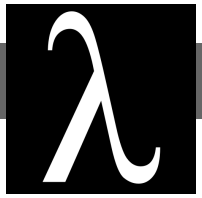



**Out: Tuesday, 23 May 2023 at 18:00 ET**
**Due: Tuesday, 30 May 2023 at 23:59 ET**

**Total Points: 111**

## Contents

# 1 Preamble

In this assignment, some lines are marked by the following comment:

```
(* DELETE THIS LINE *)
```

For example,

```
fun foo (x : int) : int =
    raise Fail "Unimplemented"  (* DELETE THIS LINE *)
```

These lines are placeholders for these functions. They're only there so that the file can load even when the function hasn't been implemented. Only remove such lines when you are ready to implement their associated function. There should be none left when you submit your assignment.

Also notice that, the left parts (i.e. the function arguments) of one or more function clauses are specified for some functions, while for others they are not. It will be your responsibility to implement the function clauses appropriately if this is the case.

## 1.1 Testing Code

In order to test your code, you should `cd` into the `code` directory and then into whatever problem you are working on. You then should open the SML/NJ REPL with the `./smlnj` command. Normally, we can just use `smlnj` (without the `./`) but when working on homework/lab problems, we want to use `./smlnj` because it configures the REPL in a way that you may find helpful (hides some warnings, loads our libraries, etc).

We provide a `sources.mlb` file for each problem. This will load in all the necessary SML files for you (useful if the problem has multiple files). You can load it by just typing `./smlnj sources.mlb`.

In order to write test cases, you should use the 150 testing framework that we introduced in Basics Lab. You can find the complete list of testing functions in the 150Basis Reference.

# 2 Types and evaluation

In this section, we will be exploring types and values. Do all the problems in this section in your written submission.

## 2.1 Typify

For each of the following expressions, state its type and value. If it is not well-typed, put "not well-typed". If the expression doesn't have a value, then put "no value". Express function values as lambda expressions (that is, as an expression of the form `fn x => e`).

If we have reason to suspect that you are using the SML/NJ REPL, you will not get credit for the question.

*Hint*: Functions are values[1].

**Task 2.1.** (1 point)

```
(15150::[])::[]
```

**Task 2.2.** (1 point)

```
fn [] => true | x::xs => xs
```

**Task 2.3.** (1 point)

```
fn [] => true | x::xs => x
```

**Task 2.4.** (1 point)

```
(let
  fun length ([]: int list): int = 0
    | length (x::xs) = 1 + length(xs)
in
  length [1, 5, 1, 5, 0]
end) + 1
```

**Task 2.5.** (1 point)

```
fn (x : int) => [x, 1 + 1]
```

[1] Where have I seen that before?

# 3 2 plus 2 is 4 minus 1 that's 3 quick maths

## 3.1 Addition & Multiplication

Let's start with the basics [2]: addition of natural numbers. Let's assume that 0 *is* a natural number for this problem. We will *recursively* define the process of adding two natural numbers as repeatedly adding 1: [3]

```
add : int * int -> int
```

REQUIRES: $m \geq 0$, $n \geq 0$

ENSURES: `add (m, n)` $\Longrightarrow m + n$

```
fun add (0 : int, n : int) : int = n
  | add (m, n) = 1 + add (m - 1, n)
```

It turns out that we can write a multiplication function in a similar way, just using addition and recursion!

**Task 3.1.** (4 points)

In `code/naturals/recursion.sml`, write the specification and the function

```
mult : int * int -> int
```

such that `mult (m, n)` recursively calculates the product of `m` and `n`, for any two natural numbers $m$ and $n$. Your implementation *may* use the function `add` and the built-in subtraction operator.

**Constraint:** Your implementation may not use the built-in addition, multiplication, or division operators.

## 3.2 Primality Test

Here, we will be utilizing recursion to determine if a natural number is prime. Recall the definition for prime numbers:

**Definition.** A natural number $n > 1$ is prime if and only if it is divisible by only itself and 1.

Given the input number $n$, a simple test for primality is to go through all of the numbers from 2 to $n - 1$ and check if each divides into $n$. This can be done with recursion, using an extra argument that keeps track of the current divisor that we should check next. We can test for divisibility using `mod`, because $n$ is divisible by $m$ if and only if $n$ `mod` $m = 0$.

Of course, we only really need to check for divisibility by 2 through $\sqrt{n}$, but we will not penalize you if your code checks for divisibility by 2 through $n - 1$.

