Haskell, Part 2

CS 440: Programming Languages and Translators, Spring 2020

A. Defining an interactive variable [not in LYaH]

- At top level, we can define a variable good to the end of the session (unless you redefine it.)
- Note defining x doesn't print out its value. To get the value, use x as an expression. The keyword "`let`" is optional.

```
> let x = [1,2,3,5,7]
>
> x -- use x as an expression to print its value
[1,2,3,5,7]
```

B. More functions: `!!`, `elem` [LYaH Ch.2 p.7,10]

- Use operation … `!! n` to return n'th element of a list

```
> x !! 0 -- return first element
1
> x !! 1
2
> x !! 8
*** Exception: Prelude.!!: index too large
```

- Use `elem` to search a list for a value

```
> elem 2 [1,2,3,5,7]        -- is 2 in the list?
True
> elem 8 [1,2,3,5,7]        -- but not 8
False
```

C. Infix to Prefix [LYaH Ch.2, p.17]; Prefix to Infix; [LYaH, Ch 3, p.4]

- Use backquotes to treat a prefix function as an operator; surround operator with (…) to treat it as a prefix function.

```
> 2 `elem` [1,2,3,5,7]
True

> (:) 3 x -- instead of 3 : x
[3,1,2,3,5,7]
> (+) 5 2
7
```

D. Infinite Lists [LYaH Ch.2, p.11]

- It's easy to definite infinite lists. You can't print them completely out; can only print a finite sublist.
    - `[1..]` is `[1,2,3, …]`
    - `[1,3..]` is `[1,3,5,7,…]`
- `repeat x` equals `[`x, x, x, etc.`]`
- Defining > *variable* = *infinite list* is okay, but don't use > *infinite list* or > *variable* (holding infinite list)

```
> r = repeat 17              -- r = [17, 17, 17, etc]
> -- Just using > r here would try to print an infinite list
```

- Using `take` *n list* to get the first n elements of the list can help

```
> take 20 r                  -- first 20 elements of r
[17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17]
> s = drop 20 [1..]     -- using an infinite list range [1, 2, 3, etc]
> take 10 s
[21,22,23,24,25,26,27,28,29,30]
```

- `cycle` x equals x ++ x ++ x ++ etc.

```
> t = cycle [1,3,5,6]        -- [1,3,5,6, 1,3,5,6, 1,3,5,6, etc]
```

- Using comprehension over infinite lists can be tricky. E.g., take a finite example

```
> p = [(x,y) | x <- [1..3], y <- [5..7]]
> p
[(1,5),(1,6),(1,7),(2,5),(2,6),(2,7),(3,5),(3,6),(3,7)]
> take 3 p  -- first 3 elements of p
[(1,5),(1,6),(1,7)]
> take 3 (drop 3 p) -- second 3 elements of p
[(2,5),(2,6),(2,7)]
> take 3 (drop 6 p)
[(3,5),(3,6),(3,7)] -- third 3 elements of p
```

- Now try the infinite version

```
> pairs = [(x,y) | x <- [1..], y <- [1..]]
> take 5 pairs
[(1,1),(1,2),(1,3),(1,4),(1,5)]
```

- Notice we never got to `y` = 2 because all the `x` = 1 pairs come first

E. Playing around with infinite lists [not in LYaH]

- It's easy to write functions that yield infinite lists

```
> repeat2 v = v : repeat2 v     -- behaves like repeat
> cycle2 x = x ++ cycle2 x      -- behaves like cycle
```

- Or just definite a variable recursively

```
> ones = 1 : ones
> pow2 = 1 : [2*x | x <- pow2]
> take 10 pow2
[1,2,4,8,16,32,64,128,256,512]
> t n = take n pow2
> t 10
[1,2,4,8,16,32,64,128,256,512] -- end 2020-01-14
```

