Haskell

Features

- Named after logician Haskell B. Curry (after whom the currying transformation is named).
- Pure functional language (imperative features like I/O integrated into a functional framework).
- No destructive assignment.
- Implicit polymorphic static typing.
- Lazy evaluation (normal-order reduction).
- First-class functions.
- All functions take only a single argument.
- Infix syntax.
- Equational definitions.
- Pattern-matching.

Factorial

```
--optional type declaration
factorial :: (Integral a) => a -> a

factorial 0 = 1
factorial n = n * factorial(n - 1)
```

- In type declaration, **a** is a type variable.
- Type declarations are optional.
- Declaration states that if **a** is constrained to be **Integral**, then **factorial** has type function from **a** to **a**.
- Equations are tried in order, using pattern matching.

List Length

cons is spelt **:** in Haskell, where **:** is an infix operator.

```
myLength [] = 0
myLength (x:xs) = 1 + myLength xs
```

```
$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :l "my-length.hs"
[1 of 1] Compiling Main ...
Ok, one module loaded.
*Main> myLength([])
0
*Main> myLength([1, 2, 3])
3
*Main> :t myLength     --type is automatically inferred
myLength :: Num p => [a] -> p
*Main>
```

Functional Programming

In FP, we have a universe of values:

- The set of values include functions, i.e. functions are first class and at the same level as integers and strings.
- Functions take values as parameters and return values. Since values include functions, parameters can include functions and return values can be functions.
- With traditional mathematical notation, applying function `f` to value `x` is denoted as `f(x)`.
- **Function composition** is a way of combining functions; i.e. `f(g(x))` denotes the result of applying function `f` to the result of applying `g` to `x`.
- First class functions allows defining **function transformers**: i.e. functions which takes functions as parameters and return functions. For example, function composition `.` is a function transformer where `(f . g)(x) = f(g(x))`.
- There are no OO classes.

Comparison with Other Languages

From Haskell vs. Ada vs. C++ vs Awk ... An Experiment in Software Prototyping Productivity.

Language	Lines of code	Lines of documentation	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

Figure 3: Summary of Prototype Software Development Metrics

Primitive Values

```
-- numbers and arithmetic
Prelude> 2 + 3
5
Prelude> 2 + 3 * 2
8
Prelude> 2 + 3 `rem` 2  -- note ` around rem
3

-- booleans
Prelude> False && True
False
Prelude> False || True
True
Prelude> not False
True

-- characters and strings
Prelude> 'a'
'a'
Prelude> "hello" -- actually, strings are lists of Char
"hello"
Prelude> "hello" ++ " world" -- ++ is concatenate
"hello world"
```

Non-Primitive Values: Lists

```
-- lists
Prelude> [ 1, 2, 3 ]
[1,2,3]
Prelude> 1 : [ 2, 3 ]  -- : is cons
[1,2,3]
Prelude> head ['a', 'b', 'c']
'a'
Prelude> tail ['a', 'b', 'c']
"bc" -- same as [ 'b', 'c' ] since strings are lists of char
Prelude> head "hello"
'h'
Prelude> tail "hello"
"ello"
Prelude> [1, 2, 3, 4] !! 2 -- zero-based indexing of list
3
Prelude> 1 : [] ++ [2] -- cons : and concatenate ++
[1,2]
Prelude> [ False, 'a' ] -- lists must be homogeneous
<interactive>:...: error:
...
```

Non-Primitive Values: Tuples

```
Prelude> ( False, 'a' ) -- tuples need not be homogeneous
(False,'a')
Prelude> fst ("hello", False)
"hello"
Prelude> snd ("hello", False)
False
-- fst, snd work only on pairs
Prelude> fst ("hello", False, 22)
<interactive>:...: error:
...
-- no function to get nth element of tuple;
-- use pattern matching or records (tuples with
-- named fields) instead.
```

Types

In GHCi, typing **`:t`** followed by an expression returns the type Haskell has inferred for that expression.

```
Prelude> :t False          -- what is the type of False?
False :: Bool
Prelude> :t 22             -- what is the type of 22
22 :: Num p => p           -- It has type p, where p is a Num
Prelude> :t (22 :: Int)    -- set type of 22 to machine int
(22 :: Int) :: Int
Prelude> :t 'x'            -- what is the type of 'x'
'x' :: Char
Prelude> :t 3.2            -- what is the type of 3.2?
3.2 :: Fractional p => p   -- a Fractional
Prelude> :t [False, True, True]
[False, True, True] :: [Bool] -- a list of Bool
Prelude> :t ['a', 'b', 'c']
['a', 'b', 'c'] :: [Char]     -- a list of Char
Prelude> :t "hello"
"hello" :: [Char]             -- strings are list of Char
Prelude> :t ["hello", "world"]
["hello", "world"] :: [[Char]]-- list of list of Char
Prelude> :t ("hello", False, 'a')
("hello", False, 'a') :: ([Char], Bool, Char)
```

Generic Types

A type can contain type variables; typically **a**, **b**, **c**, etc. Those types stand for any types (which could be constrained).

```
Prelude> :t []
[] :: [a]
Prelude> :t id
id :: a -> a
Prelude> :t fst
fst :: (a, b) -> a
Prelude> :t take
take :: Int -> [a] -> [a]
Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]
Prelude> :t (+)
(+) :: Num a => a -> a -> a  -- a constrained type
Prelude>
```

Curried Functions

Note type **`((+) :: Num a => a -> a -> a`**; the function returning operator **`->`** associates to the right, so the type is equivalent to:

```
((+) :: Num a => a -> (a -> a)
```

which is read as if **`a`** is constrained to be a **`Num`**, then **`(+)`** is a function which takes an **`a`** and returns another function. The returned function takes an **`a`** and returns an **`a`**.

