Haskell, Part 5

CS 440: Programming Languages and Translators, Spring 2020

2/3 p.12,13-14 (activity solution)

Chapter 5: Recursion

A. *Tail Recursion*

- You've seen / used recursion before, so I won't go over that. There is one topic to mention, because it can affect efficiency: Tail calls and tail recursion
- A **tail call** for procedure *P* is a call to some routine *R* where when *R* returns to *P*, then *P* returns immediately.
- A **tail-recursive call** is a recursive call that is a tail call.
- Example: In `factorial n = if n > 1 then n * factorial(n-1)` …, the recursive call is not in tail position because there's a multiplication after the call. With `f n = if n > 1 then f` (*expr*) …, the recursive call to `f` *is* a tail call because `f n` just returns what `f` (*expr*) returns.
- In Haskell, since all functions return values, a tail call is a call whose result we immediately return.
- The reason tail recursion is interesting is that tail-recursive calls can be optimized: Since all `f n` does is return `f` (*expr*), it doesn't need to keep the argument data or local data in the runtime execution stack frame, so for the recursive call, instead of opening a whole new empty frame for the recursive call, we can reuse the stack frame for `f n`.

Example: Tail-Recursive Factorial

- The tail-recursive version of `factorial` below uses a helper routine `factorial'`. The key is that one of the parameters for `factorial'` is a running partial result.

```
factorial n = factorial' n 1       -- initialize partial result to
factorial' n pr | n <= 1 = pr
                | otherwise factorial' (n-1) (n*pr)
```

- Let me abbreviate `factorial'` to just `f'`, then if we take `factorial 5` as an example,

```
factorial 5
    = f' 5 1     -- initial partial result is 1
    = f' 4 5     -- 1*5 = 5
    = f' 3 20    -- 4*5 = 20
    = f' 2 60    -- 3*4*5 = 60
    = f' 1 120   -- 2*3*4*5 = 120
    = 60         -- 1*2*3*4*5 = 120
```

- Technical note: the partial result `pr` = n_0! / `n`! where n_0 is the starting value of `n`. We get `pr` = `5!/5!` = `1`, `5!/4!` = `5`, `5!/3!` = `20`, `5!/2!` = `60`, and `5!/1!` = `120`.
- A savvy compiler can detect a tail recursive function like `factorial'` and optimize it so that it uses the same memory locations for `n` and `pr` for every recursive call, and when the first return from `factorial'` occurs, instead a sequence of returns from `n` = 1 to `n` = 2 to … to `n` = 5, we can use just one return from `n` = 1 to `n` = 5.

Example: Tail Recursive List Reversal

- Here are two definitions for routines that reverse a list: `rev1` is the `reverse'` function on p.4 of Ch.5 of LYaH, and `rev2` (with `rev2'`) is a tail-recursive reversal routine.

```
rev1 [] = []
rev1 (x:xs) = rev1 xs ++ [x]

rev2 = rev2' []              -- abbrev. for rev2 x = rev2' [] x
rev2' res [] = res           -- res = result
rev2' res (h:t)              -- move head of remaining list
   = rev2' (h:res) t         --     to head of running result
```

Execution speed: `rev2` is asymptotically faster than `rev1`

- A list concatenation *list1* ++ *list2* takes time linear to the length of *list1*: If list1 is of length n, it takes $O(n)$ time to traverse *list1* and then $O(n)$ as you go backward through *list1* adding elements to your result.
- So `rev1` takes quadratic time: The recursive calls of `rev1` take $O(1) + O(2) + O(3) + \ldots + O(n-1) = O(n^2)$ time.
- But `rev2` takes linear time. The `rev2'` routine takes $O(n)$ to go through the elements of the list, and with each element it's taking $O(1)$ time to add that element to the head of the partial result.
- You can really see the difference in Haskell. On my laptop,
  - For 10,000 elements: `last(rev1[1..10000])` and `last(rev2[1..10000])` take almost the same time, with `rev1` being noticeably a bit slower.
  - However, on 40,000 elements, `rev1` takes about 90 seconds while `rev2` is almost instantaneous.
  - On a million-element list, `rev2` is still almost instantaneous but with `rev1`, I gave up (my guess is that it would take 15 minutes).
  - On a ten-million-element list `rev2` takes about 4 seconds. I don't want to think about `rev1`.
- The moral of the story is that when people say a recursive program is slow, it's worth looking for a tail-recursive phrasing of its algorithm.