[2] BOOM . . . everyday man's on the block

[3] This way, we only have to define what it means to add 0 and add 1, and then we can extend this definition to know what it means to add 2, what it means to add 3, etc.

**Task 3.2.** (10 points)

```
isPrime : int -> bool
```

REQUIRES: $n > 1$

ENSURES: `isPrime n` $\Longrightarrow$ `true` if $n$ is prime and `false` otherwise

***Hint:*** Use a recursive helper function of a suitable type (your choice), as outlined above. Make sure you properly document and test any helper function you define.

# 4 Pascal's Triangle

## 4.1 Turtles all the way down

*Binomial coefficients* are indexed by two natural numbers, and the $(n, k)$-th binomial coefficient is usually denoted by

$$\binom{n}{k}$$

which is read "$n$ *choose* $k$". The reason we use the word "choose" is because there are $\binom{n}{k}$ ways to pick a subset of $k$ items from a set of $n$ items (e.g. if you have 15 yellow shirts and 6 friends you want to give yellow shirts to, then there are $\binom{15}{6}$ ways to pick which shirts to give to them). Notice that, in order for this interpretation to make sense, we must have that $n \geq 0$ and that $0 \leq k \leq n$. Otherwise, the binomial coefficient is undefined.

Now, if you arrange the binomial coefficients into a triangle, you get a nice pattern:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | | | | |
| 1 | 1 | 1 | | | |
| 2 | 1 | 2 | 1 | | |
| 3 | 1 | 3 | 3 | 1 | |
| 4 | 1 | 4 | 6 | 4 | 1 |

where the value of $n$ is on the left-hand side, and the value of $k$ is along the top. This is called Pascal's Triangle, and it visually displays some of the neat patterns about binomial coefficients.

There is an explicit way to define binomial coefficients in terms of factorials, but Pascal's triangle actually suggests an elegant *recursive* solution. Notice that

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

(i.e. every entry is equal to the sum of the entry above it and the entry above it and to the left. For example, $4 = 1 + 3$). In the case that $n = k$ or $k = 0$, we have that $\binom{n}{k} = 1$.

In order to make our code easier to prove, we can use this observation to define a function:

**Task 4.1.** (6 points)

In `code/pascal/pascal.sml`, implement the function

```
pascal : int * int -> int
```

REQUIRES: $0 \leq$ `k` $\leq$ `n`

ENSURES: `pascal (n, k)` $\cong$ the element of Pascal's triangle at row `n`, position `k` – in other words, $\binom{n}{k}$.

**Constraint:** While there is a closed form for Pascal's triangle, you must use a recursive solution.

## 4.2 Prove it!

***Note*: before starting this task, copy the code you have written for `pascal` into your written solutions!**

Use either the LaTeX `codeblock` environment:

```
\begin{codeblock}
  < Your copy-pasted code here... >
\end{codeblock}
```

or upload your `code/pascal/pascal.sml` file to Overleaf and use the LaTeX `codefile` command with the appropriate line range:

```
\codefile[linerange=1-10]{pascal.sml}
```

In general, it's good style to include a copy of any code you are using along with your proofs.

Check out the 15-150 Proof Guidelines for additional proof-writing guidance.

**Task 4.2.** (8 points)

Prove that, for all `n : int` such that $\texttt{n} \geq 1$, *your* implementation of `pascal` has the property that

$$\texttt{pascal (n+1,2)} \cong \sum_{i=1}^{n} i$$

You may assume that the operation `+` over `int`s in SML is implemented correctly as addition over the integers in the mathematical sense – that is, it is commutative and associative, and has the identity element 0.

You may also find the following lemma useful:

**Lemma.** For all `n : int` such that $\texttt{n} \geq 1$,

$$\texttt{pascal (n,1)} \cong \texttt{n}$$

Be sure to cite the lemma wherever you use it.

# 5 List recursion

## 5.1 Heads or tails?

Having practiced recursion on natural numbers, we are now prepared to have more `fun` with recursion on lists!

Here are some things to consider:

- What should the function do in the case of the empty list?
- What should the function do in the case of non-empty lists?
- If the list is non-empty, how do we extract the first value of the list?
- Think recursively about how you can operate on the first element of the list and then what to do to the rest of the list.

**Task 5.1.** (5 points)

`heads : int * int list -> int`

REQUIRES: true

ENSURES: `heads (x,xs)` $\Longrightarrow$ `n` where `n` is the largest number such that the first `n` elements of `xs` are equal to `x`.

