Department of Computer Science and Engineering
University of California at San Diego

CSE 230
Winter 1998

Problem Set 1 Solutions

1. [Solution due to Deborah Swanberg.] The function can easily be defined as

```
fun fulldiv (x, y) = (x div y, x mod y);
```

But, this will raise a generic exception if `y = 0`, and the function will stop executing.

An exception can be declared that will be used just for this function, and a new function can be defined that raises this exception:

```
exception ZeroDiv;

fun fulldiv (x,y) =
    if (y = 0)
       then raise ZeroDiv
       else (x div y, x mod y);
```

Or, the function `fulldiv` could be defined with a pattern:

```
fun fulldiv (x,0) = raise ZeroDiv
  | fulldiv (x,y) = (x div y, x mod y);
```

The problem is that the user must check for the exception. The function `fulldiv` cannot reasonably handle the exception, because it would have to return a value for the function on that exception. As defined, it must always return an `int*int` value, and there is no possible valid return value of this type when `y=0`.

We would like a solution that captures the error, and handles it in some way, without requiring the user to intervene. A data structure can be used that represents errors explicitly and the function can then propagate errors. The following definition achieves this:

```
datatype myint = Error | Valid of int;

fun fulldiv (x, Error) = (Error, Error)
  | fulldiv (Error, y) = (Error, Error)
  | fulldiv (x, Valid(0)) = (Error, Error)
  | fulldiv (Valid(x), Valid(y)) =
           (Valid (x div y), Valid (x mod y));
```

This function can also be defined with `if` conditions, which check for all the values. In this case, though, a function is needed which can return the integer value of `myint`, because patterns cannot be used directly in expressions, only in function definitions. One advantage of patterns is that they spare the programmer from having to write type conversion functions.

For example, the `fulldiv` definition above could be defined as:

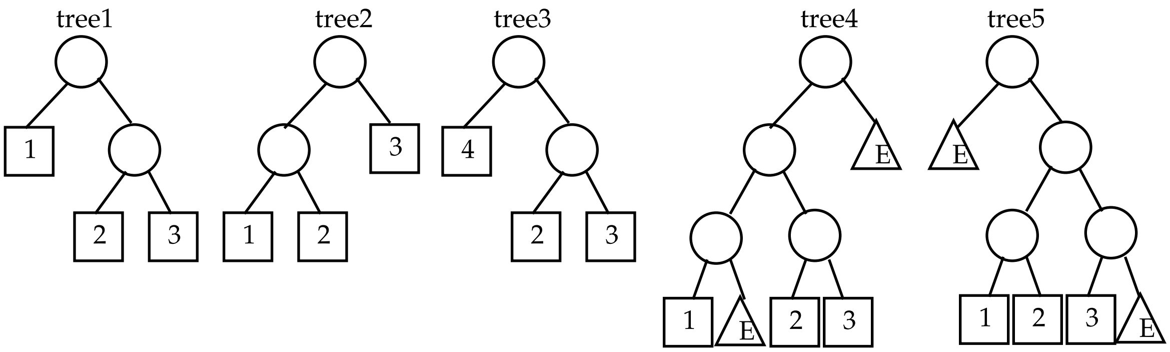
```
fun intofmyint Valid(x) = x;

fun fulldiv (x, y) =
    if y = Error
       then (Error, Error)
    else if x = Error
       then (Error, Error)
    else if y = Valid(0)
       then (Error, Error)
    else (Valid (intofmyint(x) div intofmyint(y)),
          Valid (intofmyint(x) mod intofmyint(y)));
```

2. [Solution due to David R. Nadeau.] The idea is to walk the tree to produce a list of leaf values:

```
fun fringe ( empty )      = []
  | fringe ( leaf(i) )    = [i]
  | fringe ( node(l,r) ) = fringe( l ) @ fringe( r );
```

Sample data:

```
val tree1 = node( leaf( 1 ), node( leaf( 2 ), leaf( 3 ) ) );
val tree2 = node( node( leaf( 1 ), leaf( 2 ) ), leaf( 3 ) );
val tree3 = node( leaf( 4 ), node( leaf( 2 ), leaf( 3 ) ) );
val tree4 = node( node(node(leaf(1),empty),
    node( leaf(2),leaf(3))),empty);
val tree5 = node( empty,node(node(leaf(1),leaf(2)),
    node(leaf(3),empty)));
```



Sample invocations:

```
- fringe( empty );
val it = [] : int list
```

```
- fringe( leaf(1) );
val it = [1] : int list
- fringe( tree1 );
val it = [1,2,3] : int list
- fringe( tree2 );
val it = [1,2,3] : int list
- fringe( tree3 );
val it = [4,2,3] : int list
- fringe( tree4 );
val it = [1,2,3] : int list
- fringe( tree5 );
val it = [1,2,3] : int list
```

3. [Solution due to David R. Nadeau.] The idea is to walk the two trees, comparing their fringes and returning true if they are the same, false if they are not:

```
fun samefringe( t1, t2 ) =
    let
        fun refine( [], [] ) =
                true

        |   refine( empty::xlist, ylist ) =
                refine( xlist, ylist )

        |   refine( xlist, empty::ylist ) =
                refine( xlist, ylist )

        |   refine( node(xnode1,xnode2)::xlist, ylist ) =
                refine( xnode1::xnode2::xlist, ylist )

        |   refine( xlist, node(ynode1,ynode2)::ylist ) =
                refine( xlist, ynode1::ynode2::ylist )

        |   refine( [], ylist ) =
                false

        |   refine( xlist, [] ) =
                false

        |   refine( leaf(xleaf)::xlist, leaf(yleaf)::ylist ) =
                if (xleaf <> yleaf) then false
                else refine( xlist, ylist );
    in
        refine( [t1], [t2] )
    end;
```

The function `samefringe` places the root of each tree in a list, then invokes the private function `refine`. That function progressively refines the node lists by expanding `node` elements into their children and eliminating `empty` elements. On `leaf` elements, the leaf values are compared. If they differ, `refine` falls back in its recursion, returning `false`. If they are the same, `refine` checks the remainder of the lists. If both lists are exhausted at the same time `refine` returns `true`, otherwise `false`.

The same sample data as in problem 2 is used here. Sample invocations:

```
- samefringe( empty, empty );
val it = true : bool
- samefringe( empty, leaf(1) );
val it = false : bool
- samefringe( empty, tree1 );
val it = false : bool
- samefringe( tree1, tree2 );
val it = true : bool
- samefringe( tree1, tree3 );
val it = false : bool
- samefringe( tree4, tree5 );
val it = true : bool
```

The worst case execution time occurs when both argument trees are fully searched and found to have matching fringes.

Let the argument trees be T_1 and T_2. Let the count of tree elements for each tree be $n_1 = nodes(T_1) + leaves(T_1) + empties(T_1)$, and $n_2 = nodes(T_2) + leaves(T_2) + empties(T_2)$, respectively. During refinement, each `node`, `leaf`, or `empty` is added to a list once, and removed from a list once. `leaf` elements are removed in pairs, one from each list, while `node` elements and `empty` elements are removed singly.

List addition and removal are the "inner-loop" operations of the algorithm, and require constant time. (Addition and removal occurs only from the front of a list, avoiding the `@` operator and its artifacts–see problem 13.6, p. 143 of *Elements of ML Programming*.) Total execution time is thus

$$O(2(nodes(T_1) + nodes(T_2) + empties(T_1) + empties(T_2)) + 1.5(leaves(T_1) + leaves(T_2)))$$

which we can simplify to just $O(n_1 + n_2)$.