Data structures with datatype

A shape datatype

An expression model

An infinite lazy list

A simple datatype

New types can be defined with the datatype declaration. Example:

```
- datatype Shape =
     Circle of real
   | Square of real
   | Rectangle of real * real
   | Point;
datatype Shape
  = Circle of real | Point | Rectangle of real * real | Square of real
```

This defines a new type named Shape. An instance of a Shape is a value in one of four forms:

A Circle, consisting of a real (the radius)

A Square, consisting of a real (the length of a side)

A Rectangle, consisting of two reals (width and height)

A Point, which has no data associated with it. (Debatable, but good for an example.)

Shape: a new type

At hand:

```
datatype Shape =
    Circle of real
  | Square of real
  | Rectangle of real * real
  | Point
```

This declaration defines four *constructors*. Each constructor specifies one way that a Shape can be created.

Examples of constructor invocation:

```
- val r = Rectangle (3.0, 4.0);
val r = Rectangle (3.0,4.0) : Shape

- val c = Circle 5.0;
val c = Circle 5.0 : Shape

- val p = Point;
val p = Point : Shape
```

Shape, continued

A function to calculate the area of a Shape:

```
- fun area(Circle radius) = Math.pi * radius * radius
   |  area(Square side) = side * side
   |  area(Rectangle(width, height)) = width * height
   |  area(Point) = 0.0;
val area = fn : Shape -> real
```

Usage:

```
- val r = Rectangle(3.4,4.5);
val r = Rectangle (3.4,4.5) : Shape

- area(r);
val it = 15.3 : real

- area(Circle 1.0);
val it = 3.14159265359 : real
```

Speculate: What will happen if the case for Point is omitted from area?

Shape, continued

A Shape list can be made from any combination of Circle, Point, Rectangle, and Square values:

```
- val c = Circle 2.0;
val c = Circle 2.0 : Shape

- val shapes = [c, Rectangle (1.5, 2.5), c, Point, Square 1.0];
val shapes = [Circle 2.0,Rectangle (1.5,2.5),Circle 2.0,Point,Square 1.0]
  : Shape list
```

We can use map to calculate the area of each Shape in a list:

```
- map area shapes;
val it = [12.5663706144,3.75,12.5663706144,0.0,1.0] : real list
```

What does the following function do?

```
- val f = (foldr op+ 0.0) o (map area);
val f = fn : Shape list -> real
```

A model of expressions using datatype

Here is a set of types that can be used to model a family of ML-like expressions:

```
datatype ArithOp = Plus | Times | Minus | Divide;

type Name = string        (* Makes Name a synonym for string *)

datatype Expression =
    Let of (Name * int) list * Expression
  | E   of Expression * ArithOp * Expression
  | Seq of Expression list
  | Con of int
  | Var of Name;
```

Note that it is recursive—an Expression can contain other Expressions.

Problem: Write some valid expressions.

Expression, continued

The expression 2 * 4 is described in this way:

```
E(Con 2, Times, Con 4))
```

Consider a function that evaluates expressions:

```
- eval(E(Con 2, Times, Con 4));
val it = 8 : int
```

The Let expression allows integer values to be bound to names. The pseudo-code

```
let a=10, b=20, c=30
in a + (b * c)
```

can be expressed like this:

```
- eval(Let([("a",10),("b",20),("c",30)],
      E(Var "a", Plus, E(Var "b", Times, Var "c"))));
val it = 610 : int
```

Expression, continued

Let expressions may be nested. The pseudo-code:

```
let a = 1, b = 2
in a + ((let b = 3 in b*3) + b)
```

can be expressed like this:

```
- eval(Let([("a",1),("b",2)],
        E(Var "a", Plus,
          E(Let([("b",3)],   (* this binding overrides the first binding of "b" *)
            E(Var "b", Times, Con 3)), Plus, Var "b"))));
val it = 12 : int
```

The Seq expression allows sequencing of expressions and produces the result of the last expression in the sequence:

```
- eval(Seq [Con 1, Con 2, Con 3]);
val it = 3 : int
```

Problem: Write eval.

