λ-expressions, list patterns, and comprehensions

1 Implementing factorial

Recursive method:

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

Using built in product function

```
factorial :: Int -> Int
factorial n = product [1..n]
```

Using conditional statements

```
factorial :: Int->Int
factorial n = if n==0 then 1
               else n *factorial n-1
```

```
factorial :: Int -> Int
factorial n | n == 0 =1
            |otherwise =n* factorial n-1
```

Function application binds more tightly than binary operators. This is implemented:

```
n * (factorial n) -1
```

$\Rightarrow$ never terminates

2 Lambda expressions

2.1 Nameless function

- As well as giving functions name, we can also construct them without names using lambda expressions

  ```
  -- the nameless function that takes a number x and returns x+x
  \x -> x+x
  ```

- Use of a λ for nameless functions comes from lambda calculus, which is a theory of functions
- There is a whole formal system on reasoning about computation using λ calculus
- It is also a way of formalising the idea of lazy evaluation

2.2 Use cases for unnamed functions

- Formalises idea of functions defined using currying

  ```
  add x y = x + y
  -- Equivelently
  add = \x -> (\y -> x+y)
  ```

- The latter form emphasises the idea that add is a function of one variable that returns a function
- Also useful when returning a function as a result

  ```
  const :: a -> b -> a
  const x _ =x
  -- Or, perhaps more naturally
  const x = \_ -> x
  ```

  In this function const eats an a and returns a function which eats a b and always returns the same a

- What good is a function which always returns the same value?
- Often when using higher-order functions, we need a base case that always returns the same value

  ```
  length' :: [a] -> Int
  length' xs = sum(map(const 1) xs)
  ```

  The length of a list can be obtained by summing the result of calling const 1 on every item in the list
- We will see more of this when we look at higher order functions
- Also useful where the function is only used once

  ```
  -- Generate the first n positive odd numbers
  odds : [Int] -> [Int]
  odds n = map f[0..n-1]
          where
                  f x = x*2 + 1
  ```
- Can be simplified (removing the where clause)

  ```
  odds : [Int] -> [Int]
  odds n = map(\x -> x*2 +1) [0..n-1]
  ```

2.3 Translating between the two forms

- It is always possible to translate between named functions and arguments, and the approach using λ expressions of one argument
- Just move the arguments to the right hand side and put it inside a λ, repeat with the remained until you're done

  ```
  f a b c = ...
  -- Move formal aguments to right hand side with a lambda
  f \a b c ->
  -- Move remaining arguments into new lambdas
  f = \a -> (\b -> (\c -> ...))
  ```
- Which option fits more naturally is often a style choice
- Pattern matching is supported in the argument list in exactly the same way as normal functions

  ```
  head = \(x:_) -> x
  ```
- I sometimes find it easier to think about composing functions or currying by explicitly writing λ expressions

3 Lists

3.1 Pattern matching

3.1.1 Representations of lists

- Every non-empty list is created by repeated use of the (:) operator "construct" that adds an element to the start of a list

  ```
  [1,2,3,4] = 1 : (2: (3: (4: [])))
  ```
- This is a representation of a linked list
- Operations on lists such as indexing, or computing the length must therefore traverse the list
- Operations such as reverse, length (!!) are linear in the length of the list
- Getting the head and tail is constant time, as is (:) itself

3.1.2 Pattern matching on lists

- Lists can be used for pattern matching in function definitions

```
startsWithA :: [Char] -> Bool
startsWithA ['a',_,_] = True
startsWithA _ =False
```

- Matches 3-element lists and checks if the first entry is the character 'a'

Important: Careful

Use patterns in the equations defining a function. Not in the type of a function
Pattern matches in the equations don't change the type of the function. They just say how it should act on particular expressions

- How match 'a' and not care how long the list is?
- Can't use literal list syntax. Instead, use list constructor syntax for matching

```
startsWithA :: [Char] -> Bool
startsWithA ('a':_) = True
startsWithA _ = False
```

- `('a':_)` matches any list of length at least 1 whose first entry is 'a'
- The wildcard match _ matches anything else
- This works with multiple entries too:

```
startsWithAB :: [Char] -> Bool
startsWithAB ('a':'b':_) = True
startsWithAB _ = False
```

3.1.3 Binding variables in pattern matching

- As well as matching literal values, we can also match a (list) pattern, and bind the values

```
sumTwo :: Num a => [a] -> a
sumTwo (x:y:_) = x + y
```

- Match lists of length at least two and sum their first two entries

```
sumTwo [1,2,3,4]
-- introduces the bindings
x = 1
y = 2
_ = [3,4]
```

- Reminder: can't repeat variable names in bindings (exception _)

```
-- Not allowed
sumThree (a:a:b:_) = a + a + b
-- Allowed
second (_:a:_) = a
```

3.1.4 What types of pattern can I match on?

- Patterns are constructed in the same way that we would construct the arguments to the function

```
(&&) :: Bool -> Bool -> Bool
True && True = True
False && _ = False
-- Used as
a && b
head :: [a] -> a
head (x:_) = x
-- Used as:
head [1,2,3] == head(1:[2,3])
```

- This is a general rule in constructing pattern matches ”If I were to call the function, what structure do I want to match?”
- Caveat: can only match ”data constructors”

```
-- Not allowed
last :: [a] -> a
last(xs ++ [x]) = x
```

3.2 Comprehensions

3.2.1 Syntax

- In maths, we often use comprehensions to construct new sets from old ones

$$\{2,4\} = \{x|x \in \{1.5\} \wedge (x \bmod 2 = 0)\}$$

”The set if all integers x between 1 and 5 such that x is even

- Haskell supports similar notation for constructing lists

```
[x| x <- [1..5], x `mod` 2 ==0]
```

”The list of all integers x where x is drawn from [1..5] and x is even”

- `x <- [1..5]` is called a generator
- Compare python comprehensions

```
[x for x in range(1,6) if (x%2)==0]
```

3.2.2 Generators

- Comprehensions can contain multiple generators, separated by commas

```
[(x,y) | x <- [1,2,3], y <- [4,5]]
```

- Variables in the later generator can change faster, analogous to nested loops

```
l = []
for x in [1,2,3]
        for y in [4,5]
                l.append((x,y))
## or
[(x,y) for x in [1,2,3] for y in [4,5]]
```

- Later generators can reference variables from earlier generators

```
[(x,y) | x <- [1..3], y <- [x..3]]
```

”All pairs (x,y) such that $x, y \in \{1,2,3\}$ and $y \geqslant x$

3.2.3 Guards

- As well as binding variables to guards with generators, we can restrict the values using guards
- A guard can be any function that returns a Bool
- Cards and generators can be freely interspersed, but guards can only refer to variables to their left

```
[(x,y) | x<- [1..3], even x,y <- [x..3]]
-- [](2,3), 2,3]
[(x,y) | x <- [1..3], y <- [x..3], even x, even y]
-- [(2,2)]
```

3.2.4 Pattern matching in generators

- The left hand side of a generator expression need not be a single variable, but allows pattern matching
- We'll illustrate this with the use of the library function zip

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _ = []
zip _ [] = []
zip  (x:xs) (y:ys) = (x,y) : zip xs ys
```