Hello World in Haskell
Input and Output

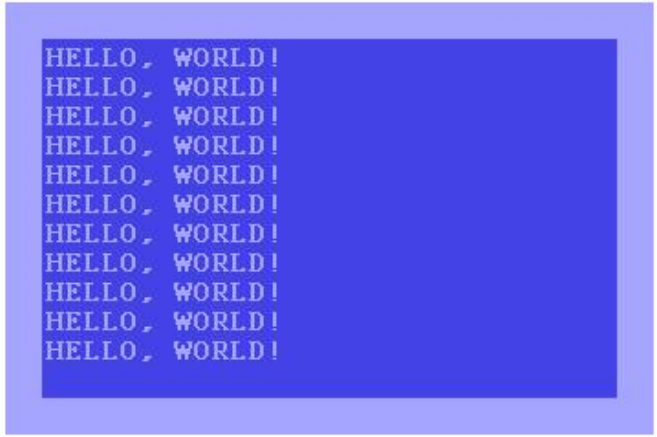
Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

Purely Functional Languages

Up until this point, our usage of Haskell has been in a world without side effects, where we need not worry about functions changing the state of the world...



Input and output is dangerous... we're changing the state of the output device (or file) we are connected to, or the file we are reading might contain state information.

Haskell has a clever system of separating parts of our program which are pure and the parts which are impure...

IO Tagged Types

Haskell only allows IO from types that are "tagged" with `IO`. We can tag a type with `IO` by writing `IO` with a space in front of it:

```haskell
-- This function is built into Haskell
getChar :: IO Char
-- definition not shown...
```

IO Tagged Types

Haskell only allows IO from types that are "tagged" with `IO`. We can tag a type with `IO` by writing `IO` with a space in front of it:

```haskell
-- This function is built into Haskell
getChar :: IO Char
-- definition not shown...
```

When we preform an IO action but return no useful data, we tag the unit type (`()`, empty parentheses) with `IO`:

```haskell
-- This function is built into Haskell
putChar :: Char -> IO ()
-- definition not shown...
```

The main Function

To run your program, Haskell expects that you write a function called `main` that preforms IO actions. Let's use the really simple function `putStrLn` which writes a line of text to the screen to see how this works:

```haskell
-- Not my first Haskell program!
main :: IO ()
main = putStrLn "Hello, World!"
```

Sequencing Actions

Using the do syntax, you can run a series of IO actions in order:

```
main = do
    putStrLn "Hello, World!"
    putStrLn "This program runs functions in order!"
```

Sequencing Actions

Using the do syntax, you can run a series of IO actions in order:

```
main = do
    putStrLn "Hello, World!"
    putStrLn "This program runs functions in order!"
```

Note

do is really just a fancy syntax for chaining together the >>= ("bind") operator, but usage of that operator is really just silly; I don't expect you to know or use it.

print

`print` is equivalent to `putStrLn . show`, in other words, it converts any object with a string representation to a string, then prints the result. For example:

```haskell
factorial' :: Int -> Int
factorial' 0 = 1
factorial' n = n * factorial' (n - 1)

main = do
    print $ factorial' 5
    print $ factorial' 10
```

The <- Operator

Similar to `let` and `where`, `<-` lets you give a variable a value, but is allowed for `IO` tagged results: the result may not always be the same value.

```haskell
main = do
    putStrLn "Please type your name!"
    name <- getLine
    putStrLn $ "Nice to meet you, " ++ name ++ "!"
```

Once the `do` block ends, the value of `name` is lost.

Making Strings Useful Things

Data which has the Read type class (for example, integers) can be created from strings using the read function.

```
main = do
    putStrLn "What number would you like factorialized?"
    num <- getLine
    print $ factorial $ read num
```

Making Strings Useful Things

Data which has the Read type class (for example, integers) can be created from strings using the read function.

```
main = do
    putStrLn "What number would you like factorialized?"
    num <- getLine
    print $ factorial $ read num
```

Why did we not have to specify the type? Haskell is clever, figured out that num must be what factorial takes, an Int. If we wanted to explicitly specify this:

```
main = do
    putStrLn "What number would you like factorialized?"
    num <- getLine
    print $ factorial $ (read num) :: Int
```

The return function

The `return` function helps us make fake IO actions, for example, if we wanted to use `<-` to give values to variables:

```
main = do
    a <- return "CSCI-400"
    b <- return "Rocks!"
    putStrLn $ a ++ " " ++ b
```

Note that you could just use `let` or `where` in this case, but you'll see good use for `return` soon...

Note that the `return` function **does not return from a function call.**

Impure map: Introducing mapM

`mapM` takes an IO action, and applies the IO action to a sequence of elements:

```
GHCi> mapM print (filter odd [1..10])
1
3
5
7
9
[(),(),(),(),()]
```

Impure map: Introducing mapM

`mapM` takes an IO action, and applies the IO action to a sequence of elements:

```
GHCi> mapM print (filter odd [1..10])
1
3
5
7
9
[(),(),(),(),()]
```

Haskell also has `mapM_`, which discards the result of the mapping if you don't care about it.

forM

`forM` could be defined as:

```haskell
forM = flip mapM
```

In other words, it's equivalent to `mapM` but takes it's arguments backwards. This leads to some cool use:

```haskell
weekdays = ["Sun","Mon","Tue","Wed","Thu","Fri"]

main = do
    shirts <- forM weekdays (\ w -> do
        putStrLn $ "What do you wear on " ++ w ++ "?"
        shirt <- getLine
        return shirt)
    mapM_ print shirts
```

Forever

`forever` takes an IO action and repeats that IO action forever.

```
main = forever $ do
    printStrLn "Are we there yet?"
```

Forever

`forever` takes an IO action and repeats that IO action forever.

```haskell
main = forever $ do
    printStrLn "Are we there yet?"
```

Food For Thought

How might we write `forever`?

Activity: Interacting with a Function

Take any pure function you've written for this course, and using `forever`, repeatedly prompt for input, and print the result of the function.

Share computers as necessary.

Conditionals in IO

Note that our usage of `if` and `else` has not changed, we just need to be careful about returning the correct type for IO, and remembering that every `if` must be paired with an `else`:

```
main = do
    putStrLn "Do you wear different shirts by weekday?"
    response <- getLine
    if response == "YES"
        then do
            shirts <- promptShirts
            mapM print shirts
        else return ()
```