Some examples:

`heads (1,[1,1,2,1,3])` $\Longrightarrow$ `2`
`heads (2,[1,1,2,1,3])` $\Longrightarrow$ `0`

**Task 5.2.** (7 points)

`tails : int * int list -> int list`

REQUIRES: true

ENSURES: `tails (x,xs)` $\Longrightarrow$ `xs'` such that `xs'` is `xs` with the first `heads (x, xs)` elements removed.

Some examples:

`tails (1,[1,1,2,1,3])` $\Longrightarrow$ `[2,1,3]`
`tails (2,[1,1,2,1,3])` $\Longrightarrow$ `[1,1,2,1,3]`

**Task 5.3.** (10 points)

`remove : int * int list -> int list`

REQUIRES: true

ENSURES: `remove (x,xs)` $\Longrightarrow$ `xs'` such that `xs'` does not contain any element `y` such that `y = x`, and `xs'` contains exactly all elements in `xs` except for `x` in the same order as they were in `xs`

Some examples:

```
remove (1,[1,1,2,1,3]) ⟹ [2,3]
remove (2,[1,1,2,1,3]) ⟹ [1,1,1,3]
remove (3,[1,5,1,5,0]) ⟹ [1,5,1,5,0]
```

**Task 5.4.** (13 points)

Now we are going to "find a Partition" [4]!

`partition : int * int list -> int list * int list`

REQUIRES: true

ENSURES: `partition (pivot,L)` $\Longrightarrow$ `(L1,L2)` such that

- For all `x` in `L1`, `x` $\leq$ `pivot`
- For all `y` in `L2`, `y` $>$ `pivot`
- All elements in `L` are in exactly one of `L1` or `L2` (i.e. `L1 @ L2` is a permutation of `L`)
- Elements in `L1` are in the same order as they were in `L`
- Elements in `L2` are in the same order as they were in `L`

**Constraint:**

- Your code ***may not*** contain more than one recursive call to `partition` in the recursive case of your function.
- You ***may not*** write/use any helper functions to implement this.
- With the exception of relational operators (e.g., `>`, `<`) and constructors (e.g. `::`), you ***may not*** use any basis library functions, including `@`.

Some examples:

```
partition (2,[1,3,4,0,5]) ⟹ ([1,0],[3,4,5])
partition (3,[1,2,3,0]) ⟹ ([1,2,3,0],[])
```

[4] This is foreshadowing a future homework.

# 6 Stepping

Deya was wandering the halls of Gates when she stumbled across a piece of paper with two random SML functions on it:

```
fun loop (x : int) : int = loop x
fun mystery (x : int, y : int) : int = (1 div x) + y
```

Intrigued by the functions, and wanting to test her SML abilities, she writes the following `let` expression:

```
let
  val y : int = 1 div 0
in
  loop 0 + y
end
```

To practice her evaluation skills, she writes out the following trace:

```
   let val y : int = 1 div 0 in loop 0 + y end
⟹loop 0 + 1 div 0                    (evaluation of let expression)
⟹loop 0 + 1 div 0                                (clause 1 of loop)
⟹loop 0 + 1 div 0                                (clause 1 of loop)
⟹...                                                (loops forever)
```

After writing her trace, Deya types the expression into the REPL to verify her work. But rather than looping forever, it said a `Div` exception was raised?

**Task 6.1.** (1 point)

Identify where Deya's trace went wrong and explain why it is incorrect.

Deya had written another expression to trace, but given her previous mistake, she isn't confident about doing another one...

**Task 6.2.** (2 points)

Show all the steps in evaluating the following expression to a value:

```
let
  val (x, y) : int * int = (30 + 3, 3)
  val z : int = 25 + 26
in
  x * y + z
end
```

***Hint.*** Feel free to omit unnecessary information with ... as long as the order of evaluation is clear.

The mysterious piece of paper Deya found also had the following statement on it:

```
mystery (0, f 0) ≅ (1 div 0) + (f 0)
```

**Task 6.3.** (2 points)

True or false? For any value `f : int -> int`,

$$\texttt{mystery (0, f 0)} \cong \texttt{(1 div 0) + (f 0)}$$

If true, then explain why. Otherwise, provide a specific counter-example for `f` and explain why it is a counter-example.

***Hint.*** Even though we don't have the definition of `f`, we are assuming that `f` is a function of type `int -> int`. Your answer should not be concerned about the well-typedness of `f`.