Currying Examples

```
Prelude> :t take
take :: Int -> [a] -> [a]
Prelude> take 2 [1, 2, 3, 4, 5]
[1,2]
Prelude> let take3 = take 3   -- applied to only one argument
Prelude> :t take3
take3 :: [a] -> [a]
Prelude> take3 [1, 2, 3, 4, 5, 6]
[1,2,3]
Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]
Prelude> zip [1, 2] [3, 4]
[(1,3),(2,4)]
Prelude> let zip' = zip [1, 2]
Prelude> :t zip'
zip' :: Num a => [b] -> [(a, b)]
Prelude> zip' [ 'a', 'b' ]
[(1,'a'),(2,'b')]
Prelude> let add1 = (1 +)  -- partial + (actually a section)
Prelude> add1 4
5
Prelude> let constListIndex = ([1, 2, 3, 4, 5] !!)
Prelude> constListIndex 3
4
```

List Append

```
-- equivalent to haskell ++ operator
append [] ys = ys
append (x:xs) ys = x : (append xs ys)
```

- **Log**

```
*Main> append [1] [2, 3, 4]
[1,2,3,4]
*Main> append ['h'] ['e', 'l', 'l', 'o']
"hello"
*Main> :t append
append :: [a] -> [a] -> [a]
*Main>
```

- If **ys** is appended to an empty list, then the result is simply **ys**.
- If **ys** is appended to a non-empty list of the form **(x:xs)**, then the result is the result of **cons**'ing (**:**) **x** in front of the result of appending **ys** to **xs**.
- Since **append** recurses on its first argument, its performance is $O(n)$ where n is the length of the first list.

Structural Recursion

- **Structural recursion** is related to the proof method of structural induction. When a data-structure is defined recursively, related functions can be written using cases corresponding to the cases in the recursive definition.

- A list is either *empty* or is a *pair* consisting of some *head* and some *tail* which is a list. Hence define function with two cases for *empty* and *pair*. In the former case, return function value for *empty* list; in the latter case, make recursive call for *tail* and combine returned result with *head* as return value of function.

- Recall that a non-negative integer is either 0 or the successor of a non-negative integer.

Structural Recursion Continued

- A non-empty array is either a 1-element array or it is a 1-element array followed by a non-empty array; basing a search function on this recursive definition leads to linear search.
- A non-empty array is either a 1-element array or it is a sequence of two arrays of length that differ at most by 1; basing a search function on this recursive definition for a sorted array leads to binary search.

List Reverse

```
-- ++ is append/concatenation
rev1 [] = []
rev1 (x:xs) = rev1 xs ++ [x]
```

- **Log**

```
*Main> rev1 [1, 2, 3]
[3,2,1]
*Main> rev1 []
[]
*Main> :t rev1
rev1 :: [a] -> [a]
*Main>
```

- The reversal of `[]` is `[]`.
- The reversal of a non-empty list is the result of appending the singleton list containing its head to the reverse of its tail.
- **`rev1`** will recurse n times when given a list of length n. However, each recursion uses **`++`** (**`append`**) which itself is $O(n)$. Hence the overall **`rev1`** is $O(n^2)$.
- Can we do better?

A Linear Reverse

```
rev2 xs = rev2Aux [] xs
   where
     rev2Aux acc [] = acc
     rev2Aux acc (x:xs) = rev2Aux (x:acc) xs
```

- **Log**

```
*Main> rev2 [1, 2, 3]
[3,2,1]
*Main> :t rev2
rev2 :: [a] -> [a]
```

- **rev2** of a list is the result of calling an auxiliary function **rev2Aux** on the original list with an accumulator parameter initialized to the empty list **[]**.

- If the original list is empty in a call to the auxiliary function, then the result is simply the value accumulated so far.

  Verify: works okay when the original list was empty as the accumulator would have been empty.

- If the original list is non-empty in a call to the auxiliary function, then the result of the auxiliary function is simply the result of calling it recursively on the tail of the original list and the accumulator set to the **cons** of the head of the original list and the incoming accumulator.