Tail Recursion is How You Write a Loop in Haskell

- Programs that use a loop in an assignment-oriented language can be phrased as tail recursive in a functional language like Haskell. If the compiler does tail-recursion optimization, then the tail-recursive routine does the same amount of work with its recursive calls and base-case testing as the loop-using program does with its top-of-loop jump and termination tests.

- Here's a very rough sketch of a general loop-using routine in a C-like language.

```
// Loop with parameter x, loop variable k, accumulating result a
f(x) {
    int k = expr0, a = expr1;      // a = eventual answer
    while (!done(k,a,x)) {
        k = expr2 ;
        a = expr3 ;
        x = expr4 ;
    }
    return a;
}
```

- Here is a tail-recursive equivalent. Claim: the *n*'th iteration of the `while` loop above and the *n*'th call of `f'` below have the same values for `k`, `a`, `x`. (Provable by induction on *n*.)

```
f x = f' expr0 expr1 x
f' k a x
    | done k a x = a;
    | otherwise =
        let k' = expr2;   // see note
            a' = expr3'; // see note
            x' = expr4'   // see note
        in
            f' k' a' x'
```

- **Notes**:
  - In the C version, if $expr_3$ uses `k`, then in the tail-recursive routine, $expr_3$', will have to take that into account. Similarly, if $expr_4$ uses `k` or `a`, then $expr_4$' must take that into account.
  - In the tail-recursive routine, the name `k'` is used instead of `k` because if `k` also appears in $expr_2$, then Haskell's lazy evaluation makes the two `k`'s the same and the new `k` is defined recursively. E.g., `let k = 'a':k` does not make a new `k` that's one character longer than the old `k`, it defines a `k` that's an infinitely long string of `a`'s. Similarly, the names `a'` and `x'` are used to avoid clashing with an `a` or `x` in $expr_3$ or $expr_4$.

Chapter 8: Making Our Own Types and Typeclasses

B. Datatype Declarations

Quick Overview of What You Can Do With `data` Declarations

- In Haskell, `data` declarations let you define types like
  - *Enumerations*: `data Color = Red | Blue`
  - *Tuple-like structures*: `data Point = MkPoint Float Float`
  - *Unions (alternatives)*: `data IorC = IntVal Int | CharVal Char`

- *Nested structures*: `data Triangle = MkTriangle Point Point Point`
- *Parameterized structures*:
  - `data Either a b = Left a | Right b`
  - `data Maybe a = Nothing | Just a`
- *Recursive structures*: `data Intlist = Node Int Intlist | Nil`

The `deriving` Clause for `data` Declarations

- When you define a datatype, you often want to print them or test them for equality or other things.
- You can attach a `deriving` clause to a `data` definition that tells the compiler to generate code for some stock typeclasses: `Read`, `Show`, `Eq` are very common; `Ord`, `Bounded`, and `Enum` are for datatypes that are more number-like..
- **Examples**:

```
data Color = Red | Blue deriving (Read, Show, Eq, Ord, Bounded, Enum)
data Point = MkPoint Float Float deriving (Read, Show, Eq, Ord)
```

- With the declarations above, you can build `Color` and `Point` values and print them out at top level, and you get tests like `Red` < `Blue`, and `MkPoint 1 2` == `MkPoint 1.0 2.0`.

```
> Blue              -- causes error without deriving Show
Blue
> Point 1 2         -- causes error without deriving Show
Point 1.0 2.0
```

