~~Jamie's~~

~~Mary Berry's~~

~~Gordon Ramsay's~~

~~Nigella's~~

Sam's

FUNctional Recipes

Do you want to make sure that you get the most wholesome fun out of your **FUN**ctional programming? Well, look no further than this short guide for guaranteed success.

Ethos

Writing a Haskell program is like painting: first of all you sketch the general shape of it, then you block in colour, then finally you fill in all the details and end up with a glorious piece of art.

I would also compare this process to completing a Sudoku. When I do these puzzles, I fill in the boxes always pushing back and procrastinating anything that constitutes more than a mild *hmmm*, slowly honing in on the solution.

Recipe

Time for the meat: a simple recipe for whipping up a Haskell function that will hit the spot.

1. **Think of a name** - Naming your child is one of the most important things you will do. Its name defines its whole destiny. You don't want it to be bullied for having a name that is too long and unwieldy; nor for a name that is too short and confusing. You want your baby to have the perfect name. Like any good parent, you will strive for Goldilocks perfection ensuring that your child's name clearly and concisely communicates their purpose in life without prattling on.

Well done! I think after all that effort you deserve a cup of tea!

2. **Write the type** - Consider what wonderful pieces of fruit your caterpillar needs to eat to become a beautiful butterfly of your choosing. Weird analogies aside, ask yourself: what arguments do you have to play with? Simply listing them in this ritualistic writing of the type can help the cogs of your brain start to turn on how you will combine these to create something of the desired output type.

Nice work! Time for a chocolate!

3. **Define it as undefined** - This is the secret. Sadly, you can't leave blanks in your functions without Haskell getting cross, BUT you can leave "blanks" in the form of `undefined`. This word is like sugar, you can never sprinkle enough of it around! Plug this little guy anywhere you don't know what to write, and Haskell will happily compile the rest of your code allowing you to check it without tackling the hard bit.

```
secretSauce = undefined
```

You can now compile your code and be gratified with successful compilation.

4. **Case split** - Is it time for the hard bit of defining yet? Hell no! Let's beat around the bush a little bit more by looking at our input and considering the different forms it can take. For example, if it is a list, we can split into two cases: `[]` and `(x:xs)`. By doing this, we are breaking the problem down and giving ourselves less things to think about at a time. Now, instead of trying to write the whole definition and getting overwhelmed, we can just focus on the smaller goal of writing one line/one case

of the definition. There is also no need to worry about over case splitting, because we can always remove cases later when we have more of an idea of what is going on.

Haribo time!

5. **Set all cases equal to undefined** - Wow wow wow don't go trying to define the cases yet! Hold your horses. Calm down. We are taking things slow and steady. First up, lets just set all the cases equal to `undefined`. This way if we have made a silly mistake in the case split, we will find out before it is too late. This is the classic compile often plan. In C, it only works to check for syntax errors, but in Haskell it does so much more: it also type checks!

So again, we can compile for some sweet sweet reassurance.

6. **Consider each case in turn** - Okay nowwww you can write the definition for each case. Hopefully, having broken the problem down, and given ourselves time to think on the matter, we can work out what to write.

Before you know it you are done!

Now, if you have gotten to 6 and you are still a bit lost. Maybe because this line of the definition has a lot going on. What you can do is introduce a let clause:

```
f ... = let
          x = undefined
        in undefined
```

I do this when I know that my result will consist of a combination of a bunch of things. I use the let to write a sort of todo list consisting of things I need. Then I can focus on each in turn and continue narrowing down the problem, and adding details till my wonderful painting is complete!

```
f ... = let
          -- things i need:
          ingredient1 = undefined
          ingredient2 = undefined
          tool = undefined
          -- idk yet:
        in undefined
```

Example

Let's say that we want to write a function that ices a bunch of cakes.

1. Because we are icing cakes. I am going to call this `iceCakes`. Inspired - I know.
2. What arguments will we have? Well a list of cakes of course! And the output that we want is a list of iced cakes:

```
iceCakes :: [Cake] -> [IcedCake]
```

3. Time for that sweet sweet sauce

```
iceCakes :: [Cake] -> [IcedCake]
iceCakes = undefined -- When in doubt, just put undefined!
```

4. Our argument is a list so let's do the classic list case split:

```
iceCakes :: [Cake] -> [IcedCake]
iceCakes []                 = -- idkkk
iceCakes (cupcake:cupcakes) = -- idkkk
```

5. Sprinkle a little more `undefined` joy...

```
iceCakes :: [Cake] -> [IcedCake]
iceCakes []                 = undefined -- The classic idkkk placeholder
iceCakes (cupcake:cupcakes) = undefined -- The classic idkkk placeholder
```

6. Almost there!

- First of all let's focus on the easy case. The `[]` case. If there are no cupcakes (possibly because I ate them all before they got to the icing stage), there will be no iced cupcakes:

```
iceCakes :: [Cake] -> [IcedCake]
iceCakes []                 = [] -- BOOM! Half way there!
iceCakes (cupcake:cupcakes) = undefined
```

- Now, what about the second case where I have a cupcake that needs icing. Well, I'm not sure what to do, but I know that I'll need some icing, a way of icing the cakes... hmmm...

```
iceCakes :: [Cake] -> [IcedCake]
iceCakes []                 = []
iceCakes (cupcake:cupcakes)
  = let
      -- I need:
      icing = undefined
      ice :: Cake -> IcedCake
      ice = undefined
    in undefined
```

Mary says that the cakes should be iced with butter icing so:

```
iceCakes :: [Cake] -> [IcedCake]
iceCakes []                 = []
iceCakes (cupcake:cupcakes)
  = let
      icing = ButterIcing -- The tasy kind.
      ice :: Cake -> IcedCake
      ice = undefined
    in undefined
```

OH! And luckily I've got a piping function flying around. We can use that!

```
iceCakes :: [Cake] -> [IcedCake]
iceCakes []                 = []
iceCakes (cupcake:cupcakes)
  = let
      icing = ButterIcing
      ice :: Cake -> IcedCake
      ice cake = pipe icing cake
    in undefined -- Ingredients gathered - time for the showstopper!
```

So close! Just gotta return something, and for that, I can just use my `ice` function on the current `cupcake`, and then keep icing the rest recursively:

```
iceCakes :: [Cake] -> [IcedCake]
iceCakes []                 = []
iceCakes (cupcake:cupcakes)
  = let
      icing = ButterIcing
      ice :: Cake -> IcedCake
      ice cake = pipe icing cake
    in ice cupcake : iceCakes cupcakes
```

Tada!

And that my friends, is how you ice cakes -uh I mean write a Haskell function

Top Tip: GHCid

Are you tired of reloading GHCi every time you change something? Well, no fear! GHCid is here!

GHCid is a daemon that runs in the background and compiles you code every time you click save on your editor, feeding any type errors you might have back to you.

Also if you are feeling super keen you can look into GHCide that is well an IDE for Haskell, coming with all sorts of cool functionality like displaying types of things upon mouse hover.