- **Trace**

```
rev2 [1, 2, 3] =
rev2Aux [] [1, 2, 3] =
rev2Aux [1] [2, 3] =
rev2Aux [2, 1] [3] =
rev2Aux [3, 2, 1] [] =
[3, 2, 1]
```

Accumulating Parameters

- A primary function is often implemented as a wrapper which simply calls an auxiliary function with additional accumulating parameters. For example, **`rev2`** is a wrapper around the auxiliary function **`rev2Aux`**.
- The accumulating parameter is given some initial value when the wrapper calls the auxiliary function. For example, when **`revs`** calls **`rev2Aux`**, it is called with 2 parameters: an accumulating parameter initialized to **`[]`** and the original list.
- As the auxiliary function recurses, the accumulating parameter for the recursive call is updated (non-destructively, since the parameter for the recursive call is different from the incoming parameter). For example, the recursive call to **`rev2Aux`** is made with the accumulating parameter set to the **`cons`** of the **`car`** of the incoming list being reversed and the incoming accumulating parameter.
- When the auxiliary function recurses, it's return value is simply the return value of the recursive call. For example, the recursive case for **`rev2Aux`** simply returns the return value of the recursive call.
- When the auxiliary function terminates its recursion, the value of the accumulating parameter is returned as the result. For example, the base case for **`rev2Aux`** simply returns the value of the accumulating parameter **`acc`**.

Fibonacci Function

Recursive Fibonacci:

```
fib1 0 = 0
fib1 1 = 1
fib1 n = fib1 (n - 1) + fib1 (n - 2)
```

- **Log**

```
*Main> fib1 1
1
*Main> fib1 5
5
*Main> fib1 10
55
*Main> :t fib1
fib1 :: (Eq a, Num a, Num p) => a -> p
*Main>
```

Recursive Fibonacci Function in C

Recursive C analog of Haskell code (in fib.c):

```
static int rec_fib(int n)
{
  return
    (n < 2)
    ? n
    : rec_fib(n - 1) + rec_fib(n - 2);
}
```

Iterative Fibonacci Function in C

Iterative C Fibonacci (in fib.c):

```c
static int iter_fib(int n) {
  if (n < 2) {
    return n;
  }
  else {
    int acc0 = 0;
    int acc1 = 1;
    int i;
    for (i = 2; i <= n; i++) {
      int t = acc0;
      acc0 = acc1;
      acc1 += t;
    }
    return acc1;
  }
}
```

Accumulator Fibonacci in Haskell

Exact analog, using auxiliary function **`auxFib2 acc0 acc1 i n`** to replace loop from C function. (the **`| i > n`** is a **guard** which helps choose an equation when pattern matching is not sufficient):

```
fib2 0 = 0
fib2 1 = 1
fib2 n = auxFib2 0 1 2 n
  where
    auxFib2 _ acc1 i n | i > n = acc1
    auxFib2 acc0 acc1 i n =
      auxFib2 acc1 (acc0 + acc1) (i + 1) n
```

- **Log**

```
*Main> fib2 1
1
*Main> fib2 5
5
*Main> fib2 10
55
```

Tail Recursion

- In conventional programming languages, recursion leads to heavy use of stack space.
- A function is **tail-recursive** if the **absolutely** last thing it does before returning is calling itself.
- A simple recursive factorial like `fact` is not tail-recursive, because recursive-call return value must be multiplied by `n` before return. `auxFib2` is tail-recursive.
- If a function is tail-recursive, then it is possible to optimize it into a loop so that it does not use additional stack space. This is referred to as **tail recursion optimization**.
- The Scheme programming language requires implementations to perform **tail optimization** converting **tail calls** to jumps when possible (this subsumes tail recursion).

Tail Recursion in C

A tail-recursive function

```
int f(params) {
  if (baseCase(params)) {
    return g(params); /* non-recursive */
  }
  else {
    return f(newParams);
  }
}
```

Tail Recursion in C Continued

Tail recursive function is replaced by

```
int f(params) {
loop:
  if (baseCase(params)) {
    return g(params); /* non-recursive */
  }
  else {
    params = newParams;
    goto loop;
  }
}
```

Tail Recursion in C Continued

Previous function is equivalent to:

```
int f(params) {
  while (!baseCase(params)) {
    params = newParams;
  }
  return g(params);  /* non-recursive */
}
```

Tail-Recursive Factorial in C

Consider iterative C factorial from fact.c:

```
int fact(int n) {
  int acc = 1;
  while (n > 1) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

Tail-Recursive Factorial in Haskell

```
iterFact n = iterFactAux 1 n
  where
    iterFactAux acc 0 = acc
    iterFactAux acc n =
      iterFactAux (n * acc) (n - 1)
```

- **Log**

```
*Main> iterFact 1
1
*Main> iterFact 0
1
*Main> iterFact 5
120
```

Syntactic Issues: Character Case and Indentation

- Types start with upper-case letter; non-types with lower-case letter.
- Implicit semi-colon at the end of every line, except when continuation lines are indented.
- A semi-colon is inserted at EOF or whenever the next line starts in the left-hand margin.
- Blocks are indicated by indentation of keywords like **`where`**, **`let`**, **`of`**, **`do`**.
- New margin is indentation of token after keyword. Block ends at return to old margin.
- Can also use explicit braces and semicolons.

Syntactic Issues: Operators

Haskell allows infix operators:

- Function application indicated by whitespace has highest precedence and is left-associative; i.e. `f a b` is `(f a) b` (note: functions take only a single argument since all functions are curried).
- `->` is right-associative; i.e. `a -> b -> c` is equivalent to `a -> (b -> c)`.
- An infix operator can be used by itself by surrounding it in parentheses.

Operator Examples

