Contents

1 Datatypes and Types
   Task 1. (1 point) ................................................. 3
   Task 2. (1 point) ................................................. 3
   Task 3. (1 point) ................................................. 3
   Task 4. (1 point) ................................................. 3
   Task 5. (1 point) ................................................. 3
   Task 6. (1 point) ................................................. 3
   Task 7. (1 point) ................................................. 4
   Task 8. (1 point) ................................................. 4

2 Total fun
   2.1 Totality and valuability ........................................ 5
   Task 9. (1 point) ................................................. 5
   Task 10. (1 point) ............................................... 5
   Task 11. (1 point) ............................................... 5
   Task 12. (2 points) ............................................. 6
   Task 13. (2 points) ............................................. 6
   Task 14. (2 points) ............................................. 6

3 Sum Nights
   3.1 Sublists ....................................................... 7
   3.2 Subset Sum ................................................... 7
   Task 15. (16 points) ............................................ 7

4 To the Left, To the Left
   Task 16. (15 points) ............................................ 9

5 #treedeep5me
   Task 17. (2 points) ............................................. 10
   Task 18. (2 points) ............................................. 11
   Task 19. (2 points) ............................................. 11
   Task 20. (6 points) ............................................. 12
   Task 21. (8 points) ............................................. 12

6 Compiler Optimizations
   6.1 Evaluation ................................................... 14
   Task 22. (5 points) ............................................. 14
   6.2 Constant Fusion ............................................... 14
   Task 23. (1 point) .............................................. 16
Task 24. (4 points) ......................................................... 16
Task 25. (5 points) ......................................................... 16
Task 26. (11 points) ......................................................... 16
1 Datatypes and Types

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

Note: Though you can just plug the expression into the SML/NJ REPL and have it do all the work for you, remember that we might ask you to solve this kind of problem on an exam. We advise you to learn how to do it by hand, because you won’t have a REPL on the exam.

Use this datatype to answer the following type questions.

datatype digit = Zero | One | Two | Three | Four
| Five | Six | Seven | Eight | Nine

Task 1. (1 point)
1 + One

Task 2. (1 point)
Five + Two

Task 3. (1 point)
fn 0 => Zero

Use the following datatypes to answer the next few type questions.

datatype professor = Jacob | Prf
datatype job = Student | Professor of professor | TA of string

Task 4. (1 point)
Student

Task 5. (1 point)
TA

Task 6. (1 point)
fun promote (x : job) =
case x of
  Student => TA "Polly"
| TA y => Professor y
| z => z

Use the following datatype to answer the next type question.

datatype tritree = Empty | Node of int * tritree * tritree * tritree

The int option type is comprised of two constructors: NONE and SOME x, where we have x : int.
Task 7. (1 point)

fun numBranches (Node(x, L, M, R)) = 1 + numBranches(L)

Task 8. (1 point)

fun max (Empty : tritree) = NONE
  | max (Node (x, L, M, R)) =
    let
      val currmax = Int.max (Int.max (max L, max M), max R)
    in
      SOME (Int.max (x, currmax))
    end

Note that Int.max : int * int -> int.
2 Total \textit{fun}

Recall the following definitions:

\textbf{Definition.} An expression $e$ is \textit{valuable} iff there exists a value $v$ such that $e \implies v$.

\textbf{Definition.} A function $f : t_1 \rightarrow t_2$ is \textit{total} iff for all values $x : t_1, f \ x$ is valuable.

An understanding of this terminology and the distinction between totality and valuability is imperative for this task and future problems. In particular, you are expected to correctly cite totality and valuability in your proofs to reflect the evaluation order of SML.

In particular, SML is eagerly evaluated, which means that function arguments are evaluated before the body of the function. Consequently, if your proof steps through the body of a function without first evaluating its argument, you must prove that its argument is indeed valuable. Sometimes this requires showing that an associated function is total.

2.1 Totality and valuability

Let $t_1, t_2, t_3$ be non-function types (that is, they can be, for instance, \texttt{int}, \texttt{string}, \texttt{int list}, or \texttt{int list list}, but not \texttt{int} \rightarrow \texttt{int} or \texttt{int list} \rightarrow \texttt{int}).

Decide whether each of the following statements are true or false. If a statement is true, briefly justify why. If you say it is false, provide a counterexample. (There are no “nonsense” problems.)