- To see how `pow2` gets built, let's calculate successive approximations of `pow2` (i.e., longer and longer prefixes of it).
  - E.g., `t 6` = `1 :` `[2*x | x <- t 5]`.
    - We have `t 5` = `[1,2,4,8,16]`
    - So `[2*x | x <- t 5]` = `[2,4,8,16,32]`
    - So `t 6` = `1 :` `[2,4,8,16,32]` = `[1,2,4,8,16,32]`
  - Symbolically, `take (n+1) pow2`
    = `take (n+1) (1 : [2*x | x <- pow2])`
    = `1 : (take n [2*x | x <- pow2])`
    = `1 : [2*x  | x <- take n pow2]`
    = `1 : ( [2*x | x <- take (n-1) pow2] ++ [2*x | x <- [last (take n pow2)] )`
    = `take n pow2 ++ [2*x <- [last (take n pow2)]`
    = `take n pow2 ++ [2 * last (take n pow2)]`
  - In English, this last line just says that `t (n+1)` is `t n` plus the new last element, which is `2 *` the last element of `t n`.
    - E.g., `t 6` = `t 5 ++ [2 * last (t 5)]`
      = `[1,2,4,8,16] ++ [2 * last [1,2,4,8,16]]`
      = `[1,2,4,8,16] ++ [2 * 16]`
      = `[1,2,4,8,16,32]`

F. *Tuples [LYaH Ch.2,p.14]*

- Tuples are similar to lists but have fixed length. They aren't the same as lists.
- In Haskell, the pair type of `a` and `b` is written `(a, b)`, not `a` × `b`
- (Note: The `:t` command below gives the type of an expression. *[LYaH Ch.3, p.1]*

```
> (1,2)
(1,2)
> :t (1,2)
(1,2) :: (Num a, Num b) => (a, b)
> [1,2] == (1,2)

<interactive>:24:10: error:
    • Couldn't match expected type ‘[Integer]’
                  with actual type ‘(Integer, Integer)’
    • In the second argument of ‘(==)’, namely ‘(1, 2)’
```

```
          In the expression: [1, 2] == (1, 2)
          In an equation for ‘it’: it = [1, 2] == (1, 2)
    > fst (1,2) -- first element of pair
    1
    > snd (1,2) -- second element of pair
    2
```

G. Zip [LYaH Ch.2, p.15]

- `zip` takes two lists and returns a list of pairs: a pair with the first elements of the two lists, a pair with the second elements, etc.

```
    > z = zip [1,2,3] ['a','b','c']
    > z
    [(1,'a'),(2,'b'),(3,'c')]
    > head z
    (1,'a')
    > z = zip [1,2,3] [6,7,9]
    > z
    [(1,6),(2,7),(3,9)]
```

H. Types in Haskell

What's a "Type"? [not in LYaH]

- Historically a type was a name used to identify a set of values implemented by hardware. (Integer, float, etc.). Nowadays, **primitive types** still do that, but we also have complex or **constructed types** that are more like logical descriptions of a collection of values and less about the internal representation of values. E.g., a list of integers might be implemented in different ways, but regardless, it will support functions like `length`, `head`, and so on.
- A **type is a name** for a collection of values, either primitive or built up from simpler types; the name refers to a collection of values.
  - Historically, types were introduced to describe the values implemented by hardware.
  - Nowadays, **primitive types** still do that, but we also have complex or **constructed types** that are more like logical descriptions of a collection of values and less about the internal representation of values.
- In Haskell, the `Int`, `Integer`, `Char`, `Float`, `Double`, and `Bool` are **type constants** (type names) for the **primitive types** (the basic built-in kinds of values).
  - (`Int` is for fixed-length integers; `Integer` is for unbounded integers — we'll see the difference momentarily).
- More complex types are built using operations named by **type constructors**.
  - Built into Haskell are `[…]`, `(…, …)`, and `->` ("arrow") for lists, *n*-tuples, and functions respectively.
  - There are also user-defined algebraic types (e.g., trees) that we'll study later.

