ ::\ v_2 \\
\\
e & ::= & v \\
& | & x \\
& | & e_1 + e_2 \\
& | & e_1 < e_2 \\
& | & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
& | & \texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2 \\
& | & e_1\, e_2 \\
& | & \langle e_1, e_2 \rangle \\
& | & e.1 \\
& | & e.2 \\
& | & e_1\ ::\ e_2 \\
& | & \texttt{case}\ e_1\ \texttt{of nil =>}\ e_2\ \texttt{|}\ x{::}y\ \texttt{=>}\ e_3 \\
\\
t & ::= & \texttt{Int} \\
& | & \texttt{Bool} \\
& | & t_1\texttt{->}t_2 \\
& | & t_1 \times t_2 \\
& | & t\ \texttt{list}
\end{array}
$$

Figure 1: Abstract Syntax for This Assignment

```
type varname = string
datatype ty  = Int_t
             | Bool_t
             | Arrow_t of ty * ty
             | Times_t of ty * ty
             | List_t of ty
datatype aop   = Plus | Minus | Times
datatype cop   = Less | Equal
datatype exp   = Num of int
               | Bool of bool
               | Var of varname
               | Arith of exp * aop * exp
               | Compare of exp * cop * exp
               | If of exp * exp * exp
               | Let of varname * exp * exp
               | Fun of varname * varname * ty * ty * exp
               | Apply of exp * exp
               | Pair of exp * exp
               | Proj1 of exp
               | Proj2 of exp
               | Nil of ty
               | Cons of exp * exp
               | Listcase of exp * exp * (varname * varname * exp)
```

This datatype should be mostly self-explanatory. You can also look at some of the predefined expressions in the `assign3-tests.sml` file to see their abstract syntax and the SML representations of this abstract syntax. Alternatively, Figure 2 gives a formal definition of the translation between abstract syntax into the SML representation. The notation $\lceil e \rceil$ will be used to denote the SML representation corresponding to the abstract syntax for e, while $\lceil t \rceil$ will be used to denote the SML representation of the type t.

The `sml-cs131-a3` binary also includes the type `'a env` (environments associating values of type `'a` with variable names) and the following items:

```
empty  : 'a env
extend : 'a env * varname * 'a -> 'a env
lookup : 'a env * varname -> 'a
```

These act exactly like the implementations shown in a previous class, except that instead of `lookup` returning either `NONE` or `SOME` it raises the `Error` exception when the variable is not found in the environment.

Finally, the `sml-cs131-a3` binary supplies the functions

```
ppty  : ty -> unit
ppexp : exp -> unit
pptyenv : ty env -> unit
ppexpenv : exp env -> unit
```

$\lceil$`Int`$\rceil$	= `Int_t`
$\lceil$`Bool`$\rceil$	= `Bool_t`
$\lceil t_1$`->`$t_2\rceil$	= `Arrow_t(`$\lceil t_1\rceil$`,`$\lceil t_2\rceil$`)`
$\lceil t_1 \times t_2\rceil$	= `Times_t(`$\lceil t_1\rceil$`,`$\lceil t_2\rceil$`)`
$\lceil t$ `list`$\rceil$	= `List_t(`$\lceil t\rceil$`)`
$\lceil\overline{m}\rceil$	= `Num(`m`)`
$\lceil\overline{\texttt{tt}}\rceil$	= `Bool(true)`
$\lceil\overline{\texttt{ff}}\rceil$	= `Bool(false)`
$\lceil$`x`$\rceil$	= `Var("x")`
$\lceil e_1 + e_2\rceil$	= `Arith(`$\lceil e_1\rceil$`,Plus,`$\lceil e_2\rceil$`)`
$\lceil e_1 - e_2\rceil$	= `Arith(`$\lceil e_1\rceil$`,Minus,`$\lceil e_2\rceil$`)`
$\lceil e_1 * e_2\rceil$	= `Arith(`$\lceil e_1\rceil$`,Times,`$\lceil e_2\rceil$`)`
$\lceil e_1 < e_2\rceil$	= `Compare(`$\lceil e_1\rceil$`,Less,`$\lceil e_2\rceil$`)`
$\lceil e_1 = e_2\rceil$	= `Compare(`$\lceil e_1\rceil$`,Equal,`$\lceil e_2\rceil$`)`
$\lceil$`if` e_1 `then` e_2 `else` $e_3\rceil$	= `If(`$\lceil e_1\rceil$`,`$\lceil e_2\rceil$`,`$\lceil e_3\rceil$`)`
$\lceil$`let x be` e_1 `in` $e_2\rceil$	= `Let("x",`$\lceil e_1\rceil$`,`$\lceil e_2\rceil$`)`
$\lceil$`fun f(x:`t_1`):`t_2 `is` $e_2\rceil$	= `Fun("f","x",`$\lceil t_1\rceil$`,`$\lceil t_2\rceil$`,`$\lceil e_2\rceil$`)`
$\lceil e_1\, e_2\rceil$	= `Apply(`$\lceil e_1\rceil$`,`$\lceil e_2\rceil$`)`
$\lceil\langle e_1, e_2\rangle\rceil$	= `Pair(`$\lceil e_1\rceil$`,`$\lceil e_2\rceil$`)`
$\lceil e.1\rceil$	= `Proj1(`$\lceil e\rceil$`)`
$\lceil e.2\rceil$	= `Proj2(`$\lceil e\rceil$`)`
$\lceil$`nil[`t`]`$\rceil$	= `Nil(`$\lceil t\rceil$`)`
$\lceil e_1$ :: $e_2\rceil$	= `Cons(`$\lceil e_1\rceil$`,`$\lceil e_2\rceil$`)`
$\lceil$`case` e_1 `of nil =>` e_2 `| x::y =>` $e_3\rceil$	= `Listcase(`$\lceil e_1\rceil$`,`$\lceil e_2\rceil$`,("x","y",`$\lceil e_3\rceil$`))`