Enumeration Types

- Enumerations are probably the simplest datatypes you can declare.

```
data Color = Red | Green | Blue
        deriving (Read, Show, Eq, Ord, Enum)
```

- The names `Red`, `Green`, and `Blue` are **data constructor constants**; you can use them as expressions and in patterns. E.g., each case of `next` below has the form next *pattern* = *expression*

```
next :: Color -> (Color, Color)
next Red = (Green, Blue)
next Green = (Blue, Red)
next Blue = (Red, Green)
```

C. Simple Tuple-like Structure Types

- Here's a declaration for an `RGB` color datatype. On the left, `RGB` is being defined as the name of the type; on the right, `MakeRGB` is a **data constructor function**, `MakeRGB :: Int -> Int -> Int -> RGB`.

```
:{
| data RGB = MakeRGB Int Int Int -- Red, Green, Blue
|     deriving (Read, Show, Eq)
:}
```

```
> :t MakeRGB
MakeRGB :: Int -> Int -> Int -> RGB
```

- People often make the type name and constructor name the same. (The LYaH book does this, for example.)
    - E.g., `data RGB = RGB Int Int Int` … *etc.*
    - You don't have to do this; people do it because it saves them from thinking up a name for the constructor.
    - If it's been done, we have to use the context in which RGB appears to determine whether we want the type or the constructor. E.g., `RGB` is a type in `f :: RGB -> Int`, but it would be a function in `let c = RGB 64 64 128 in` ….
- **Syntax note**: The names of types (like `RGB`) and type constructors always begins in upper case. Data constructor constants (like `Blue`) and data constructor functions (like `MakeRGB`) also begin in upper case whereas everyday id and function names begin in lower case.

D. How Do I Get the Parts of an RGB?

- **Question**: Say `c = MakeRGB 10 20 30`. Given `c`, how do you get, say, the redness of `c`?
- **Answer**: **You must use pattern matching** to directly get the parts of a data value.
- In patterns, data constructor functions tell the compiler what type of value to match with. (E.g., we use `MakeRGB` if we want the parts of an `RGB`.)
- Also, we need to write patterns in the parameter positions of the data constructor function. E.g., the constructor function name `MakeRGB` gets three patterns for the three components of an `RGB`; `IntVal` and `CharVal` both get one pattern.

```
> redness (MakeRGB r _ _) = r             -- MakeRGB in pattern
> brightness (MakeRGB r g b) = r + g + b  -- MakeRGB in pattern
> c = MakeRGB 10 20 30                    -- MakeRGB as function
> redness c
10
> brightness c
60
```

- **You can't write** something like `c .` *fieldname* to retrieve one of the fields of `c`. Since dot means function composition and `c` isn't a function, you get an error message. (Of course, you can write a function that takes a data value and returns one of its parts: Above, `redness` of an `RGB` was an example.
- Pattern matching also gets used if you want functions that take multiple `data` values.

```
:{
addRGB :: RGB -> RGB -> RGB
addRGB (MakeRGB r1 g1 b1) (MakeRGB r2 g2 b2)
    = MakeRGB (r1+r2) (g1+g2) (b1+b2)
:}
> c = MakeRGB 10 20 30
> addRGB c c
```



```
MakeRGB 20 40 60
```

- Of course, there's no law that says you have to do the pattern match in the function header:

```
:{
| addRGB :: RGB -> RGB -> RGB
| addRGB c1 c2 =
|     let MakeRGB r1 g1 b1 = c1 in
|     let MakeRGB r2 g2 b2 = c2 in
|     let r = r1 + r2 ;
|         g = g1 + g2 ;
|         b = b1 + b2 in
|     MakeRGB r g b
| :}
> addRGB c c
MakeRGB 20 40 60
```