\textbf{Task 9.} (1 point)

Let $f : t_1 \rightarrow t_2$ be a value. If there exists a value $x : t_1$ such that $f \ x$ is valuable, then $f$ is total.

\textbf{Task 10.} (1 point)

Let $f : t_1 \rightarrow t_2$ be a total function. Then $f$ is valuable.

\textbf{Task 11.} (1 point)

Let $x : t_1$ be a value. Then for all $f : t_1 \rightarrow t_2, f \ x$ is valuable.

\footnote{We don’t do that here.}
Task 12. (2 points)
Let \( f : t_1 \rightarrow t_2 \) be a value. If for all values \( y : t_2 \), there exists an \( x : t_1 \) such that \( f(x) \equiv y \), then \( f \) is total.

Task 13. (2 points)
Let \( f : t_1 \rightarrow (t_2 \rightarrow t_3) \) be total. Then for all values \( x : t_1 \), \( f(x) \) is valuable.

Task 14. (2 points)
Let \( f : t_1 \rightarrow (t_2 \rightarrow t_3) \) be total. Then there exists a value \( x : t_1 \) such that \( f(x) \) is total.
3 Sum Nights

3.1 Sublists

Before we begin, we first need to define what it means for a list \( L_1 \) to be a *sublist* of \( L_2 \), denoted \( L_1 \subseteq L_2 \). Intuitively, we know what this means: \([1, 3, 5, 6]\) is a sublist of \([1, 2, 3, 4, 5, 6]\) because every element of \([1, 3, 5, 6]\) is also in \([1, 2, 3, 4, 5, 6]\), and they appear in the same order. There are possibly many different ways we could formalize this (you’re encouraged to think about it), but here’s how we decided to do it (inductively, as usual):

**Definition.** For all types \( t \), \( \subseteq \) is defined as

- For all \( L : t \ list \), \( [] \subseteq L \)
- For all \( A : t \ list, L : t \ list \) and all \( x : t \), if \( A \subseteq L \), then \( x::A \subseteq x::L \)
- For all \( A : t \ list, L : t \ list \) and all \( x : t \), if \( A \subseteq L \), then \( A \subseteq x::L \)

If \( L_1 : t \ list, L_2 : t \ list \) such that \( L_1 \subseteq L_2 \), then we say that \( L_1 \) is a *sublist of* \( L_2 \).

So, we know that \([1, 3, 5, 6]\) is a sublist of \([1, 2, 3, 4, 5, 6]\) because \([3, 5, 6]\) is a sublist of \([2, 3, 4, 5, 6]\), and we know that \([3, 5, 6]\) is a sublist of \([2, 3, 4, 5, 6]\) because \([3, 5, 6]\) is a sublist of \([3, 4, 5, 6]\), and so on.

3.2 Subset Sum

Given \( L : int \ list \), we define the sum of \( L \) with the following SML code:

```sml
fun sum [] = 0
  | sum (x::xs) = x + sum xs
```

so, for example, \( \text{sum } [1,2,3] \Rightarrow 6. \)

There is a famous problem in computer science known as the *subset sum* (or, in our terminology, the *sublist sum*) problem: given a list of integers \( L \) and an integer \( n \), identify if there exists a sublist of \( L \) such that \( \text{sum } L \cong n \).

We will have you solve the subset sum problem in SML! Now, we could have you implement your *subsetSum* function to just return a *bool* (i.e. return true or false depending on whether there’s a sublist which sums to the desired value), but we want to do something better! Suppose we not only wanted to know *if* there was such a sublist but, if so, *what that sublist is*. To do this, we will use *options*, which were introduced in lab.

**Task 15.** (16 points)

In `code/subset-sum/subsetSum.sml`, write the function

```sml
subsetSum : int list * int -> int list option
```

**REQUIRES:** true

**ENSURES:**

\( \text{subsetSum } (L, n) \Rightarrow \text{SOME } L' \text{ where } L' \subseteq L \text{ and } \text{sum } L' \cong n. \)

\( \text{subsetSum } (L, n) \Rightarrow \text{NONE} \text{ if there is no such sublist } L'. \)
As a convention, the empty list \([\ ]\) has a sum of 0. Start from the following useful fact: each element of the input list is either in the sublist, or it isn’t.

**Note:** In some instances of the problem, you might find that there are multiple correct answers. For example, \texttt{subsetSum([~1, 1], 0)} could reasonably return \texttt{SOME \([~1,1]\)} or \texttt{SOME \([\ ]\)}. We will consider both correct.

