Haskell

Basic Syntax

Compiling/Running

- For example, C `main()` is the entry for any Haskell program.
- We need a main function if we're compiling our code into an executable.
  - You don't need a main if you're using the GHCi shell.
- To compile a Haskell function, you use the `ghc` command.
- Like Elixir and Python, Haskell has its interactive interpreter and debugger, `ghci`
- The program `runghc` allows you to run Haskell programs directly without manually compiling them.
- Tabs don’t work properly unless they’re eight spaces exactly.

Example

```haskell
main = putStrLn "Hello world!"
```

- To run a program, compile it (similar to GCC)

```
ghc -o a test.hs
./a
```

Arithmetic

Operator Precedence

Precedence	Operator	Description	Associativity
highest	`f x`	Function application	Left
9	`.`	Function composition	Right
8	`^` `^^` `**`	Power	Right
7	`*` `/` `quot` `rem` `div` `mod`		Left
6	`+` `-`		Left
5	`:` `++`	Append to list	Right
4	`==` `/=` `<` `<=` `>=` `>`	Comparisons	
4	`<*>` `<$>`	Functor ops	Left

Precedence	Operator	Description	Associativity
3	`&&`	Logical AND	Right
2	|| `(can't put OR in a CB)`	Logical OR	Right
1	`>>` `>>=`	Monadic ops	Left
1	`=<<` `<|>`		Right
0	`$` `$!` `seq`		Right

Functions

- Functions don't need parenthesis for their arguments.
  `sqrt 2 -> 1.4142135...`
  - They do need parenthesis if the parameter is negative.
    `func (-6)`

Infix Functions

- You can use symbolic operators as functions using parentheses `()`
  - Note; Or `||` is not shown on the table below because it gets fucked up in markdown tables, but it does work as an infix.

Name (As a Function)	Example Usage and Result
Add `(+)`	`(+) 5 10 -> 15`
Subtract `(-)`	`(-) 10 5 -> 5`
Multiply `(*)`	`(*) 5 10 -> 50`
Divide `(/)`	`(/) 10 5 -> 2.0`
And `(&&)`	`(&&) True False -> False`
Equals `(==)`	`(==) 5 5 -> True`
Not Equals `(/=)`	`(/=) 5 4 -> True`
Less Than `(<)`	`(<) 5 10 -> True`
Greater Than `(>)`	`(>) 10 5 -> True`
Less Than or Equal `(<=)`	`(<=) 5 5 -> True`
Greater Than or Equal `(>=)`	`(>=) 5 5 -> True`
Concatenate `(++)`	`(++) [1] [2, 3] -> [1,2,3]`
Cons `(:)`	`(:) 1 [2, 3] -> [1,2,3]`
Sequence `(>>)`	`(>>) (print 1) (print 2) -> (prints 1 then 2)`

Tuples

- Tuples in Haskell are denoted with parenthesis.
  `(3, 5)`
- Haskell tuples don't need to have elements of the same type.
  `(5, "Hello")`
- There are built-in functions to access a tuple's first and second elements: `fst` and `snd`.
  - This is particularly useful for coordinates.
  - `fst` and `snd` only work on pair tuples.
    `fst (5, "Hello") -> 5` snd (5, "Hello") -> "Hello"

Lists

- Lists in Haskell are homogenous, meaning all elements must be of the same type.
  - Integers placed in a float list will be inferred as floats `[1, 2.5] -> [1.0, 2.5]`
  - This doesn't work with chars in an integer list.
- Elements can be added to the beginning of a list with the `:` (cons) operator.
  `0:[1, 2] -> [0, 1, 2]`
  - This also allows you to build lists using just cons and empty lists.
    `0:1:2:3:[] -> [0, 1, 2, 3]`
  - Technically, any list definition, for example, `[1, 2, 3]`, is just syntactic sugar for building the list using cons and an empty list
- We can have lists of tuples.
  - The tuples can be heterogeneous.
  - The list has to stay homogeneous.
  - While the tuples can have different items inside them, they all have to have the same format.
    `[(1,2), (3,4), (5,6)]`
    `[(1, 'a'), (3, 'b'), (5, 'c')]`

List Functions

- `map`, `filter`, `foldr`, & `foldl` are all first-class functions

`map`

- `map` is similar to Elixirs `Enum.map`.
  - map operates on lists
    - A string is a list of chars.
  - In the example, `map Data.Char.toUpper "Hello, World!" -> "HELLO, WORLD!"`
    - `Data.Char.toUpper` is our function.
    - `"Hello, World!"` is our list.