I. Declaring a Function's Type [LYaH Ch.3, p.2]; Multi-Line Input [Not in LYaH]

- A functional type is written as *type* `->` *type*, such as `[Char] -> Int`.
  - (A function that returns the length of a string would have this type.)
- We can give an explicit type for a function using *name* `::` *type* `->` *type*, typically on the line before the function's definition. *[LYaH Ch.3, p.2]*
  - E.g., we might have `f :: Int -> Int` as the type for `f x = 2 * x + 3` .
- **To enter multiple related lines of input in the interpreter**, we have to surround the lines with two extra lines, `:{` before and `:}` afterwards. (Notice how the prompt changes below.)

```
Prelude> :{
Prelude| f :: Int -> Int
Prelude| f x = 2 * x + 3
Prelude| :}
Prelude> :t f
f :: Int -> Int
Prelude> f 5
13
```

- We've seen `:set prompt` to change the `Prelude>` prompt; to change the `Prelude|` prompt, use `:set prompt-cont` . E.g.,

```
Prelude> :set prompt "> "
> :set prompt-cont "| "
> :{
| f :: Int -> Int
| f x = 2 * x + 3
| :}
> :t f
f :: Int -> Int
> f 5
```

J. Int vs Integer [LYaH Ch 3., p.2]

- `Int` is for bounded (fixed-length) integers; `Integer` is for unbounded (infinite precision) integers.
- To show the difference, here are a couple of versions of factorial. Since 30! is too large to be an `Int`, the `Int` version of factorial overflows, but the `Integer` version does not.

```
> :{
| fact1 :: Int -> Int
| fact1 x = product [1..x]
| fact2 :: Integer -> Integer
| fact2 x = product [1..x]
| :}
> fact1 20              -- ok
```

```
2432902008176640000
> fact1 30              -- overflows
-8764578968847253504
> fact2 30              -- ok
265252859812191058636308480000000
```

K. Type Variables; Parametric Polymorphic Types [LYaH Ch.3, p.3]

- Haskell (and various other languages) have **type variables**, which are identifiers that can stand for various different types.
- E.g., the (built-in) identity function `id x = x` is said to have type `a -> a`, which simply means that `id` can be used on any type of value (the first `a`) and produces a result of the same type (the second `a`).
  - We can use any type for `a` (but we have to be consistent and use the same type for both `a`'s.).
  - E.g., `id` can be used as an `Int -> Int` function or `[Char] -> [Char]` function and so on.
  - Note the type for `a` can even be a functional type, so `id sqrt` is legal (and its result behaves just like `sqrt`, as in `id sqrt 16.0 == 4.0`).
- A **parametric polymorphic type** is a type that includes a type variable, such as `a -> a`.
  - It's said to be parametric because we substitute an actual type when we use the type, similarly to how we substitute an argument value for a parameter variable in a function call.
  - A polymorphic type can include multiple type variables. E.g., `swap (x, y) = (y, x)` has type `(b, a) -> (a, b)`
    - Note it doesn't matter what the letters are, all that matters is the pattern of how we use them: `(b, a) -> (a, b)` and `(c, d) -> (d, c)` and even `(a, b) -> (b, a)` all indicate the same type. (I used `(b, a) -> (a, b)` because that's what `ghci` reports if you type in the definition of `swap` and then ask `:t swap`.)
  - Note the kind of polymorphism here is different from the inheritance polymorphism used in object-oriented programming.

L. Parametric Polymorphism vs Templates; Boxed and Unboxed Values [added 1/17]

- Parametric polymorphism of functions and templates are similar but not identical.
- Templates:
  - In C++, a function template indicates a family of functions, parameterized by one or more typenames, so templates let you write one piece of source code that generalizes across typenames.
  - But though the source code is the same, the compiler creates different object code for different instances of the typename. (So if you had a `typename T`, you would generate different object code for `int T` versus `double T` versus etc.)

