Computer Science 324 — 24 October, 2008
St. George Campus — University of Toronto

Homework Assignment #2

Due: Thursday, 13 November, 2008, by 6pm

No submissions will be accepted after 6pm Thursday, 13 November

---

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.*

Handing in this Assignment

What to hand in on paper: Please use an **unsealed** envelope, attaching the cover page provided here 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 code. **Write your student number on the first page of this printout**.
2. For each procedure 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 procedure, what output your procedure returned, and an explanation of why the test case and output are significant in verifying the correctness of the procedure. 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 your tutorial on the due date.

What to hand in electronically: In addition to your paper submission, you must submit your code electronically. Use this command: `submit -c csc324h -a A2 a2.sml`, from the directory where `a2.sml` lives. Type **man submit** for more information. You can also use the CDF secure website: `https://www.cdf.utoronto.ca/students`.

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 names and argument(s) (including the order of arguments) specified,
- use the exact file name specified (`a2.sml`),
- not load any other files from your submitted file, and
- 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/~jward/a2guidelines.html` This document constitutes part of the official requirements for this assignment.

Bulletin Board: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the bulletin board, linked from the course home page.

Computer Science 324	24 October, 2008
St. George Campus	University of Toronto

Homework Assignment #2
Cover Sheet

Last Name:

First Name:

CDF login:

Email:

I have read, understood, and agree to the policies described in the Course Information handout, including the policy on collaboration.

Signature: ____________________

Question 1. (16 marks) Consider the following variant datatype:

```
type succ = z | s of succ;;
```

This simple type is actually enough to encode all of the natural numbers: `z` represents 0, `s(z)` represents 1, `s(s(z))` represents 2, etc.

Implement the four basic arithmetic operations on this data-type:

```
add: succ * succ -> succ
subtract: succ * succ -> succ
multiply: succ * succ -> succ
divide: succ * succ -> succ * succ
```

`divide` returns a pair consisting of the integer quotient (left) and modulus (right) of the division of its left input argument by its right input argument. You'll need to define two exceptions: `Underflow`, to raise when the second argument to `subtract` is greater than the first, and `DivbyZero`, to raise when the second argument to `divide` is `z`.

Hint: implement `divide` by protecting its recursive call with an `Underflow` exception handler.

Question 2. (30 marks) Consider the following datatype for floating-point polynomials:

```
type polynomial = poly of int * float list;;
exception DegreeLessthanZero of polynomial;;
exception WrongNumFactors of polynomial;;
exception ZeroLeadingTerm of polynomial;;
```

The integer represents the degree of the polynomial, and the list of floating point numbers represents the coefficients of each term in order, from the polynomial's degree down to 0. Not every pair of integer and float list corresponds uniquely to a polynomial. The three exceptions defined here itemize the three reasons that it may not be: the degree is less than zero, there are too many or too few floats in the list, and/or the leading term of the polynomial is zero when the degree is not zero — if we allowed this, polynomials would not have unique representations.

(a) (3 marks) Define a function `length : 'a list -> int` that returns the length of its list argument. Your implementation should be tail-recursive and run in linear time.

(b) (3 marks) Define a function `reverse: 'a list -> 'a list` that returns the reverse of its list argument. Your function should be tail-recursive and run in linear time.

(c) (3 marks) Define a function `wf_poly: polynomial -> int * float list` that confirms that its polynomial argument is well-formed. If the polynomial is well-formed, it should return the two components of the structure. If not, it should raise one of the above three exceptions, as appropriate.

(d) (4 marks) Define a function `deriv: polynomial -> polynomial` that returns the first derivative of its argument. Don't forget to call `wf_poly`.

(e) (4 marks) Define a function `defintegral: polynomial -> polynomial` that returns the polynomial g(x) that represents the integral of its argument from 0 to x. Again, be sure to call `wf_poly`. Run some tests to ensure that `deriv(defintegral(p))=p` and that `defintegral(deriv(p))` differs from `p` by at most a constant term.

(f) (6 marks) Define a function `reify: polynomial -> float -> float` that returns a CaML function that calculates the value of the input polynomial argument on its own floating-point argument. Call `wf_poly`.

(g) (7 marks) Define a function `add_poly: polynomial * polynomial -> polynomial` that returns the polynomial representing the sum of its two polynomial arguments. **Hint:** use `reverse` from (b). As usual, call `wf_poly` on both arguments.