E. Unions: `data` *Declarations With Alternatives*

- In CS generally, union types give you a choice between multiple alternatives. Technically, enumeration types are a kind of union (`Red` *or* `Blue` *or* `Green`), but people generally use the term to refer to types that include data within a structure.
  - E.g., in C, `union` types are written like `struct` types (except with `union` instead of `struct`).
  - In `struct` types, the fields are laid out in memory sequentially.
  - In `union` types, fields are allocated on top of each other; e.g., a union of integer and float stores either an integer or float but not both. In C unions, there's no way to know which alternative is actually stored; you have to know by looking at other data or by where you are in your program. Because of this, C unions are said to be **non-disjoint unions**.
- In Haskell, the `data` declaration we can types with alternatives by separating them with vertical bar. (Hence `Red | Blue | Green` for `Color`.)
- Haskell unions are disjoint: We can always look at a value and see which alternative it comes from by doing a pattern match. (So we take a `Color` value and match against `Red`, `Blue`, or `Green`.)
- To get alternatives with data, we add types after a constructor name. Here's code that declares a datatype `IorC` and then uses it. An `IorC` value holds either an `Int` or a `Char`.

```
> data IorC = IntVal Int | CharVal Char
> kindOfIorC (IntVal _) = "an int" ; kindOfIorC (CharVal _) = "a char"
> ioc1 = IntVal 17
> ioc2 = CharVal 'r'
> iocs = [ioc1, ioc2]
> map kindOfIorC iocs
["an int","a char"]
> :t iocs
iocs :: [IorC]
```

- Recall that a list can only contain one type of data, so `[17, 'r']` produces an error. By declaring `IorC`, we can use `[IntVal 17, CharVal 'r']` to create a value of type `[IorC]` and get the same effect.
  - This is an example how a strict type system needs a rich type structure to be more usable: Without alternatives, there'd be no way to get the effect of "list of integers or characters".
  - Since you can look at an `IorC` value and find out which alternative if comes from, Haskell unions are **disjoint**. This also helps with a strict type system because it enables the compiler to be sure that it's accessing the right kind of embedded data (`Int` or `Char`).

F. Nested structures

- A `data` declaration doesn't have to use only primitive data; it can also include data built by already-declared `data` types. For example,

```
> data Point = MkPoint Float Float deriving (Read, Show, Eq, Ord)
> data Triangle = MkTriangle Point Point Point deriving (Read, Show, Eq)
> p1 = MkPoint 1.0 2.5
> p2 = MkPoint 3.5 1.25
> p3 = MkPoint 2.0 4.0
> t1 = MkTriangle p1 p2 p3
> t2 = MkTriangle (MkPoint 1.0 2.5) (MkPoint 3.5 1.25) (MkPoint 2.0 4.0)
> t1 == t2
True
```

- If we write a function that takes a `Triangle`, we can (if we want) write a pattern in any of the `Point` positions, to get quick access to the coordinates.
  - Below, the `mvRight` function takes a triangle and adds a `delta_x` to the *x*-coordinates of its three points. Just to show the difference, the first two points are given names and the third point is calculated in-line.

```
:{
| mvRight :: Float -> Triangle -> Triangle
| mvRight delta_x (MkTriangle (MkPoint x1 y1) (MkPoint x2 y2) (MkPoint x3 y3))
|     = let p1 = MkPoint (x1+delta_x) y1 ;
|           p2 = MkPoint (x2+delta_x) y2
|       in
|           MkTriangle p1 p2 (MkPoint (x3+delta_x) y3)
:}
> t1
MkTriangle (MkPoint 1.0 2.5) (MkPoint 3.5 1.25) (MkPoint 2.0 4.0)
> mvRight 10.0 t1
MkTriangle (MkPoint 11.0 2.5) (MkPoint 13.5 1.25) (MkPoint 12.0 4.0)
```

- As before, we can write this routine using variables as function parameters and use the patterns in case expressions.

```
:{
| mvRight :: Float -> Triangle -> Triangle
| mvRight delta_x tri
|     = let MkTriangle p1 p2 p3 = tri;
|           MkPoint x1 y1 = p1;
|           MkPoint x2 y2 = p2;
|           MkPoint x3 y3 = p3
|       in
|           MkTriangle (MkPoint (x1+delta_x) y1)
|                      (MkPoint (x2+delta_x) y2)
|                      (MkPoint (x3+delta_x) y3)
:}
```

G. Parameterized structures

- We've seen `data` declarations create types (like `IorC` or `Point` or `Triangle`). Often, the idea behind a type is interesting enough that we generalize it.