`filter`

- `filter` removes items from a list based on specific criteria
  `filter Data.Char.isLower "Hello, World!" -> "elloorld"`

`foldr` & `foldl`

- `foldr` is similar to elixir's `Enum.reduce` in the fact that it performs a function upon a list using an accumulator
- `foldr`, in effect, replaces the cons operator with another function
  - The empty list is replaced with some initial value.
- In this example, `foldr (+) 0 [1, 2, 3, 4, 5]`
  - `(+)` is the function
  - `0` is the initial value (accumulator)
  - `[1, 2, 3, 4, 5]` is the list
  - `foldr (+) 0 1:2:3:4:5:[]` -> `1 + 2 + 3 + 4 + 5 + 0` -> `15`
  - You can use `foldr` to calculate factorials.
    `foldr (*) 1 [1, 2, 3, 4, 5]`-> `15`
- `foldr` is right-associative;
  - `foldr (+) 0 [1, 2, 3, 4, 5]` -> $(1 + (2 + (3 + (4 + (5 + 0))))) = 15$
  - This doesn't matter for addiction, but for subtraction and other non-commutative functions, it does
    - `foldr (-) 1 [4, 8, 5]` -> $(4 - (8 - (5 - 1))) = 0$
- `foldl` is left associative
  - `foldl (-) 1 [4, 8, 5]` -> $(((1 - 4) - 8) - 5) = -16$

`head` & `tail`

```
- `head` returns the first element of a list
      `head [1, 2, 3] -> 1`
      `head [1] -> 1 `
- `tail` returns the rest of the list (as a list)
      `tail [1, 2, 3] -> [2, 3]`
      `tail [1] -> []`
      `tail [] -> Exception: empty list.`
```

List Generation

- There are a few ways to generate lists using syntactic sugar.
  - Double periods `n..m` signify a range from n to m
  - `list = [1, 2, 3, 4, 5, 6, 7, 8, 9]` is the same as `list = `[1..9]`
  - Commas `,` can be placed interchangeably with double periods.
    - For example, `list = [1,3..9]` is equal to `[1,3,5,7,9]`

- An infinite list can be made using double periods without specifying an `m` value.
  - The list will generate forever until it is interrupted.
  - If we bind the variable x to the expression to generate an infinite list, Haskell won't evaluate it until it has to `x = [1..]`
    - Displaying the list requires evaluation.
    - Finding the length of the list requires counting the elements.
  - We can perform operations on a finite subset of the infinite list.
    `take 3 x -> [1,2,3]`
    `take 3 (drop 5 x) -> [6,7,8]`
- `zip` generates a new list of tuples using two lists.
  - If the first list is finite, the second can be infinite.
    `zip "Hello" x - > [('H',1), ('e',2), ('1',3), ('1',4), ('o',5)]`

Strings

- Strings are an empty list of chars that get special treatment.
- Strings are made by adding chars using the cons operator on an empty list.
  `'H':'e':'l':'l':'o':[] -> "Hello"`
- We can concatenate strings and lists using the `++` operator.
  `"Hello, " ++ "World!" -> "Hello, World!"`
- To concatenate different types into a string, you use the `show` function.
  `show 500 ++ " warrants for my arrest" -> "500 warrants for my arrest`
- You use the `read` operator to read a numeric value from a string.
  `read "506" - 8 -> 498`
  - `read` sends an exception when no numeric value is present

Modules and Functions

Function

- Functions can be defined using the `=` operator.
  - In the example, `square x = x * x`;
    - We define a function called `square` that takes in one argument (`x`)
    - `square` computes `x * x` and returns it
    - To call `square`, we just pass it a parameter.
      - For example
        `square 2 -> 2`
        `sqaure (sqrt 2) -> 2.0000000000004`
- Functions can have multiple parameters.
  - For example;
    `sum a b c d = a + b + c + d`

    - To call this function, we need four arguments.
      `sum (square 2) (cube 2) 3 4 -> 19`

Function Composition

- Instead of calling nested functions in the typical way
  `fac(fib(4)) -> 6`
- You can make calls using the period `.` for example;
  `(fac.fib) 4 -> 6`
- They do the same thing.

