



Revision Notes

Basics

Function notation

In Mathematics, function application is denoted by parentheses and multiplication by juxtaposition or space, e.g., $f(a, b) + cd$. In Haskell, function application is denoted by using space, and multiplication by `*`, giving `f a b + c*d`.

Function application binds strongest: `f a + b` means `f (a) + b` not `f (a+b)`.

To use a function that takes two arguments between these arguments instead of before, i.e. `4 `div` 2` instead of `div 4 2`, you must use back quotes - ```.

Naming

Function and argument names must start with a *lower-case letter*. By convention, list variables usually have an 's' suffix, e.g., `xs` or `ns`.

Layout

Layout is important - each definition must begin in precisely the same column. This avoids the need for explicit grouping with curly braces.

N.B., using where align variable names, not equals, e.g.:

where		where
a = 2	NOT	a = 2
xs = [1..3]		xs = [1..3]

Types

Types are automatically calculated at compile time, using a process called *type inference*. Checking at compile time is safer and faster as it removes the need for checks at run time.

Basic types are `Bool`, `Int` (fixed precision), `Integer` (arbitrary precision), `Float`, `Char`.

A *list* is a sequence of values of the same type. A *tuple* is a sequence of values of different type. The type of a tuple encodes its size.

Functions with multiple arguments are possible by returning *functions as results*, e.g.

```
add' :: Int -> (Int -> Int)
add' x y = x+y
```

`add'` takes an integer `x` and returns a function, `add' x`. In turn, this function takes an integer `y` and returns the integer result, `x+y`.

Function who take their arguments one after the other are called *Curried functions*.

'`->`' associates to the right, so `Int -> Int -> Int` means `Int -> (Int -> Int)`.

Consequently, function application associates to the left.

A function which contains more than one type, i.e. `length :: [a] -> Int` are called *Polymorphic Functions*.

If a function's type is constrained, the function is said to be *Overloaded*. E.g., `sum :: Num a => [a] -> a`

Haskell type classes include `Num`, `Eq`, and `Ord`.

Defining Functions

Conditional Expressions

Using if-then-else construction:

```
signum :: Int → Int
signum n = if n >= 0 then -1 else
            if n == 0 then 0 else 1
```

In Haskell, conditional statements must *always* have an else branch to avoid any ambiguities.

Guarded Expressions

```
signum :: Int → Int
signum n | n >= 0    = -1
         | n == 0    = 0
         | otherwise = 1
```

Haskell tests each guard one by one. *The first one to match is taken.*

Pattern Matching

```
(&&)      :: Bool → Bool → Bool
True && True  = True
True && False = False
False && True = False
False && False = False
```

Although the following is shorter, using the wildcard symbol, `_`:

```
True && True  = True
_ && _        = False
```

But the following is more efficient, as it avoids evaluating the second argument if the first is False:

```
False && _    = False
True && x      = x
```

Patterns are always matched *in order*, so the following always returns False:

```
_ && _        = False
True && True  = True
```

Variables must not be repeated on the lhs:

```
x && x        = x
_ && _        = False
```

Integer Patterns

```
pre      :: Int → Int
pre (n+1) = n
```

`n+k` patterns must match integers `>= 0` and must be parenthesised (as application binds stronger than `+`)

List Patterns

The operator `' : '` (cons) adds an element to the start of a list, so `[1,2,3,4] = 1:(2:(3:(4:([]))))`

Similarly to integer patterns, lists can be defined using `(x:xs)`, e.g.:

```
tail (_:xs) = xs
```

`(x:xs)` must always be parenthesised and cannot match the empty list.

Lambda Expressions

Functions can be constructed without naming the function - `λx → x+1` return the successor of the input. Lambda is represented at the keyboard by `'\'`.

Revision Notes

Lambda expressions are used to give a formal meaning to curried functions:

```
add x y = x + y
```

```
add = λx → (λy → x+y)
```

```
compose f g x = f (g x)
```

```
compose f g = λx → f (g x)
```

Also useful for avoiding naming functions that are only used once:

```
map f [0..99]
```

```
where
```

```
  f x = x*2 + 1
```

can be simplified to

```
map (λx → x*2 + 1) [0..99]
```

Sections

To use a sign before two arguments that is normally between them, enclose it in parentheses, i.e. $(+)$ 4 2 instead of $4 + 2$. This is called a section. Some uses of sections:

(+1) - successor function

(*2) - double function

(/2) - halving function

(1/) - reciprocation function

List Comprehensions

```
[(x,y) | x ← [1..3], y ← [1..2]]
```

The expressions to the right of the bar is called the *generator*. In this case, there are multiple generators. Their order matters - later generators are like deeper nested loops - they change more quickly.