```
*Main> 1 + 2
3
*Main> :t (+)
(+) :: Num a => a -> a -> a
*Main> :t (+3)
(+3) :: Num a => a -> a
*Main> 1:[]
[1]
*Main> :t (:)
(:) :: a -> [a] -> [a]
*Main> :t (2:)
(2:) :: Num a => [a] -> [a]
*Main>
```

Sum Types

```
-- Colors is a data-type which is one of Red, Blue or Green.
-- Red, Blue, Green are 0-ary type constructors.
data Colors
  = Red
  | Blue
  | Green
```

The Need for Typeclasses

```
*Main> Red
<interactive>:22:1: error:
    • No instance for (Show Colors) arising from a
      use of ‘print’
```

Show is used to obtain a **String**-representation of a data-type.

```
-- Define Colors as an instance of typeclass Show
-- (think of a typeclass like an interface)
instance Show Colors where
  show Red = "Red"
  show Blue = "Blue"
  show Green = "Green"


*Main> Red
Red
```

Show is a **typeclass** (not to be confused with an OO class). It should be thought of more like an **interface** or **trait**.

Composite Data: Sum Types

```
data Shape =
  Square Int |
  Rect Int Int
  -- let Haskell automatically derive Show and Eq
  deriving (Eq, Show)

area (Square side) = side * side
area (Rect width height) = width * height

sumAreas [] = 0
sumAreas (shape:shapes) = area shape + (sumAreas shapes)
```

Log:

```
*Main> :l shapes
[1 of 1] Compiling Main             ( shapes.hs, interpreted )
Ok, one module loaded.
*Main> :t area
area :: Shape -> Int
*Main> :t sumAreas
sumAreas :: [Shape] -> Int
*Main> area (Square 4)
16
*Main> area (Rect 4 5)
20
*Main> sumAreas [ (Square 4), (Rect 4 5) ]
36
*Main>
```

Binary Trees

In trees.hs

```
data Tree a
  = Leaf a
  | InternalNode (Tree a) a (Tree a)

sumTree :: Tree Int -> Int
sumTree (Leaf value) = value
sumTree (InternalNode left v right) =
  sumTree left + v + sumTree right
```

Log:

```
*Main> sumTree (InternalNode (Leaf 1) 2 (Leaf 3))
6
*Main> sumTree
        (InternalNode
          (Leaf 1) 2
          (InternalNode (Leaf 3) 4 (Leaf 5)))
15
*Main>
```

Typedefs

Define type synonyms using **`type`** keyword.

```
type Name = String
type SSN = String
type Age = Int
type Person = (Name, SSN, Age)
donald = ("Donald Duck", "123-45-6789", 87)
```

Use of Guards

Guards can be used instead of a top-level **`if`** within an equation.

```
max :: Ord a => a -> a -> a
max x y
  | x > y = x
  | otherwise = y
```

Guards with Pattern Matching

Allows pattern matching within expressions:

```
my_filter :: (a -> Bool) -> [a] -> [a]
my_filter p [] = []
my_filter p (x:xs)
  | p x = x : my_filter p xs
  | otherwise = my_filter p xs
```

Case Expressions

```
my_length :: [a] -> Int
my_length xs =
  case xs of
    [] -> 0
    (x:xs) -> 1 + my_length xs
```

Can be combined with guards.

The Maybe Type

Standard prelude contains:

```
-- Maybe is a data-type; Nothing a 0-ary type constructor;
-- Just is a 1-ary type constructor.
data Maybe a =
  Nothing |
  Just a
```

- Can be used to convert a partial function to type **`a`** to a total function to **`Maybe a`**.
- **`Nothing`** used to indicate failure.
- **`Just a`** used to indicate success.

Safe Division

```
safeDivision :: Float -> Float -> Maybe Float
safeDivision x y
  | y == 0 = Nothing           -- guard
  | otherwise = Just (x/y)
```

List Comprehension

List comprehension returns a list of elements created by evaluation of generators. `<-` used for $\in$.
Examples from zvon.org.

```
*Main> [x + 2*x + x/2 | x <- [1, 2, 3, 4]]
[3.5,7.0,10.5,14.0]
*Main>  [ odd x | x <- [1..9]]
[True,False,True,False,True,False,True,False,True]
*Main> [ x*y | x <- [1,2,3,4], y <- [3,5,7,9]]
[3,5,7,9,6,10,14,18,9,15,21,27,12,20,28,36]
*Main> [x | x <- [1,5,12,3,23,11,7,2], x>10]
[12,23,11]
*Main> [(x,y) | x <- [1,3,5], y <- [2,4,6], x<y]
[(1,2),(1,4),(1,6),(3,4),(3,6),(5,6)]
*Main>
```

QuickSort

```
qsort []     = []
qsort (x:xs) =
  qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
     where
       elts_lt_x   = [y | y <- xs, y < x]
       elts_greq_x = [y | y <- xs, y >= x]
```

Log:

```
*Main> :l qsort
...
*Main> qsort [5, 3, 2, 6]
[2,3,5,6]
*Main> qsort ["c", "d", "abc"]
["abc","c","d"]
*Main> :type qsort
qsort :: Ord a => [a] -> [a]
*Main>
```

Permutations