    - The compiler has to do this because the low-level code that needs to be generated might be different across different types. (Integer addition and floating-point addition use different assembler instructions.)
- Parametrically polymorphic functions:
    - In Haskell (and similar languages), there's only one piece of code associated with a polymorphic function, and this code works for all types of arguments.
    - E.g., take the function `f x = [x]`, which takes a value and returns a singleton list containing it. Different calls of `f`, such as `f 17` and `f "hello"`, can have different argument types, but there's only one piece of code for `f`, and it gets used for all arguments.
- Boxed and unboxed values:
    - The code that handles `x` has to be the same regardless of the type of value of `x`, so we need a uniform representation of data. In particular, basic hardware-supported data uses bitstrings of different length, so implementations **box** (a.k.a. **wrap**) the data.
    - E.g., in Java, regular integers are of type `int`, but boxed integers are of class `Integer`. When you call a routine that expects a value that's from a class, you pass an `Int` so that the called routine always gets internally, the same kind of data.
    - Implementing a boxed `Integer` takes more space than a raw bitstring `int` (you need a pointer to an object that contains an `int`), so you might need to box and unbox data frequently, which may chew up extra memory space.
    - In older versions of Java, you had to explicitly box data (via a constructor, as in `Integer x = new Integer(17);`) and unbox data (by referencing the object field containing the data, as in `x.intValue()`.). In contemporary Java, boxing and unboxing is done more automatically, which makes program writing easier but may hide implementation inefficiencies.
- Back to Haskell: A polymorphic function like `f x = [x]` expects a boxed representation of `x` at runtime. (Think of it as passing a pointer to a raw integer or float if you call `f 2` or `f 2.5`.) Boxing does require extra space, but since Haskell values are immutable (unchangeable), we can share multiple pointers to the same raw data. Unboxing of data (to do arithmetic, for example) occurs deep in the implementation, not during passing around of data.

M. Type Inference [LYaH Ch.3, p.1]; Explicit Type Annotations [LYaH Ch.3, p.6]

- In Haskell, you can declare the type of a function or expression; the typechecker will make sure that your declared type is consistent with the code. (If it isn't you'll get an error message of course.) It's quite common to include an explicit type annotation.

```
> :{
|  makePalindrome :: [Char] -> [Char]
|  makePalindrome s = s ++ "_" ++ reverse s
| :}
> makePalindrome "hello"
"hello_olleh"
```

- We've been omitting these explicit type annotations because Haskell is very good at **type inferencing** — deducing the type of an expression or function.

```
> makePalindrome2 s = s ++ "_" ++ s
> :t makePalindrome2
makePalindrome2 :: [Char] -> [Char]
```

- The `makePalindrome` function has a **monomorphic type** (just the one type), but for polymorphic code, Haskell will figure out the **most general type** that can be used. *[not in LYaH]*
- Below, Haskell infers the type `b` -> (`b`, `b`) for, `mkpair` infers because it finds nothing to restrict the type of the first `x` (hence the `b` in `b` -> …) and then uses that type to generate the most general type for the result (the (`b`, `b`) in … `b` -> (`b`, `b`)). But for the application `mkpair 'x'`, Haskell infers the type (`Char`, `Char`) because `'x' :: Char`. Similarly, for `mkpair "hi"`, Haskell infers `([Char]`, `[Char])` as the result type.

```
> mkpair x = (x, x)
> :t mkpair
mkpair :: b -> (b, b)
> mkpair 'x'
('x','x')
> :t mkpair 'x'
mkpair 'x' :: (Char, Char)
> :t mkpair "hi"
mkpair "hi" :: ([Char], [Char])
```

- The inferred type doesn't have to be monomorphic. Below, `mkpair` is used on a list value; we don't have a restriction on what kind of list, so the list and the result have a polymorphic type.

```
> mkdouble_pair x = mkpair [x]
> :t mkdouble_pair
mkdouble_pair :: a -> ([a], [a])
```