Dependant Generators

Later generators can depend on earlier ones:

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

Guards

Comprehensions can include guards to limit the values produced by earlier generators, e.g.:

```
[x | x ← [1..10], even x]
```

The *Zip* function maps two lists to a list of pairs of their corresponding elements, e.g.:

```
zip [1,2,3] [a,b,c] = [(1,a), (2,b), (3,c)]
```

A *String* is simply a list of characters, so any list operators can be used on strings, e.g., "Hel" ++ "lo"

Recursive Functions

Some common recursive definitions:

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

```
product    :: [Int] → Int
product []  = 1
product (x:xs) = x * product xs
```

```
length     :: [a] → Int
length []  = 0
length (_:xs) = 1 + length xs
```

```
reverse    :: [a] → [a]
reverse []  = []
reverse (x:xs) = reverse xs ++ [x]
```

```
take       :: Int → [a] → [a]
take 0 _   = []
take _ []  = []
take (n+1) (x:xs) = x : take n xs
```

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

```
(++)      :: [a] → [a] → [a]
[] ++ ys  = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

To work out a recursive definition, simply apply the definitions until you reach the base case. E.g.,

```
take      :: Int → [a] → [a]
take 0 _  = []
take _ [] = []
take (n+1) (x:xs) = x : take n xs
```

```
take 2 [2,4,6,8,10]
= 2 : take 1 [4,6,8,10]
= 2 : 4 : take 0 [6,8,10]
= 2 : 4 : []
= [2,4]
```

Quicksort is defined by two rules: i) the empty list is already sorted; ii) Non-empty lists can be sorted by sorting the tail values \leq the head, sorting the tail values $>$ the head, and then appending the resulting lists on either side of the head value. I.e.:

```
qsort     :: [Int] → [Int]
qsort []  = []
qsort (x:xs) = qsort [a | a ← xs, a <= x] ++ [x] ++ qsort [b | b ← xs, b > x]
```

Higher Order Functions

A function is called *Higher-order* if it takes a function as an argument and returns a function as it's result, e.g. `twice f x = f (f x)`

The *map* function applies a function to every element of a list.

```
map (/2) [2,4,6] = [1,2,3]
```

The *filter* function selects every element of a list that satisfies a predicate.

```
filter even [1..10] = [2,4,6,8,10]
```

Foldr

A number of functions on lists can be defined using this pattern:

```
f [] = v
```

```
f (x:xs) = x ⊕ f xs
```

i.e., the empty list maps to something, v and the non-empty list maps to some function \oplus (this symbol is pronounced 'plusle' applied to the head and f of it's tail. *Foldr* expresses this pattern taking the function, \oplus and the empty list result as arguments, e.g.:

```
product [] = 1
```

```
product (x:xs) = x * product xs
```

turns into

```
product = foldr (*) 1
```

It is easiest to think of *foldr* as replacing each `:` of a given list by a given function and the empty list by a given value, i.e.

```
sum [1,2,3]
```

```
foldr (+) 0 [1,2,3]
```

```
foldr (+) 0 (1:(2:(3:[])))
```

```
1+2+3+0
```

```
6
```

Foldr can be used to represent many more function than you might at first think:

