PROGRAMMING I
Part I: Functional Programming in Haskell

Tony Field

Course Outline

I Expressions and Basic Types

II Functions

III List Processing

IV User-defined Types

V Classes

Suggested Books:

- *Haskell: The Craft of Functional Programming*
  Simon Thompson
  Addison Wesley 1996
- *The Haskell School of Expression*
  Paul Hudak, 2000
  Cambridge University Press
- *Introduction to Functional Programming using Haskell*
  Richard Bird
  Prentice Hall 1988
- *Programming Challenges – the Programming Contest Training Manual*
  Steven S. Skiena and Miguel A. Revilla
  Springer 2003

- *Functional Programming*
  Tony Field and Peter Harrison
  Addison Wesley 1988
- Also, take a look at the Haskell web site: http://haskell.org/

Part I: Expressions and Basic Types

- Mathematical expressions are already familiar to you. In Haskell they are the basis of all computation, e.g.

```
4 - 3
sin 24.9 * cos ( pi * 1.175 / 4 )
```

- Expressions like these can be typed directly at the Haskell system and the answer will be printed immediately, e.g.

```
Hugs.Base> sin 24.9 * cos ( pi * 1.175 / 4 )
-0.139208
```

- '`-`' and '`*`' are called *operators* or *infix functions*
- '`cos`' is a *prefix* function (we usually drop the word "prefix")
- '`pi`' is a predefined constant (an approximation to π)

- The 'whole' numbers like 4 and 3 are called *integers* (abbreviated to `Int`)
- `Int`s occupy a fixed amount of space (32 bits); the smallest `Int` that can be represented is `-2147483648` and the largest is `+2147483647`
- 2.75, 24.9 and 1.175 are *real* numbers which are approximated by *floating point* numbers (`Float` in Haskell)
- As with `Int`s, a fixed amount of space (32 bits) is used to represent each `Float`
- Only a subset of the real numbers can be represented so floating-point arithmetic is not always accurate
- The smallest `Float` that can be represented is `-3.40282347E+38` and the largest is `+3.40282347E+38`

Bracketing

- If necessary, brackets (*parentheses*) can be used to get the right meaning. For example

```
2 - 3 * 4 / sin 2 * pi
```

is bracketed implicitly by Haskell as

```
( 2 - ( ( ( 3 * 4 ) / ( sin 2 ) ) * pi ) )
```

because:
  - '`*`' and '`/`' have higher *precedence* than '`-`'
  - '`*`' and '`/`' are *left-associative*
  - Prefix function application has higher precedence than (all) infix function applications
- We can put brackets where we want to make the meaning clear

Qualified Expressions

- We can also name values and use the name instead of the value
- This can be done in Haskell using a `let` expression, e.g. instead of `24.9 * cos 1.175` we could equally write

```
let f = 24.9 in f * cos 1.175
let f = 24.9 ; theta = 1.175 in f * cos theta
let g = cos in 24.9 * g 1.175
```

All three produce the same *value* 9.60002 (an object of type `Float`)

- `f, g` and `theta` are called 'variables' or 'identifiers'
- `let` expressions are sometimes called *qualified* expressions; the bits before the '`in`' are the *qualifiers* and the final expression is called the *resultant*

Characters and Truth Values

- In addition to `Int` and `Float` Haskell supports other *base* types including:
  - Characters (`Char`), e.g. `'a', 'b', 'A', 'X', '!'` etc.
  - Truth values (`Bool`), which are either `True` or `False`
- Some Haskell operators work on `Bools`, including *and* (`&&`), *or* (`||`) and *not* (`not`); for example

```
Hugs.Base> not False
True
Hugs.Base> False && ( not True )
False
Hugs.Base> not ( True && ( False || not True ) )
True
```


- Values of type `Bool` are produced by *comparison* operators:

`==` Equal, as in `5 == ( 4 + 1 )`

`/=` Not equal, as in `'a' /= 'p'`

`>` Greater than, as in `12 > 9`

`<` Less than, as in `( 12.8 * 9 ) < 2`

`<=` Less than or equal, as in `5 <= 6`

`>=` Greater than or equal, as in `44 >= 45`

- So, we can put things together, e.g.

```
Hugs.Base> ( 1 < 9 ) || ( ( 4 == 7 ) && ( 'a' > 'm' ) )
True
Hugs.Base> sin 3 > cos ( 2 * pi ) || 4 / 5 <= sin 0.8
False
```


- And also by some of the built-in prefix functions, e.g.

  `even` – Returns `True` iff a given number is even

  `odd` – Returns `True` iff a given number is odd

  `isSpace` – Returns `True` iff a given character is `' '`

  `isDigit` – Returns `True` iff a given character is one of `'0'..'9'`

- For example (note that `isSpace` and `isDigit` are defined in module `Char`):

```
Hugs.Base> :l Char
Char> even ( 13^2 ) && isDigit '*' || isSpace '9'
False
Char> not ( odd 7 && even 11 )
True
```


Conditionals

- *Conditional* expressions are of the form

```
if P then Q else R
```

  where `P`, `Q` and `R` are expressions

- `P` must have a `Bool` result; the types of `Q` and `R` must be the *same*
- if `P` evaluates to `True` then the overall result is `Q`; if it evaluates to `False` then the result is `R`, e.g.

```
Hugs.Base> if False then 5 - 3 * 4 else 2
2
Hugs.Base> let p = 'a' > 'z' in if p then p else False
False
```

? Can you simplify `if p then p else False`?

Tuples

- Sometimes it is convenient to be able to group a fixed number of values together, e.g.
  - Pairs of `Float` for representing cartesian/polar coordinates
  - Triples of `Float` for representing a 3D vector
  - Triples of `Int` for representing the time in h:m:s format
- We can build such *tuples* by enclosing the required components in brackets, e.g.

  `( 2.78, 14.9 )` is a two-tuple (or *pair*) of `Float`

  `( 1.0, 0.0, 0.0 )` is a three-tuple (or *triple*) of `Float`

  `( 2, 10, 16 )` is a triple of `Int`

- Tuple *types* are written using the same syntax as the tuples themselves, e.g. `( 2, 1, 6 )` is "of type" `( Int, Int, Int )`
- The tuple elements can have different types, e.g. `( True, 2, 'u' )` is of type `( Bool, Int, Char )`
- Tuples may be nested, e.g. `( 'c', ( 1, False ) )` has the type `( Char, ( Int, Bool ) )`
- There is no notion of a *one-tuple*, so `(True)` is the same as `True`

Lists

- Lists are used to represent collections (or 'sequences') of objects
- An empty list is written as `[]`
- A constant non-empty list can be written using square brackets with items separated by commas:

```
[ 1.0, 2.0, 3.0 ]       has type  [ Float ]
[ 'f' ]                 has type  [ Char ]
[ False, False ]        has type  [ Bool ]
[ [ 1.0, 2.0 ], [] ]    has type  [ [ Float ] ]
[ ( 80, True ) ]        has type  [ ( Int, Bool ) ]
```


- Lists should not be confused with sets in conventional mathematics, e.g.
  - The order in which the elements appear is important (lists can be *indexed* in a meaningful way)
  - Values may occur more than once
- The elements of a list can be of any type, so long as each element has the *same* type, so `[ 2, True, 4, 'u' ]` is *not* valid
- Compare this with tuples where the elements may have mixed type

Special Syntax: Strings

- Lists of characters (i.e. `[ Char ]`) are called *strings* (type `String`) and can be written by enclosing the characters in double quotation marks, e.g.

```
Hugs.Base> [ 'h', 'a', 'n', 'd', 'b', 'a', 'g' ]
"handbag"
Hugs.Base> :type "handbag"
"handbag" :: String
Hugs.Base> :t "handbag"
"handbag" :: String
```

? How is `'k'` different from `"k"`?

