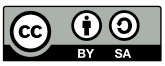
Notes from 9 Sep



2019-10-21 09:53
e8fd88f

REPL tip

The repl.it environment requires there to be a `main` function, even if you only want to experiment with other functions. This isn't too much of a problem, you can just define

```
main = putStrLn "Okay"
```

but the *minimal* main function is actually

```
main = return ()
```

so use that as a placeholder if you don't need it to do anything at all.

Operator sections

This refers to specifying an infix operator with only one argument, which then becomes a *function* of the other argument. For example:

- `(/3)` is a section of the division operator, where the numerator is the next argument.
- `(3/)` is also a section of the division operator, but now the denominator is the next argument.

Usage:

```
λ> boo = (/3)
λ> moo = (3/)
λ> boo 10
3.3333333333333335
λ> moo 10
0.3
λ> boo 12
4.0
λ> moo 12
0.25
```

We can define functions this way, without ever referring to the arguments:

```
half = (/2)
square = (**2)
twice = (*2)
```

Function composition

In algebra, we sometimes use a small circle as an operator that can *compose* two functions. The rule is:

$$(f \circ g)(x) = f(g(x))$$

So it's like a pipeline of the two functions – first apply g to the argument x, and then apply f to that result. So if:

$$f(x) = 2x + 1$$

and

$$g(x) = \sqrt{\frac{x}{2}}$$

Then:

$$(f \circ g)(x) = 2\sqrt{\frac{x}{2}} + 1$$

and:

$$(g \circ f)(x) = \sqrt{\frac{2x+1}{2}}$$

Note that the function to the *right* of the operator is applied *first*!

In Haskell, the function composition operator is a single dot (period) character. We usually put spaces around the dot, although that isn't strictly necessary. Here are some examples:

```
λ> (half . square) 10
50.0
λ> (square . half) 10
25.0
λ> (square . half . square) 10
2500.0
λ> (half.square.half) 10
12.5
```

In some situations, it may be more intuitive for a sequence of functions to be applied left-to-right instead of right-to-left. We can define a different composition operator for that purpose, simply like this:

```
(|>) = flip (.)
```

The pre-defined `flip` function takes any function with two parameters and switches the order of the parameters:

```
λ> (-) 9 2
7
λ> flip (-) 9 2
-7
```

Once we flip the composition operator, the function on the left will be applied first:

```
λ> (half |> square) 10
25.0
λ> (square |> half) 10
50.0
λ> (square |> half |> square) 10
2500.0
λ> (half |> square |> half) 10
12.5
```

For symmetry, we can provide an alias for composition in the standard right-to-left order:

```
(<|) = (.)
```

Functions with multiple clauses

Functions can be defined with multiple clauses – different expressions that apply to different cases. One way to do this is with pattern guards, which are introduced with a pipe/bar character (`|`) before the equals sign:

```
collatz n
  | even n = n `div` 2
  | otherwise = 3*n + 1
```

Following each bar is a Boolean expression, called a **guard.** The first guard that results in `True` indicates the expression that is will be executed. The keyword `otherwise` will match if no previous guard matched.

Here are some derivations using the sample function above:

```
collatz 18 ↬
  -- plug in 18 for n in the guards
  -- evaluate: even 18 ↬ True
18 `div` 2 ↬
9

collatz 5 ↬
  -- plug in 5 for n in the guards
  -- evaluate: even 5 ↬ False
  -- move on to otherwise case
3*5 + 1 ↬
```

```
15 + 1 ↪
16
```

Let's define a few more numeric functions this way:

```
fact n
  | n < 0 = error "Negative argument"
  | n == 0 = 1
  | otherwise = n * fact (n-1)

pow x n
  | n < 0 = error "Negative argument"
  | n == 0 = 1
  | otherwise = x * pow x (n-1)
```

The `error` function signals a run-time error. It will look something like this:

```
*** Exception: Negative argument
CallStack (from HasCallStack):
  error, called at <interactive>:60:1 in interactive:Ghci44
```

In Haskell we usually try pretty hard to avoid generating run-time errors. Later we can explore some techniques for ensuring they don't happen.