```
length [1,2,3]
```

```
length (1:(2:(3:[])))
```

```
1 + (1 + (1 + (0)))
```

```
= 3
```

Hence, `length = foldr ($\lambda x n \rightarrow 1+n$) 0`

```
reverse [1,2,3]
```

```
reverse (1:(2:(3:[])))
```

```
(([] ++ [3]) ++ [2]) ++ [1]
```

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

Hence, `reverse = foldr ($\lambda x xs \rightarrow xs ++ [x]$) []`

Other Functions

`(.)` is function composition, so `f . g = $\lambda x \rightarrow f (g x)$`

all decides if all elements in a list satisfy a predicate - `all even [2,4,6] = True`

any decides if any elements in a list satisfy a predicate - `any isSpace "Hello There" = True`

takeWhile takes elements of a list while a predicate holds - `takeWhile isAlpha "hi there" = "hi"`

dropWhile removes elements while a predicate holds - `dropWhile isSpace " hi" = "hi"`

Interactive Programs

Haskell enables interactivity by defining a special type to distinguish pure expressions from impure actions that may involve side effects. This type is `IO a` - the type of actions that return result of type `a`.

Primitive Actions

`getChar :: IO Char`

reads a character from the keyboard, echos it to the screen and returns the character as a result.

`putChar :: Char → IO ()`

writes a character to the screen and returns no result value.

`return :: a → IO a`

returns the value of `a` without any interaction.

Sequencing Actions

A sequence of actions can be combined as a single composite action using the keyword `do`:

`getTwo :: IO (Char, Char)`

`getTwo = do x ← getChar`

`y ← getChar`

`return (x,y)`

Other Library Actions

`getLine :: IO String`

`putStr :: String → IO ()`

`putStrLn :: String → IO ()`

Functional Parsers

A Parser can be viewed as taking a `String` and returning a tree: `Parser :: String → Tree`

It may not consume all of it's input: `Parser :: String → (Tree, String)`

It may be parsable in several ways (including 0): `Parser :: String → [(Tree, String)]`

It may not always return a `String` so we generalise: `Parser a :: String → [(a, String)]`

The parser `Item` fails if the input is empty and consumes the first character otherwise:

`item :: Parser Char`

`item = λinp → case inp of`

`[] → []`

`(x:xs) → [(x,xs)]`

The parser `mzero` always fails:

`mzero :: Parser a`

`mzero = λinp → []`

The parser `p +++ q` behaves as `p` if it succeeds, and `q` if not:

`(+++) :: Parser a → Parser a → Parser a`

`p +++ q = λinp → take 1 (p inp ++ q inp)`

The function `papply` applies a parser to a `String`:

`papply :: Parser a → String → [(a,String)]`

`papply p inp = p inp`



Some other library parsers include:

Parsing a character that satisfies a predicate:
`sat :: (Char → Bool) → Parser Char`
`sat p = do c ← item`
 if p c then
 return c
 else
 mzero

Parsing a digit:
`digit :: Parser Char`
`digit = sat isDigit`

Parsing a specific character:
`char :: Char → Parser Char`
`char c = sat (c ==)`

Applying a parser zero or more times:
`many :: Parser a → Parser [a]`
`many p = many1 p +++ return []`

Applying a parser one or more times:
`many1 :: Parser a → Parser [a]`
`many1 = do x ← p`
 xs ← many p
 return (x:xs)

Parsing a specific string:
`string :: String → Parser String`
`string [] = return []`
`string (c:cs) = do char c`
 string cs
 return (c:cs)

Example:

A Parser that consumes one or more digits from a string:

```
digits :: Parser String
digits = do char '['
           d ← digit
           ds ← many (do char ',',
                        digit)
           char ']'
           return (d:ds)
```

Turning a context-free grammar into a Haskell parser

Remember the grammar for arithmetic expressions (see G51MAL):

```
expr → term '+' expr | term '-' expr | term
term → factor '*' term | factor '/' term | factor
factor → digit | '(' expr ')'
```

First, this is factorised for reasons of efficiency:

```
expr → term ( '+' expr | '-' expr | ε)
term → factor ( '*' term | '/' term | ε)
factor → digit | '(' expr ')'
```

Revision Notes

It is then a straight-forward step to translate this into a functional parser:

```
expr :: Parser Int
expr = do t <| term
      do char '+'
        e <| expr
        return (t + e)
      +++ do char '-'
        e <| expr
        return (t - e)
      +++ return t

term  :: Parser Int
term  = do f <| factor
      do char '*'
        t <| term
        return (f * t)
      +++ do char '/'
        t <| term
        return (f `div` t)
      +++ return f

factor :: Parser Int
factor = do d <| digit
        return (digitToInt d)
      +++ do char '('
        e <| expr
        char ')'
        return e
```

Defining Types

Type Declarations

A new for an existing can be defined:

```
type String = [Char]
```

Type declaration can also be used to make types easier to read:

```
type Pos = (Int, Int)
```

and so

```
origin    :: Pos
```

```
origin    = (0,0)
```

```
left      :: Pos → Pos
```

```
left (x,y) = (x-1, y)
```

Type declarations can also have parameters:

```
type Pair a = (a,a)
```

and so

```
copy      :: a → Pair a
```

```
copy x    = (x,x)
```

Type declarations can be nested:

```
type Pos   = (Int, Int)
```

```
type Trans = Pos → Pos
```

But they cannot be recursive:

```
type Tree = (Int, [Tree])
```

Revision Notes

Data Declarations

A new type can be defined by specifying its set of values using a data declaration:

```
data Answer = Yes | No | Unknown
```

Here, Yes, No and Unknown are called the constructors of the type Bool. N.B. Type and constructor names must begin with an Upper Case Letter.