Special Syntax: Arithmetic Sequences

- The special form `[ a,b..c ]` builds the list of numbers `[ a, a+(b-a), a+2(b-a), ... ]` and so on until the value `c` is exceeded, e.g.

```
Hugs.Base> [ 1..5 ]
[1,2,3,4,5]
Hugs.Base> [ 2,4..11 ]
[2,4,6,8,10]
Hugs.Base> [ 10,9..0 ]
[10,9,8,7,6,5,4,3,2,1,0]
Hugs.Base> [ 0,0.5..4 ]
[0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0]
```

- Note: it works with `Int`s and `Float`s

Special Syntax: List Comprehensions

- A list *comprehension* takes the form

```
[ e | x1 <- l1, ..., xm <- lm, P1, ..., Pn ]
```

where

`e` is an expression

`xi` is a variable ($1 \leq$ `i` $\leq$ `m`)

`li` is a list ($1 \leq$ `i` $\leq$ `m`)

`Pi` is a *predicate* (i.e. `Bool`-valued expression) ($1 \leq$ `i` $\leq$ `n`)

- It is read "the list of all `e` where `x1` comes from list `l1`, ..., `xm` comes from list `lm`, and where `P1`, ..., `Pn` are all True"

- The terms of the form `x <- l` are called *generators*
- The target variable of a generator can only be used to the *right* of the generator and to the *left* of the '`|`'
- The terms after the '`|`' can appear in any order, subject to the above

```
Hugs.Base> [ x^2 | x <- [ 1..10 ], even x ]
[4,16,36,64,100]
Hugs.Base>  [ x | even x,  x <- [ 1..10 ] ]
ERROR: Undefined variable "x"
Hugs.Base> [ x+y | x <- [1..3], y <- [1..3] ]
[2,3,4,3,4,5,4,5,6]
Hugs.Base> [ ( x, y ) | x <- [ 1..3 ], y <- [ 1..x ] ]
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
```

- Aside: the operators `==`, `/=`, `>`, `<`,`>=`, `<=` are defined on lists in an obvious way (we'll see exactly how later), e.g.

```
Hugs.Base> [ 1, 1 ] == [ 1 ]
False
Hugs.Base> [ True, False ] == [ True, False ]
True
Hugs.Base> "False" /= "False"
False
Hugs.Base> [ 1, 7, 9 ] < [ 2, 5, 8 ]
True
Hugs.Base> "big" < "bigger"
True
Hugs.Base> "big" < "big"
False
```


Part II: Functions

- Programming is all about the packaging and subsequent use of computational "building blocks" of varying size and complexity
- In Haskell, the building blocks are *functions*; you have already seen some *built-in* functions like `+, *, div, sqrt, cos` etc. but we can define our own
- A function `f` is a rule for associating each element of a source type `A` with a unique member of a target type `B` (cf domain and range in mathematics); we express this thus: `f :: A -> B`
- `f` is said to "take an argument" (or "have a parameter") of type `A` and "return a result" of type `B`
- If the function takes several arguments their types are listed in sequence, e.g. `g :: A -> B -> C -> D`

- We say what the function does, using one or more *rules* (sometimes called *equations*)
- A rule has a *left-hand side* which lists the argument(s) and a *right-hand side* which is an expression
- The rule looks like a conventional mathematical function definition except that we omit brackets around the argument(s), e.g

```
add1 :: Int -> Int
add1 x = x + 1
```

- Note that function names must begin with a lower-case letter

- Some more examples...

```
isUpperCase :: Char -> Bool
isUpperCase ch = ch >= 'A' && ch <= 'Z'

diff :: Float -> Float -> Float
diff x y = if x >= y then x - y else y - x

fromOrigin :: ( Float, Float ) -> Float
fromOrigin ( x, y ) = sqrt ( x^2 + y^2 )

isEven :: Int -> Bool
isEven x = if x `mod` 2 == 0 then True else False
```

? Can you improve the right-hand-side of `isEven`?

- Note: Sometimes there are constraints on the values the arguments can take, e.g. the function to compute $\log x$ requires that $x > 0$
- When it is unclear what the constraints are it is conventional to list them using *preconditions*
- Preconditions are treated simply as annotations – they are *not* intended to be executed
- We use a Haskell *comment*, thus:

```
log :: Float -> Float
-- Pre: x > 0
log x = ...
```


- Once we have defined a function, we can *apply* it to given argument(s) provided the argument(s) have the right type
- The application of function `f` to an argument `a` is done by the *juxtaposition* `f a`, e.g.

```
Main> add1 569
570
Main> isEven 15
False
Main> isEven ( add1 19 )
True
```

- However, `add1 'b'` is a *type error* (note that badly typed programs cannot be executed!)

Guarded Rules

- Conditionals can also be written using *guards*, for example:

```
evenYN :: Int -> Char
evenYN x | x `mod` 2 == 0  =  'Y'
         | otherwise       =  'N'
```

```
diff :: Float -> Float -> Float
diff x y | x >= y    = x - y
         | otherwise = y - x
```

- Each guarded term of the form `... | g = e` is called a *clause*
- Remark: The quick way to define `diff` is to use the built-in function `abs`:

```
diff x y = abs ( x - y )
```


- A function rule can have arbitrarily many guards, e.g.

```
sign :: Int -> Int
sign x
  | x < 0  = -1
  | x == 0 = 0
  | x > 0  = 1
```

- Note the layout: we can lay out the clauses any way we want so long as they *all* lie textually to the right of the '`s`' of `sign`
- The above layout convention is preferred, especially for definitions with many clauses

Local Definitions

- In the same way that `let` expressions introduce definitions local to an expression, `where` *clauses* introduce definitions local to a rule
- This is useful for breaking a function down into simpler named components, e.g.

```
turns :: Float -> Float -> Float -> Float
turns start end r
  = totalDistance / distancePerTurn
    where
      totalDistance   = kmToMetres * ( end - start )
      distancePerTurn = 2 * pi * r
      kmToMetres      = 1000
```

- Note that the `where` must be to the right of the left-hand side

- A `where` clause can also avoid replication and redundant computation, e.g.

```
normalise :: ( Float, Float ) -> ( Float, Float )
normalise ( x, y )
  = ( x / sqrt ( x^2 + y^2 ), y / sqrt ( x^2 + y^2 ) )
```

The common subexpression can be factored out thus:

```
normalise :: ( Float, Float ) -> ( Float, Float )
normalise ( x, y )
  = ( x / s, y / s )
  where
    s = sqrt ( x^2 + y^2 )
```

`sqrt ( x^2 + y^2 )` will now be evaluated *once*

- They are also useful for naming the components of a tuple using *pattern matching*, e.g.

```
quotrem :: Int -> Int -> ( Int, Int )
quotrem x y = ( x `div` y, x `mod` y )

yardstoMFY :: Int -> ( Int, Int, Int )
yardstoMFY y
  = ( m, f, y'' )
    where
      ( m, y' )  = quotrem y 1760
      ( f, y'' ) = quotrem y' 220
```

This translates a distance in yards to a distance in (a whole number of) miles (1760 yards), furlongs (220 yards) and yards
E.g. `yardstoMFY 2640 = ( 1, 4, 0 )`, i.e. 1.5 miles exactly

- Note that you can define local *functions* too, e.g.

```
yardstoMFY :: Int -> ( Int, Int, Int )
yardstoMFY y
  = ( m, f, y'' )
    where
      ( m, y' )  = quotrem y 1760
      ( f, y'' ) = quotrem y' 220

      quotrem :: Int -> Int -> ( Int, Int )
      quotrem x y = ( x `div` y, x `mod` y )
```

- Here, `quotrem` cannot be used outside the definition of `yardstoMFY`

- Remark: To aid readability we can name types using a *type synonym*, e.g. (assuming `g` is already defined),

```
type Mass     = Float
type Position = ( Float, Float )
type Force    = ( Float, Float )

force :: Mass -> Mass -> Position -> Position -> Force
force m1 m2 ( x1, y1 ) ( x2, y2 )
  = ( f * dx / r, f * dy / r )
  where dx = abs ( x1 - x2 )
        dy = abs ( y1 - y2 )
        r = sqrt ( dx^2 + dy^2 )
        f = g * m1 * m2 / r^2
```

- Rule: Type synonyms must begin with a capital letter

Scope

- The *scope* of an identifier is that part of the program in which the identifier has a meaning
- All identifiers defined at the "top level" (i.e. non-local) are in scope over the entire program (they are *global*)
- Within each rule, each argument identifier *and* each local identifier is in scope everywhere throughout the rule
- Identifiers introduced in (nested) where clauses attached to local rules are in scope only in that local rule i.e. *not* in the outer rule as well

- Identifiers in `where` clauses supersede argument identifiers with the same name
- Similarly, identifiers in a nested `where` clause supersede those with the same name in an outer `where` clause, and so on, e.g.

```
f :: Int -> Int -> Int
f x y
  = x + y
  where
    y = x^2
      where
        x = 3
```

- Here, the function has the same meaning as `f x y = x + 9`; the `y` argument identifier is in scope nowhere!