Figure 2: SML Representations of Abstract Syntax

which print types, expressions, type environments, and value environments respectively. You may find these useful for debugging. (For example, while debugging my solution to #4 I temporarily changed the code

```
fun evalDyn (env,e) =
   (case e of
       Num _ => e
```

to

```
fun evalDyn (env,e) =
   (ppexp e;
    case e of
       Num _ => e
```

so that it would print a trace of every call to `evalDyn`.)

1 Typechecking

The typing rules for the extended language are as in the NQSML handout, plus the typing rules for pairs and projections given in Assignment 2, plus new rules for the list-related expressions. The new rules are as follows:

$$\frac{}{\Gamma \vdash \texttt{nil}[t] : t\ \texttt{list}} \tag{1}$$

$$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t\ \texttt{list}}{\Gamma \vdash (e_1\ ::\ e_2) : t\ \texttt{list}} \tag{2}$$

$$\frac{\Gamma \vdash e_1 : u\ \texttt{list} \qquad \Gamma \vdash e_2 : t \qquad \Gamma, x{:}u, y{:}(u\ \texttt{list}) \vdash e_3 : t}{\Gamma \vdash \texttt{case}\ e_1\ \texttt{of nil =>}\ e_2\ \texttt{|}\ x{::}y\ \texttt{=>}\ e_3 : t} \tag{3}$$

Define the function

```
typeof : (ty env) * exp -> ty
```

which given an environment associating variable names with types and an expression, returns the type of the expression if it has one and otherwise raises the `Error` exception. (This exception has been predefined for you.) Remember that for expressions like $e_1 + e_2$ even though you immediately know if this has a type it must be `Int`, you still have to check that e_1 and e_2 are well-typed with type `Int`.

Several of the cases for this function have been given to you in the file `assign3.sml`. Fix the function so that it returns the correct answer for the remaining cases instead of raising the `Unimplemented` exception.

$n[x{\mapsto}v] = n$

$\overline{\texttt{tt}}[x{\mapsto}v] = \overline{\texttt{tt}}$

$\overline{\texttt{ff}}[x{\mapsto}v] = \overline{\texttt{ff}}$

$(e_1 + e_2)[x{\mapsto}v] = (e_1[x{\mapsto}v]) + (e_2[x{\mapsto}v])$

$(e_1 < e_2)[x{\mapsto}v] = (e_1[x{\mapsto}v]) < (e_2[x{\mapsto}v])$

$(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)[x{\mapsto}v] = \texttt{if}\ (e_1[x{\mapsto}v])\ \texttt{then}\ (e_2[x{\mapsto}v])\ \texttt{else}\ (e_3[x{\mapsto}v])$

