

Worth: 10%

Due: By 1pm on Wednesday 25 February

For this assignment, you will write a number of different ML functions and data types. You must use good functional style in your code, including good external and internal comments (to explain the purpose of your code and your decisions on how to implement it), as well as good visual layout (formatting, indenting, blank lines) to make your code easy to read and understand.

1. Write a function `squeeze = fn : 'a list -> 'a list` that removes all occurrences of repeated consecutive elements from a list—repeated elements that are not consecutive are retained.

For example,

```
- squeeze [];
stdIn:... Warning: type vars not generalized because of
  value restriction are instantiated to dummy types (X1,X2,...)
val it = [] : ?X1 list
- squeeze [1];
val it = [1] : int list
- squeeze [1,1,1];
val it = [1] : int list
- squeeze [1,2,2,2,1];
val it = [1,2,1] : int list
```

Submit your code for this question in file “`squeeze.sml`”.

2. Write a function `compose = fn : int -> ('a -> 'a) -> 'a -> 'a` such that for all $n \geq 0$, we have `compose n f x = f(n) x = f (f (... (f x)...))`, where there are n applications of `f`, nested appropriately.

Your function should raise a `Negative` exception if $n < 0$, but without testing for this condition more than once on any invocation.

Submit your code for this question in file “`compose.sml`”.

3. Consider the polymorphic singly-linked list data type we defined in class:

```
datatype 'a LL = Nil | Node of 'a * 'a LL;
```

Write a function `multapply = fn : ('a -> 'b) LL -> 'a LL -> 'b LL LL` such that the call `multapply fns args` applies every function in the linked list `fns` to every argument in the linked list `args`, and returns a linked list of the linked lists of results.

More precisely, if `fns = Node(f0, Node(f1, ..., Node(fN, Nil)...))`, then

```
multapply fns args = Node(apply f0 args, Node(apply f1 args,
..., Node(apply fN args, Nil)...))
```

where

```
apply f Node(a0, Node(a1, ..., Node(aK, Nil)...)) =
  Node(f a0, Node(f a1, ..., Node(f aK, Nil)...))
```

Submit your code for this question in file “`multapply.sml`” (include the code for `datatype 'a LL` in your file).

4. Recall the definition of the Fibonacci numbers F_0, F_1, F_2, \dots :

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad \text{for all } n \geq 2.$$

It is possible to extend this definition to negative indices, by requiring that the equality $F_n = F_{n-1} + F_{n-2}$ hold for all $n \in \mathbb{Z}$. In this way, we get:

$$F_{-1} = 1 \quad \text{because } F_1 = F_0 + F_{-1}, \quad F_{-2} = -1 \quad \text{because } F_0 = F_{-1} + F_{-2}, \quad \dots$$

In this question, we generalize the definition further so that there are three base cases instead of two, and each value depends on the three previous ones—the result is a sequence we call the “Tribonacci” numbers:

$$T_{-1} = a, \quad T_0 = b, \quad T_1 = c, \quad T_n = T_{n-1} + T_{n-2} + T_{n-3} \quad \text{for all } n \in \mathbb{Z},$$

where $a, b, c \in \mathbb{Z}$ are arbitrary constants.

Now, it is easy to write an ML implementation of the standard Fibonacci numbers:

```
fun slow_fib 0 = 0
  | slow_fib 1 = 1
  | slow_fib n = fib (n-1) + fib (n-2);
```

However, there are a few drawbacks to this implementation:

- As you know, the value of `slow_fib n` grows exponentially with `n`. (In fact, it quickly exceeds the maximum size of 32-bit integers, and the call `slow_fib 45` results in an `Overflow` exception.)
- As you also know, it is inefficient: the computation of `slow_fib n` requires many repeated calls to `slow_fib k` for values `k < n`. (In fact, the running time for `slow_fib n` grows proportionally with the value of `slow_fib n`, so it is exponential.)

It is relatively easy to deal with the first problem by using a type other than `int` (32-bit integers). ML provides such a type `LargeInt.int` that can store arbitrarily large integer values (limited only by the memory of the computer). To make this easier to use, we can provide a synonym for it:

```
type bigint = LargeInt.int;
```

The second problem is also relatively easy to deal with: instead of using recursion, compute the values using a loop... except that there is no loop in ML! A standard loop to compute Fibonacci numbers would look something like this (in Python):

```
F = [0, 1] # consecutive Fibonacci numbers (initially F_0, F_1)
while n > 1:
    F = [F[1], F[0] + F[1]]
    n -= 1
# At this point, F[n] contains the answer.
```

In ML, the trick is to carry out the same computation using recursion, by writing a function that returns a pair of values—I will let you work out the details for yourself, as part of this question. (Note that the loop above only works for non-negative values of `n`—it would have to be extended in the right way to handle negative values.)