```
--selections :: [a] -> [(a, [a])]
selections [] = []
selections (x:xs) =
  (x, xs) :
  [ (z, x:zs) | (z, zs) <- selections(xs) ]

permutations [] = [[]]
permutations xs =
  [ y:zs |
   (y, ys) <- selections xs,
   zs <- permutations ys ]
```

Log:

```
*Main> :l permutations
...
*Main> :t permutations
permutations :: [a] -> [[a]]
*Main> permutations []
[[]]
*Main> permutations [1, 2, 3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
*Main> permutations [1]
[[1]]
*Main>
```

Zip

`zip` is a function which turns two lists into a list of 2 tuples. **`zipWith`** maps a binary function over two lists at once.

```
*Main> zip [1, 2, 3] ["ab", "cd", "ef"]
[(1,"ab"),(2,"cd"),(3,"ef")]
*Main> zip [1, 2, 3] ["ab", "cd"]
[(1,"ab"),(2,"cd")]
*Main> zipWith (*) [1, 2, 3] [4, 5, 6]
[4,10,18]
*Main> zipWith (*) [1, 2, 3] [4, 5]
[4,10]
*Main>
```

Function Composition

- Composition of two functions f and g is denoted using $f \,.\, g$. $(f \,.\, g)x = f(gx)$.
- Since function application has higher precedence than composition operator `.`, `succ . succ 1` is not `(succ . succ) 1`, but `succ . (succ 1)`, which will usually result in a type error.
- Composition combines 2 functions, whereas application applies a function to a argument.

Lambda Functions

- In C, in a statement like `a = (x + y)*z`, we use the sub-expression `(x + y)` without giving it a name.
- In C, it is impossible to define a function without giving it a name. So functions do not have the same flexibility as arithmetic expressions; we say they are not **first-class**.
- If functions are to be first-class, then it must be possible to have an expression which is a function, without giving it a name. This leads to **lambda functions**.
- In the following we have an anonymous function expression which is then given a name.

```
lambdaLength =
  \ ls -> if null ls
     then 0
     else 1 + lambdaLength(tail ls)
```

- Very useful in situations where we need a small one-off function, as shown on the next slide.

Higher-Order Functions: map

```
*Main> :type map
map :: (a -> b) -> [a] -> [b]
*Main> map (\x -> x > 4) [1..9]
[False,False,False,False,True,True,True,True,True]
*Main> map (\x -> x + 4) [1..9]
[5,6,7,8,9,10,11,12,13]
```

Note that `map f xs = [ f x | x <- xs]` . Often it is preferable to use list comprehensions rather than explicit **`map`** and/or **`filter`**.

Infinite Data Structures

Haskell uses **lazy evaluation**: parameters are not evaluated at time-of-call; instead they are evaluated when needed within called function.

```
numsFrom :: Int -> [Int]
numsFrom n = n : numsFrom (n+1)

-- x^y returns x to the power-of y for integer y.
squares :: [Int]
squares = map (^2) (numsFrom 0)

*Main> take 5 squares
[0,1,4,9,16]
```

Zip Fibonacci

The following produces a sequence of **all** the Fibonacci numbers using three different variations.

zip-fib.hs

```
-- using map
fib1 :: [Int]
fib1 =
  1 : 1 : map (\(x,y) -> x+y) (zip fib1 (tail fib1))

-- using a list comprehension
fib2 :: [Int]
fib2 =
  1 : 1 : [x+y | (x,y) <- zip fib2 (tail fib2)]

-- using zipWith
fib3 :: [Int]
fib3 = 1 : 1 : zipWith (+) fib3 (tail fib3)
```

Zip Fibonacci Log

Since the sequence is infinite, we need to use **`take`** and **`drop`** to examine it:

```
*Main> take 10 fib1
[1,1,2,3,5,8,13,21,34,55]
*Main> take 10 (drop 5 fib1)
[8,13,21,34,55,89,144,233,377,610]
*Main> take 10 fib2
[1,1,2,3,5,8,13,21,34,55]
*Main> take 10 fib3
[1,1,2,3,5,8,13,21,34,55]
*Main>
```

Higher-Order Functions: filter

Think of **`filter`** as **`select`**, which avoids any ambiguity as to the boolean meaning of the predicate.

```
*Main> :t filter
filter :: (a -> Bool) -> [a] -> [a]
*Main> filter even [1..10]
[2,4,6,8,10]

-- use function as infix operator by putting within ` `
*Main> filter (\n -> (n `mod` 3) == 1) [1..10]
[1,4,7,10]
*Main>
```

Note that **`filter p xs = [ x | x <- xs, p x]`** . Often it is preferable to use list comprehensions rather than explicit **`map`** and/or **`filter`**.

Edit Distance

From Thompson's book in edit.hs, find set of transformations to transform a source string to a destination string. Example log:

```
*Main> transform "abc1" "ac2"
[Edit_Copy,Edit_Delete,Edit_Copy,Edit_Change '2']
*Main> transform "abcd" "1a2"
[Edit_Insert '1',Edit_Copy,Edit_Insert
 '2',Edit_Kill]
*Main>
```

Edit Distance: Data Type

