

# Haskell Crash Course Part III

01-TREES

5/6 WED

## Writing Applications

Lets write the classic "Hello world!" program.

For example, in Python you may write:

*None → None*

```
def main(): () ()
    print "hello, world!"

main()
```

and then you can run it:

```
$ python hello.py
hello world!
```

*main :: () → ()*  
*main = \\_ → ()*

'build'

02-WHILE

data Dir a = File a  
| SubD a [Dir a]

In → OUTPUT

⇓  
side-effect

Haskell is a **Pure** language.

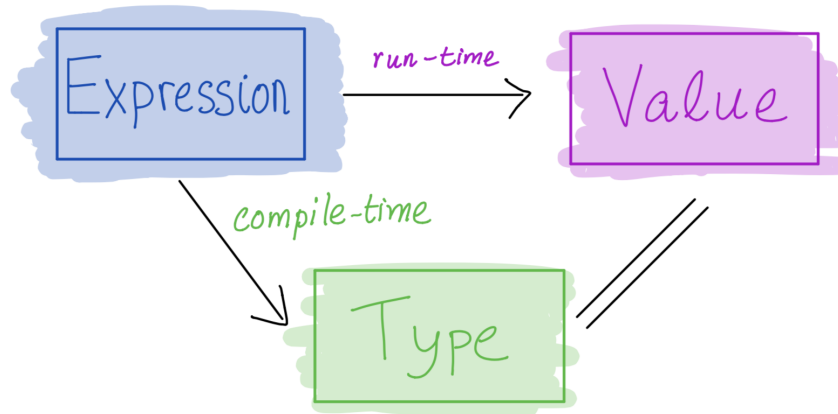
Not a *value* judgment, but a precise *technical* statement:

The “Immutability Principle”:

- A function must *always* return the same output for a given input
- A function’s behavior should *never change*

(foo 3) →

# No Side Effects



Haskell's most radical idea: `expression => value`

- When you evaluate an expression you get a value and
- **Nothing else happens**

Specifically, evaluation must not have an **side effects**

- **change** a global variable or
- **print** to screen or
- **read** a file or
- **send** an email or

In → Out

- *launch* a missile.

*But... how to write “Hello, world!”*

But, we want to ...

- print to screen
- read a file
- send an email

Thankfully, you *can* do all the above via a very clever idea: **Recipe**

# Recipes

This analogy is due to Joachim Brietner (<https://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html>)

Haskell has a special type called `IO` – which you can think of as `Recipe`

```
type Recipe a = IO a
```

A value of type `Recipe a`

- is a **description** of a *computation* that can have *side-effects*
- which **when executed** performs some effectful I/O operations
- to **produce** a value of type `a`.


*Recipe a*

*= "Description of a comp. with effects that produces a value 'a' "*

# Recipes have No Side Effects

A value of type `Recipe` is

- A **description** of a computation that can have side-effects



VS.

**INGREDIENTS**

FOR THE CAKE:

- 2 ½ cups/310 grams self-rising flour, sifted (see note)
- ½ cup/68 grams cocoa powder, sifted
- 1 ½ cups/295 grams sugar
- 4 large eggs, lightly beaten
- 1 ½ cups/360 milliliters whole milk
- 1 cup plus 2 tablespoons/255 grams unsalted butter, melted and slightly cooled
- 7 ounces/200 grams dark chocolate, melted and slightly cooled
- 2 teaspoons vanilla extract
- 1 teaspoon flaky sea salt, white or black.

FOR THE GANACHE:

- 1 cup/240 milliliters sour cream
- 14 ounces/400 grams milk

**PREPARATION**

**Step 1**

Heat oven to 350 degrees. Line 2 8-inch round cake tins with parchment paper. Place the flour, cocoa, sugar, eggs, milk, butter, dark chocolate and vanilla in a large bowl and whisk until smooth. (You may need to use a spatula to start, but use a whisk once the ingredients begin to combine.) Divide the mixture evenly between the tins and bake for 35 to 40 minutes or until a wooden skewer inserted into the center comes out clean. Allow to cool in the tins for 10 minutes before turning out onto wire racks to cool completely.

**Step 2**

Make the ganache: Place the sour cream and melted chocolate in a large bowl. Whisk to combine and refrigerate for 10 to 15 minutes or until firm. Place 1 of the cakes on a cake stand or plate. Spread with half the ganache. Top with the remaining cake and ganache. Sprinkle with the salt to serve.

**Tip**

To make your own self-rising flour, combine 2 1/2 cups/330 grams all-purpose flour, 1 tablespoon plus 3/4 teaspoon baking powder, and 1/2 teaspoon plus 1/8 teaspoon fine salt. Use the entire amount in place of the self-rising flour listed in the ingredients.

saltDarkChoco :: Cake

*"real thing"*

howToSaltChoco :: Recipe Cake

*"recipe"*

Cake vs. Recipe

(L) chocolate *cake*, (R) a *sequence of instructions* on how to make a cake.

They are different (*Hint*: only one of them is delicious.)

Merely having a `Recipe` `Cake` has no effects! The recipe

- Does not make your oven *hot*
- Does not make your your floor *dirty*

## *Only One Way to Execute Recipes*

Haskell looks for a special value

```
main :: Recipe ()
```

The value associated with `main` is handed to the **runtime system and executed**

main



Baker Aker

The Haskell runtime is a *master chef* who is the only one allowed to cook!



# How to write an App in Haskell

Make a `Recipe ()` that is handed off to the master chef `main`.