Expression, continued

Solution:

```
fun lookup(nm, nil) = 0
    | lookup(nm, (var,value)::bs) = if nm = var then value else lookup(nm, bs);

fun eval(e) =
    let
        fun eval'(Con i, _) = i
          | eval'(E(e1, Plus, e2), bs) = eval'(e1, bs) + eval'(e2, bs)
          | eval'(E(e1, Minus, e2), bs) = eval'(e1, bs) - eval'(e2, bs)
          | eval'(E(e1, Times, e2), bs) = eval'(e1, bs) * eval'(e2, bs)
          | eval'(E(e1,Divide,e2), bs) = eval'(e1, bs) div eval'(e2,bs)
          | eval'(Var v, bs) = lookup(v, bs)
          | eval'(Let(nbs, e), bs) = eval'(e, nbs @ bs)
          | eval'(Seq([ ]), bs) = 0
          | eval'(Seq([e]), bs) = eval'(e, bs)
          | eval'(Seq(e::es), bs) = (eval'(e,bs); eval'(Seq(es),bs))
        in
                eval'(e, [ ])
    end;
```

How can eval be improved?

An infinite lazy list

A *lazy list* is a list where values are created as needed.

Some functional languages, like Haskell, use *lazy evaluation*—values are not computed until needed. In Haskell the infinite list 1, 3, 5, ... can be created like this: [1,3 ..].

```
% hugs
Hugs> head [1,3 ..]
1

Hugs> head (drop 10 [1,3 ..])
21
```

Of course, you must be careful with an infinite list:

```
Hugs> length [1,3 ..]
(...get some coffee...check mail...^C)
{Interrupted!}

Hugs> reverse [1,3 ..]
ERROR - Garbage collection fails to reclaim sufficient space
```

An infinite lazy list, continued

ML does not use lazy evaluation but we can approach it with a data structure that includes a function to compute results only when needed.

Here is a way to create an infinite head/tail list with a datatype:

```
datatype 'a InfList = Nil
                    |       Cons of 'a * (unit -> 'a InfList)

fun head(Cons(x, _)) = x;
fun tail(Cons(_, f)) = f();[1]
```

Note that 'a is used to specify that values of any (one) type can be held in the list.

A Cons constructor serves as a stand-in for op::, which can't be overloaded.

Similarly, we provide head and tail functions that mimic hd and tl but operate on a Cons.

---

[1] Adapted from *ML for the Working Programmer* L.C. Paulson

An infinite lazy list, continued

```
datatype 'a InfList = Nil
        |   Cons of 'a * (unit -> 'a InfList)

fun head(Cons(x,_)) = x;
fun tail(Cons(_,f)) = f();
```

Here's what we can do with it:

```
- fun byTen n = Cons(n, fn() => byTen(n+10));
val byTen = fn : int -> int InfList

- byTen 100;
val it = Cons (100,fn) : int InfList

- tail it;
val it = Cons (110,fn) : int InfList

- tail it;
val it = Cons (120,fn) : int InfList
```

Try it!

An infinite lazy list, continued

More fun:

```
fun toggle "on" = Cons("on", fn() => toggle("off"))
  | toggle "off" = Cons("off", fn() => toggle("on"))
```

```
- toggle "on";
val it = Cons ("on",fn) : string InfList

- tail it;
val it = Cons ("off",fn) : string InfList

- tail it;
val it = Cons ("on",fn) : string InfList

- tail it;
val it = Cons ("off",fn) : string InfList
```

Problem: Write drop(L,n):

```
- drop(byTen 100, 5);
val it = Cons (150,fn) : int InfList
```

An infinite lazy list, continued

Problem: Create a function repeatValues(L) that infinitely repeats the values in L.

```
- repeatValues;
val it = fn : 'a list -> 'a InfList

- repeatValues (explode "pdq");
val it = Cons (#"p",fn) : char InfList

- tail it;
val it = Cons (#"d",fn) : char InfList

- tail it;
val it = Cons (#"q",fn) : char InfList

- tail it;
val it = Cons (#"p",fn) : char InfList

- tail it;
val it = Cons (#"d",fn) : char InfList
```