Topic Pot Pourri: Scope, Lists, and Recursion

Prof. Susan Older

11 September 2019

Another Look at Function Definitions

Consider the following definition:

```
simple :: Integer -> Integer -> Integer
simple a b = a + 3*b
```

- The formal parameters a and b are names used as placeholders for input values that may be passed into the function.
- The single equation above represents a (very large!) collection of mathematical relationships:

```
simple 10 7   =  10 + 3*7
simple 5 200  =  5 + 3*200
simple (-2) 4 =  -2 + 3*4
simple 60 500 =  60 + 3*500
        ⋮
```


Another Look at Conditional Equations

Consider the following definition:

```
tame :: Integer -> Integer -> Integer
tame a b
  | b < 10    = simple a (b+2)
  | otherwise = b + 18
```

- The conditional equation above represents a (very large!) collection of mathematical relationships:

```
tame 13 7   =  simple 13 (7+2)
tame 6 200  =  200 + 18
tame (-2) 4 =  simple (-2) (4 + 2)
tame 300 5  =  simple 300 (5+2)
        ⋮
```


Scope: Which Name is Which?

Consider the following code:

```
x, y :: Integer
x = 10
y = 12

thrice :: Integer -> Integer
thrice x = 3*x

gentle :: Integer -> Integer
gentle x = x + y

extra :: Integer -> Bool
extra y = x > y
```

There are eight definitions of names:

- Five top-level definitions: x, y, thrice, gentle, extra
- Two definitions of name x as a formal parameter: see thrice and gentle
- One definition of name y as a formal parameter: see extra

What is a definition's scope?

The portion of the code where references to that name refer to that specific definition.

Scope in Haskell: How Does it Work?

Consider the following code:

```
x, y :: Integer
x = 10
y = 12

thrice :: Integer -> Integer
thrice x = 3*x

gentle :: Integer -> Integer
gentle x = x + y

extra :: Integer -> Bool
extra y = x > y
```

The formal parameter `x` in `thrice`:

- Has as scope the body of `thrice`
- "Cuts a hole in the scope" of the top-level definition of `x`

The formal parameter `y` in `extra`:

- Has as scope the body of `extra`
- "Cuts a hole in the scope" of the top-level definition of `y`

Top-level definitions have entire program *(minus any holes cut by inner definitions)* as their scope

Local Variables in Haskell

Local variables are visible only inside a definition:

```
maxSq :: Float -> Float
               -> Float
maxSq x y
   | sq x > sq y = sq x
   | otherwise   = sq y
  where
    sq :: Float -> Float
    sq w = w*w
```

```
maxSq' :: Float -> Float
                -> Float
maxSq' x y
   | sqx > sqy = sqx
   | otherwise = sqy
  where
    sqx = x*x
    sqy = y*y
```

The ramifications:

- `sq` is visible/usable only within definition of `maxSq`
- `sqx` and `sqy` are visible/usable only within definition of `maxSq'`

Local Variables: Reasons to Use Them

- Enhance the legibility of your code, especially when the same calculation is performed multiple times
- Control access to a helper function
- Clean up your name space

An example of their use:

```
-- convert a measurement in inches to feet-and-inches

convert :: Float -> (Integer, Float)
convert meas
    | meas < 0   = (0,0)        -- avoid negative measures
    | otherwise  = (feet, inches)
    where
      feet   = floor (meas / 12)
      inches = meas - 12*(fromInteger feet)
```


Introducing: Lists

List Types

Suppose `t` is a type. Then `[t]` is a type.

`[Bool]` `[Integer]` `[Float]` `[Char]` `[[Bool]]` `[[Integer]]` `[[Float]]` `[[Char]]` (and so on)

List Values

The values of type `[t]` are lists whose elements all have type `t`.

`[True,False,False,True,False]`	::	`[Bool]`
`[5,10,15,24]`	::	`[Integer]`
`[6.318, -2.5, 100.079]`	::	`[Float]`
`[[1,2,3],[10],[76,9],[3]]`	::	`[[Integer]]`
`['q','w','e','r','t','y']`	::	`[Char]` (= `String`)
`"qwerty"`	::	`[Char]` (= `String`)

The Simplest List is the Empty List: [ ]

The empty list [ ] contains no elements.

What is the type of [ ]?

[ ] is polymorphic (Greek for "many shapes"):

```
[ ] :: [a]
```

In this usage, a is a type variable. Any valid type can be plugged in for a:

```
[ ] :: [Bool]
[ ] :: [Integer]
[ ] :: [Float]
[ ] :: [[Bool]]
 ⋮
```


Building up Lists: The List Constructor

: is pronounced "cons" and builds up new lists:

```
(:) :: a -> [a] -> [a]
```

```
(:) :: Bool -> [Bool] -> [Bool]
(:) :: Integer -> [Integer] -> [Integer]
(:) :: Float -> [Float] -> [Float]
 ⋮
```

- 30:[] ⇝ [30]
- 5:[10,20,30] ⇝ [5, 10, 20, 30]
- True:[True, False] ⇝ [True, True, False]
- 'C':['u','s','e'] ⇝ ['C', 'u','s','e'] (= "Cuse")
- 1:2:3:[] ⇝ [1, 2, 3] (cons is right-associative)
- 17:[True, False] Type error!

Coming Full Circle: What Does This Function Do?

Consider the following code:

```
series :: Integer -> a -> [a]
series n item
  | n <= 0    = []
  | otherwise = item : series (n-1) item
```

This code represents a collection of mathematical relationships:

```
series (-2) False = []
series (-1) False = []
   series 0 False = []
   series 1 False = False : series 0 False
   series 2 False = False : series 1 False
   series 3 False = False : series 2 False
                  ⋮
```


Recursion: Simple Idea, Powerful Technique

Definition

Recursion is the process of defining a mathematical object (such as a set or a function) in terms of itself.

How do you generate a list containing n copies of item?

- If n is zero (or negative), return the empty list.
- If n is positive, then:
  - Generate a list with n-1 copies of item
  - Place an extra copy of item on the front of the list.

```
series :: Integer -> a -> [a]
series n item
  | n <= 0    = []
  | otherwise = item : series (n-1) item
```