Haskell, Part 4

CS 440: Programming Languages and Translators, Spring 2020

A. Ch 5 (recursion)

- We're skipping over this for now; we'll go back to it after higher-order functions.

B. Ch 6 Higher-Order functions [LYaH Ch.6, p.1]

- Recall: A **higher-order** function is a function that takes another function as a parameter or returns a function as a result. If you look at the type of a higher-order function, you can tell whether or not it takes functional arguments or produces a functional result by where and how many arrows appear in its type.

```
> f_squared f x = f (f x)
> :t f_squared
f_squared :: (t -> t) -> t -> t
```

- Since arrow is right-associative, we can write the type of `f_squared` as `(t -> t) -> (t -> t)`. If we break down this type, the arrow in the middle tells us `f_squared` is a function, the arrow to the left of middle indicates a function argument, and the arrow to the right of middle indicates a function as result.

```
(t -> t)  ->  (t -> t)
-------- ----- --------
argument arrow  result
```

Function Composition

- One example of a higher-order function is function composition. In everyday math, infix circle is used for function composition. The Haskell operator is infix `dot`. I.e., `f . g` is the function that takes an argument `x` and returns `f(g(x))`. A definition is `(.) f g x = f(g x)`.

```
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

- From the type of `(.)`, we can see that it takes a function (of type `b -> c`) as argument and produces a function (of type `(a -> b) -> a -> c`) as the result. This result is itself higher-order: It takes a function (of type `a -> b`) as an argument and produces a function (of type `a -> c`) as its result.

N-Argument Functions [LYaH Ch.6, p.1]

- In a typical language, a function like + is of type `Int` × `Int` → `Int`. I.e., it takes a pair of values and returns a value. In Haskell, this type is written `(Int, Int) -> Int`, but Haskell actually uses a different type.

- In Haskell, a function like + takes its arguments *one after another*. The type of `(+)` is `a -> a -> a` (where `a` is a `Number` type). So (+) is a function that takes one argument (i.e., the left operand for addition), and returns a function. This returned function takes its one argument and uses it as the right operand for addition, and returns the sum.
- E.g., if we take the prefix version of addition, written (+) and apply it to 5, we get a function (`add5` below) that takes an argument like 7 and returns the sum 12.

```
> (+) 5 7
12
> add5 = (+) 5
> :t add5
add5 :: Num a => a -> a
> add5 7
12
```

- It is possible in Haskell to write functions that take all their arguments *at the same time*, by writing a function that takes a tuple for its parameter. We give separate names to the function parameters by forming a tuple pattern with variables. E.g.,

```
> f(x,y) = x + 2*y
> :t f
f :: Num a => (a, a) -> a
> f(6,2)
10
```

- But `f` really takes only one argument, which happens to be an ordered pair, so we can pass `f` any expression that evaluates to a pair of numbers:

```
> p = (6,2)
> f p
10
> :t p
p :: Num a => (a, a)
```

C. Currying and Uncurrying [LYaH Ch.6, p.1]

- Functions that take multiple arguments one after another are said to be "**curried**".
- The name has nothing to do with spices, it comes from Haskell Curry, the mathematician / logician / CS person for whom the language Haskell is named.
- Examples: Below, `f` is curried and `g` is uncurried but they produce the same final result.

```
> f x y = x - y
> g(x,y) = x - y
> f 5 3
2
> g(5,3)
2
```

```
> :t f
f :: Num a => a -> a -> a
> :t g
g :: Num a => (a, a) -> a
>
```

- In Haskell, we pretty much always use curried functions, which is why we typically write *fcn arg1 arg2* etc. when calling a multi-argument function.
- The `curry` and `uncurry` functions convert a function from one to the other. Here are their types (I've added extra parentheses to emphasize that they take a 2-argument function and return a 2-argument function).
  - `curry :: ((a, b) -> c) -> (a -> b -> c)`
  - `uncurry :: (a -> b -> c) -> ((a, b) -> c)`

- Currying the function `g` above gives you a function that behaves like `f`; uncurrying `f` gives you a function that behaves like `g`.[1] We say `f'` and `g'` are partially applied versions of `g` and `f` respectively

```
> g' = curry g
> g' 5 3
2
> f' = uncurry f
> f'(5,3)
2
```

- We don't have to define `f'` and `g'` as intermediate names.