```
data Edit
  = Change Char          -- change current src char to Char
  | Copy                 -- copy current src char
  | Delete               -- remove current src char
  | Insert Char          -- insert Char in dest
  | Kill                 -- kill rest of src String
  deriving (Eq, Show)
```

Edit Distance transform

```
transform :: String -> String -> [Edit]

transform [] [] = []
transform string [] = [Kill]
transform [] string = map Insert string
transform (a:x) (b:y)
  | a == b =    Copy : transform x y
  | otherwise =
     best [ Delete : transform x (b:y),
            Insert b : transform (a:x) y,
            Change b : transform x y ]
```

Edit Distance best and cost

```
best :: [[Edit]] -> [Edit]
best [a] = a
best (a:x)
  | cost a <= cost b = a
  | otherwise        = b
    where b = best x

cost :: [Edit] -> Int
cost = length . filter (/=Copy)
```

Higher-Order Functions: foldl

Reduce list to a single value from the left:

```
*Main> :type foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
-- (acc -> element -> acc) -> acc -> [element] -> acc
*Main> foldl (+) 0 [1, 2, 3, 4]
10   -- (((0 + 1) + 2) + 3) + 4
*Main> foldl (\a e -> a+e) 0 [1, 2, 3, 4]
10
*Main> foldl (-) 0 [1, 2, 3, 4]
-10  -- (((0 - 1) - 2) - 3) - 4
*Main> foldl (-) 15 [1, 2, 3, 4]
5    -- (((15 - 1) - 2) - 3) - 4
-- usually need to parenthesize unary - expr
*Main> foldl (-) (-15) [1, 2, 3, 4]
-25  -- (((-15 - 1) - 2) - 3) - 4
```

Higher-Order Functions: foldr

Reduce list to a single value from the right.

```
*Main> :type foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
-- (element -> acc -> acc) -> acc -> [element] -> acc
*Main> foldr (+) 0 [1, 2, 3, 4]
10     -- 1 + (2 + (3 + (4 + 0)))
*Main> foldr (\e a -> e+a) 0 [1, 2, 3, 4]
10
*Main> foldr (-) 0 [1, 2, 3, 4]
-2      -- 1 - (2 - (3 - (4 - 0)))
*Main> foldr (-) (-5) [1, 2, 3, 4]
-7      -- 1 - (2 - (3 - (4 - (-5))))
```

foldl versus foldr

```
*Main> foldl (\x -> \y -> x + length y) 0
        ["abc", "d", "ef"]
6
*Main> foldl (\x -> \y -> length x +  y) 0
         ["abc", "d", "ef"]
ERROR - Type error in application
*Main> foldr (\x -> \y -> length x +  y) 0
         ["abc", "d", "ef"]
6
*Main>
```

Currying

Given a function f of type $f : (X \times Y) \rightarrow Z$, then $\text{curry}(f) : X \rightarrow (Y \rightarrow Z)$. Currying results in higher-order functions. In simple.hs:

```
simple a b c = a * (b + c)
```

```
*Main> :l "code/simple.hs"
*Main> :type simple
simple :: Num a => a -> a -> a -> a
*Main> :type (simple 5)
simple 5 :: Num a => a -> a -> a
*Main> :type (simple 5 3)
simple 5 3 :: Num a => a -> a
*Main> :type (simple 5 3 2)
simple 5 3 2 :: Num a => a
*Main> (((simple 5) 3) 2)
25
-- paren unnecessary since fn application assoc to left
*Main> simple 5 3 2
25
*Main>
```

Currying Infix Operators using Sections

- `(x / y)` is equivalent to a function `f1 x y = x / y`.
- `(x /)` is equivalent to a function `f1 y = x / y`.
- `(/ y)` is equivalent to a function `f1 x = x / y`.

Currying Infix Operators using Sections Log

```
*Main> (10 / 5)
2.0
*Main> (10 /) 5
2.0
*Main> (/ 5) 10
2.0
*Main> (/) 10 5
2.0
*Main>
```

Another Sections Example

posints.hs

```
posInts1  :: [Integer] -> [Bool]
posInts1 xs = map gt0 xs
              where gt0 x = x > 0

posInts2  :: [Integer] -> [Bool]
posInts2 xs = map (> 0) xs

posInts3  :: [Integer] -> [Bool]
posInts3 = map (> 0)
```

Currying Simplification

In listsumprod.hs

```
listSum, listProd :: [Float] -> Float
listSum xs = foldl (+) 0 xs
listProd xs = foldl (*) 1 xs
```

Possible to define function without explicitly mentioning arguments This is known as **point-free style** since arguments (points) are not mentioned. In listsumprod-curry.hs

```
listSum, listProd :: [Float] -> Float
listSum = foldl (+) 0
listProd = foldl (*) 1
```

Reverse Revisited

reverse.hs

```
rev3 xs = foldl revOp [] xs
  where
    revOp acc x = x:acc

-- Using flip f x y = f y x defined in Prelude.
-- revOp acc x = flip (:) acc x
-- 2 applications of currying simpli□cation:
-- revOp = flip (:)
rev4 :: [a] -> [a]
rev4 = foldl (flip (:)) []
```

User-Defined Operators

- Operator names consists of special characters
- Precedence declarations given by *fixity declarations* **`infixr`** n (right associative), **`infixl`** n (left associative), **`infix`** n (non-associative) for $n \in 1 \ldots 9$, with 9 being the strongest (function application has precedence level 10 and is left associative).
- Operator **`$`** also does function application but has **lowest** precedence 0 and is right associative.