It’s easy to produce correct and unnecessarily complicated functions to compute subset sums. It’s almost certain that your solution will have \(O(2^n)\) work, so don’t try to optimize your code too much. There is a very clean way to write this in a few elegant lines.
4 To the Left, To the Left

Consider the following functions:

```ml
datatype tree = Empty | Node of tree * int * tree
infixr 5 @
fun [] @ R = R
| (x::xs) @ R = x :: (xs @ R)

fun head ([] : int list ) : int option = NONE
| head (x::_) = SOME x

fun inord ( Empty : tree ) : int list = []
| inord ( Node (L,x,R)) = inord L @ x:: inord R

fun leftmost ( Empty : tree ) : int option = NONE
| leftmost ( Node (L,x,R)) =
  case L of
    Empty => SOME x
  | Node _ => leftmost L
```

Task 16. (15 points)

Prove that for all values \( T : \text{tree} \) we have that

\[
\text{leftmost } T \cong \text{head} \ (\text{inord } T)
\]

You may also find the following lemmas useful:

**Lemma 1.** For all non-\( \text{Empty} \) values \( T : \text{tree} \), \( \text{inord } T \cong x::xs \) for some values \( x : \text{int} \), \( xs : \text{int list} \).

**Lemma 2.** \( @ \) is total.

**Lemma 3.** \( \text{inord} \) is total.

Note: As an exercise, try proving this last lemma.

**Hint:** Consider following the structure of the code!
We define a subtree inductively as follows:

**Definition** (Subtree). Let T : tree be a value.

1. T is a subtree of T.
2. Let T' : tree be a value. If T' is a subtree of T and T' \cong Node(L, x, R), then L and R are subtrees of T.

As usual, convince yourself that this definition of subtrees is equivalent to the intuitive one.

The *lowest common ancestor* (lca) between two nodes in a tree is the root of the smallest subtree that contains both nodes. To further your understanding, consider the following examples:

```
    3
   / \
  1   4
 /     \
0 2     5
```

Then,

- 3 is the lowest common ancestor of 0 and 4,
- 3 is the lowest common ancestor of 2 and 5,
- 3 is the lowest common ancestor of 0 and 3,
- 4 is the lowest common ancestor of 4 and 5,
- 1 is the lowest common ancestor of 0 and 2,
- and 1 is the lowest common ancestor of 1 and 0.

In this section, you will be given a tree that does not contain any duplicate values, and two target values that may or may not be in the tree. Our goal is to implement a function to solve the lca problem.

It seems costly to repeatedly check whether a value exists in our tree. Our approach will instead use the path through the tree to each of our target nodes. To achieve this, we introduce the following datatype:

```
datatype direction = LEFT | RIGHT
```

where LEFT indicates traversal through a node's left subtree, and RIGHT indicates traversal through a node's right subtree.

**Task 17.** (2 points)

In code/lca/lca.sml, write the function
find : tree * int -> direction list option
REQUIRES: true
ENSURES: find (T, v) ⊑ NONE if v is not in T, else SOME L, where L is a list of directions that can be used to traverse from the root of T to v

To clarify the behavior of find, consider the following example:

$$\begin{align*}
\text{find (3, 2)} &\equiv \text{SOME [LEFT, RIGHT]} \\
\text{find (3, 6)} &\equiv \text{NONE}
\end{align*}$$

Say that we find a path to one of our target nodes in the tree and produce the direction list L. If we traverse our input tree according to the directions in any prefix of the list L, we arrive at an ancestor of our target. Since our goal is to find the lowest common ancestor of our two target nodes, we want to use the longest prefix that is common to both paths. To achieve this, we will use two more helper functions.

**Task 18.** (2 points)
In code/lca/lca.sml, write the function

follow : tree * direction list -> tree option
REQUIRES: true
ENSURES: follow (T, L) ⊑ NONE if traversing T according to the directions in L do not lead to a valid subtree of T, else SOME T’, where T’ is the subtree of T that is obtained by traversing T according to L

**Task 19.** (2 points)
In code/lca/lca.sml, write the function
common : direction list * direction list -> direction list

REQUIRES: true

ENSURES: common (L1, L2) \cong L, where L is the longest prefix that is common to both L1 and L2

Your implementation of common should have \(O(n)\) work and span, where \(n = \max(|L1|, |L2|)\).