Evaluation

- Haskell evaluates an expression by reducing it to its simplest equivalent form (called its *normal form*) and printing the result
- Evaluation can be thought of as rewriting or *reduction* (meaning simplification); a reducible expression is called a *redex*
- Reduction works by repeatedly reducing redexes until no more redexes exist; the expression is then in normal form
- E.g. consider `double (3 + 4)`, where

```
double :: Int -> Int
double x = x + x
```

- One possible reduction sequence is:

```
double (3 + 4)
-> double 7            by built-in rules for +
-> 7 + 7               by the rule for double
-> 14                  by built-in rules for +
```

- `14` cannot be further reduced (it is in normal form) and will be printed by the evaluator
- Another possible reduction sequence:

```
double (3 + 4)
-> (3 + 4) + (3 + 4)   by the rule for double
-> 7 + (3 + 4)         by built-in rules for +
-> 7 + 7               by built-in rules for +
-> 14                  by built-in rules for +
```


- Thus evaluation is a simple process of *substitution and simplification*, using primitive rules for the built-in functions and additional function rules supplied by the programmer
- If an expression has a *well-defined value*, then the order in which the evaluation is carried out does not affect the result (the *Church-Rosser* property)
- But, the evaluator selects a redex (from the set of possible redexes) in a consistent way. This is called its evaluation/reduction *strategy*
- Haskell's reduction strategy is called *lazy evaluation*, equivalent to choosing the *leftmost-outermost* redex each time

- Lazy evaluation reduces a redex *only* if the value of the redex is required to produce the normal form, e.g.

```
f :: Float -> Float -> Float
f x y = if x < 0 then 0 else y
```

- If `x` is negative, the second argument (`y`) is not required, hence

```
Main> f 3 5
5
Main> f 3 ( 6 / 0 )
Program error: primDivDouble 6.0 0.0
Main> f ( -5 ) ( 6 / 0 )
0
```

- More of this later...

Recursive Functions

- Let us consider functions for taking the second, third and fourth powers of a given `Float`:

```
square :: Float -> Float
square x = x * x

cube ::  Float -> Float
cube x = x * x * x

fourthpower ::  Float -> Float
fourthpower x = x * x * x * x
```

- What about computing x^n for an *arbitrary* value of $n \geq 0$?
- Problem: Written out explicitly, the number of terms in right-hand side expression would depend on the value of n

- The solution is to use a *recurrence relationship*—in this case one that defines x^n in terms of x^{n-1}:

$$
\begin{aligned}
x^n &= \overbrace{x \times x \times x \times \ldots \times x}^{n \text{ times}} \\
&= x \times x^{n-1}
\end{aligned}
$$

- This suggests the *recursive* Haskell function:

```
power :: Float -> Int -> Float
-- Pre: n >= 0
power x n = x * power x ( n - 1 )
```

- However, this is not quite right, e.g.


```
power 2 3
-> 2 * power 2 ( 3 - 1 )  =  2 * power 2 2
-> 2 * 2 * power 2 1
-> 2 * 2 * 2 * power 2 0
-> 2 * 2 * 2 * 2 * power 2 -1
-> ...
```

- Oops! We want things to stop at `power 2 0`, since this should give 1
- The case `power 2 0` is called a *base case*
- Note: the function is not designed to work for `n`<0

- Hence:

```
power :: Float -> Int -> Float
-- Pre: n >= 0
power x n = if n == 0 then 1 else x * power x ( n-1 )
```

- This function/definition is said to be *recursive*, since it calls itself
- Note that a measure of the *cost* of the function `power` is the number of multiplications required to compute `power n` for an arbitrary n
- Here the "cost" is n and we say that the function's *complexity* is "order n", written $O(n)$

$\boxed{?}$ How would you make `power` work for all integers `n`?

- This is how we build *all* recursive functions

1. Define the base case(s)
2. Define the recursive case(s)
   (a) Split the problem into one or more subproblems
   (b) Solve the subproblems
   (c) Combine results to get required answer

- The subproblems are solved by a *recursive* call to the (same) function

- Important: the subproblems *must* be “smaller” than the original problem otherwise the recursion never stops, e.g.

```
loop :: Int -> Int
loop x
  | x == 0  =  0
  | x > 0   =  1 + loop x
```

- For example...

```
loop 4
-> 1 + loop 4
-> 1 + 1 + loop 4
-> ...
```

- This is called an *infinite loop* or a *black hole*; the program runs forever, or until it runs out of memory

Example: `power2` which computes the same thing as `power` but more efficiently

- Idea: use the fact that x^n can be written $x^{n/2} \times x^{n/2} = (x^{n/2})^2$ if n is even and $x \times (x^{\lfloor n/2 \rfloor})^2$ if n is odd

- Graphically, for *even* n:

$$\begin{aligned} x^n &= \overbrace{x \times x \times \ldots \times x}^{n \text{ times}} \\ &= \underbrace{\overbrace{x \times \ldots \times x}^{n/2 \text{ times}}}_{x^{n/2}} \times \underbrace{\overbrace{x \times \ldots \times x}^{n/2 \text{ times}}}_{x^{n/2}} \end{aligned}$$

- Similarly for *odd* n

- The term $x^{\lfloor n/2 \rfloor}$ is referred to several times, so we’ll define it using a `where`; also we’ll arbitrarily use guards instead of conditionals:

```
power :: Float -> Int -> Float
-- Pre: n >= 0
power x n
  | n == 0 = 1
  | even n = k * k
  | odd n  = x * k * k    (or x * power x (n-1))
  where
    k = power x ( n ‘div‘ 2 )
```

? What is the cost now, in terms of the number of multiplications, for a given n?

- Another example: Newton’s method for finding the square roots of numbers. This repeatedly improves approximations to the answer until the required degree of accuracy is achieved.

- Given x, if a_n is the n^{th} approximation to $\sqrt{x}$ then

$$a_{n+1} = \frac{a_n + x/a_n}{2}$$

  gives the next approximation

- Let’s define a function `newtonSqrt` which given a number x returns a “good” approximation to $\sqrt{x}$

- Here we will use $x/2$ as the first approximation of $\sqrt{x}$, i.e. $a_0 = x/2$

- We want to stop when the approximation is "close" to $\sqrt{x}$
- We can check this by comparing a_n^2 to x; if $|a_n^2 - x| < \epsilon$ for some small value of ϵ (here set at 0.00001), we'll terminate the recursion, e.g.

```
newtonSqrt :: Float -> Float
-- Pre: x >= 0
newtonSqrt x
  = findSqrt ( x / 2 )
    where
      findSqrt :: Float -> Float
      findSqrt a
        | abs ( x - a * a ) < 0.00001  =  a
        | otherwise = findSqrt ( ( a + x/a ) / 2 )
```


- For example:

```
newtonSqrt 12
-> findSqrt 6.0
-> findSqrt 4.0
-> findSqrt 3.5
-> findSqrt 3.464286
-> findSqrt 3.464102
-> 3.464102
```

since `(12-3.464102*3.464102) < 0.00001`

Part III: List Processing

- The list square bracket notation (`[ , , , ]`) is actually a shorthand
- At the simplest level lists are put together using two types of building block:

  `[]` (pronounced "nil" or "empty-list") is used to build an empty list

  `:` (pronounced "cons") is an infix operator which adds a new element to the front of a list

- These work like any other function, but are called *constructors* for reasons which will become apparent

- New lists can be built by repeated use of '`:`', starting with `[]`, e.g.

```
Hugs.Base> []
[]
Hugs.Base> True : []
[ True ]
Hugs.Base> 1 : 2 : []
[ 1, 2 ]
```

- Thus, the expression `[ x1, ..., xn ]` is just a convenient shorthand for `x1 :  ...  :  xn :  []` (we can use either)
- Note also that `:` associates to the right so that `x :  x' :  xs` is interpreted as `x :  ( x' :  xs )`

Polymorphism

- Importantly, the constructors `[]` and `:` can be used to build lists of *arbitrary* type. They are therefore said to be *polymorphic*
- To express this in type definitions, we use a *type variable*, which is an identifier beginning with a lower-case letter
- For example, the types of the two list constructors are:

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

- Here `a` is a type variable; the second line reads: "`(:)` is a function which takes an object of *any* type `a` and a list of objects of (the same) type `a` and delivers a list of objects of (the same) type `a`"

- A variable in a type (e.g. `a` above) stands for any type (*for all* a, or $\forall a$), but once determined each `a` in the type must be the same
- For example, `Int -> [ Int ] -> [ Int ]` is a valid *instance* of type `a -> [ a ] -> [ a ]`, but `Char -> [ Int ] -> [ Int ]` is not
- Many other Haskell prelude functions are polymorphic, e.g.