$x[x{\mapsto}v] = v$

$y[x{\mapsto}v] = y \qquad \text{if } x \neq y$

$(\texttt{let}\ y\ \texttt{be}\ e_1\ \texttt{in}\ e_2)[x{\mapsto}v]$

$= \texttt{let}\ y\ \texttt{be}\ e_1[x{\mapsto}v]\ \texttt{in}\ e_2[x{\mapsto}v] \quad \text{if } x \neq y$

$= \texttt{let}\ y\ \texttt{be}\ e_1\ \texttt{in}\ e_2 \quad \text{otherwise}$

$(\texttt{fun}\ f(y{:}t_1){:}t_2\ \texttt{is}\ e_2)[x{\mapsto}v]$

$= \texttt{fun}\ f(y{:}t_1){:}t_2\ \texttt{is}\ (e_2[x{\mapsto}v]) \quad \text{if } x \neq f \text{ and } x \neq y$

$= \texttt{fun}\ f(y{:}t_1){:}t_2\ \texttt{is}\ e_2 \quad \text{otherwise}$

$(e_1\, e_2)[x{\mapsto}v] = (e_1[x{\mapsto}v])\,(e_2[x{\mapsto}v])$

$(\langle e_1, e_2\rangle)[x{\mapsto}v] = \langle e_1[x{\mapsto}v], e_2[x{\mapsto}v]\rangle$

$(e.1)[x{\mapsto}v] = (e[x{\mapsto}v]).1$

$(e.2)[x{\mapsto}v] = (e[x{\mapsto}v]).2$

$\texttt{nil[}t\texttt{]}[x{\mapsto}v] = \texttt{nil[}t\texttt{]}$

$(e_1 :: e_2)[x{\mapsto}v] = (e_1[x{\mapsto}v]) :: (e_2[x{\mapsto}v])$

$(\texttt{case}\ e_1\ \texttt{of nil =>}\ e_2\ \texttt{|}\ y{::}z\ \texttt{=>}\ e_3)[x{\mapsto}v]$

$= \texttt{case}\ (e_1[x{\mapsto}v])\ \texttt{of nil =>}\ (e_2[x{\mapsto}v])\ \texttt{|}\ y{::}z\ \texttt{=>}\ (e_3[x{\mapsto}v]) \quad \text{if } x \neq y \text{ and } x \neq z$

$= \texttt{case}\ (e_1[x{\mapsto}v])\ \texttt{of nil =>}\ (e_2[x{\mapsto}v])\ \texttt{|}\ y{::}z\ \texttt{=>}\ e_3 \quad \text{otherwise}$

Figure 3: Substitution of Closed Values

2 Substitution

Define the function

```
val substClosed : exp * var * exp -> exp
```

by recursion on the first expression such that `substClosed(`$\lceil e \rceil$`, "x",` $\lceil v \rceil$`)` returns the abstract syntax for the term $e[\mathtt{x} \mapsto v]$. *You may assume that the argument v is closed (has no free variables).* This makes the substitution code simpler because there is no possibility of variable capture. Figure 3 contains the definition of substitution in the extended syntax, simplified by the assumption that the value v contains no free variables.

The file `assign3.sml` contains the code for several cases of this function. Fix this code so that it returns the correct answer for the remaining cases instead of raising the `Unimplemented` exception.

3 A Simple Evaluator

In the previous assignment you were given a big-step (or "natural") semantics for the NQSML language. Here are the rules again, along with the rules for the extensions to the language.

$$\frac{}{v \Downarrow v} \tag{4}$$

$$\frac{e_1 \Downarrow \overline{\mathtt{m}_1} \qquad e_2 \Downarrow \overline{\mathtt{m}_2}}{e_1 + e_2 \Downarrow \overline{\mathtt{m}_1 + \mathtt{m}_2}} \tag{5}$$

$$\frac{e_1 \Downarrow \overline{\mathtt{m}_1} \qquad e_2 \Downarrow \overline{\mathtt{m}_2}}{e_1 - e_2 \Downarrow \overline{\mathtt{m}_1 - \mathtt{m}_2}} \tag{6}$$

$$\frac{e_1 \Downarrow \overline{\mathtt{m}_1} \qquad e_2 \Downarrow \overline{\mathtt{m}_2}}{e_1 * e_2 \Downarrow \overline{\mathtt{m}_1 * \mathtt{m}_2}} \tag{7}$$

$$\frac{e_1 \Downarrow \overline{\mathtt{m}_1} \qquad e_2 \Downarrow \overline{\mathtt{m}_2}}{e_1 < e_2 \Downarrow \overline{\mathtt{m}_1 < \mathtt{m}_2}} \tag{8}$$