Lambda Functions

- They are like any other language.
- The lambda function in Haskell uses a `\` and `->`
- Lambda functions don't need names.
- They are good for passing as arguments when that's the only place you need them.
    - They work very well with map.

Examples

Lambda with one input

```
> square = \x -> x*x
> square 8
64
```

Lambda with two inputs

```
> f = \x y -> 2 * x + y
> f 3 4
10
```

Lambda with no names

```
-- these functions always return true or false
> (\x -> True) True
True
> (\x -> True) False
True

-- this function returns the opposite of whatever it is given
> (\x -> not x) False
True
```

```
> (\x -> not x) True
False
```

Using the `map` function

```
> map (\x -> -x) [1,-3, 5, 6, -9]
[-1,3, -5, -6, 9]
```

Returns

- You must use a list or tuple to return multiple things and unpack them later.

Example

```
module Test where

sumAndProduct :: Int -> Int -> (Int, Int)
sumAndProduct x y = (x + y, x * y)

main :: IO ()
main = do
-- pattern matching the local variables to the return
  let (sumResult, productResult) = sumAndProduct 3 4
  putStrLn ("The sum is " ++ show sumResult)
  putStrLn ("The product is " ++ show productResult)
```

Local Names

- To bind a local variable to a value in a function, you can use the `let` keyword.
- This allows that variable only to be accessed within that module.
- If you do use the `let` keyword in a function, make sure to use `do` beforehand.
  - This is because `let` counts as its own expression and you can only have one expression per function.

`do/let

`do/let` is used when you want a temp variable midway through or at the end of a function

- For example, a string you want to return.`

```
module Test where

main :: IO ()
main = do
```

```haskell
  let message = "Hello, world!"
  putStrLn message
```

`let/in`

You can also use the `let/in` keywords instead of using `do`

- This allows you to specify where the local variable is used.
- You use `let/in` when you have variables at the top of the function that you want to define before.

```haskell
module Test where

roots a b c =
  let disc = sqrt(b * b - 4 * a * c)
  in ((-b + disc) / (2 * a), (-b - disc) / (2 * a))
```

`where`

- The `where` lets you create local bindings for values or functions at the end of a function or a pattern matching expression.
  - `where` bindings are not technically expressions
- Any Bindings made in a `where` clause are only in scope for the function or case alternative they're attached to.
  - Meaning they're only accessible within the body of the function
- You can define multiple bindings in a `where` clause, and they can all see each other.

```haskell
area width height = widthAdjusted * heightAdjusted
  where
    widthAdjusted = width * scaleFactor  -- Local binding
    heightAdjusted = height * scaleFactor -- Another local binding
    scaleFactor = 1  -- scaleFactor is visible to both widthAdjusted and
heightAdjusted
```

Modules

- Modules in Haskell and Elixir are very similar.
- We can load modules into GHCi.
- They allow us to access the module's functions and expressions.

Example Module

```haskell
module Test where

square x = x*x
cube x = x*x*x
sum x y = x + y
```

- To load a module, you use `:load` in GHCi
- In this example, the command would be `:load Test.hs`
- After loading, all functions and expressions from the module will become available in GHCi

Control Flow

`if then else`

- `if then else` is just like any other if statement except for a few minor caveats
  - You need to have a `then` after each `if`
  - You need to have an `else` no matter what
  - You can have nested `else if`'s
- Indenting matters in Haskell, the first `if` should match the indentations of the `else`

Example

- In this example, we have a function named sign that returns.
  - -1 if x is negative
  - 1 is x is positive
  - 0 if x is zero

```haskell
module Test where

sign x =
  if x < 0 then -1
  else if x > 0 then 1
  else 0
```