`id :: a -> a`	The identity function
`fst :: ( a, b ) -> a`	Pair index
`snd :: ( a, b ) -> b`	Pair index

- Note that the type of `fst` and `snd` involve two type variables since pair elements can be of any type

- `[]` and `:` can be used in function definitions to build lists, e.g.

```
ints :: Int -> [ Int ]
-- Pre: n >= 0
ints n
  | n == 0  =  []
  | n > 0   =  n : ints ( n - 1 )
```

- Given a parameter `n` this generates the list `[ n, n-1, ..., 1 ]`

  $\boxed{?}$ How would you change `ints` to produce the numbers in increasing order, e.g. `ints 3 = [ 1, 2, 3 ]`?

- Another example: a variation of `newtonSqrt` that returns the list of all intermediate approximations to $\sqrt{x}$:

```
newtonSqrt :: Float -> [ Float ]
-- Pre: x >= 0
newtonSqrt x
  = findSqrt ( x / 2 )
    where
      findSqrt a
        | abs ( x - a * a ) < 0.00001  =  [ a ]
        | otherwise = a : findSqrt ( ( a + x/a ) / 2 )
```

e.g. `newtonSqrt 12 -> [6.0,4.0,3.5,3.464286,3.464102]`

- What about functions which *consume* lists? We now need a mechanism for taking lists apart...

Method 1: `null`, `head` and `tail`

- Let's write a function to sum the elements of a list of `Int`s using some built-in list processing functions:
  - `null :: [ a ] -> Bool` asks whether a list is empty;
  - `head :: [ a ] -> a` returns the head of a given list
  - `tail :: [ a ] -> [ a ]` returns the tail of a given list
- Summing an empty (`null`) list must return 0; for a non-empty list, add the head of the list to the result of summing the tail, thus:

```
sumInts :: [ Int ] -> Int
sumInts xs
  = if null xs
    then 0
    else head xs + sumInts ( tail xs )
```


- For example:

```
sumInts [ 10, 20, 30 ]
-> if null [ 10, 20, 30 ]
   then 0
   else head [ 10, 20, 30 ] + sumInts ( tail [ 10, 20, 30 ] )
-> head [ 10, 20, 30 ] + sumInts ( tail [ 10, 20, 30 ] )
-> 10 + sumInts [ 20, 30 ]
-> 10 + if null [ 20, 30 ] then ... else ...
-> 10 + head [ 20, 30 ] + sumInts ( tail [ 20, 30 ] )
-> 10 + 20 + sumInts [ 30 ]
-> 10 + 20 + if null [ 30 ] then ... else ...
-> 10 + 20 + head [ 30 ] + sumInts ( tail [ 30 ] )
-> 10 + 20 + 30 + if null [] then 0 else ...
-> 10 + 20 + 30 + 0
-> 60
```


Method 2: Pattern Matching

- Note that there are *exactly* two ways to build a list (`[]` and `:`) and hence *exactly* two ways to take them apart
- If we need to take a list apart then, when we look at the list, either:
  1. The list is empty, i.e. "of the form" `[]`
  2. The list is non-empty, i.e. "of the form" `( x : xs )` for some `x` and `xs`
- There are *no* other possibilities!
- Here, "of the form" means "matches the pattern"
- Another way to define `sumInts` is by *pattern matching*...

- There are two possible patterns, so we have two rules:

```
sumInts :: [ Int ] -> Int
sumInts []          =  0
sumInts ( x : xs ) =  x + ( sumInts xs )
```

- Think of the whole of each left-hand side as being a *pattern*; patterns are tested from top to bottom, and left to right internally
- If the pattern matches the expression we are trying to evaluate, we return the result of evaluating the right-hand side
- Note: the pattern `( x : xs )` also serves to name the two 'things' attached to the first `:`, namely the head and tail of the given list
- Generally, pattern matching is preferred to the use of `null`, `head` and `tail`

- Pattern matching simplifies how we think about reduction (although it's actually implemented similarly to before), e.g.

```
sumInts [ 10, 20, 30 ]
-> 10 + sumInts [ 20, 30 ]
-> 10 + ( 20 + sumInts [ 30 ] )
-> 10 + ( 20 + ( 30 + sumInts [] ) )
-> 10 + ( 20 + ( 30 + 0 ) )
-> 60
```


More on the Haskell Prelude

- Here are some of the more commonly-used predefined functions over lists

```
null    :: [ a ] -> Bool
head    :: [ a ] -> a
tail    :: [ a ] -> [ a ]
length  :: [ a ] -> Int
elem    :: Eq a => a -> [a] -> Bool   (see later)
(!!)    :: [ a ] -> Int -> a
(++)    :: [ a ] -> [ a ] -> [ a ]
take    :: Int -> [ a ] -> [ a ]
drop    :: Int -> [ a ] -> [ a ]
zip     :: [ a ] -> [ b ] -> [ ( a, b ) ]
unzip   :: [ ( a, b ) ] -> ( [ a ], [ b ] )
```


- Recall: The function `null` delivers `True` if a given list is empty (`[]`); `False` otherwise
- The function `null` is (must be!) defined using pattern matching...

```
null :: [ a ] -> Bool
null []          = True
null ( x : xs ) = False
```

- This is (almost) how `null` is defined in the Haskell prelude
- Alternatively, as there is no need to name the head and tail,

```
null :: [ a ] -> Bool
null []  = True
null any = False
```

- Recall that patterns are tested in order (from top to bottom)

- Recall: The function `head` selects the first element of a list, and `tail` selects the remaining portion

```
head :: [ a ] -> a
head ( x : xs ) = x

tail :: [ a ] -> [ a ]
tail ( x : xs ) = xs
```

$\boxed{?}$ What is `tail [ 1 ]` and what happens if we type `head []`?

- The function `length`
  returns the length of a list (i.e. the number of elements it contains):

```
Hugs.Base> length "brontosaurus"
12
Hugs.Base> length [ ( True, True, False ) ]
1
Hugs.Base> length []
0
```

- A recursive definition using pattern matching...

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


- The function `elem` determines whether a given element is a member of a given list:

```
Hugs.Base> elem 'c' "hatchet"
True
Hugs.Base> elem (1,2) [ (3,4), (5,6) ]
False
```

- One of many possible definitions using pattern matching...

```
elem :: Eq a => a -> [ a ] -> Bool
elem x []         = False
elem x ( y : ys ) = x == y || elem x ys
```


- The `!!` operator (sometimes pronounced "at") performs list *indexing* (the head is index 0):

```
Hugs.Base> [ 11, 22, 33 ] !! 1
22
Hugs.Base> "Tea" !! 0
'T'
Hugs.Base> "Tea" !! 5
Program error: Prelude.!!: index too large
Hugs.Base> "Error" !! -1
Program error: Prelude.!!: negative index
```


- Here is a recursive def-
  inition using pattern matching (not the definition in the prelude)...

```
infixl 9 !!
(!!) :: [ a ] -> Int -> a
[] !! n = error "Prelude.!!: index too large"
( x : xs ) !! n
  | n == 0 = x
  | n > 0  = xs !! ( n-1 )
  | n < 0  = error "Prelude.!!: negative index"
```

- Note the syntax for introducing new left- (`infixl`) and right- (`infixr`) associative operators with a given precedence (here 9)
- Operator names must be unique and built from the symbols

```
! # $ \% . + * @ | > < ~ - : ^ \ = / ? &
```

- The binary operator `++` (pronounced "concatenate" or "append") joins two lists of the same type to form a new list e.g.

```
Hugs.Base> [ 1, 2, 3 ] ++ [ 1, 5 ]
[ 1, 2, 3, 1, 5 ]
Hugs.Base> "" ++ "Rest"
"Rest"
Hugs.Base> [ head [ 1, 2, 3 ] ] ++ tail [ 2, 8 ]
[ 1, 8 ]
```

- The recursive definition...

```
infixr 5 ++
(++) :: [ a ] -> [ a ] -> [ a ]
[] ++ ys          = ys
( x : xs ) ++ ys = x : ( xs ++ ys )
```


- `take n xs` returns the first `n` elements of `xs`
- `drop n xs` returns the remainder of the list after the first `n` elements have been removed

```
Hugs.Base> take 4 "granted"
"gran"
Hugs.Base> drop 2 [ True, False, True ]
[True]
Hugs.Base> take 1 "away" ++ drop 1 "away"
"away"
Hugs.Base> drop 8 "letters"
""
```

? How would you define `take` and `drop`?