```
Constructors can have parameters: data Shape = Circle Float | Rect Float Float
```

```
Data types themselves can also have parameters: data Maybe a = Nothing | Just a
```

Although *type* declarations cannot be recursive, *data* declarations can:

```
data Nat = Zero | Succ Nat
```

e.g., Succ (Succ (Succ Zero)) :: Nat and represents the number 3.

```
nat2int :: Nat -> Int
```

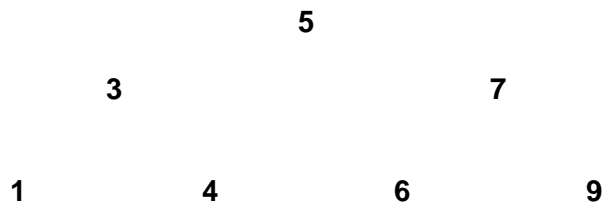
```
nat2int Zero = 0
```

```
nat2int (Succ n) = 1 + nat2int n
```

We can define a type to represent arithmetic expressions such as addition and multiplication:

```
data Expr = Val Int | Add Expr Expr | Mul Expr Expr
```

Binary Trees



```
Given a data type, data Tree = Leaf Int
                             | Node Tree Int Tree
```

We can represent this tree as:

```
Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9))
```

Function that decides in a certain value is contained in a tree:

```
find :: Int -> Tree -> Bool
find x (Leaf n) = x==n
find x (Node l n r) = x==n
                    || find x l
                    || find x r
```

Function that returns a list of all values in the tree:

```
flatten :: Tree -> [Int]
flatten (Leaf n) = [n]
flatten (Node l n r) = flatten l
                    ++ [n]
                    ++ flatten r
```

A binary tree is said to be a *search tree* if it flattens to an ordered list.

Knowing this, we can amend our definition of find:

```
find :: Int -> Tree -> Bool
find x (Leaf n) = x==n
find x (Node l n r) | x==n = True
                    | x<n = find x l
                    | x>n = find x r
```

The Countdown Problem

Rules: all numbers, including intermediate results must be integers > 0.
each of the source numbers can be used at most once.

Define the operators:

```
data Op = Add | Sub | Mul | Div
```

Apply an operator:

```
apply      :: Op -> Int -> Int -> Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

Decide if the result of applying an operator to two integers greater than zero is also an integer greater than zero:

```
valid      :: Op -> Int -> Int -> Bool
valid Add _ _ = True
valid Sub x y = x > y
valid Mul _ _ = True
valid Div x y = x `mod` y == 0
```

Expressions:

```
data Expr = Val Int | App Op Expr Expr
```

Evaluate an expression:

```
eval      :: Expr -> [Int]
eval (Val n)      = [n | n > 0]
eval (App o l r) = [apply o x y | x -> eval l, y -> eval r, valid o x y]
```

Returns a list of all possible ways of selecting zero or more elements from a list:

```
subbags :: [a] -> [[a]]
So, subbags [1,2] = [[], [1], [2], [1,2], [2,1]]
```

Returns a list of all the values in an expression:

```
values    :: Expr -> [Int]
values (Val n)      = [n]
values (App _ l r) = values l ++ values r
```

Decides if an expression is a solution for a given list of source numbers and a target result:

```
solution  :: Expr -> [Int] -> Int -> Bool
solution e ns n = elem (values e) (subbags ns) && eval e == [n]
```

Returns a list of all possible ways of splitting a list into two non-empty parts:

```
nesplit :: [a] -> [[a],[a]]
So, nesplit [1,2,3,4] = [[1],[2,3,4]],[[1,2],[3,4]],[[1,2,3],[4]]]
```

Returns a list of all possible expressions whose values are precisely a given list of numbers:

```
exprs    :: [Int] -> [Expr]
exprs [] = []
exprs [n] = [Val n]
exprs ns = [e | (ls,rs) -> nesplit ns
               , l      -> exprs ls
               , r      -> exprs rs
               , e      -> combine l r]
```

Revision Notes

Combines two expressions using each operator:

```
combine      :: Expr -> Expr -> [Expr]
combine l r = [App o l r | o -> [Add,Sub,Mul,Div]]
```

Returns a list of all possible expressions that solve an instance of the countdown problem:

```
solutions    :: [Int] -> Int -> [Expr]
solutions ns n = [e | cs -> subbags ns, e -> exprs cs, eval e == [n]]
```

Lazy Evaluation

- Avoids doing unnecessary evaluation
- Allows programs to be more modular
- Allows us to program with infinite lists

Expressions are evaluated or reduced by successively applying definitions until no further simplification is possible:

```
square n = n * n
square (3+4)
square 7
7 * 7
49
```

OR

```
square n = n * n
square (3+4)
(3+4) * (3+4)
7 * (3+4)
7 * 7
49
```

In Haskell, two different (both terminating) ways of evaluating the same expression will always give the same final result.

