Computer Science 324 | 22 October, 2004
St. George Campus | University of Toronto

Homework Assignment #3
Due: Monday, 8 November, 2004, by 5pm

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due. This means that no question about this assignment will be answered, whether it is asked on the newsgroup, by email, or in person.*
Total Marks: There are 40 marks available in this assignment.
Handing in this Assignment

What to hand in on paper: Please use an **unsealed** envelope, attaching the cover page provided to the front. Note that without a properly completed and **signed** cover page, your assignment will not be marked. Put inside:

1. A printout of your solutions for all questions.
2. For each function that you write, you must describe the testing strategy that you used. This should include a table listing the test cases that you designed for your function, what output your function returned, and an explanation of why the test case and output are significant in verifying the correctness of the function. Read the following for a discussion of good testing practices:

   `http://www.cs.toronto.edu/~gpenn/csc324/software.testing.pdf`

You must hand in this part of the assignment in the CSC324 drop box in the Bahen Computer Lab, BA2220.

What to hand in electronically: In addition to your paper submission, you must submit your code electronically. The functions (including helper functions) for each part of each individual question are to be submitted separately, with the filename as stated in the question. To submit these files electronically, use the CDF secure website:
`https://www.cdf.utoronto.ca/students`
or use the CDF **submit** command. Type **man submit** for more information.

Warning: marks will be deducted for incorrect submission. Note that if the code submitted electronically differs from the code submitted on paper, we will only mark the electronically submitted version (if, in such a case, you put comments, etc. only on paper, we will mark the question as if no comments, etc. were provided).

Since we will test your code electronically, you must:

- *make certain that your code runs on CDF*,
- use the exact function, constructor and type names and argument specified,
- use the exact file names specified in the questions,
- not use `use` in any of your submitted files,
- not display anything but the function output (no text messages to the user, fancy formatting, etc. — just what is in the assignment handout).

Other Information You must read the requirements for code and marking information on the following web page: `http://www.cs.toronto.edu/~anya/a3guidelines.html`
This document constitutes part of the **official requirements** for this assignment.
FAQ Page and Newsgroup Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 3 FAQ page, linked from your section's CSC324 home page. You are also responsible for monitoring the CSC324 newsgroup.

Question 1. (1 mark) Fill out the cover sheet correctly.

Question 2. (14 marks) For each of the following functions described, state what type ML would assign to a correct implementation of it.

a. (3 marks) A function which, for a given pair of functions `f` and `g`, determines whether $f(g(x), g(y)) = g(f(x, y))$, for all pairs `(x,y)` in a given list (of pairs).

b. (3 marks) A function which takes a comparison predicate and a list as input and returns true if and only if the list is sorted according to the comparison predicate.

c. (3 marks) A function which takes a list of functions, f_1, ..., f_n (for any $n \geq 0$), and returns a function that computes $\lambda x.f_1(f_2(\dots f_n(x)))$ (this is the usual way of writing lambda terms).

d. (5 marks) Same function as `(c)`, but instead of taking a list of functions, it takes a tuple of functions for a fixed $n = 3$. What restrictions on the types of the f_i did you have to make in (c)? Did you have to make those restrictions here?

Do not submit these solutions electronically.

Question 3. (5 marks) Implement a function that works as described in (2b), except that it should take a *certificate* as an extra argument. A certificate is a scheme for proving a claim. Your function should return true when the list is sorted, but it should never return false. Instead, if the list is not sorted, it should raise (as an exception) a proof that the list is not sorted. To prove that a list is not sorted, we only need to exhibit a pair of elements that are not in the proper order. So the exception that you raise should contain such a pair.

Name the function `sorted`, and submit it in a file named `sorted.sml`. You may assume that the comparison operator is irreflexive and transitive.

Example:

```
- fun less(x,y) = x<y;
val less = fn : int * int -> bool

- exception NotSortedInt of int * int;
exception NotSortedInt of int * int

- sorted(less,[1,2,3],NotSortedInt);
val it = true : bool

- sorted(less,[1,3,2],NotSortedInt)
  handle NotSortedInt(e1,e2) => (print(Int.toString(e1)^" "^Int.toString(e2)^"\n");false);
3 2
val it = false : bool

- exception NotSortedIntList of int list * int list;
exception NotSortedIntList of int list * int list

- fun shorter (l1,l2) = length(l1) < length(l2);
val shorter = fn : 'a list * 'b list -> bool

- sorted(shorter,[[1,2],[1,2,3],[1,2,3,4]],NotSortedIntList);
val it = true : bool
```

Question 4. (15 marks) **Unit Conversion** In this question you will use the ML type system to write a simple and straightforward program for measurement conversion. Table 1 shows several quantities along with their units. For each quantity, there is a standard unit. Some quantities have alternative units.

Table 1: Quantities and their units

Quantity	**Standard Unit**	**Alternative Units**
Time	second	hour, minute, day
Power	watt	horsepower
Current	ampere	
Potential	volt	
Resistance	ohm	
Flux	weber	

We have the following conversion rules between these standard units: `Power=Potential×Current, Potential=Resistance×Current, Flux=Potential×Time.`

Also, we have the following prefixes for quantities: `kilo, mega, milli` and `micro`, which multiply the quantity by a factor of 1e3, 1e6, 1e-3, and 1e-6, respectively.

You will define a datatype called "`measure`" that consists of the quantities mentioned in the first column of table 1. To simplify matters, every quantity should implicitly be represented by its standard unit. The value of each unit will be a `real` number. You must use type synonyms to represent standard units.