- `zip` takes two lists and forms a single list of pairs by combining the elements pairwise
- `unzip` does the opposite!
- Note: for `zip` if one list is longer than the other then the surplus elements are discarded

```
Hugs.Base> zip [ 5, 3 ] [ 4, 9, 8 ]
[(5,4),(3,9)]
Hugs.Base> zip "it" "up"
[('i','u'),('t','p')]
Hugs.Base> unzip [ ( "your", "jacket" ) ]
(["your"],["jacket"])
```

? How would you define `zip` and `unzip`?

More (non-prelude) Examples

- Example: Ordered insertion—the function `insert` takes an `Int` and an *ordered* list of `Int` and returns a new ordered list with the new `Int` in the right place

```
insert :: Int -> [ Int ] -> [ Int ]
-- Pre:  The given list is ordered
insert x []    = [ x ]
insert x ( y : ys )
   | x < y     = x : ( y : ys )
   | otherwise = y : ( insert x ys )
```

- For example: `insert 3 [ 1, 4, 9 ]` proceeds as follows:

```
-> 1 : ( insert 3 [ 4, 9 ] )
-> 1 : ( 3 : [ 4, 9 ] )
-> [ 1, 3, 4, 9 ]
```

- If we repeatedly insert (unsorted) items into a sorted list, the final list will also be sorted, hence:

```
isort :: [ Int ] -> [ Int ]
isort []          =  []
isort ( x : xs ) =  insert x ( isort xs )
```

- This 'algorithm' is called *(linear) insertion sort*

- An example:

```
isort [4, 9, 1]
-> insert 4 (isort [9, 1])
-> insert 4 (insert 9 (isort [1]) )
-> insert 4 (insert 9 (insert 1 (isort []) ) )
-> insert 4 (insert 9 (insert 1 []) )
-> insert 4 (insert 9 [1])
-> insert 4 [1, 9]
-> [1, 4, 9]
```

? How many calls to '`:`' are made on average to sort a list with n items?

- Example: Haskell's `splitAt`—this takes an `Int` index n and a list of type `[ a ]` and splits the list at position n
- How does Haskell implement it? A quick solution:

```
splitAt :: Int -> [ a ] -> ( [ a ], [ a ] )
-- Pre: n >= 0
splitAt n xs
  = ( take n xs, drop n xs )
```

- However, this traverses the first `n` elements of `xs` *twice*!
- We can avoid this but the resulting function is more complicated...


```
splitAt :: Int -> [ a ] -> ( [ a ], [ a ] )
-- Pre: n >= 0
splitAt n []
  = ( [], [] )
splitAt n ( x : xs )
  = if n == 0 then ( [], x : xs )
              else ( x : xs', xs'' )
  where
    ( xs', xs'' ) = splitAt ( n - 1 ) xs
```

? Exercise: Remove the precondition and trap the 'negative argument' error.

- Example: list merge–the function `merge` takes two ordered lists of `Int` and merges them to produce a single ordered list

```
merge :: [ Int ] -> [ Int ] -> [ Int ]
-- Pre: both argument lists are ordered
merge [] []
  = []
merge [] ( x : xs )
  = x : xs
merge ( x : xs ) []
  = x : xs
merge ( x : xs ) ( y : ys )
  = if x < y
    then x : merge xs ( y : ys )
    else y : merge ( x : xs ) ys
```


- Now, as if by magic...

```
msort :: [ Int ] -> [ Int ]
msort []
  = []
msort xs
  = merge ( msort xs' ) ( msort xs'' )
  where
    ( xs', xs'' ) = splitAt ( length xs `div` 2 ) xs
```

- This is called *merge sort*; it sorts a list of numbers into ascending order, like `isort`

$\boxed{?}$ This definition isn't quite right. What's wrong and how would you fix it?

$\boxed{?}$ Which is the best sorting function: `isort` or `msort`? Why?

Aside: Enumerated Types

- Enumerated types are special forms of *algebraic data types*—see later
- They introduce a new type and an associated set of elements, called *constructors*, of that type, e.g.

```
data Day =  Mon | Tue | Wed | Thu |
            Fri | Sat | Sun
```

- This says that `Day` is a new type and that objects of type `Day` may either be `Mon`, `Tue`, ..., `Sun`
- Constructor names must be unique within a program
- When Haskell spots a constructor it knows immediately its type, e.g. `Fri` is immediately recognisable as an object of type `Day`

- Important: type and constructor names must begin with a capital letter but are otherwise completely arbitrary
- Some more examples:

```
data Kerrrpowwwww = Plink | Plonk

data Switch = Off | On

data Color = Black | Blue    | Green  | Cyan
           | Red   | Magenta | Yellow | White
```

- It's up to us to choose type and constructor names that make sense to us

- Functions can be defined on objects of type `Day`, `Kerrrpowwww`, `Switch` etc. using pattern matching, e.g.

```
bothOn :: Switch -> Switch -> Bool
bothOn On  On  = True
bothOn On  Off = False
bothOn Off On  = False
bothOn Off Off = False
```

- Recall: patterns are tested in order (from top to bottom)
- The last three rules can thus optionally be replaced by a single rule, e.g.: `bothOn s s' = False`

- Note: we could use conditionals to test elements of an enumerated type, but it's much less elegant, e.g.

```
bothOn :: Switch -> Switch -> Bool
bothOn s 's' = s == On && s' == On
```

(Yuk!)

- Secret: you can think of constructors as numeric tags, e.g. Hugs encodes `Off` and `On` using the integers 0, and 1; similarly `Kerrrpowwww` but the type system ensures there is no confusion between, for example, 0 for `Off` and 0 for `Plink`
- Note that the type `Bool` is just an enumerated type! Indeed, the Haskell prelude includes exactly this:

```
data Bool = False | True
```


- Note that Haskell's boolean functions can be straightforwardly defined using pattern matching, e.g.

```
not :: Bool -> Bool
not False = True
not True  = False

infixr 3 &&
False && b = False
True  && b = b

etc.
```

$\boxed{?}$ What if we wrote four equations: `True && True = True`, `True && False = False`, `False && True = False`, `False && False = False`. Is this the same as the above?

Higher-order List Processing

- In Haskell functions can be passed as parameters to other functions
- Functions that take one or more other functions as parameters are called *higher-order* functions
- Some useful examples...

```
map     :: ( a -> b ) -> [ a ] -> [ b ]
filter  :: ( a -> Bool ) -> [ a ] -> [ a ]
zipWith :: ( a -> b -> c ) -> [ a ] -> [ b ] -> [ c ]
foldr1  :: ( a -> a -> a ) -> [ a ] -> a
foldr   :: ( a -> b -> b ) -> b -> [ a ] -> b
```

- Note: the first argument of each of these is a *function*, hence the type

- The function `map` applies a given function (passed as a parameter) to every element of a given list, e.g.

```
Hugs.Base> map succ [ 1, 2, 3, 4 ]
[2,3,4,5]
Hugs.Base> map head [ "Random", "Access", "Memory" ]
"RAM"
```

- Note that the expression `map f xs` is equivalent to the list comprehension `[ f x | x <- xs ]` so one way to define `map` would be

```
map :: ( a -> b ) -> [ a ] -> [ b ]
map f xs = [ f x | x <- xs ]
```


- We can define `map` recursively using pattern matching as well...

```
map :: ( a -> b ) -> [ a ] -> [ b ]
map f []          = []
map f ( x : xs ) = f x : map f xs
```

- Let's see how it works with an example:

```
map not [ True, False, False ]
-> not True : map not [ False, False ]
-> not True : not False : map not [ False ]
-> not True : not False : not False : map not []
-> not True : not False : not False : []
-> [ False, True, True ]
```


- The function `filter` filters out elements of a list using a given *predicate* (a function that returns a `Bool`)
- The predicate is applied to each element of a given list; if the result is `False` the element is excluded from the result, e.g.

```
Hugs.Base> filter even [ 1 .. 10 ]
[2,4,6,8,10]
Hugs.Base> let f x = x > 6 in filter f [4,7,6,9,1]
[7,9]
Hugs.Base> let f x = x /= 's' in filter f "scares"
"care"
```

- The expression `filter f xs` is equivalent to the list comprehension `[ x | x <- xs, f x ]`

? How would you define `filter` recursively?