***Hint.*** Think about the order of evaluation. Also recall the definition of extensional equivalence. Look back at the Evaluation section of Basics HW if you are stuck.

# 7 De*list*ifying proofs

We have seen proofs involving functions with `int` inputs and outputs. It is natural to ask if we can prove the correctness of functions that take in inputs or produce outputs of other types. In this problem you will write a proof involving list functions and their specifications.

Consider the following function, which takes in `n : int` and `k : int` as arguments and makes a list with `n` copies of `k`:

> `repeat : int * int -> int list`
>
> REQUIRES: $\texttt{n} \geq 0$
>
> ENSURES: `repeat (n,k)` $\Longrightarrow$ `L` such that `L` has `n` elements and contains only `k`

```
fun repeat (0 : int, k : int) : int list = []
  | repeat (n : int, k : int) : int list = k :: repeat (n - 1, k)
```

and the following function, which evaluates to the number of elements in a list `L`:

> `size : int list -> int`
>
> REQUIRES: true
>
> ENSURES: `size L` $\cong$ number of elements in `L`

```
fun size ([] : int list) : int = 0
  | size (x::xs) = 1 + size xs
```

You will be proving the part of `repeat`'s specification that asserts, for any `k : int` and any `n : int`, `repeat (n,k)` returns a list with `n` elements.

**Task 7.1.** (10 points)

Prove[5] for all values `n: int` and `k: int` where $\texttt{n} \geq 0$,

$$\texttt{size (repeat (n,k))} \cong \texttt{n}$$

Please **do not** rely on your intuitive understanding of `size` and `repeat`'s behavior or attempt to prove the correctness of size and repeat to prove the thereom. A correct proof will require several steps that cite lines of code, not just specifications.

You can assume that the operations `+` and `-` over `int`s in SML are implemented correctly as addition and subtraction over the integers in the mathematical sense – for example, you can assume that addition is commutative and associative, and has the identity element 0.

[5] In general, it's important to cite that your arguments evaluate to values when stepping through code in your proofs (this is where the concept of ***totality*** comes in handy). Since we haven't covered totality yet, you can omit these citations for this task, ***but you will need to make these citations in future homeworks!***

# 8 A *Spec*ial Problem

Consider the following `div` function:

```
fun div (n : int, d : int) : int =
  if n < d
  then 0
  else 1 + div(n - d, d)
```

**Task 8.1.** (10 points)

Prove that for all values `n : int` and `d : int`, where `n` $\geq 0$ and `d` $> 0$, there exists a value `q : int` such that

$$\texttt{div (n, d)} \implies \texttt{q} \text{ and } q * d \leq n < (q+1) * d.$$

This proof involves a variety of variables from the get-go, both in the code and the theorem. When naming new variables you **must** define and quantify them. If you find a single variable name being used for several contexts, it is more appropriate to use distinct variable names. It is worth noting, however, that you shouldn't need an excessive number of variables (e.g. q' ', q' ' ') for this proof.

***Hint***: Think about the structure of the code when structuring your proof!

**Constraint:** Your inductive step should not have any case-work. Remember: your proof should reflect the structure of your code.

**Constraint:** As with any proof in this class, you may not use any lemmas that are not given to you in the writeup. In particular, you may not use Euclid's division lemma for this task. (If you don't know what that is, then great! You aren't allowed to use it in your proof anyway.)

# 9 FlipFlop

**Task 9.1.** (18 points)

In `code/flipOne/flipOne.sml`, write the function

> `flipOne : bool list -> bool list list`
>
> REQUIRES: true
>
> ENSURES: `flipOne L` $\Longrightarrow$ a list containing all the ways to flip exactly one `true` in `L` to `false`.

If that's confusing, here are some examples:

1. `flipOne [true, true]`
   We can flip either the first or second `true`. The order of the `bool list`s does not matter, so both of the following are acceptable:

   `flipOne [true,true]` $\Longrightarrow$ `[[false,true], [true,false]]`

   `flipOne [true,true]` $\Longrightarrow$ `[[true,false], [false,true]]`

2. `flipOne [false, true]`
   We can flip the second `true`. The first `false` is ignored.

   `flipOne [false,true]` $\Longrightarrow$ `[[false,false]]`

3. `flipOne []`
   There are no `true`s to flip.

   `flipOne [false,false]` $\Longrightarrow$ `[]`

*Hint*: This problem is tougher than it looks! There are many ways to solve it, and there are no restrictions on your solution.

**This assignment has a total of 111 points.**