Question 3. (44 marks) In this question, you will write code to 2-colour an undirected graph, by converting it to a functional representation in which nodes pass messages to each other to enforce the typical constraint that no two adjacent nodes may have the same colour. Let us begin with some datatypes:

```
type colour = black | white | unknown;;
type reply = c of colour | ok of unit;;
type message = status | link of (message -> reply) | paint of colour;;
type edge = e of int * int;;
type adjGraph == edge list;;
type lambdaGraph == (message -> reply) list;;
```

You will ultimately take an *adjacency representation* of a graph as input, convert it to the *lambda representation*, which is the one in which nodes are represented by functions that pass messages to each other, and then colour that graph. In the adjacency representation, nodes are represented by positive integers. If someone gives us a graph that uses other integers, we must correct them:

```
exception NonPosInt of int;;
exception NotTwoColourable;;
```

Not all graphs are 2-colourable. Odd cycles such as this one are never 2-colourable:

```
let oddcycle = [e(1,2);e(2,3);e(3,1)];;
```

Acyclic graphs, like this tree, are always 2-colourable:

```
let tree = [e(1,5);e(2,5);e(3,6);e(4,6);e(5,7);e(6,7)];;
```

If the input graph is not 2-colourable, you must raise the exception, `NotTwoColourable`.

In the functional representation, there are three kinds of messages that you can pass to your nodes. The first (`status`) asks, "what is your colour?" Notice that there is a value in the datatype `colour` for `unknown`, meaning that a colour has not been assigned yet. The second kind of message (`link`) tells the node to link itself to another node. Whenever a node is assigned a colour, it will broadcast a message to the nodes that it is linked to, assigning them the opposite colour. We will be linking adjacent nodes to each other, thus enforcing the 2-colouring constraint. The third message (`paint`) tells the node what colour it has been assigned.

Every node must reply to every message it receives. The reply to a `status` message should be the colour it is assigned (or `unknown`). The reply to a `link` message should be `ok` — this has a `unit` argument to make your code more readable, since you can return `ok(`*command*`)`, where *command* is a side-effect-producing command that evaluates to the unit object. The reply to a `paint` message should also be `ok`.

The functional representations of nodes must remember their known neighbours and their colour value in between function calls. This means that they will have local reference variables like the object-oriented stack implementation that we covered in lecture. Now you have variant datatypes, however, so your implementation will not be so poor.

You will probably need these:

```
let unit_of_ok = function ok(()) -> ();;
let opposite = function white -> black
                      | black -> white
                      | unknown -> unknown;;
let compose(f)(g) = (function x -> f(g(x)));;
```

Composing `unit_of_ok` with a message-pass to a node converts the `ok` to a unit object, which makes it easier to throw away without CaML complaining.

(a) (4 marks) Write a function `nth: int -> 'a list -> 'a` such that `nth(n)(list)` returns the `n`th element of `list`, numbered left-to-right beginning with 1. If the integer is out of range, you must raise an exception. Define your own exception(s) for this purpose. Your code should be tail-recursive and run in linear time.

(b) (4 marks) Write a function `max_list: 'a list -> 'a` that returns the maximum element in a list (of integers — although the built-in `max` does not restrict itself to integers). You will need this to find the largest integer in the adjacency representation of your input graph, which will tell you how many nodes the graph has. Your code should be tail-recursive and run in linear time.

(c) (25 marks) Write a function `init_lambdaG: int -> (message -> reply) list`, which initializes the functional representation of a graph with the number of nodes given by its argument, n. This will be a list of length n, each member of which is a *different* function that does not know its colour and has no known neighbours. Be sure that you initialize every position to a different such function — if you use a local variable, for example, it will be very easy to bind every position in the list to the same function, which would imply that all nodes must have the same colour.

(d) (6 marks) Define a function `link_edges: (adjGraph * lambdaGraph) -> unit` which links all of the nodes in the `lambdaGraph` according to the edges specified in the `adjGraph`. Remember that you are dealing with undirected graphs here, so link in both directions.

(e) (5 marks) Define a function `lambdaG_of_adjG: adjGraph -> lambdaGraph` which converts an adjacency representation of a graph to the functional representation. This should find the number of nodes with `max_list`, call `init_lambdaG` to initialize the functional representation, and then add then link the nodes using `link_edges`.

(f) (10 marks) Finally, write the function `two_colour: adjGraph -> colour list`, which colours its input graph by calling `lambdaG_of_adjG`, and then assigning colours to the nodes of the resulting functional representation. How many nodes must you explicitly assign colours to here? Remember that the input graphs may not be connected, i.e., some nodes may not be accessible from others along paths of edges.

This function will return a list of colours. The n^{th} position in the list indicates the colour assigned to node number n in the original adjacency representation. It does not matter which colours you assign to which nodes, as long as you assign every node either `black` or `white`, and no two adjacent nodes the same colour. Here is a sample transcript:

```
#let oddcycle = [e(1,2);e(2,3);e(3,1)];;
oddcycle : edge list = [e (1, 2); e (2, 3); e (3, 1)]
#let tree = [e(1,5);e(2,5);e(3,6);e(4,6);e(5,7);e(6,7)];;
tree : edge list =
 [e (1, 5); e (2, 5); e (3, 6); e (4, 6); e (5, 7); e (6, 7)]
#two_colour oddcycle;;
Uncaught exception: NotTwoColourable
#two_colour tree;;
- : colour list = [white; white; white; white; black; black; white]
```