```
> curry g 5 3
2
> uncurry f (5,3)
2
```

- You may already have guessed, but `curry` and `uncurry` are inverses of each other.

```
> uncurry (curry g) (5,3)
2
> (uncurry . curry) g (5,3) -- recall dot is function composition
2
> curry (uncurry f) 5 3
2
> (curry.uncurry) f 5 3     -- don't need spaces around the dot
2
```

D. The Higher-Order Function Map [LYaH Ch.6, p.6]

- The `map` function is a higher-order function that applies a given function to every element of a list.

[1] I don't think I've mentioned that you can use apostrophes in identifiers, so `f'` is Haskell for f'.

```
> map sqrt [1..5]
[1.0,1.4142135623730951,1.7320508075688772,2.0,2.23606797749979]
> k x = 8 + x           -- a function on one argument
> map k [1..5]          -- add 8 everywhere
[9,10,11,12,13]
> map ((+) 8) [1..5]    -- k is same as function result of (+) 8
[9,10,11,12,13]         -- like [(+) 8 1,(+) 8 2, etc]
```

- So `map :: (a -> b) -> [a] -> [b]`. The first argument is a function on a values (so `map` is higher-order), a list of a values, and it returns the list of results. Using type variables a and b indicates that the argument and result types of the function can be different (but they aren't required to be).
- `map` can be defined using a list comprehension, `map2 f x = [f v | v <- x]`. We can expand an example using referential transparency:

```
map2 ((+) 8) [1..3]
= [(+) 8 x | x <- [1..3]]
= [(+) 8 1, (+) 8 2, (+) 8 3]
= [9, 10, 11]
```

E. The Higher-Order Function Filter [LYaH Ch.6, p.6]

- Similar to `map` is `filter`; like `map`, `filter` takes a function argument and a list of argument values for the function, and it runs the function on every element of the list.
- `filter` is different in that the function has to be of type (`a -> Bool`), so it's a test function; it's also different from `map` because it returns a sublist of the function arguments, namely, the values that pass the test.
- E.g., if we `map` an *is-this-positive*? test function across a list of numbers returns a list of the true/false results of the test. If we `filter` the test function across the same list, we get a list of the members that have `True` as the corresponding result from `map`.

```
> positive x = x > 0  -- (if x > 0 then True else False)
> map positive [3, 5, -1, 2, -9, 7, -2, -3]
[True,True,False,True,False,True,False,False]
> filter positive [3, 5, -1, 2, -9, 7, -2, -3]
[3,5,2,7]
```

- Another example: Find values divisible by 3

```
> divisible_by_3 x = x `mod` 3 == 0 -- using mod in infix
> divisible_by_3 6
True
> filter divisible_by_3 [27..83]
[27,30,33,36,39,42,45,48,51,54,57,60,63,66,69,72,75,78,81]
```

- Find last value in a list that passes a test

```
> last (filter divisible_by_3 [27..83])
81
```

- Like map, filter can be defined using a list comprehension:

```
filter2 f xs = [x | x <- xs, f x]
```

Lambda (Unnamed) Functions and the Lambda Calculus) [LYaH Ch.6 p.9]

- It can be annoying to write a function like `multiple_of_3` just to use it in one spot.
- We can use **unnamed functions** instead. Below, `\ x -> x `mod` 3 == 0` is a function that takes an `x` and returns true if `x mod 3` is zero.

