Name: ______________________________

ID number: ______________________________

CS 312, Fall 2002

Exam 1

October 17, 2002

Problem	1	2	3	4	Total
Grader					
Grade	/20	/25	/25	/30	

There are four (4) problems on this exam. Please check now that you have a complete exam booklet with eight (8) numbered pages after the cover sheet. *Write your name or id number on each page of the exam.* Be sure to try all of the problems, as some are more difficult than others (i.e., don't waste a lot of time on a problem that is giving you a hard time – move on to another problem and then return to it later). The order of the problems is roughly by increasing difficulty.

This exam is closed book. No papers, books or notes may be used.

There is space provided to answer each question. You may use the back of the page, or request extra paper if you need more space.

To help ensure that you don't accidentally miss any of the questions, we have marked those sections where an answer is requested with a ⟸ in the right margin.

The staff reserves the right to ignore illegible answers. "Pretty printing" (proper indentation) of your code will aid in the grading process.

1. (20 Points) For each of the following expressions determine what value would be returned by SML, or indicate that an error would occur.

   (a)
```
let val x:int list = [4,2] in
    hd(x) + hd(tl(x))
end
```

```
val it = 6 : int
```
⟸

   (b)
```
let val x:int list = [4,2] in
    hd(x) * tl(x)
end
```

```
stdIn:14.1-14.39 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * int list
  in expression:
    hd x + tl x
```
⟸

   (c)
```
let val x:int = 4 in
    let val ford:int->int = fn(z:int) => x*z in
        let val x:int = 2 in
            ford(x)
        end
    end
end
```

```
val it = 8 : int
```
⟸

(d)
```
let val ford:int->int =
        fn(x:int) =>
           let val x:int = x + x in
             let val arthur:int->int = (fn(x:int) => x*4) in
               let val marvin:int->int = (fn(z:int) => x+z) in
                 arthur(marvin(x))
               end
             end
           end
    in
      ford(2)
end

val it = 32 : int
```
⟸

(e)
```
let val ford:int->int =
 (fn(z:int) =>
   (fn(self:int->int) => z+4)(fn(self:int) => z+4))
  in
   let val marvin:(int->int)->int =
         (fn(arthur:int->int) => arthur(ford(2)))
      in
       marvin(fn(y:int) => y*3)
  end
end

val it = 18 : int
```
⟸

2. (25 points)

The procedure `mapsquares` takes a list of integers and returns a new list where each element of the new list is the square of the corresponding element in the input list. So, for example, we have

`mapsquares([1,2,5,4])` $\Rightarrow$ `[1,4,25,16]`

We will define `mapsquares([])` $\Rightarrow$ `[]`.

(a) Give a short recursive definition of `mapsquares`. To simplify the next part of this problem, please do not use tail-recursion, pattern matching or case statements.

```
fun mapsquares (x: int list): int list =
  if null(x) then []
    else (hd x * hd x)::mapsquares(tl x)
```

$\Longleftarrow$

(b) Prove that your definition of the `mapsquares` function is correct, using induction and the substitution model. You may assume that all ML arithmetic and list primitives perform correctly. In using the substitution model, you only need to show the expression you are evaluating, and any steps that involve either a substitution or the use of the variable you are doing induction on. To receive full credit you must justify each step.

(1) $P(n)$ = Given an arbitrary list x of length n, `mapsquares(x)` returns a list z s.t. $\forall i, 1 \le i \le n.(\#i\ x)^2 = (\#i\ z)$.

(2) We are doing induction on the length of the lists, which is the set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.

Proof by induction

(3) Base case:

The base case corresponds to $n = 0$, or in other words lists of length 0. We instantiate $P(n)$ with $n = 0$ to get the following:

$P(0)$ = Given an arbitrary list x of length 0, `mapsquares(x)` returns a list z s.t. $\forall i, 1 \le i \le 0.(\#i\ x)^2 = (\#i\ z)$.

Since the only list of length 0 is `[]`, and no integer i satisfies $1 \le i \le 0$, we can rewrite $P(0)$ as follows:

$P(0) =$ `mapsquares([]) returns []`

We use an short, semi-informal version of the substitution model to prove the base case. We omit type annotations for better legibility.

$eval($`mapsquares([])`$)$

substitute function body

$= eval($`(fun mapsquares (x) = if null(x) then []`
`else (hd x * hd x)::mapsquares(tl x)) []`$)$

evaluate and substitute argument in function body

$= eval($`if null([]) then [] else (hd [] * hd []):mapsquares(tl [])`$)$

evaluate test condition: $eval($`null([])`$) = true$ by definition of the null operator

condition is true $\Rightarrow$ *evaluate 'then' branch*

$= eval($`[]`$) =$ `[]`

an empty list evaluates to itself

Thus we have indeed that $eval($`mapsquares([])`$) =$ `[]`, which proves $P(0)$.

Induction Hypothesis

Our Induction Hypothesis is that $P(n)$ holds for an arbitrary natural number n. (see above for formal definition).