```
:i +
infixl 6 +
(+) :: Num a => a -> a -> a  -- typeclass member
*Main> :i *
infixl 7 *
(*) :: Num a => a -> a -> a  -- typeclass member
*Main> max 4 $ max 5 $ length "323"
5
```

Qualified Types

- We could type **+**

```
(+) :: Integer -> Integer -> Integer
```

  but that would not allow us to add floats, or complex numbers.

- We would have separate addition functions **`addInteger`**, **`addFloat`**, **`addComplex`**, but that would not be satisfactory.

Qualified Types

- Giving **(+)** the polymorphic type **a -> a -> a** would be too general, because the type-variable **a** is implicitly universally quantified.
- Solution is to use a *qualified type*:

```
(+) :: Num a => a -> a -> a
```

  which is read as for all types **a** that are members of the typeclass **Num**, **(+)** has type **a -> a -> a**.

Equality

- Equality cannot be computed for all types; for example, one cannot determine the equality of two infinite lists or two functions. Hence *computational equality* is weaker than *full equality*.
- So we can compute equality for some types but not for others. We say that a type implements equality if it is a member of type typeclass `Eq` which defines the function: `(==) :: Eq a => a -> a -> Bool`.
- `Integer` and `Char` are instances of `Eq`. `42 == 43` and `'a' == 'a'` are well-typed but `42 == 'a'` is not.

Equality Continued

- Type constraints can be propagated through polymorphic data types. Hence `[10, 12] == [10, 12]` and `['a', 'b'] == "ac"` are well-typed. On the other hand, `[10, 12] == "ac"` is not.
- Qualified types also propagate through function definitions. Consider `member` function:

```
member x [] = False
member x (y:ys) = (x == y) || member x ys
```

  has type `Eq a => a -> [a] -> Bool`.

Defining Typeclasses

- The essence of **`Eq`**.

```
class Eq a where
  (==) :: a -> a -> Bool
```

  says a type **`a`** is an instance of the typeclass **`Eq`** iff there is an operation **`(==) :: a -> a -> Bool`** defined on it."

- We can say that a particular type is an instances of **`Eq`**:

```
instance Eq Integer where
  x == y = IntegerEq x y
```

Equality over User Data Types

Given

```
data Tree a =
  Leaf a
  | Node (Tree a) (Tree a)
```

define equality as:

```
instance Eq a => Eq (Tree a) where
  Leaf a == Leaf b = a == b
  Node t1 t2 == Node s1 s2 = t1 == s1 && t2 == s2
  _ == _ = False
```

Full Definition of Equality

Definition of **`Eq`** in Haskell's *Standard Prelude*:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
  x == y = not (x /= y)
```

defines 2 operations with *default methods* for each operator.

Polymorphism versus Typeclasses

- **Polymorphism** captures similar structure over different values. For example, a sequence of integers, sequence of strings, etc. can be captured by a polymorphic *List*.
- **Typeclasses** capture similar operations over different structures. For example, equality of integers, equality of trees, etc. can be captured by a class `Eq`.

Typeclass Inheritance

`Ord` ordering inherits all the operations in **`Eq`**:

```
-- Eq is a super typeclass of Ord.
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min             :: a -> a -> a
```

Note the previous definition of **`quicksort`** had type: **`Ord a => [a] -> [a]`**

Standard Prelude Typeclasses

Haskell comes with standard typeclasses like **`Eq`**, **`Ord`** and **`Show`**.

- **`Eq`** for **`==`**, **`/=`**.
- **`Ord`** for **`<`**, **`<=`**, **`>`**, **`>=`**, **`max`**, **`min`**, **`compare`**, **`Ordering`**
- **`Show`** for converting types to character strings.
- **`Read`** for converting character strings to types.
- **`Num`** for numeric types.

Haskell I/O

hello.hs

```
hello =
  do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ "Hello " ++ name ++ "!"
```

Haskell I/O Log

```
*Main> hello
Hello, what's your name?
Tim
Hello Tim!

*Main> :type hello
hello :: IO ()
*Main> :type putStrLn
putStrLn :: String -> IO ()
*Main> :type getLine
getLine :: IO String
*Main>
```

I/O Actions

- If a return value has type involving `IO`, then it means that function has a side-effect.
- All I/O actions have the type `IO t`.
- Produces side-effect when *performed*, not *evaluated*.
- I/O is performed within another I/O action or at the top-level.

I/O Actions Continued

- **`do`** allows *sequencing* I/O actions.
- **`<-`** is an operator which extracts a value of type **`t`** from **`IO t`**. Hence **`getLine :: IO String`**.