`case`

- `case` is used when you have a variable that can have many results
  - It's used to pattern-match specific values.
- `case` is just like it is in Elixir
  - `_` are wild, and they are a catch-all

Examples

```
module Test where

isNum x =
  case x of
    0 -> 0
    1 -> 1
    2 -> 2
    _ -> -1

sign x = do
  let q = x
  if q < 0 then -1
  else if q > 0 then 1
  else 0
```

```
module Test where

chkClr rgb =
  case rgb of
    (255, _, _) -> "RED"
    (_, 255, _) -> "GREEN"
    (_, _, 255) -> "BLUE"
    (_, _, _)   -> "None"
```

Piecewise Functions

- Like in Elixir, you can specify different functions for different sets of inputs.
    - This makes recursion super easy.

Examples

```
module Test where

fac 0 = 1
fac x = x * fac (x-1)

fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

```
module Test where

chkAxis (0, _) = (0, 1)
chkAxis (_, 0) = (1, 0)
chkAxis (a, b) = (a, b)
```

- In this example, the function returns the unit vector if the point lies on an axis.
  - If it doesn't, then it just returns the points.

Guards

- guards in Haskell are denoted with `|`
- Guards are much different than elixir; they match a condition and execute the expression next to it.
  - They are another version of if-else but for the beginning of functions.
- You can use otherwise as a catch-all, so if none of the conditions match, the `otherwise` will kick into effect.

Example

- in this example, the function `cmp2` uses guards to check

```
module Test where

cmp2 x y
  | x < y = "First is smaller"
  | x > y = "Second is smaller"
  | otherwise = "Equal"
```

Recursion

- We can use recursion just like any other language using piecewise functions and control flow.
- Tail recursion is not as big of a deal as in elixir because the function call model is different
  - Function calls don’t necessarily create a new stack frame.

Examples

```
module Test where

-- | Computes the length of a list.
-- Here, we treat the input argument as a pair containing
-- the head and tail of the list.
llen :: [a] -> Int
llen [] = 0               -- Base case: empty list has length 0
llen (xh:xt) = 1 + llen xt -- Recursive case: add 1 for the head, then process
the tail
```

```
module Test where

-- | Checks if a number is non-negative.
```

```
pos x = x >= 0 -- Returns true if x >= 0, false otherwise

-- | Custom filter function using recursion.
-- First argument is a Boolean function.
-- Second input is a list.
filt p [] = [] -- Base case: if the list is empty, return an empty list
-- Otherwise, we call the function p with the head of the list.
filt p (xh:xt) =
  if p xh
    then xh : filt p xt -- If the condition is true, include xh and continue
filtering
    else filt p xt      -- If the condition is false, just continue filtering
-- In both cases, make the recursive call with the tail.
```

Types

Types in Haskell

- Haskell uses static type-checking
  - This means that variables cannot change type during execution.
- Every expression is assigned a type.
- A compile error occurs if a function's arguments aren't the expected type.
- Types in Haskell are inferred.
  - We don't need to explicitly state what the type of a variable is.
  - The interpreter can infer from the context of the program.
    - for example in `x = "Hello"`, x is a string.
- We can explicitly specify types in Haskell if we want
  - This is good practice when we want to know what types we want.
- to show what type a variable is you use `:t`

```
-- The type of 1 is polymorphic, being any type that is an instance of the Num
type class.
1 :: Num p => p

-- The type of 1.0 is polymorphic, being any type that is an instance of the
Fractional type class.
1.0 :: Fractional p => p

-- The type of 'a' is Char, indicating a single character.
'a' :: Char

-- The type of "Hello" is [Char], which is equivalent to a String.
"Hello" :: [Char]
```

```haskell
-- The type of the expression 1 > 2 is Bool, indicating a boolean expression.
1 > 2 :: Bool
```

- `Num p => p` means that numbers (`Int`, `Float`, etc.) are in the type class `Num`.
  - A type class is just a generic type.
  - This allows Haskell to treat `7` however it wants as long as it works with the type class `Num`.
- Haskell tries to keep types as generic as possible.
  - We can't pass a `Float` to a function that needs `Int`.
    - But assuming a variable is a `Num`, it can be used wherever `Num` is allowed.
- You use the `::` operator to bind a variable to a type.
  - There's no real reason to do this.
  - The inference engine will figure it out and knows much better than you.