- You can use explicit type annotations to restrict what would otherwise be a more general type for an expression or function. Here are three versions of `mkpair` that require a list argument; the different function declarations are equivalent, as you can see from their types. Note `mkpair_list` creates a different piece of code from `mkpair` (it takes `x` and calls `mkpair` on it). In contrast, `mkpair_list2` and `mkpair_list3` name exactly the same piece of code as `mkpair_list`.

```
> :{
|  mkpair_list :: [a] -> ([a], [a])
|  mkpair_list x = mkpair x         -- declaration with parameter
|
|  mkpair_list2 :: [a] -> ([a], [a])
|  mkpair_list2 = mkpair      -- declaration without parameter
|
|  mkpair_list3 = mkpair :: [a] -> ([a], [a])   -- on 1 line
|  :}
```

```
> :t mkpair_list
mkpair_list :: [a] -> ([a], [a])
> :t mkpair_list2
mkpair_list2 :: [a] -> ([a], [a])
> :t mkpair_list3
mkpair_list3 :: [a] -> ([a], [a])
```

- (If you don't like typing `:t` all the time, you can cheat :-)

```
> x = (mkpair_list, mkpair_list2, mkpair_list3)
> :t x
x :: ([a1] -> ([a1], [a1]), [a2] -> ([a2], [a2]),
      [a3] -> ([a3], [a3]))
```

N. Operator Overloading and Type Classes

Type Classes Generalize Operator Overloading

- In Haskell, as in other languages, you can use + on various different types of data — integers, floats, and so on. In (e.g.) C, this is called **operator overloading**. There, we say that + can take two integers and return an integer or two floats and return a float and so on.
- Overloading is a *kind of polymorphism* ("+" has multiple types because it has multiple meanings). There are different pieces of code for each meaning, so C has to figure out which meaning you want by looking at the types of the arguments / result; the code for adding integers is different from the code for adding floats.
- Haskell has a feature called **typeclasses** that formalizes and generalizes operator overloading by providing constraints on types. A type class is a collection of types that are guaranteed to support some particular operations. A type becomes an **instance** of a class when it's shown to support all of the required operations.
- E.g., the type class `Num` is for types that support `+`, `-`, `*`, and some related functions. The built-in types `Int`, `Float`, and `Double` are all instances of `Num`. The code denoted by `+` is different for the different types, which is why typeclasses support operator overloading.
- User-defined types can also be instances of `Num`. You're required to write an instance declaration, which simply shows that you've defined all the operators / functions required by the class.
- A function supported by a type class doesn't have to have the same implementation across different instances. E.g., `Int` and `Double` are both instances of `Num`, so they both have + operators, but the two types can (and do) have different implementations of +. This is similar to operator overloading in (e.g.) C, where there are different versions of + depending on which types of values you pass to it.

Primitive Types [LYaH Ch.3, p.2] and Some Standard Type Classes [LYaH Ch.3, pp.4, 7]

- Haskell's primitive types include `Int`, `Float`, `Double`, `Char`, `Bool`, and `Integer`. (Recall that `Int` is for 64-bit integers; `Integer` is for unbounded ("infinite precision") integers.)
- All five of these types support equality testing and comparisons <, >, etc. Class `Eq` is for `==` and `/=`; class `Ord` extends this to include <, >, etc. ("`Ord`" is short for "ordinal", which means "can be ordered".)

  - The basic function for `Ord` is `compare` :: `a` -> `a` -> `Ordering`, where `Ordering` is a type with three values, `LT`, `EQ`, and `GT`. Then `x` < `y` means `compare x y` == `LT`, etc.
- `Integer`, `Int`, `Float`, and `Double` differ from `Char` and `Bool` by being numeric types; as instances of type class `Num`, they support addition, subtraction, multiplication, absolute value, etc.
- `Float` and `Double` are also instances of `Fractional`, so they support floating-point division (`/`). They are also instances of `Floating`, so they support `sqrt`, `**` (exponentiation), `log`, and the trigonometric functions.
- (I skipped `Enum` and `Bounded`; I think these other typeclasses are more useful to us.)