Purity

Restrict side-effects. **`nameToGreet`** below is *pure* and has type **`String -> String`**.

hello2.hs

```
nameToGreet name =
  "Hello " ++ name ++ "!"

hello2 =
  do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ nameToGreet name
```

Lazy I/O

- Function `hGetContents :: Handle -> IO String` returns all the contents from current position in Handle.
- Does not produce any I/O when called.
- I/O is performed lazily as return value is processed.
- Can conceptually have all contents of a 4GB file in memory. The contents can then be processed using pure functions with I/O occurring lazily behind the scenes.

Safe Division Revisited

Previously we had defined safe division to return a **`Maybe Num`**. What happens if we try to use safe division for a general arithmetic expression.

```
data ArithExp =
  Leaf Int |
  Add ArithExp ArithExp |
  Sub ArithExp ArithExp |
  Mul ArithExp ArithExp |
  Div ArithExp ArithExp


safeEval :: ArithExp -> Maybe Int

safeEval (Leaf i) = (Just i)
safeEval (Add e1 e2) =
  case (safeEval e1, safeEval e2) of
    (Just e1', Just e2') -> Just (e1' + e2')
    otherwise -> Nothing
safeEval (Sub e1 e2) =
  case (safeEval e1, safeEval e2) of
    (Just e1', Just e2') -> Just (e1' - e2')
    otherwise -> Nothing
safeEval (Mul e1 e2) =
  case (safeEval e1, safeEval e2) of
    (Just e1', Just e2') -> Just (e1' * e2')
    otherwise -> Nothing
safeEval (Div e1 e2) =
  case (safeEval e1, safeEval e2) of
    (Just e1', Just e2') ->
      if (e2' == 0) then Nothing else Just (e1' `div` e2')
    otherwise -> Nothing
```

Safe Division Revisited Retrospective

```
*Main> safeEval (Add (Leaf 2)  (Div (Leaf 10) (Leaf 2)))
Just 7
*Main> safeEval (Add (Leaf 2)  (Div (Leaf 10) (Leaf 0)))
Nothing
```

- Having to pattern match on **`Maybe`** possibilities can get tedious.
- Basically, we want a sequence of **`Maybe`** computations, but if any individual computation returns **`Nothing`**, then we want to abort the remaining computations and return **`Nothing`**.
- Many similar situations. For example, if we have a sequence of error-returning computations and an individual computation returns an error, then we want to abort the remaining computations and return the error.
- Need to abstract this out into a general structure.

Monads

- A monad is like a container, representing a computational context.
- A monad `M` wrapping some value of type `a` is denoted as `M a`.
- `Maybe a = Nothing | Just a` is such a container.
- `Either e z = Left e | Right z` is another such container. This can be used to wrap a result which may be an error `e` or a result `z`.
- `IO` is another such container.
- The `do` and `<-` syntactic sugar can be used to chain computations within a monad.
- A monad must provide certain functions which must obey specific *monad laws*.

Safe Division Using do Notation

```
data ArithExp =
  Leaf Int |
  Add ArithExp ArithExp |
  Sub ArithExp ArithExp |
  Mul ArithExp ArithExp |
  Div ArithExp ArithExp


safeEval :: ArithExp -> Maybe Int

safeEval (Leaf i) = (Just i)
safeEval (Add e1 e2) = do
  e1' <- safeEval e1
  e2' <- safeEval e2
  Just (e1' + e2')
safeEval (Sub e1 e2) = do
  e1' <- safeEval e1
  e2' <- safeEval e2
  Just (e1' - e2')
safeEval (Mul e1 e2) = do
  e1' <- safeEval e1
  e2' <- safeEval e2
  Just (e1' * e2')
safeEval (Div e1 e2) = do
  e1' <- safeEval e1
  e2' <- safeEval e2
  if e2' == 0 then Nothing else Just (e1' `div` e2')
```

Testing

Having static typing and pure functions makes it possible to perform **property-based testing**:

- Instead of having tests specify particular input/output pairs for a function under test, specify properties which the function must have.
- The test framework can automatically generate random inputs for testing the property.
- If a test fails, the framework can *shrink* the test until it can find a minimal test which also fails.
- QuickCheck for Haskell is one of the first property-based testing frameworks.
- The *hello world* for QuickCheck:

```
ghci> quickCheck(\ls -> reverse (reverse ls) == ls)
+++ OK, passed 100 tests.
```

  Reversing a list twice results in the same list.

Not Covered

Some of the topics not covered include:

- Modules.
- Functors, Applicatives, Semigroups, Monoids.
- Details of monads.

References

- Miran Lipovaca, Learn You a Haskell for Great Good!. Highly recommended.
- Bryan O'Sullivan, Don Stewart, and John Goerzen, Real World Haskell, O'Reilly, 2009.
- haskell.org Particularly see the documentation
- Haskell Reference
- Paul Hudak, *The Haskell School of Expression: Learning Functional Programming through Multimedia*, Cambridge University Press, 2000.
- Bryan O'Sullivan, John Goerzen and Don Stewart, *Real World Haskell*, O'Reilly, 2009.
- School of Haskell
- Simon Thompson, *Haskell: The Craft of Functional Programming*, Second Edition, Addison-Wesley, 1999.