Let's attempt to derive the values of expressions using these guarded recursive functions.

```
fact 5 □
  -- substitute 5 for n in guards
  -- evaluate: 5 < 0 □ False
  -- evaluate: 5 == 0 □ False
  -- move on to otherwise case
5 * fact (5-1) □    -- just arithmetic
5 * fact 4 □
  -- substitute 4 for n in guards
  -- evaluate: 4 < 0 □ False
  -- evaluate: 4 == 0 □ False
  -- move on to otherwise case
5 * 4 * fact (4-1) □          -- arithmetic
5 * 4 * fact 3 □              -- otherwise case
5 * 4 * 3 * fact 2 □          -- arithmetic, otherwise
5 * 4 * 3 * 2 * fact 1 □      -- arithmetic, otherwise
5 * 4 * 3 * 2 * 1 * fact 0 □  -- arithmetic
  -- evaluate: 0 < 0 □ False
  -- evaluate: 0 == 0 □ True, use that case
5 * 4 * 3 * 2 * 1 * 1 □
```

```
20 * 3 * 2 * 1 * 1 □
60 * 2 * 1 * 1 □
120 * 1 * 1 □
120 * 1 □
120
```

```
pow 2 6 □
2 * pow 2 5 □
2 * 2 * pow 2 4 □
2 * 2 * 2 * pow 2 3 □
2 * 2 * 2 * 2 * pow 2 2 □
2 * 2 * 2 * 2 * 2 * pow 2 1 □
2 * 2 * 2 * 2 * 2 * 2 * pow 2 0 □
2 * 2 * 2 * 2 * 2 * 2 * 1 □
4 * 2 * 2 * 2 * 2 * 1 □
8 * 2 * 2 * 2 * 1 □
16 * 2 * 2 * 1 □
32 * 2 * 1 □
64 * 1 □
64
```

Exercise: why is it helpful to have a case for `n < 0` in these functions? What happens if we omit that case and we try to evaluate `fact (-1)`?

Other tips for Assignment 1

For the `palindrome` function, you basically need `reverse` and the equality operator `(==)` (double equals, like in many programming languages). The `reverse` function just reverses the order of a list (or string):

```
□> reverse [1..4]
[4,3,2,1]
□> reverse "liart"
"trail"
```

Here are some of the operators that generate Booleans:

- && is the logical AND (same as in C++)
- `||` is logical OR (same as C++)
- `not` is logical negation (`!` in C++)
- == is equality of two values (same as C++)
- `/=` is not-equals (`!=` in C++)
- <=, <, >, >= are inequality operators, as in C++
- `even` and `odd` take integers and return `True` if they are even/odd respectively.

To manipulate characters for the `rot13` problem, we need to add or subtract 13 from a character. You can't directly add a character and an integer (like in C), but you could apply `succ` or `pred` 13 times:

```
λ> ( succ . succ . succ . succ . succ . succ . succ . succ
   . succ . succ . succ . succ . succ) 'F'
'S'
```

Of course, that's pretty ugly. A more elegant way is to use `ord` and `chr` from `Data.Char` module, to convert to integer and back:

Then you can add in between:

```
λ> ord 'F'
70
λ> chr 70
'F'
λ> chr (ord 'F' + 13)
'S'
```

The other thing we'll need for `rot13` is to determine if the letter is in the front half or back half of the alphabet. The `elem` function is a good tool for that, it simply determines whether an element is contained in a list:

```
λ> elem 5 [1..10]
True
λ> elem 10 [1..5]
False
λ> elem 'k' "apple"
False
λ> elem 'p' "apple"
True
λ> elem 'S' ['N'..'Z']
True
```