**Task 20.** (6 points)

Finally, use find, follow, and common to implement the function

lca : tree * int * int -> tree option

REQUIRES: T contains no duplicates, a \(\not\equiv\) b

ENSURES: lca (T, a, b) \cong NONE if a or b are not in T, else SOME (Node (L, x, R)) s.t. Node (L, x, R) is a subtree of T, and either:

- a is in L and b is in R
- b is in L and a is in R
- a = x and b is in L or R
- b = x and a is in L or R

**Task 21.** (8 points)

Write and solve recurrences for the work and span of your implementation of find, follow, and lca in terms of the number of nodes \(n\) in the input tree. For this analysis, assume that the input tree is balanced. That is, for all subtrees of the form Node (L, x, R), the number of nodes in L and R differ by at most 1. Please include a copy of your implementation in your written submission, and show your work when solving for the big-\(O\) bounds.

You may assume that common (L1, L2) has \(O(n)\) work and span, where \(n = \max(|L1|, |L2|)\).
6 Compiler Optimizations

In this problem, we will be working with the following datatype, which represents an arithmetic expression over integers.

```haskell
datatype exp
  = Var of string
  | Int of int
  | Add of exp * exp
  | Mul of exp * exp
  | Not of exp
  | IfThenElse of exp * exp * exp
```

Here is the meaning of each constructor in the datatype:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var x</td>
<td>A variable with name x. When evaluating the expression, this variable must be assigned a value in the environment.</td>
</tr>
<tr>
<td>Int n</td>
<td>A constant integer with value n.</td>
</tr>
<tr>
<td>Add (a, b)</td>
<td>The sum of two expressions, a and b.</td>
</tr>
<tr>
<td>Mul (a, b)</td>
<td>The product of two expressions, a and b.</td>
</tr>
<tr>
<td>Not a</td>
<td>The “truthy” negation of an integer expression. If a is 0, Not a should be 1. Otherwise, Not a should be 0.</td>
</tr>
<tr>
<td>IfThenElse (i, t, e)</td>
<td>“Truthy” casing on integers. If i is nonzero, the expression evaluates to t. Otherwise, it evaluates to e.</td>
</tr>
</tbody>
</table>

Here are some example representations of operations.

<table>
<thead>
<tr>
<th>Arithmetric</th>
<th>exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 + 3</td>
<td>Add (Int 2, Int 3)</td>
</tr>
<tr>
<td>2 + x</td>
<td>Add (Int 2, Var &quot;x&quot;)</td>
</tr>
<tr>
<td>3x + 1</td>
<td>Add (Mul (Int 3, Var &quot;x&quot;), Int 1)</td>
</tr>
<tr>
<td>3 · 2 + x</td>
<td>Add (Mul (Int 3, Int 2), Var &quot;x&quot;)</td>
</tr>
<tr>
<td>“if x ≠ 0 then y else 8”</td>
<td>IfThenElse (Var &quot;x&quot;, Var &quot;y&quot;, Int 8)</td>
</tr>
<tr>
<td>“1 if x = 0, else 0”</td>
<td>Not (Var &quot;x&quot;)</td>
</tr>
<tr>
<td>“0 if x = 0, else 1”</td>
<td>Not (Not (Var &quot;x&quot;)).</td>
</tr>
</tbody>
</table>

**Environments** We also need a way to represent the “environment”, which contains the variable bindings. We will use the `environ` type alias:

```haskell
type environ = (string * int) list
```
Note that there are three helper functions defined in `code/expressions/exp.sml` that you might find useful as you work through the following tasks.

- `lookup (env, x)` finds the integer bound to variable `x` in environment `env`.
- `vars e` evaluates to a list of all variable names in expression `e`.

Full specifications and implementations can be found in the file itself.

### 6.1 Evaluation

Before we do anything, it would be helpful to define an evaluation function for our datatype. Our function will take in an environment, which is just a list of `string * int` pairs representing which integer each variable name is bound to. You may assume the list has no duplicates.