- `main` can be arbitrarily complicated
- composed of **smaller** sub-recipes



## A Recipe to Print to Screen

`putStrLn :: String -> Recipe ()`

The function `putStrLn`

*(putStrLn msg)*

- takes as input a `String`
- returns as output a `Recipe ()`

`putStrLn msg` is a `Recipe ()` - when executed prints out `msg` on the screen.

```
main :: Recipe ()  
main = putStrLn "Hello, world!"
```

... and we can compile and run it

```
$ ghc --make hello.hs  
$ ./hello  
Hello, world!
```

## *QUIZ: How to Print Multiple Things?*

Suppose I want to print two things e.g.

```
$ ghc --make hello.hs  
$ ./hello2  
Hello!  
World!
```

Can we try to compile and run this:

```
main = (putStrLn "Hello!", putStrLn "World!")
```

- A. Yes!
- B. No, there is a type error!
- C. No, it compiles but produces a different result!

## *A Collection of Recipes*

Is just ... a *collection* of Recipes!

```
recPair :: (Recipe (), Recipe ())  
recPair = (putStrLn "Hello!", putStrLn "World!")
```

```
recList :: [Recipe ()]  
recList = [putStrLn "Hello!", putStrLn "World!"]
```

... we need a way to **combine** recipes!

## Combining? Just ***do*** it!

We can *combine* many recipes into a single one using a **do** block

```
foo :: Recipe a3  
foo = do r1      -- r1 :: Recipe a1  
        r2      -- r2 :: Recipe a2  
        r3      -- r3 :: Recipe a3
```

(or if you *prefer* curly braces to indentation)

```
foo = do { r1;      -- r1 :: Recipe a1
         r2;      -- r2 :: Recipe a2
         r3       -- r3 :: Recipe a3
        }
```

The **do** block combines sub-recipes  $r_1$ ,  $r_2$  and  $r_3$  into a *new* recipe that

- Will execute each sub-recipe in *sequence* and
- Return the value of type  $a_3$  produced by the last recipe  $r_3$

## Combining? Just **do** it!

So we can write

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

or if you prefer

```
main = do { putStrLn "Hello!";  
           putStrLn "World!"  
         }
```

## *EXERCISE: Combining Many Recipes*

Write a function called `sequence` that

- Takes a *list* of recipes  $[r_1, \dots, r_n]$  as input and
- Returns a *single* recipe equivalent to `do {r1; ...; rn}`

```
sequence :: [Recipe a] -> Recipe a  
sequence rs = ???
```

When you are done you should see the following behavior

```
-- Hello.hs
```

```
main = sequence [putStrLn "Hello!", putStrLn "World!"]
```

and then

```
$ ghc --make Hello.hs
```

```
$ ./hello
```

```
Hello!
```

```
World!
```

## *Using the Results of (Sub-) Recipes*

Suppose we want a function that **asks** for the user's name

```
$ ./hello
```

```
What is your name?
```

```
Ranjit           # <<<<< user enters
```

```
Hello Ranjit!
```

We can use the following sub-recipes

```
-- | read and return a line from stdin as String
```

```
getLine :: Recipe String
```

```
-- take a string s, return a recipe that prints s
```

```
putStrLn :: String -> Recipe ()
```

But how to

- *Combine* the two sub-recipes while
- *Passing* the result of the first sub-recipe to the second.



## *Naming Recipe Results via “Assignment”*

You can write

```
x <- recipe
```

to *name* the result of executing `recipe`

- `x` can be used to refer to the result in *later* code

## *Naming Recipe Results via “Assignment”*

Lets, write a function that *asks* for the user’s name

```
main = ask

ask :: Recipe ()
ask = do name <- getLine;
      putStrLn ("Hello " ++ name ++ "!")
```

Which produces the desired result

```
$ ./hello
What is your name?
Ranjit           # user enters
Hello Ranjit!
```

## *EXERCISE*

Modify the above code so that the program *repeatedly* asks for the users's name *until* they provide a *non-empty* string.

```
-- Hello.hs
```

```
main = repeatAsk
```

```
repeatAsk :: Recipe ()
```

```
repeatAsk = _fill_this_in
```

```
isEmpty :: String -> Bool
```

```
isEmpty s = length s == 0
```

When you are done you should get the following behavior

```
$ ghc --make hello.hs
```

```
$ ./hello
```

```
What is your name?
```

```
# user hits return
```

```
What is your name?
```

```
# user hits return
```

```
What is your name?
```

```
# user hits return
```

```
What is your name?
```

```
Ranjit # user enters
```

```
Hello Ranjit!
```

## EXERCISE

Modify your code to *also* print out a **count** in the prompt

```
$ ghc --make hello.hs
```

```
$ ./hello
```

```
(0) What is your name?
```

```
# user hits return
```

```
(1) What is your name?
```

```
# user hits return
```

```
(2) What is your name?
```

```
# user hits return
```

```
(3) What is your name?
```

```
Ranjit
```

```
# user enters
```

```
Hello Ranjit!
```



# *That's all about IO*

You should be able to implement `build` from `Directory.hs`

Using these library functions imported at the top of the file

```
import System.FilePath (takeDirectory, takeFileName, (</>))  
import System.Directory (doesFileExist, listDirectory)
```

The functions are

- `takeDirectory`
- `takeFileName`
- `(</>)`
- `doesFileExist`
- `listDirectory`

google the documentation to learn about how to use them.

(<https://ucsd-cse230.github.io/sp20/feed.xml>) (<https://twitter.com/ranjitjhala>)  
(<https://plus.google.com/u/0/104385825850161331469>) (<https://github.com/ranjitjhala>)

Generated by Hakyll (<http://jaspervdj.be/hakyll>), template by Armin Ronacher (<http://lucumr.pocoo.org>),  
suggest improvements here (<https://github.com/ucsd-progsys/liquidhaskell-blog/>).