```haskell
-- We can also specify that 5.0 is of type Double.
5.0 :: Double :: Double

-- We can explicitly declare 5 as an Int, and GHCi confirms the type.
5 :: Int
-- Output: 5

-- We can explicitly declare 5 as a Double, and GHCi shows it as 5.0.
5 :: Double
-- Output: 5.0

-- Without an explicit type, 'm' is polymorphic, constrained to the Num type
class.
m = 5
-- When queried, GHCi infers m to be of a type that is an instance of Num.
m :: Num p => p

-- We can explicitly set 'm' to be of type Double.
m = 5 :: Double
-- Now when queried, GHCi confirms that 'm' is of type Double.
m :: Double
```

Type Classes

- In most languages, the `==` operator is overloaded to work with many possible types.
- In cases where we want to compare two values of the same type (for example, type a), we would use a-compare
  - a, in this case, is a type variable, which means it doesn't refer to a specific type but rather stands in for any type.
  - If a concrete type, a, belongs to a certain type class, we say a is an instance of that type class.

    - `Int` is an instance of `Eq`, for example.
- Type classes define a set of operations that can be performed on a type.
- In Haskell, the `==` operator is not overloaded like in C++; instead, it's part of a type class.
- Numeric-type equality operations and string equality operations are performed differently.
  - Numeric type equality compares the numerical values (e.g., 5 == 5).
  - String equality means comparing the sequence of characters in two string objects to make sure every char is the same (e.g., "hello" == "hello").
- When you define an instance of the `Eq` type class for a new data type, you implement `==` (and `/=` for inequality) for that type.
  - it's like `.equals()` in java.
  - This operator isn't defined for all types, just some.
  - It takes in two arguments of the same type and returns a Boolean.
  - We associate `==` with a specific type class containing all the types where `==` is defined.
  - This type class is called `eq` in Haskell.
- The `Num` type class allows numeric values to be `Int` or `Float`.
  - It contains all numbers and the operations that can be performed over them (like addition)
- The `Show` type class consists of all the types for which the show function can convert their values into strings.

Function Type Signatures

- In Haskell, type signatures explicitly state the types of functions.
- They serve as documentation and ensure type correctness throughout the program.
- A function's type signature is typically written on the line above the function definition.
  - This signature defines the types of the function's arguments and return value.

```
add :: Int -> Int -> Int
add x y = x + y
```

- Haskell features strong, static type inference,
  - If you don't explicitly define the type signature, the Haskell compiler will infer it based on how the function is used in the code.
- Providing a type signature is a good practice because it makes your code more readable and helps debug.
- A function type signature can specify type variables, which makes the function polymorphic.
  - Polymorphic means it can operate on any type that fits within the constraints (if any) provided:
  - Type class constraints are powerful because they allow us to write general and reusable functions.

```haskell
compareThem :: (Ord a) => a -> a -> Bool
compareThem x y = x > y
```

- You can have type variables be part of more than one class.
  - You can include multiple class constraints in a type signature,
  - To do this, you can separate the type classes with commas within a single set of parentheses:
- `a` is an instance of both `Num` and `Ord`.
  - this lets you make sure that you can perform arithmetic operations like `+` and also comparisons like `>` on the arguments passed to the function

```haskell
sumAndCompare :: (Num a, Ord a) => a -> a -> Bool
sumAndCompare x y = (x + y) > x
```

- `Int` is indeed an instance of the `Ord` type class, and as such, it supports all the comparison operations like `>`, `<`, `>=`, and `<=`.
  - `Ord` is not just limited to `Int`; it includes other ordered types, such as Char, Double, and even user-defined types with an ordering defined.
- `Num` does not include comparison operations.
  - So, to use both `Num` operations and comparisons, you need to specify both `Num` and `Ord`.