- The function `zipWith` applies a given binary function pairwise to the elements of two given lists, e.g.

```
Hugs.Base> zipWith (+) [ 1, 2, 3 ] [ 9, 5, 8 ]
[10,7,11]
Hugs.Base> zipWith elem "abp" [ "dog", "cat", "pig" ]
[False,False,True]
Hugs.Base> zipWith max [(2,1),(4,9)] [(1,1),(8,5)]
[(2,1),(8,5)]
```

- Recall: `(op)` is the prefix version of `op`; also, `elem e xs` is `True` iff `e` is an element of list `xs`
- The expression `zipWith f xs ys` is equivalent to the list comprehension `[ f x y | ( x, y ) <- zip xs ys ]`

? How would you define `zipWith` recursively?

- Example: Mastermind—given two sequences of coloured pegs (guess and secret) you score one *black* stick for each peg that has the right colour in the right place
- If the pegs are represented by `Int`s there are many ways to compute the black stick score, e.g. (assuming colours are characters)...


```
blacks :: String -> String -> Int
-- Pre: length g = length s
blacks g s
  = length [ x | (x,y) <- zip g s, x == y ]

-- Recall: sumInts sums the elements of a list of Ints
blacks g s
  = sumInts [ 1 | (x,y) <- zip g s, x == y ]

blacks g s =
  = length ( filter (id) ( zipWith (==) g s ) )
```

$\boxed{?}$ How does the third solution work?

$\boxed{?}$ How would you score the white pegs?!

- The function `foldr1` 'inserts' a given binary operator 'in between' each list element, i.e.

$$foldr1\ \otimes\ [x_1, x_2, ..., x_n] \longrightarrow x_1 \otimes x_2 \otimes ... \otimes x_n$$

- The resulting expression is bracketed from the *right*, i.e.

$$x_1 \otimes (x_2 \otimes (...(x_{n-1} \otimes x_n)...)))$$

- A variation called `foldl1` brackets from the left, e.g.

```
Hugs.Base> foldr1 (+) [ 3, 5, 7, -3, 9 ]
21
Hugs.Base> foldr1 (-) [ 4, 3, 6 ]
7
Hugs.Base> foldl1 (-) [ 4, 3, 6 ]
-5
```


- Note: `foldr1` is actually a special case of a more general folding function call `foldr` (the same goes for `foldl1`)
- `foldr` allows a *right unit* (here called b) to be specified

$$foldr\ f\ b\ [x_1, x_2, ..., x_n] \longrightarrow f\ x_1 (f\ x_2 (...(f\ x_n\ b)...))$$

- This subtly changes the type to

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

- For example:

```
Hugs.Base> foldr (+) 0 [ 3, 5, 7, -3, 9 ]
21
Hugs.Base> let f x y = y in foldr f 99 [ 3, 2, 6 ]
99
Hugs.Base> foldr (:) [] [ 3, 5, 7, -3, 9 ]
[3,5,7,-3,9]
```

- Example: three more functions from the prelude...

```
dropWhile :: ( a -> Bool ) -> [ a ] -> [ a ]
dropWhile p [] = []
dropWhile p ( x : xs )
  = if p x then dropWhile p xs else x : xs

takeWhile :: ( a -> Bool ) -> [ a ] -> [ a ]
takeWhile p [] = []
takeWhile p ( x : xs )
  if p x then x : takeWhile p xs else []

iterate :: ( a -> a ) -> a -> [ a ]
iterate f x = x : iterate f ( f x )
```


- For example,

```
Hugs.Base> take 10 ( iterate succ 0 )
[0,1,2,3,4,5,6,7,8,9]
Hugs.Base> take 6 ( iterate tail "suffix" )
["suffix","uffix","ffix","fix","ix","x"]
Hugs.Base> takeWhile even [ 2, 4, 7, 6 ]
[2,4]
Hugs.Base> dropWhile isSpace "     Begin"
"Begin"
```

- Note that lazy evaluation is essential for evaluating expressions involving `iterate`

Binary Operator Extensions

- Some binary operators have corresponding generalisations over lists, for example:

Operator	Generalisation
`+`	`sum :: Num a => [ a ] -> a`
`*`	`product :: Num a => [ a ] -> a`
`&&`	`and :: [ Bool ] -> Bool`
`||`	`or :: [ Bool ] -> Bool`
`++`	`concat :: [ [ a ] ] -> [ a ]`
`max`	`maximum :: Ord a => [a] -> a`
`min`	`minimum :: Ord a => [a] -> a`

- Note: see later for a proper explanation of the types

- Examples...

```
Hugs.Base> sum [1..6]
21
Hugs.Base> product [ 2, 4, 1, 6 ]
48
Hugs.Base> and [ True, False, True ]
False
Hugs.Base> or [ x < 3 | x <- [ 5, 4, 8, 1, 9 ] ]
True
Hugs.Base> concat [ "Three ", "small ", "lists" ]
"Three small lists"
Hugs.Base> maximum [ 1, 4, 3, 1, 9 ]
9
```

- Note: these operators can be defined recursively using pattern matching, e.g.

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

and :: [ Bool ] -> Bool
and []          = True
and ( b : bs ) = b && and bs
```

- However, in each case all we're doing is 'inserting' a standard binary operator in between each list element, with or without a right element
- But this is just what the family of `fold` functions do!

- So, alternatively:

```
sum xs      = foldl (+) 0 xs
product xs  = foldl (*) 1 xs
and xs      = foldr (&&) True xs
or xs       = foldr (||) False xs
concat xs   = foldr (++) [] xs
maximum xs  = foldl1 max xs
minimum xs  = foldl1 min xs
```

- Note: `maximum []` and `minimum []` are not defined - hence the use of `foldl1`. $\boxed{?}$ Could we use `foldr1` instead?

Currying and Partial Application

- Functions can also return other functions as their result (another type of higher-order function)
- Q: How can we evaluate something and end up with a new function? A: Partial application...
- Consider the function

```
plus :: Int -> Int -> Int
plus x y = x + y
```

- Why do we write `Int -> Int -> Int` and why is function application expressed by juxtaposition?
- The answer is that `plus` introduces *two* single-argument functions:

1. `plus` is really a single-argument function of type `Int -> ( Int -> Int )`
2. If `a :: Int` then `plus a` is a *function* of type `Int -> Int`

- So, `plus 4` is a perfectly meaningful expression—it is the function which adds 4 to things!
- This suggests we can map partial applications over lists; let's try:

```
Hugs.Base> map ( plus 4 ) [ 1, 3, 8 ]
[ 5, 7, 12 ]
Hugs.Base> map ( elem 'e' ) [ "No", "No again", "Yes" ]
[False,False,True]
```

- An application which only partially completes the arguments of a function `f` is called a *partial application* of `f`

- The idea of treating all multi-argument functions "one argument at a time" is called *currying* after the mathematician **HASKELL B. Curry**!!
- Partial applications of operators are called *sections* and Haskell has some special notation to help. For example:

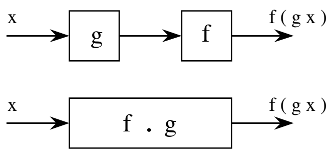
`(1/)`	is the 'reciprocal'	function
`(/2)`	is the 'halving'	function
`(^3)`	is the 'cubing'	function
`(+1)`	is the 'successor'	function
`(!!0)`	is the 'head'	function
`(==0)`	is the 'is-zero'	function


```
Hugs.Base> map (== 0) [ 4, 0, 8, 0 ]
[False,True,False,True]
Hugs.Base> map (^2) [ 1..4 ]
[1,4,9,16]
Hugs.Base> map (!! 2) [ "one", "two", "three" ]
"eor"
Hugs.Base> takeWhile (<20) ( iterate (+3) 1 )
[1,4,7,10,13,16,19]
Hugs.Base> filter (/=0) ( map (`mod` 2) [1..10] )
[1,1,1,1,1]
```


- So functions really are 'first-class' citizens in Haskell! Indeed, even function composition can be expressed as a higher-order function:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = h  where  h x = f ( g x )
```

- `( f . g )` is the composition of functions `f` and `g`. Diagrammatically:



- Example: here are two equivalent definitions of functions `notNull` and `allZero`

```
notNull :: [ a ] -> Bool
notNull xs = not ( null xs )

notNull xs = ( not . null ) xs


allZero :: [ Int ] -> Bool
allZero xs = and ( map (==0) xs )

allZero xs = ( and . map (==0) ) xs
```

- Here is an alternative definition of `newtonSqrt`:

```
newtonSqrt :: Float -> Float
newtonSqrt x
  = ( head . dropWhile badAppx . iterate next ) ( x/2 )
  where
    next a    = ( a + x / a ) / 2
    badAppx a = abs ( x - a^2 ) > 0.00001
```


Extensionality

- A useful rule for simplifying some definitions is the *extensionality* rule from mathematics:

$$\text{if } \forall x,\ f\ x = g\ x \text{ then } f = g$$

- This means, for example, that in our `notNull` and `allZero` functions we can instead *cancel* the `xs` and write

```
notNull = not . null

allZero = and . map (==0)
```


- Similarly for some of our earlier examples, e.g.