Typeclass	***Subclass(es)***	***Supports***	***Instances Include Built-In Types***
`Num`		`+ - * abs ...`	`Integer Int Float Double`
`Fractional`	`Num`	`/`	`Float Double`
`Floating`	`Fractional`	`sqrt ** log sin ...`	`Float Double`
`Integral`	`...`	`quot rem div mod toInteger`	`Integer Int`
`Eq`		== /=	`Integer Int Float Double Char Bool`
`Ord`	`Eq`	`< > <= >= compare`	`Integer Int Float Double Char Bool`

- The `Show` class [LYaH Ch.3, p.5]: This class supports a function `show` :: `a` -> [`Char`] that takes a value and returns a printable representation of it. The types `Integer Int Float Double Char Bool` are all instances of `Show`. If a types `a`, `a1`, `a2` … are showable, then so are [`a`], (`a1`, `a2`), (`a1`, `a2`, `a3`), … (up to 15-tuples). To print a value, the interpreter prints the string `show` *value*; since no function types are instances of `Show`, the call `show sqrt` (e.g.) fails, which is why you can't print functions.
- The `Read` class [LYaH Ch.3, p.5]: This class supports a read function, `read ::` `Read a` => `String` -> `a`. ("`String`" is a synonym for "[`Char`]".) The type of `read` says that the result could be of any type, but the format of the string you `read` will yield some particular type.
- E.g., you can read `"17"` as an `Int` or `Integer` or `Float`, etc., but you can't read it as a `Char`. (This seems reasonable.) On the other hand you can read `"'a'"` as a `Char` but not an `Int`. (Sorry, no automatic ASCII conversion..)

```
> read "17" :: Int
17
> read "17" :: Float
17.0
```

```
> read "'a'" :: Char
'a'
> read "17" :: Char
*** Exception: Prelude.read: no parse
> read "'a'" :: Int
*** Exception: Prelude.read: no parse
```

- Producing a typeclass value is a little tricky. You can read `"17"` as an `Int` or an `Integer` but to read it as a more general `Integral t => t`, you have to include the constraint `Read t`.

```
> read "17" :: Integer
17
> read "17" :: Integral t => t
<interactive>:61:1: error:
    • Could not deduce ... -- (long error message)
> read "17" :: (Integral t, Read t) => t
17
> read "17.0" :: (Fractional t, Read t) => t
17.0
```

-- 2020-01-16

O. The `:browse` *and* `:info` *commands (not in LYaH)*

- In ghci, you can use `:browse` to see all the types and functions available for you to use. It makes for a very long list, overwhelming in detail, but you can glean useful information. For typeclasses, it includes the functions / types / values that the typeclass supports.

```
> :browse
        -- many lines omitted
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
        -- many lines omitted
```

- The `:info` *name* command gives you different information depending on the kind of name. Given the name of a function, you get its type (like `:t` *function_name*) and the typeclass it's part of, if any. E.g., sqrt turns out to be part of the `Fractional` typeclass.

```
> :info sqrt
class Fractional a => Floating a where
  ...
  sqrt :: a -> a
  ...
  -- Defined in ‘GHC.Float’
```

- For a type name, you get information about where the type is declared and what typeclasses it's an instance of. E.g., the `Char` type is an instance of `Eq`, `Ord`, `Show`, `Read`, and other typeclasses we didn't look at.

```
> :info Char
data Char = GHC.Types.C# GHC.Prim.Char# -- Defined in ‘GHC.Types’
instance Eq Char -- Defined in ‘GHC.Classes’
instance Ord Char -- Defined in ‘GHC.Classes’
instance Show Char -- Defined in ‘GHC.Show’
instance Read Char -- Defined in ‘GHC.Read’
  -- (lines omitted)
```

- For a typeclass, you get the names of any typeclasses it depends on, a list of what the typeclass includes, and the types that are an instance of it.

```
>:info Fractional
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
      -- (some lines omitted)
instance Fractional Float -- Defined in ‘GHC.Float’
instance Fractional Double -- Defined in ‘GHC.Float’
```

P. Polymorphic constants [LYaH Ch.3, p.7]; No automatic type conversion

- **Polymorphic constants**: In Haskell, the constants 0, 1, 2, etc. have polymorphic type: They are `Num` values, not plain integers. If you want one to have a monomorphic type, you have to control their usage (or use explicit type annotations).

```
> :t 17
17 :: Num p => p
```