```
> -- divisible_by_3 x = x `mod` 3 == 0
> -- divisible_by_3 = \ x -> x `mod` 3 == 0 -- same as previous line
> divisible_by_3 6
True
> filter (\ x -> x `mod` 3 == 0) [27..83]
[27,30,33,36,39,42,45,48,51,54,57,60,63,66,69,72,75,78,81]
```

- **The lambda calculus** discusses function definition and execution using unnamed, lambda functions.
- A **lambda function** tells you how to calculate a result given an argument but it doesn't give the function a name. (Hence, "unnamed" function.)
- **Notation**: λ *id*.*expr* means "a function that takes a value for the identifier and yields the value of the expression. (The expression is also called the **body**.)
  - Example: $\lambda\, x . x$ is the identity function.
  - The body can itself be a function. E.g., $\lambda\, f . \lambda\, x. f(f\, x)$ is the f squared function ($f^2 = f \circ f$) .
  - Iterated λ functions can be abbreviated: $\lambda\, f\, x . f(f\, x)$ means $\lambda\, f . \lambda\, x. f(f\, x)$
- **The Haskell notation** for λ *id*.*expr* is `\` *id* `->` *expr*. (Backslash is used for lambda and the arrow separates the identifier and body.)
  - E.g., `\ f -> \ x -> f(f x)` is the Haskell notation for $\lambda\, f . \lambda\, x. f(f\, x)$.
  - Iterated lambdas can be abbreviated: E.g., `\ f x -> f(f x)` means $\lambda\, f\, x . f(f\, x)$.
- **Haskell function definitions as syntactic sugar:** A function declaration like `f x` = *exp* is short for the definition `f` = `\x ->` *exp*.
  - This is one way in which Haskell supports first-class functions: Aside from the pleasant syntax, a function definition is just like a declaration of any other variable. Declaring *id* = *expr* works the same way regardless of whether the expression is a primitive `Int` or a pair or list or (now, we see) even a function.
- So these all have the same meaning:

```
> f a b c = a * b + c
> f = \a b c -> a * b + c
> f = \a -> \b -> \c -> a * b + c
> f 3 5 8
23
```

- **Referential transparency and lambda application**[2]:

[2] The application of a lambda function to an argument is also known as "β-reduction" — there are different transformations on lambda functions and β-reduction is one of them.

    - (\ x -> *expr*) *arg* turns into the *expr* with *arg* replacing *x* everywhere.
    - E.g., with `f = \a -> \b -> \c -> a * b + c`,
        - `f 3` is `\b -> \c -> 3 * b + c`
        - f 3 5 is `\c -> 3 * 5 + c`
        - `f 3 5 8` is `3 * 5 + 8` evaluates to 23
    - When we have `f a b c = a * b + c` and replace `f 3 5 8` by `3 * 5 + 8`, we're doing the same thing as the above (just faster).
- **Using lambdas in Haskell**: Lambdas are very useful for short things you use once. Why define

```
> positive x = x > 0  -- usual definition syntax
> positive = \x -> x > 0 -- using lambda
> filter positive [3, 5, -1, 2, -9, 7, -2, -3]
[3,5,2,7]
> filter (\x -> x > 0) [3, 5, -1, 2, -9, 7, -2, -3]
[3,5,2,7]
```

F. Folding Lists [LYaH Ch.6, p.11]

- Folding a list lets you combine its elements using some operation, like adding together a list of numbers.
- `foldl` takes a binary operation, a starting value, and the list to fold. With the starting value on the left, the operation is repeated left to right.

```
> foldl (-) 0 [3,5,8] -- equals (((0 - 3) - 5) - 8)
-16
> (((0 - 3) - 5) - 8)
-16
```

- `foldr` goes right-to-left, with the starting value at the **right** end.

```
> foldr (-) 0 [3,5,8]  -- equals (3 - (5 - (8 - 0)))
6
> (3 - (5 - (8 - 0)))
6
```

- [Not in LYaH} Giving the ghci command `:t` the flag `+d` tells ghci to provide default types for complicated types. E.g.,

```
> :t (+)
(+) :: Num a => a -> a -> a
> :t +d (+)
(+) :: Integer -> Integer -> Integer
```

- Restricted to just looking at lists, the types of `foldl` and `foldr` are as follows:[3]

```
> :t +d foldl
foldl :: (b -> a -> b) -> b -> [a] -> b
```

---

[3] The non-default types for `foldl` and `foldr` use types more general than `[a]` and `[b]`. Instead, they use `t a` and `t b` where t is a *type operation* (takes a type, returns a type) and the type operation is an instance of typeclass `Foldable`. Eg., the full type of `foldl` is `Foldable t => (b -> a -> b) -> b -> t a -> b`