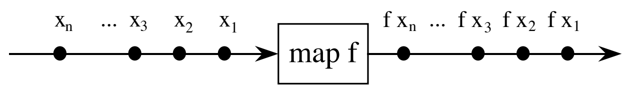
```
sum xs    = foldr (+)  0 xs
and xs    = foldr (&&) True xs
concat xs = foldr (++) [] xs
```

can be written

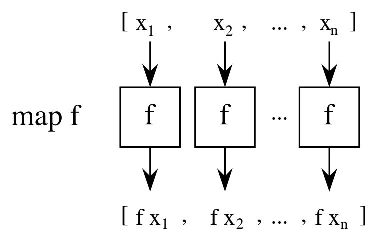
```
sum    = foldr (+) 0
and    = foldr (&&) True
concat = foldr (++) []
```

- This exploits the fact that function application associates to the left, i.e. $f\ x\ y\ z \equiv (\ (\ f\ x\ )\ y\ )\ z$

- Now for some fun! There are several ways to think of higher-order functions like `map`, and diagrams often help. For example: for example:

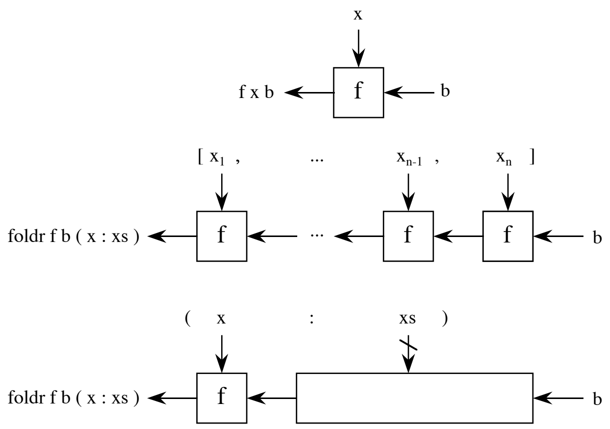


a. map as a Pipeline



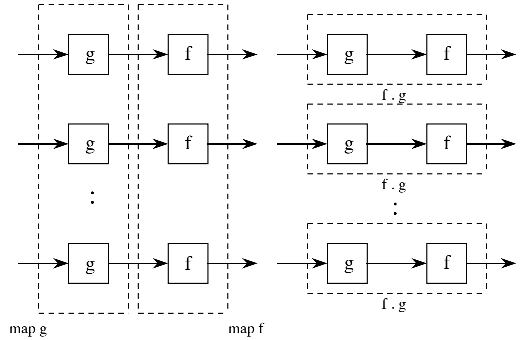
b. Parallel map

- As with `map`, we can also think of `fold` functions diagramatically; this also helps to explain the recursive definition, e.g. for `foldr`:



= f x (foldr f b xs)

- We can compose higher-order functions to form complex interacting processes. For example, consider a pipeline in which we want each element of an input stream (`[x1,..,xn]`) to be processed by `g` then by `f`



- Both solutions will do, so the two diagrams must specify the same function. Hence, we establish:

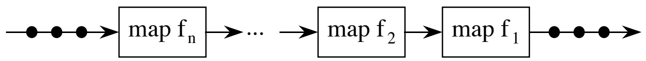
$$map\ f\ .\ map\ g \quad = \quad map\ (f\ .\ g\ )$$

- Note: we can prove this formally using extensionality, i.e. by applying the left- and right-hand sides to the same argument:

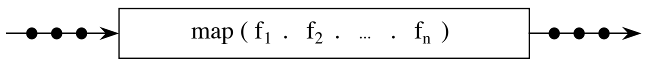
$$\begin{aligned}
(map\ f\ .\ map\ g)\ [\ x_1, ...x_n\ ] &= map\ f\ (map\ g\ [\ x_1, ..., x_n\ ]) \\
&= map\ f\ [\ g\ x_1, ..., g\ x_n\ ] \\
&= [\ f(g\ x_1), ..., f(g\ x_n)\ ] \\
map\ (f\ .\ g\ )[\ x_1, ..., x_n\ ] &= [\ (f\ .\ g)x_1, ..., (f\ .\ g)x_n\ ] \\
&= [\ f(g\ x_1), ..., f(g\ x_n)\ ]
\end{aligned}$$

for $n > 0$. (The case for [ ] is trivial)

- Note that this generalises to many pipeline stages. Let's draw `map` another way, this time treating it as a pipeline:



foldr (.) id (map map [f_1, ..., f_n])

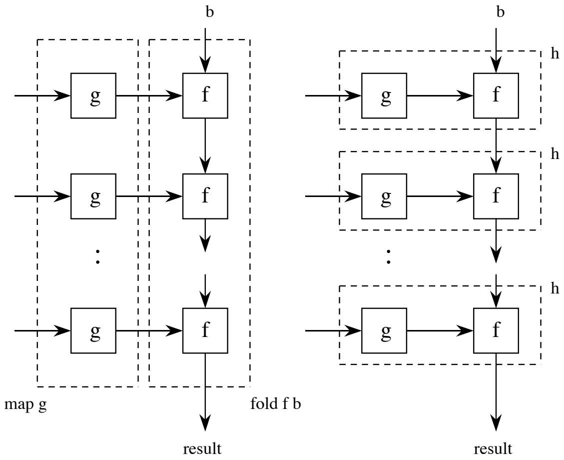


map (foldr (.) id [f_1, ..., f_n])

- From which we see that

$$(foldr\ (.)\ id)\ .\ map\ map \quad = \quad map\ .\ (\ foldr\ (.)\ id\ )$$

- As a final example, let's make it a little harder:



- The one on the left corresponds to:

```
foldr f b . map g
```

and the one on the right to:

```
foldr h b  where  h x y = f ( g x ) y
```

- Notice, however, that by extensionality

```
h x y = f ( g x ) y
=>  h x = f ( g x ) = ( f . g ) x
=>  h = f . g
```

Hence the rather unobvious equivalence:

$$foldr\ f\ b\ .\ map\ g \quad = \quad foldr\ (\ f\ .\ g\ )\ b$$

- You'll see these ideas again in course 318 (Custom Computing) which uses functional languages to design hardware systems built from *field-programmable gate arrays (FPGAs)*

Part IV: Algebraic Data Types

- We have already seen *enumerated* types, for example:

```
data Bool = False | True

data Color = Black | Blue    | Green  | Cyan
           | Red   | Magenta | Yellow | White
```

- In general, constructors can also take arguments and both they (and the associated type) can be polymorphic

- Example: a "hand-made" list type:

```
data List a = Nil | Cons a ( List a )
```

1. `Nil` is a constructor of type `List a` (i.e. `Nil ::  List a`)
2. `Cons` is a constructor of type `a -> List a -> List a`
3. Constructors are thus like ordinary functions except they have no rules
4. Constructors are defined implicitly when they appear in a `data` definition

- The type `List a` is isomorphic to Haskell's list type `[ a ]` – indeed, Haskell's prelude essentially has this:

```
data [ a ] = [] | a : [ a ]
```

although this is *not* legal syntax!

- If we stick with our own definition of lists (`List a`) we'll need to use `Nil` and `Cons` instead of `[]` and ':' e.g.

```
Nil
Cons 6 Nil
Cons "this" ( Cons "that" Nil )
```

generate objects of type `List Int` and `List String` resp.