$$\frac{e_1 \Downarrow \overline{\mathtt{m}_1} \qquad e_2 \Downarrow \overline{\mathtt{m}_2}}{e_1 = e_2 \Downarrow \overline{\mathtt{m}_1 = \mathtt{m}_2}} \tag{9}$$

$$\frac{e_1 \Downarrow \overline{\mathtt{tt}} \qquad e_2 \Downarrow v_2}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Downarrow v_2} \tag{10}$$

$$\frac{e_1 \Downarrow \overline{\mathtt{ff}} \qquad e_3 \Downarrow v_3}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Downarrow v_3} \tag{11}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2[x \mapsto v_1] \Downarrow v_2}{\mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2 \Downarrow v_2} \tag{12}$$

$$\frac{e_1 \Downarrow (\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_1') \qquad e_2 \Downarrow v_2 \qquad (e_1'[x{\mapsto}v_2])[f{\mapsto}(\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_1')] \Downarrow v}{e_1\, e_2 \Downarrow v} \tag{13}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\langle e_1, e_2\rangle \Downarrow \langle v_1, v_2\rangle} \tag{14}$$

$$\frac{e \Downarrow \langle v_1, v_2\rangle}{e.1 \Downarrow v_1} \tag{15}$$

$$\frac{e \Downarrow \langle v_1, v_2\rangle}{e.2 \Downarrow v_2} \tag{16}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2} \tag{17}$$

$$\frac{e_1 \Downarrow \texttt{nil[}t\texttt{]} \qquad e_2 \Downarrow v_2}{(\texttt{case}\ e_1\ \texttt{of nil =>}\ e_2\ \texttt{|}\ x{::}y\ \texttt{=>}\ e_3) \Downarrow v_2} \tag{18}$$

$$\frac{e_1 \Downarrow (v_1 :: v_2) \qquad e_3[x{\mapsto}v_1][y{\mapsto}v_2] \Downarrow v_3}{(\texttt{case}\ e_1\ \texttt{of nil =>}\ e_2\ \texttt{|}\ x{::}y\ \texttt{=>}\ e_3) \Downarrow v_3} \tag{19}$$

Define the function

```
val evalSimple : exp -> exp
```

such that `evalSimple(`$\lceil e \rceil$`)` returns $\lceil v \rceil$ when $e \Downarrow v$, and raises the `Error` exception if the expression cannot be evaluated. Your code should have essentially the same form as the natural semantics. For example, to evaluate the expression `let` x `be` e_1 `in` e_2, we recursively evaluate e_1 to get a v_1, substitute v_1 into e_2 (using `substClosed`) and return the result of evaluating the substituted expression.

The file `assign3.sml` contains the code for several cases of this function. Fix this code so that it returns the correct answer for the remaining cases instead of raising the `Unimplemented` exception.

4 Environments and Dynamic Scope

As discussed in class, a more efficient approach to implementation is to avoid substitution by simply remembering the current values of variables (that is, by maintaining an environment which maps variable names to values). This can also be modeled directly with a natural semantics, as described in class. The relation $(\eta, e) \Downarrow v$ defined by the following inference rules can be read as "e evaluates to v if the values of its free variables are given by the environment η". The notation $\eta, x{=}v$ stands for the environment η extended by associating the value v with the variable x. The notation $\eta(x)$ stands for the value which the environment η contains for the variable x.

$$\frac{}{(\eta, v) \;\Downarrow\; v} \tag{20}$$

$$\frac{}{(\eta, x) \;\Downarrow\; \eta(x)} \tag{21}$$

$$\frac{(\eta, e_1) \;\Downarrow\; \overline{\mathtt{m}_1} \qquad (\eta, e_2) \;\Downarrow\; \overline{\mathtt{m}_2}}{(\eta, e_1 + e_2) \;\Downarrow\; \overline{\mathtt{m}_1 + \mathtt{m}_2}} \tag{22}$$

$$\frac{(\eta, e_1) \;\Downarrow\; \overline{\mathtt{m}_1} \qquad (\eta, e_2) \;\Downarrow\; \overline{\mathtt{m}_2}}{(\eta, e_1 - e_2) \;\Downarrow\; \overline{\mathtt{m}_1 - \mathtt{m}_2}} \tag{23}$$