```haskell
incrementAndCompare :: (Num a, Ord a) => a -> a -> Bool
incrementAndCompare x y = (x + 1) > y
```

Custom Data Types

- To create custom data types in Haskell, you use the `data` keyword.
- This allows us to add custom behaviour to our data.
  - for example, adding sum and dot products to tuples to use them as coordinates
  - `Pt3` is the custom data name.
  - The `Float Float Float` is our value for the constructor.
  - `Pt3` is also the name of our constructor function

```haskell
module Test where
        data Pt3 = Pt3 Float Float Float
        ptx (Pt3 x y z) = x
        pty (Pt3 x y z) = y
        ptz (Pt3 x y z) = z
```

Overloading constructors

We can overload constructors by using guards in our constructors.

- If we do this, we also need to change our access functions.

```
module Test where
        data Pt = Pt3 Float Float Float
                | Pt2 Float Float

        ptx (Pt2 x _) = x
        ptx (Pt3 x _ _) = x

        pty (Pt2 _ y) = y
        pty (Pt3 _ y _) = y

        ptz (Pt3 _ _ z) = z
```

Deriving Show

To add the default display behaviour from `Show`. We need to add the `deriving` keyword after our constructor.

```
module Test where
        data Pt = Pt3 Float Float Float
                | Pt2 Float Float
                deriving(Show)
```

Adding Symbolic Operators

To add the symbolic operators so you don't have to make custom `myAdd` functions and other garbage is easy.

- Equality is defined for instances of type class `Eq`.
    - Once we define `==`, we don't have to define `/=` because Haskell assumes it for use.
    - We can still choose to define it if we wish.
- `+`, `-,` etc. are defined for instances of type class `Num`.
    - `abs` and `signum` are defined for `Float`
        - `x1` and `y1` are `Float`, so it will be easy to define them
    - `fromInteger` is a coercion function.
        - It dictates how our custom type can be created from an `Int`.
        - It takes an `Int` and returns a `Pt`.
- To create our implementation of `Show` we can use string concatenation to create the output.

```
module Test where
```

```
        data Pt = Pt2 Float Float

        instance Eq Pt where
          (Pt2 x1 y1) == (Pt2 x2 y2) = (x1==x2 && y1==y2)
          (Pt2 x1 y1) /= (Pt2 x2 y2) = not (x1==x2 && y1==y2)

        instance Num Pt where
          (Pt2 x1 y1) + (Pt2 x2 y2) = Pt2 (x1+x2) (y1+y2)
          (Pt2 x1 y1) - (Pt2 x2 y2) = Pt2 (x1-x2) (y1-y2)
          (Pt2 x1 y1) * (Pt2 x2 y2) = Pt2 (x1*x2) (y1*y2)
          abs (Pt2 x1 y1) = Pt2 (abs x1) (abs y1)
          signum (Pt2 x1 y1) = Pt2 (signum x1) (signum y1)
          fromInteger n = let a = (fromInteger n) in Pt2 a a

        instance Show Pt where
                show (Pt2 x y ) =
                "< " ++ (show x) ++ ", " ++ (show y) ++ " >"
```

Pure Code, Monads, & Actions

Pure Code

- A pure function is a function with no side effects.
  - A function has a side effect if it has an observable interaction with the outside world aside from returning a value.
  - This can include;
    - Modifying global variables
    - Raising exceptions
    - Writing data to display or a file
- Pure code means that every function in the code is a pure function.
- Functions in Haskell are always pure.
  - Functions are evaluated,
- Functions can also be called/evaluated from within actions.