**Task 22.** (5 points)

In `code/expressions/eval.sml`, write:

```sml
val eval : environ * exp -> int

REQUIRES: Each variable in `e` has one entry in `env`  
ENSURES: `eval (env, e) ⇒ n`, where `n` is the integer representing the value of `e` given  
environment `env`  
```

For example, it should be the case that:

```sml
eval ([("x",45),("y",3)], Add (Int 15, Mul (Var "x",Var "y"))) ⇒ 150
```

To test your code, you will need to load both `exp.sml` and `eval.sml` into the REPL. You may do this by running the following command in the `code/expressions/` directory:

`smlnj exp.sml eval.sml`

If you attempt to view deeply-nested `exp` values, the REPL will stop printing a few layers in, instead showing a `#` sign. You can (temporarily) increase the print depth by typing the following into the REPL:

`Control.Print.printDepth := 100`

### 6.2 Constant Fusion

Many programming language implementations have optimizations so that the code you write will run faster. For example, if you write the Python program

```python
def foo(x):
    return x + (2 * 3)
```

an optimization called *constant fusion* may be applied, transforming the function to

```python
def foo(x):
    return x + 6
```
In this case, the constant-fused expression has precomputed \((2 \times 3)\), so we can avoid the multiplication step at runtime when the function is called. We’ve implemented a similar optimization \(^2\) in `code/expressions/fuse.sml` that fuses multiplications for our integer expression language:

```sml
fun fuse (Var x) = Var x
     | fuse (Int n) = Int n
     | fuse (Add (a,b)) = Add (fuse a, fuse b)
     | fuse (Mul (a,b)) =
             case (vars a,vars b) of
                 ([],[]) => Int (eval ([],a) * eval ([],b))
             | (_, _) => Mul (fuse a,fuse b)
     |
     | fuse (Not a) = Not (fuse a)
     | fuse (IfThenElse (i,t,e)) = IfThenElse (fuse i,fuse t,fuse e)

val Add (Var "x",Int 6) = fuse (Add (Var "x", Mul (Int 2,Int 3)))
```

However, before enabling this optimization, we want to ensure that the constant-fused code is actually equivalent to the original code. Thankfully, we can prove this fact and sleep soundly for the rest of our lives.

The next four tasks will refer to the validity of `fuse`.

**Theorem** (Validity of `fuse`). For all `env` and `e` satisfying the REQUIRES for `eval`,

\[
\text{eval (env, fuse e)} \equiv \text{eval (env, e)}
\]

You may make use of the following lemmas:

**Lemma 4** (Totality of `vars`). The `vars` function is total.

**Lemma 5** (Totality of Fusion). The `fuse` function is total.

**Lemma 6** (Idempotence of Fusion). For all `e : exp`:

\[
\text{fuse e} \equiv \text{fuse (fuse e)}
\]

**Lemma 7** (Closed Expression Valuability). For all `e : exp`, if `vars e \equiv []`, then for all `env : environ`, we have that `eval (env,e)` is valuable.

**Lemma 8** (Environment Independence of Closed Expressions). For all `e : exp`, if `vars e \equiv []`, then for all `env : environ`, we have that `eval (env,e) \equiv eval ([],e)`.

\(^2\)This is just one of many possible optimizations! For example, you could consider a similar optimization for addition, or optimizing expressions like `Add (Add (x,x),x)` to `Mul (Int 3,x)`. We will only consider the given optimization for constant multiplication on multiplication in this problem, but feel free to experiment with coding (and proving) other optimizations.

\(^3\)Note that this is a special case of a stronger lemma. Namely, given `e : exp`, if `env` and `env'` bind all variables in `e` to the same integers and satisfy the REQUIRES of `eval`, then `eval (env,e) \equiv eval (env',e)`.
Feel free to abbreviate \texttt{IfThenElse} as \texttt{ITE}.

Note that this proof may depend on your implementation of \texttt{eval}. Make sure you are convinced of its correctness!

\textbf{Task 23.} (1 point)
Identify which cases are base cases for the structural induction proof of the theorem.

\textbf{Task 24.} (4 points)
State the inductive cases and their IHs for the structural induction proof of the theorem.

\textbf{Task 25.} (5 points)
Prove the \texttt{IfThenElse} case for the structural induction proof of the theorem.

\textbf{Task 26.} (11 points)
Prove the \texttt{Mul} case for the structural induction proof of the theorem.

\textbf{Constraint:}

- While you may use abbreviations to shorten your proof, make sure this is \textit{not} at the expense of correctness clarity. In particular, please make the structure of your proof is clear: be sure to \textit{explicitly} state all cases and inductive hypotheses.
- You may \textit{not} cite the theorem in any step of your proofs.

This assignment has a total of 94 points.