You will also write four *infix* operators `++` (addition), `--` (subtraction), `**` (product), and `//` (division) over the `measure` type with the usual precedence and associativity. Note that for both operators `++` and `--` the operands must be of the same quantity. Nevertheless, the operands of `**` and `//` operators must conform to one of the conversion rules mentioned above. Note that each conversion rule introduces two possibilities for each of the `**` and `//` operators. For example, the rule `Power=Potential×Current` also implies the following: `Power=Current×Potential`, `Potential=Power/Current`, and `Current=Power/Potential`. If the operands of these operators do not conform to the above rules, you must raise an exception called **UnitUnDef** that carries a `string*measure*measure` tuple. The `string` will be the violating operator (one of `++`,`--`,`**`, or `//`) and the `measure` components will be the operands involved. Your program should not generate any error, nor print any error messages, but rather should raise an appropriate exception.

Here are some examples of *valid* expressions: (Note: read these examples carefully and try all of them, in the given order, to figure out how to design the necessary types/functions. These examples should also remind you of the conversion rates to/from the alternative units like `day, hour, horsepower`, in case you have forgotten.)

```
-val t1=second 1.0;
val t1 = Time 1.0 : measure

-val t2=hour(1.0);
val t2 = Time 3600.0 : measure  (* time is stored internally in seconds *)

-val t3 = t1++minute(20.0);
val t3 = Time 1201.0 : measure

-val r1 = ohm(0.5);
val r1 = Resistance 0.5 : measure
```

```
-val r2 = milli(ohm 50.0);
val r2 = Resistance 0.05 : measure

-val r3 = r1--r2;
val r3 = Resistance 0.45 : measure

-val v1 = micro(volt 150.0);
val v1 = Potential 0.00015 : measure

-val i1 = v1 //r1;
val i1 = Current 0.0003 : measure

-val i2 = mega(ampere 0.1);
val i2 = Current 100000.0 : measure

-val f1 = v1**t2;
val f1 = Flux 0.54 : measure

-val w1 = horsepower 1.0;
val w1 = Power 745.7 : measure

-val w2 = micro(r2**i2**i2);
val w2 = Power 500.0 : measure

-val w3 = kilo (ampere 0.1) ** volt(0.2);
val w3 = Power 20.0 : measure

-val w4 = f1//t1**i2;
val w4 = Power 54000.0 : measure
```

The following are some examples showing that your program should raise `UnitUnDef` exception due to unsupported types:

```
-day(1.5)++watt(1.0);   (* cannot add different quantities  *)
uncaught exception UnitUnDef

-day(0.1)**ohm(1.2);    (* no quantity exists for the product *)
uncaught exception UnitUnDef

-f1**i2//t1;            (* no quantity exists for Flux*Current *)
uncaught exception UnitUnDef
```

Note that although `f1**i2//t1` is mathematically equivalent to the last valid expression `f1//t1**i2` above, nevertheless your program will not support such an expression. This is because, for the sake of simplicity, we require that every intermediate result produced by an application of `//` or `**` must have a unit defined in Table 1.

For this question, hand in a file called `conversion.sml`. This question will be auto-marked, so it is extremely important to use the exact names (including upper/lower cases) provided in Table 1. For your convenience, more test cases with the expected output can be found at the following web page:

`http://www.cs.toronto.edu/~hojjat/324f04/a3/tests.txt`

Question 5. (5 marks) **Expression Tree**
In this question, you will define a datatype called `mtree` for storing an expression in a binary tree and a function called `calc` that evaluates the value corresponding to such a tree. All leaves in an expression tree are of type `measure` (as defined in the previous question), and all internal nodes are one of the `++,--,**,//` operators. Your `calc` function must handle exceptions raised by any of these operators. In response to such exceptions, `calc` should simply return a special `UnDef` value (you will need to modify the `measure` definition a little bit).

Below are some examples with the expected output. Again, read these examples carefully and use the exact constructor/function names provided (as this question will be auto-tested, too).

```
-val t1= Leaf (second 1.0);
val t1 = Leaf (Time 1.0) : mtree

-val t2= Leaf (minute 0.1);
val t2 = Leaf (Time 6.0) : mtree

-val t3= Node (op ++,t1,t2);
val t3 = Node (fn,Leaf (Time 1.0),Leaf (Time 6.0)) : mtree

-calc t1;
val it = Time 1.0 : measure

-calc t2;
val it = Time 6.0 : measure

-calc t3;
val it = Time 7.0 : measure

-val t4 = Node (op //, Leaf (weber 14.0), t3);
val t4 = Node (fn,Leaf (Flux 14.0),Node (fn,Leaf #,Leaf #)) : mtree

-calc t4;
val it = Potential 2.0 : measure

-val t5 = Node(op **, t3, Leaf(ampere 1.5));
val t5 = Node (fn,Node (fn,Leaf #,Leaf #),Leaf (Current 1.5)) : mtree

-calc t5;
val it = UnDef : measure

-val t6 = Node(op ++,t5,t5);
val t6 = Node (fn,Node (fn,Node #,Leaf #),Node (fn,Node #,Leaf #)) : mtree

-calc t6;
val it = UnDef : measure
```

Copy your `conversion.sml` program from the previous question to a new file called `conversiontree.sml`. Make the necessary modifications to the `measure` definition. Add your definitions of `mtree` and `calc` to the file. Submit `conversiontree.sml`.