The Either type constructor

- As an example, the idea behind `IorC` is that we have either an `Int` or a `Char`; if we decide we want either an `Int` or `String`, rather than define a second type `IorString`, we can make use of the library type constructor `Either`, defined as

```
data Either a b = Left a | Right b
```

- The `a` and `b` to the left of the equal are type variables, parameters for `Either`. `Either` is not a type, it's a type constructor: To get a type, you substitute types for `a` and `b`:

```
> ival = Left 12 :: Either Int Char
> cval = Right 'z' :: Either Int Char
> :t [ival,cval]
[ival,cval] :: [Either Int Char]
```

- If we hadn't explicitly annotated types for `ival` and `cval`, the types would have been polymorphic:

```
> (ival2, cval2) = (Left 12, Right 'z')
> :t (ival2, cval2)
(ival2, cval2) :: Num a1 => (Either a1 b, Either a2 Char)
```

- We can define functions that take `Either Int Char` parameters in the same way we defined functions on `IorC`. In `kindOfIorC` below, we explicitly say that the parameter is an `Either Int Char`

```
> :{
| kindOfIorC :: Either Int Char -> String
| kindOfIorC (Left n) = "integer " ++ show n
| kindOfIorC (Right c) = "character " ++ show c
| :}
> :t kindOfIorC
kindOfIorC :: Either Int Char -> String
> map kindOfIorC [Left 17, Right 'c']
["integer 17","character 'c'"]
```

- The `kind2` function shows what Haskell infers if we leave off the specific type annotation. We have to use showable types of values for the left and right alternatives of `Either a b`.

```
> :{
| kind2 (Left n) = "k2 left " ++ show n
| kind2 (Right c) = "k2 right " ++ show c
| :}
> :t kind2
kind2 :: (Show a, Show b) => Either a b -> String
> map kind2 [Left 17, Right 'c']
["k2 left 17","k2 right 'c'"]
```

-------------------- 2020-01-28 ?

The Maybe Type Constructor

- A very useful built-in type constructor is `Maybe`; it's used when you have computations that might fail to produce a value, but you don't want this to cause runtime errors.

```
data Maybe a = Nothing | Just a
```

- E.g., here's a safer square root routine: If its argument is negative, it produces `Nothing`; if the argument is nonnegative, it produces `Just` (`sqrt` *arg*).

```
safesqrt :: Double -> Maybe Double
safesqrt x | x < 0 = Nothing
           | otherwise = Just (sqrt x)
```

- When using `safesqrt`, we can do a pattern match. (Below, the `safeneg` functions do the same thing; they're just declared differently.)

```
:{
| safeneg :: Maybe Double -> String
| safeneg Nothing = Nothing
| safeneg (Just val) = Just (-val)
|
| safeneg :: Maybe Double -> String
| safeneg x = case x of
|         Nothing -> Nothing
|         Just val -> Just (-val)
:}
```

- Just for fun, here's some runs of `safeneg` on mostly the same list:

```
> map safeneg [Nothing,Just 1,Just 4,Just 9,Just 16,Just 25]
[Nothing,Just (-1),Just (-4),Just (-9),Just (-16),Just (-25)]
> map safeneg (Nothing : map (Just . (\n -> n*n)) [1..5])
[Nothing,Just (-1),Just (-4),Just (-9),Just (-16),Just (-25)]
> [safeneg x | x <- Nothing : [Just (n*n) | n <- [1..5]]]
[Nothing,Just (-1),Just (-4),Just (-9),Just (-16),Just (-25)]
> map (safeneg . Just) [1,4,9,16,25]
```

```
[Just (-1),Just (-4),Just (-9),Just (-16),Just (-25)]    -- fixed 3
2020-01-30
```