$$\frac{(\eta, e_1) \;\Downarrow\; \overline{\mathtt{m}_1} \qquad (\eta, e_2) \;\Downarrow\; \overline{\mathtt{m}_2}}{(\eta, e_1 * e_2) \;\Downarrow\; \overline{\mathtt{m}_1 * \mathtt{m}_2}} \tag{24}$$

$$\frac{(\eta, e_1) \;\Downarrow\; \overline{\mathtt{m}_1} \qquad (\eta, e_2) \;\Downarrow\; \overline{\mathtt{m}_2}}{(\eta, e_1 < e_2) \;\Downarrow\; \overline{\mathtt{m}_1 < \mathtt{m}_2}} \tag{25}$$

$$\frac{(\eta, e_1) \;\Downarrow\; \overline{\mathtt{m}_1} \qquad (\eta, e_2) \;\Downarrow\; \overline{\mathtt{m}_2}}{(\eta, e_1 = e_2) \;\Downarrow\; \overline{\mathtt{m}_1 = \mathtt{m}_2}} \tag{26}$$

$$\frac{(\eta, e_1) \;\Downarrow\; \overline{\mathtt{tt}} \qquad (\eta, e_2) \;\Downarrow\; v_2}{(\eta, \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) \;\Downarrow\; v_2} \tag{27}$$

$$\frac{(\eta, e_1) \;\Downarrow\; \overline{\mathtt{ff}} \qquad (\eta, e_3) \;\Downarrow\; v_3}{(\eta, \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) \;\Downarrow\; v_3} \tag{28}$$

$$\frac{(\eta, e_1) \;\Downarrow\; v_1 \qquad ((\eta, x{=}v_1), e_2) \;\Downarrow\; v_2}{(\eta, \mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2) \;\Downarrow\; v_2} \tag{29}$$

$$\frac{\begin{array}{c}(\eta, e_1) \;\Downarrow\; (\mathtt{fun}\ f(x{:}t_1){:}t_2\ \mathtt{is}\ e_1') \qquad (\eta, e_2) \;\Downarrow\; v_2 \\ ((\eta, x{=}v_2, f{=}(\mathtt{fun}\ f(x{:}t_1){:}t_2\ \mathtt{is}\ e_1')), e_1') \;\Downarrow\; v\end{array}}{(\eta, e_1\, e_2) \;\Downarrow\; v} \tag{30}$$

$$\frac{(\eta, e_1) \;\Downarrow\; v_1 \qquad (\eta, e_2) \;\Downarrow\; v_2}{(\eta, \langle e_1, e_2\rangle) \;\Downarrow\; \langle v_1, v_2\rangle} \tag{31}$$

$$\frac{(\eta, e) \;\Downarrow\; \langle v_1, v_2\rangle}{(\eta, e.1) \;\Downarrow\; v_1} \tag{32}$$

$$\frac{(\eta, e) \;\Downarrow\; \langle v_1, v_2\rangle}{(\eta, e.2) \;\Downarrow\; v_2} \tag{33}$$

$$\frac{(\eta, e_1) \;\Downarrow\; v_1 \qquad (\eta, e_2) \;\Downarrow\; v_2}{(\eta, e_1 :: e_2) \;\Downarrow\; v_1 :: v_2} \tag{34}$$

$$\frac{(\eta, e_1) \;\Downarrow\; \mathtt{nil[}t\mathtt{]} \qquad (\eta, e_2) \;\Downarrow\; v_2}{(\eta, \mathtt{case}\ e_1\ \mathtt{of\ nil\ =>}\ e_2\ \mathtt{|}\ x{::}y\ \mathtt{=>}\ e_3) \;\Downarrow\; v_2} \tag{35}$$

$$\frac{(\eta, e_1) \Downarrow (v_1 :: v_2) \qquad ((\eta, x{=}v_1, y{=}v_2), e_3) \Downarrow v_3}{(\eta, \texttt{case}\ e_1\ \texttt{of nil =>}\ e_2\ \texttt{|}\ x{::}y\ \texttt{=>}\ e_3) \Downarrow v_3} \tag{36}$$

Write the function

```
evalDyn : (exp env) * exp -> exp
```

which takes an environment (mapping variable names to values) and an expression, and evaluates that expression using dynamic scoping as in the inference rules above. This function should also raise the `Error` exception if the expression cannot be evaluated.

The file `assign3.sml` contains the code for several cases of this function. Fix this code so that it returns the correct answer for the remaining cases instead of raising the `Unimplemented` exception.