Finally, here is the actual question!

Write a function `slow_trib = fn : bigint * bigint * bigint -> bigint -> bigint` (using the type synonym `bigint` introduced above), such that `slow_trib (a,b,c) n = Tn` (as defined above), for all integers `n` (negative, zero, or positive). Your function should simply follow the recursive definition and it is fine for it to run in time exponential in `n`.

Next, write a second function `trib = fn : bigint * bigint * bigint -> bigint -> bigint` such that `trib (a,b,c) n = Tn` for all integers `n`, except that `trib` must run time *linear* in `n`. Make sure to include enough internal comments to explain/justify that your computation is correct. Submit your code for this question in file “`trib.sml`” (include the code to define type `bigint` in your file).

5. Define a datatype `wff` to represent “well-formed formulas” of the propositional calculus. Your datatype should have constant constructors for the values “True” and “False”, a constructor for propositional variables with one argument of type `string` (for the name of the variable), and one constructor for each of the following propositional connectives:

- \neg (unary negation)
- \wedge (binary conjunction)
- \vee (binary disjunction)
- \Rightarrow (binary implication)
- \Leftrightarrow (binary biconditional)
- \oplus (binary exclusive or)

As you have discovered when working with user-defined data types, the ML interpreter abbreviates output values by not showing expressions deeper than some number of levels. For example,

```
- datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a tree;
datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
- val T = Node (Node (Leaf 3, Leaf 0), Node (Leaf ~1, Node (Leaf 5, Leaf 0)));
val T = Node (Node (Leaf #,Leaf #),Node (Leaf #,Node #)) : int tree
```

The simplest way to be able to see the full structure of expressions is to write a function that turns them into strings, and then print the result.

Write such a function `wffstr = fn : wff -> string` that generates an ASCII string representation of its argument. Use the following conventions:

- “False” is represented by the string “F” and “True” is represented by the string “T”,
- a variable with name `v` is represented simply by `v` itself,
- \neg is represented by “~”,
- \wedge is represented by “ /\ ”,
- \vee is represented by “ \/ ”,
- \Rightarrow is represented by “ => ”,
- \Leftrightarrow is represented by “ <=> ”,
- \oplus is represented by “ <+> ”,

and put parentheses around each sub-formula defined by a binary connective. For example, the string representation of the wff $\neg((p \wedge q) \Rightarrow (p \oplus q))$ should be “~((p /\ q) => (p <+> q))” (the double backslash in “ \/ ” and “ /\ ” is necessary in ML strings because backslash is used as an escape character to represent special symbols—such as “\n” for end-of-line).

The *standard form* of a wff is obtained by applying the following rules repeatedly, as much as possible—until this is no longer possible.

- $\neg(p \oplus q) = p \Leftrightarrow q$
- $\neg(p \Leftrightarrow q) = p \oplus q$
- $\neg(p \Rightarrow q) = p \wedge \neg q$
- $\neg(p \vee q) = \neg p \wedge \neg q$
- $\neg(p \wedge q) = \neg p \vee \neg q$
- $\neg\neg p = p$
- $\neg T = F$
- $\neg F = T$

For example, the standard form of $\neg((p \wedge q) \Rightarrow (p \oplus q))$ is $(p \wedge q) \wedge (p \Leftrightarrow q)$ —if you are not sure that you understand how this was obtained, make sure that you ask (and review your notes from CSC236H).

Write a function `standardize = fn : wff -> wff` that returns the standard form of its argument.

Submit your code for this question (including your data type and both functions) in file “`wff.sml`”.

Bonus!

Write a function `fast_trib = fn : bigint * bigint * bigint -> bigint -> bigint` such that for all integers n , `fast_trib (a,b,c) n = Tn` (see Question 4 for the relevant definitions). Your function must run in time **logarithmic** in n . Justify carefully that your function correctly computes the generalized Tribonacci numbers, and include a careful analysis of your function’s running time.

Submit your code for this question in file “`bonus.sml`” (include the code for type `bigint` in your file).