```
findBigger x y = if x > y then x else y -- pure function

main = do
  putStrLn "Enter first number:"
  nStr <- getLine
  let num1 = (read nStr :: Double)
  putStrLn "Enter second number:"
  nStr <- getLine
  let num2 = (read nStr :: Double)
  let big = findBigger num1 num2
  putStrLn ("Larger: " ++ show big)
```

- The best practice is to separate pure code from actions

```
module Test where

-- Pure function
testPos numString = do
  let x = read numString :: Double
  if x < 0 then False else True

-- IO action
positive = do
  putStr "Enter a number: "
  num <- getLine
  return (testPos num)
```

Actions

- Haskell separates pure functions from computations where side effects must be considered.
  - It encodes functions that produce side effects with a special type.
- Actions (specifically IO actions), when executed, are not pure.
  - Actions are executed or run.
  - Actions are values and can be returned by functions or passed as arguments.
  - Actions have a type.
  - Actions cannot be executed from within pure functions.
  - Actions can only be executed from within other actions.
  - Functions that return actions are often casually referred to as actions.
    - This is technically wrong, though, so try not to do this.
- A compiled Haskell program begins by executing a single IO action, which is `main::IO()`
  - The main function is a single action.
  - This action is executed when the program is run.
  - A Haskell program, by itself, is a single action that is executed when we run the program.
  - any number of additional actions can be executed From within this action

```
> :t putStrLn
putStrLn :: String -> IO ()
```

- `putStrLn` accepts a String argument.
- The actual act of printing to the screen does not occur as a result of a function call.
  - Printing to the screen is an action.
  - Actions are values; they have a type!
  - What it returns is an action of type `IO()`
    - When the IO() action is executed, it returns `()`.

                - This can be read as an empty tuple.
        - The action, when executed, produces a side effect.
        - The `putStrLn` function, strictly speaking, does not.
- An action can be thought of as a recipe.
    - This recipe (in the case of IO) is a list of instructions that produce side effects.
    - Creating the recipe does not have side effects.
        - The recipe can be the output of a pure function.
        - Same inputs to the function, same recipe.
    - Actions are values, just like strings and numbers.
        - They are completely inert – they do not affect the real world until executed.

IO Actions

- You can use the `<-` operator to pull out the result from executing an IO action.
    - We can bind a name to that action.
    - The variable is then bound to the result of the IO action

```
> x <- getLine
test --this is input
> x
"test"
```

- We can combine actions and execute one action per line using the `do` keyword.
- `do` is syntactic sugar for `>>`

```
main = do
putStrLn "Hello"
putStrLn "World!"
-- these are the same function
main =
putStrLn "Hello" >>
putStrLn "World!"
```

- If the first action produces a result, it is discarded when using `>>` or do
- if we want to use the result, we can use the `>>=` operator to pipe the result into the next action

```
main =
putStrLn "Hello" >>
putStrLn "World!" >>

getLine >>= putStrLn
```

- We grab a string from the input using `getLine` and then pipe it to `putStrLn` so it's outputted to the terminal.
  - `getLine` returns an action that produces a string
  - `putStrLn` takes a string as an argument.
- We can even pipe to lambda functions to modify the outputs

```
main =
  putStrLn "What is your name?"
  >> getLine
  >>= \name -> putStrLn ("Hello, " ++ name ++ "!")
```

- The value of a `do` block is the value of the last expression evaluated.
  - Be careful when executing actions in `main` because the return type must be an action, or you will get an error.

```
main = do
  putStrLn "Enter a number:"
  numStr <- getLine
  let num = (read numStr :: Double)
  let sq = sqrt num
  return ()
```

Non-main actions

- For any function that creates an IO action, the return type must be some type of `IO`.
- It can't be Boolean
  - We can get around this by using the `return` keyword.
  - it creates an action that, when executed, returns a bool