```
> :t +d foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
```

- In `foldl (-) 0 [3, 5, 8]` and `foldr (-) 0 [3,5,8]`, we use `Int` for type variables `a` and `b` and get
  - `foldl :: (Int -> Int -> Int) -> Int -> [Int] -> Int`
  - `foldr :: (Int -> Int -> Int) -> Int -> [Int] -> Int`
- Note that since we use `Int` for both `a` and `b`, the types of `foldl` and `foldr` are the same.
- Since + is associative, `foldl` and `foldr` return the same result when given the same arguments: `foldl` (+) `0 [3,5,8]` and `foldr` (+) `0 [3,5,8]` both return `16`.
- Since + is commutative, you can reorder the elements: `foldl` (+) `0 [3,5,8]` = `foldl` (+) `8 [0,5,3]`.
- On the other hand, **–** is not associative, so `foldl` and `foldr` can return different values on the same arguments. (We saw this above) Since **–** is not commutative, reordering the elements can change the result: `foldl (-) 0 [1, 2]` ≠ `foldl (-) 0 [2, 1]`.

The `Foldable` type class

- The types of `foldl` and `foldr` actually use `t a` instead of `[a]`, where `Foldable t`. (I.e., `t` is an instance of `Foldable`.) Here, `t` is a **type constructor** (an operator that takes one type and gives you back another), not a type itself. When we go from type `a` to type `[a]`, we're applying the list type constructor […] to the type `a` to get another type.
- `Foldable` is for building types that behave like lists: They have to allow folding and mapping, finding an element, `sum` and `product` (when `Foldable` is given a numeric type) and `minimum` and `maximum` (when `Foldable` is given an order-able (`Ord`) type).
- We'll define our own type constructors when we get to algebraic datatypes (soon!)
- In `foldl (-) 0 [3, 5, 8]`, we use `Int` for the type variables `a` and `b` in `(b -> a -> b) -> b -> t a -> b`, and we use [...] (list-of) as `t`.

Another Example of folding

- The *Learn You* ... book has an example that uses folding to define our own `elem` function. (`elem y ys` is true if `y` is a member of list `ys`.) The use of `foldl` here uses different types for `a` and `b` in
  ```
  foldl :: (b -> a -> b) -> b -> [a] -> b
  ```
  Specifically, for `b` we use `Bool` and `a` we use `Int` (more generally, any `Eq` type)
  ```
  foldl :: (Bool -> Int -> Bool) -> Bool -> [Int] -> Bool
  ```
- So we'll define an `elem2` function and have `elem2 y ys` look for a `y` amongst the `ys`.
  - The function we pass to `foldl` accumulates the boolean result of the question "Have we found a `y` yet?" For the initial test we pass `False` (no `y` in `[]`) and then test the 1st element of the list. The result comes back `True` or `False` depending on whether or not it was `y`, and then we continue. Note once the accumulated result becomes `True`, it never becomes `False` after that.
  - Define `found y acc next = if acc then True else next == y` -- i.e., (`acc || next == y`)
  - Then `elem2 y ys = foldl (found y) False ys`
    - Starting with `false`, search left-to-right to see if we can find a `y`.

- As we search, the accumulated result is the boolean "Have we found `y` yet?"

```
> found y acc next = (next == y || acc)
> elem2 y ys = foldl (found y) False ys
:t elem2
elem2 :: (Foldable t, Eq a) => a -> t a -> Bool

> elem2 0 []
False
> elem2 3 [1,2,3,4]
True
```

- Let's look at how `elem2 3` behaves.
- We have `elem2 3 ys = foldl (found 3) False ys` and `found 3 acc next = (acc || next == 3)`
- To shorten things, define `f = found 3`, then *acc* `` `f` `` *val* returns true if the accumulated value is already true, otherwise it checks the value against 3, returning true for the new accumulated search result if it does find 3 and false if it doesn't.