There are two common strategies to decide which subexpression to evaluate first:

Given: loop = tail loop

Innermost Reduction

```
fst (1, loop)
= fst (1, tail loop)
= fst (1, tail (tail loop))
... this strategy does not terminate
```

Outermost Reduction

```
fst (1, loop)
= 1
```

- Outermost reduction may give a result when innermost reduction fails to terminate
- If there exists an reduction sequence that terminates, then outermost reduction also terminates with the same result.

Looking again at the square (3+4) example, it is evident that outermost reduction is less efficient - the subexpression (3+4) is duplicated when square is reduced. This problem is solved by using pointers to indicate the *sharing* of expressions during evaluation:

```
square (3+4)
= ( * ) (3+4)
= ( * ) 7
= 49
```



Revision Notes

- Lazy Evaluation is a combination of Outermost evaluation and Sharing.
- It never requires most reduction than innermost reduction.

Infinite Lists (e.g., ones = 1: ones)

Innermost reduction:

Lazy evaluation:

head (ones) = head (1:ones) = head (1:1:ones)

head (ones) = head (1:ones) = 1

Using Lazy Evaluation, expressions are only evaluated *as much as required* to produce the final result.

Lazy Evaluation allows us to make programs more modular, by separating control from data:

take 5 ones
take 5 [1..]

control data

The data is only evaluated as much as required by the control part.

Example - generating prime numbers using the 'Seive of Eratosthenes'

1. Write down the list 2,3,4,5..., 2. Mark the first value as p, 3. Delete all multiples of p, 4. Return to 2.

seive :: [Int] → [Int]
seive p:xs = p:seive [x | x ← xs, x `mod` p /= 0]

primes :: [Int]
primes = seive [2..]

By freeing the generation of primes from the constraint of finiteness, we obtain a modular definition on which different boundary conditions can be imposed - take 10 primes, takeWhile (<15) primes, etc.

Revision Notes

Reasoning about Programs

Given

$$\begin{aligned} \text{reverse } [] &= [] && \text{eq1} \\ \text{reverse } (x:xs) &= \text{reverse } xs ++ [x] && \text{eq2} \end{aligned}$$

Show that

$$\forall xs. \text{reverse } (\text{reverse } xs) = xs$$

Base Case:

$$\begin{aligned} \text{reverse } (\text{reverse } []) & \\ \text{reverse } [] & && \text{by eq1} \\ [] & && \text{by eq1} \end{aligned}$$

Step Case:

$$\begin{aligned} \text{reverse } (\text{reverse } xs) &= xs \quad | - \\ \text{reverse } (\text{reverse } (x:xs)) &= (x:xs) \\ \text{reverse } (\text{reverse } xs ++ [x]) & && \text{by eq2} \\ \text{Assume: } \text{reverse } (ys ++ xs) &= \text{reverse } xs ++ \text{reverse } ys && \text{eq3} \\ \text{reverse } [x] ++ \text{reverse } (\text{reverse } xs) & && \text{by eq3} \\ \text{reverse } [x] ++ xs & && \text{by Induction hypothesis} \\ [x] ++ xs &= (x:xs) \end{aligned}$$

Given

$$\begin{aligned} (f . g) x &= f (g x) && \text{eq1} \\ \text{map } f [] &= [] && \text{eq2} \\ \text{map } f (x:xs) &= f x : \text{map } f xs && \text{eq3} \end{aligned}$$

Show that

$$\forall xs. \text{map } f (\text{map } g xs) = \text{map } (f . g) xs$$

Base Case:

$$\begin{aligned} \text{map } f (\text{map } g []) & \\ \text{map } f ([]) & && \text{by eq2} \\ [] & && \text{by eq2} \end{aligned}$$

Step Case:

$$\begin{aligned} \text{map } f (\text{map } g xs) &= \text{map } (f . g) xs \quad | - \\ \text{map } f (\text{map } g (x:xs)) &= \text{map } (f . g) (x:xs) \\ \text{map } f (g x : \text{map } g xs) & && \text{by eq3} \\ f (g x) : \text{map } f (\text{map } g xs) & && \text{by eq3} \\ f (g x) : \text{map } (f . g) xs & && \text{by Induction hyp} \\ \text{map } (f . g) (x:xs) & && \text{by eq3 in reverse} \end{aligned}$$