(4) Induction Step

Assuming $P(n)$ in our Induction Hypothesis, we want to prove $P(n+1)$:

$P(n+1) =$ Given an arbitrary list x of length $n+1$, `mapsquares(x)` returns a list z s.t. $\forall i, 1 \leq i \leq n+1.(\#i\ x)^2 = (\#i\ z)$.

An arbitrary list of length $n+1$ can be obtained, for example, by prepending an arbitrary element of the base type to an arbitrary list of n elements. Thus if y is an arbitrary list of integers having length n, $x = a :: y$ (a being an arbitrary *int*) is an arbitrary list of *ints* of length $n+1$. Note that since y has length n, the induction hypothesis applies to it.

eval(`mapsquares(a::y)`)

function body substitution

= *eval*(`(fun mapsquares (x) = if null(x) then []`
`else (hd x * hd x)::mapsquares(tl x)) [a::y]`)

evaluate and substitute argument in function body

= *eval*(`if null(a::y) then a::y else`
`(hd (a::y) * hd (a::y))::mapsquares(tl (a::y))`)

> *evaluate test condition: eval*(`null(a::y)`) $= false$ because list a::y contains at least one element (specifically, a), and thus it can't be an empty list.

condition is false ⇒ evaluate 'else' branch

= *eval*(`(hd (a::y) * hd (a::y))::mapsquares(tl (a::y))`)

evaluate hd and tl applications - note that we are taking shortcuts!

= *eval*(`(a * a)::mapsquares(y)`)

As discussed above, the induction hypothesis applies to y, as y is a list of length n. We have the following relations:

> If `mapsquares(y)` $= z \Rightarrow$ (by I.H) $\forall i, 1 \leq i \leq n.(\#i\ y)^2 = (\#i\ z)$.

Since $(\#i\ y) = (\#(i+1)\ x)$ (i.e. y is the tail of x), we have from above that $\forall i, 1 \leq i \leq n.(\#i\ y)^2 = (\#(i+1)\ x)^2 = (\#i\ z)$. In other words, `mapsquares(y)` returns a list containing the squares of the last n elements of x, in their original order.

Now, $a * a$ is the square of the first element of x, i.e. $(a * a) = (\#1\ x)$.

We can now refer to the concatenation `(a * a)::mapsquares(y)` $= z'$, and state that: $(\#1\ x)^2 = (\#1\ z')$, and that $\forall i, 1 \leq i \leq n.(\#(i+1)\ z') = (\#i\ z) = (\#(i+1)\ x)$. By rewriting these expressions we get:

$(\#1\ x)^2 = (\#1\ z')$, and that $\forall i, 2 \leq i \leq n+1.(\#(i\ z') = (\#i\ x)$,

which is equivalent to

$\forall i, 1 \leq i \leq n+1.(\#i\ x)^2 = (\#i\ z')$,

which proves P(n + 1).

QED

Note that operator $(\#i\ l)$ (for getting the i^{th} element of a list) does not exist in SML. We use this form here as a notational convenience. The same observation applies to x^2, used here to indicate exponentiation.

3. (25 Points) Some of the following expressions can be made to evaluate to 42 by filling in $\boxed{\texttt{TYPE}}$ and $\boxed{\texttt{EXP}}$ correctly. For example, the expression

```
let val HHGTTG:[TYPE] = [EXP] in
    HHGTTG(21)
end
```

can be made to evaluate to 42 by
$\boxed{\texttt{TYPE}}$ = `int->int`
$\boxed{\texttt{EXP}}$ = `fn(x:int) => x*2`.

For each of the following expressions, specify $\boxed{\texttt{TYPE}}$ and $\boxed{\texttt{EXP}}$, or state that the expression cannot be made to evaluate to 42 and justify your claim.

(a) (5 points)

```
let val HHGTTG:[TYPE] = [EXP] in
  let val ford:int list->int = (fn(x:int list) => hd(x)) in
    let val arthur:int list->int =  fn(y:int list) => hd(tl(y)) in
     (ford(HHGTTG)+1)*(arthur(HHGTTG)-1)
    end
  end
end
```

$\boxed{\texttt{TYPE}}$ = *int list*
$\boxed{\texttt{EXP}}$ = `[0,43]` or `[1,22]` or `[2,15]` or `[6,7]` or `[~3, ~20]` ...

(b) (5 points)

```
let val HHGTTG:[TYPE] = [EXP] in
  let val ford:int list->int = (fn(x:int list) => hd(x)) in
    let val arthur:int list->int =  fn(y:int list) => hd(tl(y)) in
      ford(HHGTTG)*HHGTTG(arthur)
    end
  end
end
```

No solution.

From the application of `ford` function, which applied `hd` to its argument, we infer that `HHGTTG` must be an *int list*. But then expression `HHGTTG(arthur)` can not typecheck, as `HHGTTG` should be (more precisely: evaluate to) a function.