H. Recursive Structures

- If the body of a data declaration uses the type being defined, we get a recursive datatype.

```
:{
| data List a = Node a (List a) | Nil deriving (Show, Read, Eq)
|
| null Nil = True
| null (Node _ _) = False
|
| len Nil = 0
| len (Node _ x) = 1 + len x
|
| len1 x = len1' 0 x
| len1' res Nil = res
| len1' res (Node _ t) = len1' (res+1) t
:}
> x = Node 1 (Node 2 (Node 3 (Node 4 (Node 5 Nil))))
> len x
5
> len1 x
5
```

- For another example of a recursive datatype, here's a simple binary tree type, where the nodes and leafs are labeled by values.

```
> data Tree a = Leaf a | Node a (Tree a) (Tree a) deriving (Read, Show, Eq)
> t1 = Leaf "abc"
> t2 = Node "ab" t1 (Leaf "de") -- using t1, a leaf
> :t t1
t1 :: Tree [Char]
> :t t2
t2 :: Tree [Char]
```

- Here's a function that checks to see if a given value is ≥ every value in the tree. (Note we can only run this function on trees where the values are of a type that supports `Ord`.)

```
:{
| ge_tree :: Ord t => t -> Tree t -> Bool
| ge_tree val (Leaf val') = val >= val'
| ge_tree val (Node val' t1 t2) =
|     val >= val' && ge_tree val t1 && ge_tree val t2
:}
> t2
Node "ab" (Leaf "abc") (Leaf "de")
```

```
> ge_tree "z" t2
True
```

I. Type Declarations

- The `type` declaration is the other way to declare a type, but it doesn't declare a new kind of structure the way a `data` declaration does. A `type` declaration lets you give an alternate name to a type.

  `type` *name* = *type_expression*

- As with `typedef` in `C`, a type declaration lets you give a more descriptive name to a type expression that could otherwise be misinterpreted. For example, an integer serving as an identifier looks like any other integer, but by declaring `type Id = Int`, we can use `Id` in places where we intend an identifier instead of, say, a street number.
- A type declaration only declares a synonym: Formally, `Id` and `Int` have **structural equivalence** because they're built using the same type structure. Haskell uses structural equivalence on `type`-declared types when checking equality. E.g.,

```
> type Id = Int
> type Id' = Int
> x = 5 :: Id
> y = 5 :: Id'
> z = 5 :: Int
> x == y && y == z && z == x   -- Not a type error
True
> (5 :: Id) == (5 :: Id')      -- more briefly
True
```

- `data`-declared types use **name equivalence** for equality: Values of `data`-declared types that have the same structure are not equal. E.g.,

```
> data Temp1 = Temp1 Int
> data Temp2 = Temp2 Int
> x1 = Temp1 77
> x2 = Temp2 77
> x1 == x2
[Error message Couldn't match expected type ‘Temp1’ with actual type ‘Temp2’ ]
```


Activity Questions, Lecture 5

Tail Recursion

1. What is tail-recursion and tail-recursion optimization?
2. Write a tail recursive version of $sum(n) = 0$ if $n \le 1$ and $n+sum(n\text{-}1)$ otherwise.
3. Using tail-recursion, write a linear-time version of a function `add_total` $[v_1, v_2, \ldots, v_n]$ that returns $[v_1 + s, v_2 + s, \ldots, v_n + s]$ where s is the sum $(v_1 + v_2, + \ldots + v_n)$. E.g., `add_total [1..5] = [16,17,18,19,20]`.

   (Hint: Write one function that traverses a list and returns its reverse and the sum of its values, then write a function that takes a list and a value and returns the reverse of that list with the value added to every element.)