- **No automatic type conversions**: Haskell does not have the kinds of automatic type conversion you might expect, such as integer to floating point or integer to longer integer. E.g., to convert an `Int` to `Integer`, there's a class-supported function `toInteger` :: `Integral a` => `a` -> `Integer`

```
> int17 = 17 :: Int
> integer17 = 17 :: Integer
> int17 + integer17                    -- fails!
> sum = toInteger int17 + integer17     -- works ok
> :t sum
sum :: Integer
```

- But then why does `17` / `17.0` work?
- This goes back to polymorphic constants. `17.0` is of type `Fractional t` => `t`, and one of the types `17` can have type is also `Fractional t` => `t`. The / operator has the type `Fractional t` => `t` -> `t` -> `t`, so `17` (treated as a Fractional) / `17.0` works fine. You can see this if you explicitly declare that your `17` is a `Fractional`. If you declare `17` to be an `Int`, then you get a type mismatch.

```
> 17 / 17.0
1.0
> (17 :: Fractional t => t) / 17.0
1.0
```

```
> (17 :: Int) / 17.0 -- fails because there's no (/) :: Int -> ...
operator.
<interactive>:50:1: error:
    • No instance for (Fractional Int) arising from a use of ‘/’ ...
```

CS 440: Lecture 2 Activity Questions

1. a. What is type of value is `[(+), (-), (*), div, rem]` ?

   b. What happens if you type `x + y = 12` into ghci? How much is `2+2` now? How do you get back the original behavior of `+` ?

2. Recall the fibonacci sequence: $f(0) = 1, f(1) = 1, f(2) = f(0) + f(1), \ldots, f(n{+}2) = f(n) + f(n{+}1)$. In Haskell, declare `fib = (1, 1) : [(fnp1, fn + fnp1) | (fn, fnp1) <- fib]`. Try using referential transparency to argue what `take k fib` and `fib !! k` are (for an arbitrary $k \geq 0$).

3. Suppose we wanted to write a function `cvt` that would take strings like "`Int 17`", "`Integer 17`", "`Float` 17.0", etc., and return the read-in value, with its correct type? (We'd be looking for something like)

```
> :t cvt "Int 17"
17 :: Int
> :t cvt "Integer 17"
17 :: Integer
> :t cvt "Float 17.0"
17.0 :: Float
```

   Why is it not possible to write such a `cvt` function? (Hint: What type would if have?)

4. Say we declare a version of the identity function restricted to pairs and use it in various ways.

```
> idp = id :: (a, b) -> (a, b)
> f1 (x, y) = idp (x, y)
> f2 p = idp p
> f3 (x, y) = (id x, id y)
> f4 (x, y) = (idp x, idp y)
```

   a. What are the types of `f1`, `f2`, `f3`, and `f4`?

   b. Try applying these functions to, say, `17`, `(17, "18")`, `((2, 3.5), ('z', "abc"))`. Which ones cause problems and why?

Write definitions for three list-handling functions

5. `firsthalf x` = first half of list x. E.g., `firsthalf [1,2,3,4] = [1,2]`
   If the list is of odd length, ignore the middle element: `firsthalf [1,2,3,4,5] = [1,2]`
   Use `length x`, division by 2, and `take`. `div y 2` is division by 2 dropping remainder

6. `secondhalf x` = second half of list x. E.g., `secondhalf [1,2,3,4] = [3,4]`

If the list is of odd length, ignore the middle element: `secondhalf [1,2,3,4,5] = [4,5]`
`rem y 2` is remainder of integer division.
Possible hint: `z + rem y 2` is `z+0` or `z+1` depending on `y` being even or odd

7. `pal x` = Is x a palindrome? (does first half equal reverse of second half?)