(c) (5 points)

```
let val HHGTTG: TYPE  = EXP  in
  let val ford:int list->int = (fn(x:int list) => hd(x)) in
    let val arthur:int list->int =  fn(y:int list) => hd(tl(y)) in
      (ford(HHGTTG))(arthur(HHGTTG))
    end
  end
end
```

No solution.

Expression `ford(HHGTTG)` and the definition of function ford imply that `HHGTTG` must be an *int list*, and that `ford(HHGTTG)` returns an *int*. But this means that expression `(ford(HHGTTG)) (arthur(HHGTTG))` is not correct, as `ford(HHGTTG)` should evaluate to a function. The type restrictions imposed by the definition of function ford thus making it impossible to find a solution to the problem.

(d) (10 points)

```
let val HHGTTG: TYPE  = EXP  in
  let val ford:int list->int = (fn(x:int list) => hd(x)) in
    let val arthur:int list->int =  fn(y:int list) => hd(tl(y)) in
      (HHGTTG(ford))(HHGTTG(arthur))
    end
  end
end
```

No solution (unless one uses parameterized types).

From expression `(HHGTTG(ford)) (HHGTTG(arthur))` we infer that `HHGTTG` must be a function, and that `(HHGTTG(ford))` must return a function. Because we can't use parameterized types, expression `HHGTTG(ford)` constrains the type of `HHGTTG's` first argument to be (*int list* $\rightarrow$ *int*). This restriction is also satisfied in expression `HHGTTG(arthur)`, as both `ford` and `arthur` have the same signatures.

Now, we can see that `HHGTTG(ford)` must return a function whose type must be $'a \rightarrow int$. We use $'a$ here to denote the as-of-yet unknown type of of `HHGTTG(arthur)`, which is the first argument for the function returned by `HHGTTG(ford)`. As `HHGTTC(ford)` and `HHGTTG(arthur)` are in fact of the same type, we infer that $'a$ should be the same type as $'a \rightarrow int$. This turns out to be impossible, however.

Stated a little more formally, we can say that `HHGTTG(ford)` and `HHGTTG(arthur)` constrains `HHGTTG` to the type (*int list* $\rightarrow$ *int*) $\rightarrow \alpha$ where we must solve for α. `(HHGTTG(ford)) (HHGTTG(arthur))` tells us that `HHGTTG(ford)` should return a function, and so we have a new type constraint that $\alpha \equiv (\beta \to \gamma)$. However, `HHGTTG(ford)` takes in as input `HHGTTG(arthur)`, which means that $\alpha \equiv \beta$ (because `HHGTTG(arthur)` is of type α and is also an input to a function that takes in a β). Substituting, we now have that $\beta \equiv (\beta \to \gamma)$, and we cannot solve that particular type equation without additional help.

Here is a solution using parameterized types:

```
let
  val HHGTTG: 'a -> 'b -> int = (fn (x: 'a) => fn (y: 'b) => 42)
in
  let
    val ford: int list -> int = (fn (x: int list) => hd(x))
  in
    let
      val arthur: int list -> int = (fn (y: int list) => hd (tl
(y)))
    in
      (HHGTTG(ford)) (HHGTTG(ford))
    end
  end
end
```

We can also solve this if we were allowed to declare a recursive datatype and to modify some of the types; this is left as an exercise.

4. (30 points)

The procedure `listsumsto` takes a (non-empty) list of integers and returns a new list where each element of the new list is the sum of all the previous elements of the input list. So, for example, we have

`listsumsto([1,2,5,4])` $\Rightarrow$ `[1,3,8,12]`

You may assume that `listsumsto` is only called on a non-empty list of integers. Hint: it may be useful to use `List.rev`, which reverses a list, and also to write one or more helper functions within the main functions you define.

(a) Write a recursive (but not tail-recursive) definition of the function `listsumsto`.

```
fun listsumsto (x: int list): int list =
  let
    fun helper (x: int list, sum: int) =
      case x of
        []   => []
      | h::t => (h + sum)::helper(t, h + sum)
  in
    helper (x, 0)
  end
```

(b) Write a tail-recursive definition of the function `listsumsto`.

```
fun listsumsto (x: int list): int list =
  let
    fun helper (x: int list, ans: int list) =
      case x of
        []   => tl(List.rev ans)
      | h::t => helper(t, (h + hd ans)::ans)
  in
    helper (x, [0])
  end
```

(c) Recall that the definition of `foldl` is

```
fun foldl (f:'a*'b->'b) (accum:'b) (x:'a list) : 'b =
  case x of nil => accum
| hd::tl => foldl f (f(accum,hd)) tl
```

Write a non-recursive definition of the function `listsumsto` using a single call to `foldl`.

```
fun listsumsto (x: int list): int list =
  tl(List.rev (foldl (fn (v, l) => (v + hd l)::l) [0] x))

fun listsumsto (x: int list): int list =
  List.rev (#1 (foldl (fn (v, (l, s)) => ((v + s)::l, v + s))
                     ([], 0) x))
```