Algebraic types

1. The declaration `data T1 = A1 | B1` declares what kind of type? List the values of type `T1`.
2. The declaration `data T2 = A2 | B2 Int` declares what kind of type? Give examples of values of type `T2`.
3. Repeat, on `data T3 = A3 Int | B3 Int T3`. Also, what kind of data structure does `T3` model?
4. Repeat, on `data T4 = A3 Int | B3 Int T4 | C3 Int T4`. What kind of data structure does `T4` model?
5. Is there anything weird about the type `data T5 = A5 T5` ?
6. Is there anything weird about the type `data T6 = A6 | B6 T6` ?
7. What does the clause `deriving (Eq, Read, Show)` do when added to a `data` declaration?

For Problem 8 and 9, look back to a datatype declared earlier:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
     deriving (Read, Show, Eq)
```

8. Define a recursive function that returns the height of a binary tree. (A tree that's just a leaf has height zero [2/3].)
9. Repeat Problem 8 using a helper function `height' tree n` that returns `n + height(tree)`. (Since you need two recursive calls, the routine won't be fully tail-recursive.)

Solutions to Selected Activity Questions

Tail Recursion

2.
```
sum n = sum' n 0
sum' 0 s = s                    -- sum n s = sum of [1..n], plus s
sum n s = sum (n-1) (s+n)
```

3.
```
-- add_total_list [x1, x2, ..., xn]
--     = [x1+tot, x2+tot, ..., xn+tot] where tot = x1+x2+…+xn
--
add_total list =
    let (_, list_plus_total, total) =
            rev_and_inc (rev_and_sum ([], list, 0))
    in list_plus_total

-- rev_and_sum(rpt, [x1, x2, ..., xn], ptot)
--      = ([xn, ..., x2, x1] ++ rpt, [], x1+x2+...+xn + ptot)
--
-- variables: rpt = "reverse of partial tail", ptot = "partial total"
--
rev_and_sum(rpt, [], ptot) = (rpt, [], ptot) ; rev_and_sum(rpt, val:vals,
    ptot) = rev_and_sum(val:rpt, vals, ptot+val)

-- rev_and_inc([x1,x2,...,xn], ript ,tot) =
--    ([], [xn+val, ..., x2+val, x1+val] ++ ript, tot)
--
-- variable ript = "reverse of incremented partial tail"
--
rev_and_inc([], ript, tot) = ([], ript, tot); rev_and_inc(val:vals, ript,
    tot) = rev_and_inc(vals, (val+tot):ript, tot)
```

Algebraic types

1. Enumerated, `A1`, `B1`.
2. Structured. Values of type T2 are `A2` (a constant) and `B2` *int* for any int.
3. Structured. We get `A3` *int* and `B3` *int*. We basically have a set of two different kinds of integers, so we tag them with `A3` or `B3` to say which kind.
4. Same as 3 but with three different kinds of integers.
5. `data T5 = A5 T5` is a legal declaration, but there's no way to create a value of type `T5` (unless you already have one, which we don't). This type is sometimes called *empty*, *void*, or *false*.
6. No, nothing weird about `data T6 = A6 | B6 T6` aside from it being recursive. `data T6 = A6 T6 | B6 T6` would be similar to `T5` except we now we'd have two ways to not create a value.

7. Adding `deriving` (`Eq`, `Read`, `Show`) makes the new type instances of those classtypes. Haskell will automatically generate the functions for the classtypes (`==`, `/=`, `<`, `<=`, etc., `show`). (Using `:info` *classtype* in ghci will list all the items provided by that classtype.)

For Problems 8 and 9, we have

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
    deriving (Read, Show, Eq)
```

I'll name the functions `height1` and `height2` just to make them different

8.
```
height1 (Leaf _) = 0        -- leafs have height 0 not 1
height1 (Node _ left right) =
    1 + max (height1 left) (height1 right)
```

9.
```
height2 tree = height' tree 0
height' (Leaf _) h = h      -- height' tree n = height of tree + n
height' (Node _ left right) h =
    max (height' left (h+1)) (height' right (h+1))
```

(There isn't much difference between the two definitions, since we need two recursive calls and we're doing work after the calls.)