8. In Haskell, function composition is infix dot (i.e, period). In ghci, what happens on the inputs below? Any errors? Why?

```
> sqrt (sqrt 16)
> (sqrt . sqrt) 16
> sqrt sqrt 16
```

9. In ghci, define the following.
   a. Which give errors and why?

```
> fs = (sqrt, sqrt)
> head fs 81
> fst fs 81
```

10. In ghci, enter the following. What are the types of `flst`, `f1`, `f2`, and `g`? Which of the applications of `f1`, `f2`, etc. calculate a result? Which cause errors, and why?

```
> flst = [sqrt, sqrt]
> f1 = head flst
> f2 = last flst
> g = tail flst
> f1 16
> f2 16
> g 16
> head g 81
> last g 81
```

Activity Solution

1. a. `[(+), (-), (*), div, rem] :: Integral a => [a -> a -> a]`

   b. > x+y = 12 declares a binary + operator that always returns 12, so now > 2+2 yields 12. I don't know of any way to recover the original behavior of + (aside from quitting ghci and restarting it).

3. The problem with a `cvt` routine that can return an `Int`, `Integer`, `Float`, or `Char`, is that we can't form a return type that combines those types. If you run `:info` on each of those types, you'll see what typeclasses they belong to. We could return an `Int` or `Integer` using typeclass `Integral` or `Int` and `Integer` and `Float` using `Real` (a typeclass we didn't look at) but no typeclass combines `Char` and anything numeric. (We could try defining our own typeclass, but that requires a `datatype` definition, which we haven't seen yet.)

4. (Identity function on ordered pairs)

   a. Using `:t f1`, `:t f2`, and `:t f3`, we find that `f1`, `f2`, and `f3` are of type (`a`, `b`) -> (`a`, `b`). The type of f4 is more complicated because it takes an ordered pair of ordered pairs: `f4 ::` ((`a`, `b`), (`c`, `d`)) -> ((`a`, `b`), (`c`, `d`)).

   b. All four of `f1`, …, `f4` produce errors on argument `17` because it's not an ordered pair. We can use `f1`, `f2`, and `f3` on (`17`, `"18"`), which get treated as functions of type `Num a` => (`a`, [`Char`]). All four work on ((2,3,5), (`'z'`, `"abc"`)) and are treated as functions of type (`Num` a, `Num` b, `Num` c) => (`a`, `b`, `c`) -> (`Char`, [`Char`]). (There are three `Num` constraints because the `2`, `3`, and `5` can be different types of numbers.)

5. `firsthalf x = take (div (length x) 2) x`
   Take first *N* items of `x` where *N* is `div` (`length x`) `2`, i.e., half the length of 2 ignoring remainder from division of 2.

6. For `secondhalf`, let's look at an initial buggy version then fix it.
   (Buggy): `secondhalf x = drop (div (length x) 2) x`. This works okay for even-length lists but it includes the middle element for odd-length lists. i.e., `secondhalf [1,2,3,4,5] = [3,4,5]`.

   If *N* = `div (length x) 2`,, then for even-length `x`, we want to drop the first *N* elements. For odd-length `x`, we want to drop the first *N*+1 elements. Turns out we can drop the first *N* + `rem` (`length x`) `2` elements because the remainder is 0 for even lengths and 1 for odd lengths. So a correct solution is

   ```
   secondhalf x = drop (div (length x) 2 + rem (length x) 2) x
   ```

Some tests:

- `secondhalf [1,2,3,4,5,6] = [4,5,6]`
- buggy `secondhalf [1,2,3,4,5,6,7] = [4,5,6,7]` (bug: includes middle element 4)
- fixed `secondhalf [1,2,3,4,5,6,7] = [5,6,7]` (omits middle element)

7. The palindrome testing is straightforward. We need the parentheses below because function application is left associative.

```
pal x = firsthalf x == reverse (secondhalf x)
```

Some tests:

- Return true: `pal [], pal [1], pal [1,1], pal [1,2,1], pal [1,2,3,2,1]`
- Also true (since strings are lists of characters): `pal "", pal "1", pal "11", pal "121"`
- Return false: `pal "12", pal "1312"`