```
module Test where

positive = do
  putStr "Enter a number: "
  num <- getLine
  return ((read num :: Double) < 0) -- here we need the return keyword
```

Monads

- Monads are a type class.
- A "monadic" type is an instance of a type class Monad.
  - for example `IO`

- "Type `xxx` is monad" means that xxx is an instance of type class monad and implements the following functions.
  - `>>=` passes the result on the left into the function on the right.
  - `>>` Ignores the result on the left
  - `return` wraps data in a monad.
- "action" is another name for a monadic value.
- Monads are good for things other than side effect-producing IO.
- When looking at main, Haskell looks rather imperative...
  - But Haskell sets itself apart from imperative languages.
  - It creates a separate type of programming construct for operations that, when executed, produce side effects.
  - We can always be sure which parts of the code will alter the state of the world and which parts won't.
    - Imperative languages make no guarantees whatsoever regarding function purity.
- Monads can be used for exception handling, non-determinism, etc.

Return

- `return` in Haskell is not a keyword; it is a function.
  - It doesn't break the control flow or loops.
- `return` in Haskell takes in a value and creates an action that produces that value when executed.
  - In the example `return ()`, the action produces a `()`.

`>>` vs `>>=`

- `>>=` chains actions together where the result of the left side is input to the right side.
- `>>` chains actions together and ignores the result of the left side.
- `>>` can be defined in terms of `>>=`.
  - `a >> b` is the same thing as `a >>= \_ -> b`.

`Maybe`

- Represents a computation that might not produce a result.
  - A computation that might go wrong.
    - For example, calling tail on an empty list.
- We can use maybe to create a safety wrapper for functions that might fail depending on the input.
- `Maybe` is a custom data type.
  - It's an instance of `Monad`.
- `Maybe a` can be `Nothing` or `Just a`
- We can define safe functions for `head` and `tail` using guards and maybe.

    - Instead of failing on empty lists, the function evaluates to `Nothing`.
    - If a tail or head can be found, evaluate to `Just head x` or `Just tail x`

```
module Test where

safeTail x
  | (length x > 0) = Just (tail x)
  | otherwise = Nothing

safeHead x
  | (length x > 0) = Just (head x)
  | otherwise = Nothing
```

- When we call `safeHead` on a non-empty list, we get `Just head`.
    - Maybe is a type and has custom variables.
    - `Just` is a wrapper for the head of the list, wrapped in a `Maybe` monad.
- To get a value out of `Just`, you need to unwrap it.
    - It's super similar to pulling values out of our Pt data type.
    - You can also unwrap `Nothing`.

```
> safeTail []
Nothing
> safeHead []
Nothing
> x = safeHead [1, 2, 3, 4, 5]
> x
Just 1
> y = \(Just a) -> a
> y x
1
> y = \(Nothing) -> 0
> y (safeHead [])
0
```

- If you decide on a numeric value for errors, be careful not to confuse the zero error code as the head of the list, for example.
    - This is part of why we use `Just` and `Nothing`, so we don't encounter this error.
        - `Just` can contain anything.
        - `Nothing` is useful as an error value.
- `Maybe` can make code safer by gracefully dealing with failure, but we should use it for everything.
    - Not everything has a chance to fail.
    - Wrapping everything with `Maybe` makes your code harder to read.

Example of Cascading Failure

- Let's say we have a list of tuple pairs with names and numbers.
- we want to search the table for a name
    - If we find a name, return the number.
    - If no name matches, return something indicating nothing.
        - This is where we can use `Maybe`.
- We can do this by using `lookup`.
    - `lookup` is a safe way to get a value for a given key from an associative list. If the key is in the list, lookup returns `Just value`; otherwise, it returns `Nothing`.
        - When performing multiple lookups across lists where keys from one list correspond to values in another, you might use values from the third list (in a chained lookup).
- There's a potential for failure at each step if a key doesn't exist in the subsequent list.
    - If any lookup in the chain fails (returns Nothing), the entire operation fails and returns Nothing.
        - This is known as cascading failure.
        - The >>= operator is used to pass the successful result of the previous operation as input to the next operation.
        - If the operation fails at any step, like if one of the lookups returns `Nothing`.
            - `Nothing` is passed through the rest of the code, and no further lookups are performed.
        - When the first argument to (>>=) is `Nothing`, it returns `Nothing` while ignoring the given function.

```haskell
module Test where

getPlace :: String -> Maybe String
getPlace name =
  lookup name book1 >>= (\code ->
  lookup code book2 >>= (\num ->
  lookup num book3))

fm m = case m of
  Nothing -> ""
  Just x -> x

book1 = [("Alex", 555), ("John", 444), ("Tim", 333), ("Mark", 222)]
book2 = [(555, 1), (444, 2), (333, 3), (111, 4)]
book3 = [(1, "First"), (2, "Second"), (5, "Third"), (4, "Fourth")]
```