```
> f = found 3
> foldl f False [1,2,3,4]       -- find a 3?
> True
> foldl f False [1,2]           -- don't find a 3
> False
> acc1 = False `f` 1            -- check 1st element of list
> acc1
False
> acc2 = acc1 `f` 2             -- check 2nd element of list
> acc2
False
> acc3 = acc2 `f` 3             -- 3rd element of list finds a 3
> acc3
True                            -- new accumulated value is true
> acc4 = acc3 `f` 4             -- no matter what next value is
> acc4
True                            -- search result is still true
```

---------- 2020-01-23

Activity Questions, Lecture 4

1. Higher-Order Functions

1. What is a **higher-order** function?
2. What is the associativity of arrow? (I.e., how do you parenthesize the type `a -> b -> c`)
3. What is a **curried** / uncurried function? What is partial application of a curried function?
4. You can't print functions (regardless of order) because function types aren't instances of … ?
5. How do you do function composition?
6. What does `map` do? `map f x` = what list comprehension?
7. What does `filter` do? `filter f x` = what list comprehension?

2. Lambda Functions

1. What is the syntax for an unnamed lambda in Haskell? In the lambda calculus?
2. `\x -> \y ->` *expr* can be abbreviated as … ?
3. Why use lambda expressions?
4. What is the usual way to write the declaration `f = \x -> \y ->` *expr* ?
5. The declaration `f =` *lambda function* illustrates what principle?

3. Folding lists

1. What do `foldl` or `foldr` with arguments `f x` [v_1, v_2, v_3, …, v_n] return? If `x :: t1` and the `v`'s are `:: t2` , what type does `f` have to have under `foldl`? `foldr`?
2. Give a simple recursive definition for `foldl`; give one for `foldr`.
3. If `f` is associative, then what properties hold with `foldl` and `foldr`? What if `f` is commutative?
4. When we say that lists are instances of `Foldable`, we mean ____ is an instance of _____ ?

Solutions to Selected Activity Questions

1. Higher-Order Functions

1. A higher-order function is a function that takes a function parameter or produces a function result (or both). Note the function can be part of a larger structure. E.g., a function that takes or produces a list of functions counts as higher-order.
2. The -> type operator is right associative, so `a -> b -> c` means (`a -> (b -> c)`). (So if you want the type (`a -> b) -> c`, then the parentheses are required.)
3. A curried function takes multiple arguments in sequence. It has a type like *arg_type* -> *arg_type* -> … -> *result_type*. An uncurried function takes multiple arguments simultaneously. It has a type like (*arg_type*$_1$, *arg_type*$_2$, … *arg_type*$_n$) -> *result_type*. A partially-applied curried function is a function call where we've supplied some of the arguments but not all of them. Example: (+) is curried and (+) 1 is partially-applied; (+) 1 2 is fully-applied and equals 3
4. You can't print functions because function types aren't instances of typeclass `Show`.
5. `map f x = [f v | v <- x]`
6. `filter f x = [v | v <- x, f v]`

2. Lambda Functions

1. The lambda calculus function λ *x*. *expr* is written in Haskell as `\ x ->` *expr*.
2. `\x -> \y ->` *expr* can be abbreviated as `\x y ->` *expr*.
3. Lambda functions can be useful for short functions you only use a small number of times.
4. We usually write `f = \x -> \y ->` *expr* as `f x =` *expr*.
5. Allowing declarations like `f =` *lambda function* is part of having functions be first-class: They can be used like any other kind of expression. (Compare with `x = 17`, `y = [17,18,19]`, and `z = (2, "ab")`, which all have the form *id* = *expr*.)

3. Folding lists

1. `foldl f x` [v_1, v_2, v_3, …, v_n] = `(…(((x `f` v1) `f` v2) `f` v3) … `f` vn)`, and `foldr f x` [v_1, v_2, v_3, …, v_n] = `( v1 `f`(v2 `f` (v3 … `f` (vn `f` x)…)))`. If `x :: t1` and the v's `:: t2`, then for `foldl`, `f :: t1 -> t2 -> t1`; for `foldr`, `f :: t2 → t1 → t1`.
2.
```
foldl f x [ ] = x
foldl f x (h:t) = foldl f (f x h) t
foldr2 _ x [ ] = x
foldr2 f x (h:t) = f h (foldr2 f x t)
```

3. If `f` is associative then `foldl f` v_0 [v_1, v_2, v_3, …, v_n] = `foldr f` v_n [v_0, v_1, v_2, …, v_{n-1}]. Both folds list values v_0 through v_n. If f is associative, the different parenthesizations used by `foldl` and `foldr` don't matter. If `f` is associative and commutative, then permuting the values $v_0 - v_n$ for `foldl` or the values v_n, v_0, v_1, v_2, …, v_{n-1} for `foldr` doesn't change the result either.
4. When we say that lists are instances of `Foldable`, we mean that the type constructor `[]` (list of) is an instance of `Foldable`.

   Obscure fact (not on exam): Haskell accepts `[]` *type* as an alternative way to write [*type*]. E.g.,

   ```
   > "abc" :: [] Char
   "abc"
   ```