- We can also pattern match on terms involving `Nil` and `Cons` e.g.

```
myLength :: List a -> Int
myLength Nil = 0
myLength ( Cons x xs ) = 1 + myLength xs

mySum :: List Int -> Int
mySum Nil = 0
mySum ( Cons x xs ) = x + mySum xs
```

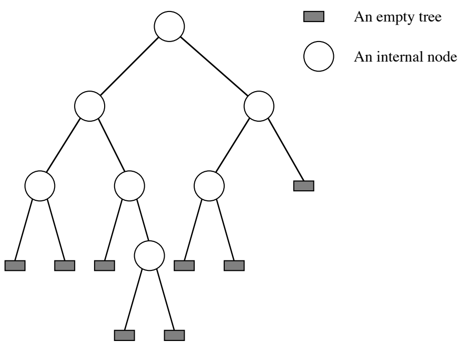

- Just to prove a point about names...

```
data SillyList a = Metriacanthosaurus
                 | Hipsilophodon a ( SillyList a )

sillyLength :: SillyList a -> Int
sillyLength Metriacanthosaurus
  = 0
sillyLength ( Hipsilophodon x xs )
  = 1 + sillyLength xs
```


Trees

- Trees are powerful generalisations of lists and have a two-dimensional branching structure
- Here is the general shape of a *binary* tree:



- Objects of some given type are located at each node

- We can describe the structure of trees using an algebraic data type
- Let's call the constructor for an empty tree `Empty` and that for an internal node `Node`
- We'll allow any type of object to sit in the nodes, so we'll make our trees polymorphic:

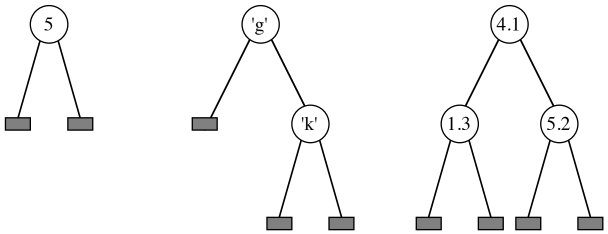
```
data Tree a = Empty | Node ( Tree a ) a ( Tree a )
```

- Note we could rearrange the arguments of `Node`, e.g.

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

- It doesn't matter so long as we are consistent; we'll use the former

- For example



(a) (b) (c)

- (a) is a `Tree Int`, (b) is a `Tree Char` and (c) is a `Tree Float`
- We can write Haskell expressions that represent these, e.g. (b) corresponds to `Node Empty 'g' ( Node Empty 'k' Empty )`
  ? What about (a) and (c)?

- As with lists we can write functions on `Trees` using pattern matching
- There are two types of tree, hence two types of pattern to consider
- Example: `treeSize` for computing the number of nodes in a tree

```
treeSize :: Tree a -> Int
treeSize Empty           = 0
treeSize ( Node l x r ) = 1 + treeSize l + treeSize r
```

- Compare this with `length` for lists; here `treeSize` has *two* "sub-trees" to explore beneath each `Node` hence *two* recursive calls to `treeSize`

- Another example: `flatten` which will reduce a `Tree a` to a list of type `[ a ]` by performing an *in-order* traversal of the tree
- In-order traversal visits the nodes from left to right

```
flatten :: Tree a -> [ a ]
flatten Empty
  = [ ]
flatten ( Node t1 x t2 )
  = flatten t1  ++  ( x : flatten t2 )
```

- Note that the flattened version of `t1` is the leftmost argument of `++` and therefore will appear *first* in the resulting list; hence the left-to-right order

? How many calls to '`:`' are required to flatten a prefectly balanced tree containing $n = 2^k - 1$ elements? How would you redefine `flatten` so that exactly *one* '`:`' is required for each element?

- Example: a function to insert a number into an ordered tree
- An ordered tree satisfies the property that for every node:
  - Every element of the left subtree is smaller than (or equal to) the element at the node
  - The element at the node is smaller than every element in the right subtree

```
insert :: Int -> Tree Int -> Tree Int
insert n Empty
  = Node Empty n Empty
insert n ( Node t1 x t2 )
  | n <= x    = Node ( insert n t1 ) x t2
  | otherwise = Node t1 x ( insert n t2 )
```

? Note that `insert` as defined is *not* polymorphic. Why?

- Example: a function to construct an ordered tree from an unordered list of numbers:

```
build :: [ Int ] -> Tree Int
build = foldr insert Empty
```

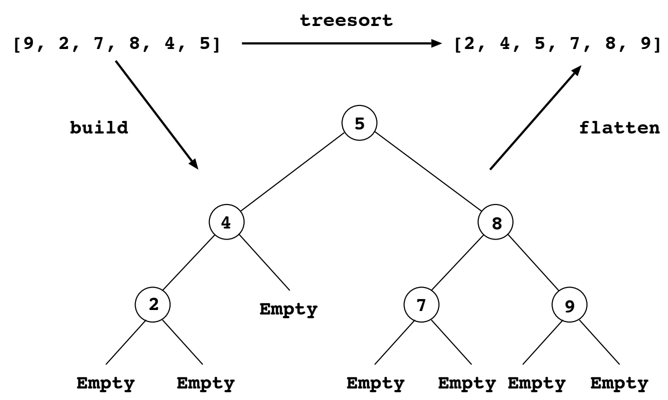
- Hence, yet another program for sorting a list of numbers:

```
treeSort :: [ Int ] -> [ Int ]
treeSort = flatten . build
```

? On average, how many nodes have to be visited for each insertion?

? If you use the optimised version of `flatten` (see above) how many constructor calls (list and tree constructors) are required on average to sort a list of n elements?

- The composition of `build` and `flatten` can be seen clearly with a pretty picture:



Optional extra (needed for Exercise Sheet 4)

- We can also define higher-order functions that model recursive patterns in more complex data structures
- Let's define a slightly different type of tree to that earlier and build some equivalents to `map` and `fold` for lists
- Our new trees will hold values only at the leaves

```
data Tree t = Empty  |
              Leaf t |
              Node ( Tree t ) ( Tree t )
              deriving ( Show )
```


- The equivalent of the `map` function on lists will transform the items at the leaves, but preserves the tree's shape:

```
mapt :: ( a -> b ) -> Tree a -> Tree b
mapt f Empty
  = Empty
mapt f ( Leaf x )
  = Leaf ( f x )
mapt f ( Node t1 t2 )
  = Node ( mapt f t1 ) ( mapt f t2 )
```

- For example, for a `Tree Num` we can double to each element stored by applying the function `mapt ( * 2 )` to the tree

- The equivalent of `fold` reduces a tree to a new value, but there are several variants, e.g.

```
foldt :: ( a -> b -> b ) -> b -> Tree a -> b
foldt f b Empty        = b
foldt f b ( Leaf x )   = f x b
foldt f b ( Node l r ) = foldt f ( foldt f b l ) r
```

- This accumulates the reduced tree from left to right, hence:

`foldt max 0`	Find the maximum
`foldt (+) 0`	Sum all elements
`foldt (:)  []`	Flatten from right to left

Part V: Classes

- In contrast to polymorphic functions such as `length`, some functions, e.g. `==`, are *overloaded*:
  - they can be used at more than one type
  - their definitions are different at different types
- The collection of types over which a function is defined is called a *class*
- The set of types over which `==` is defined is called the *equality class*, `Eq`
- We say that `==` is a *member function* of `Eq`
- Note that `/=` is also a member of `Eq`
- The Haskell equality class is defined by:


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

- Note the *default* definition of `/=`; as we add new types to the equality class, `/=` will de defined automatically (in terms of `==`)
- The member *types* of a class (the types that `t` can be above) are called *instances* of that class; for `Eq` the instances include `Int, Float, Bool, Char`
- It is this which enables use to write things like
  `True == True || 'a' == 'b' && 13 /= 7`

Extending a Class

- Suppose we wish to check whether two `Switch` values are equal. We could define a new function, e.g.

```
eqSwitch :: Switch -> Switch -> Bool
eqSwitch On On   = True
eqSwitch Off Off = True
eqSwitch s1 s2   = False
```

- This is fine, but it would be much more convenient if we could use `==` instead, as in `Off == On` for example
- The problem is that the type `Switch` is *not* by default a member of `Eq`!
- However, we can add it in one of two ways:

1 Explicitly by adding a definition of '`==`' on values of type `Switch`:

```
instance Eq Switch where
  On  == On   =  True
  Off == Off  =  True
  s1  == s2   =  False
```

(Note that `/=` is defined in terms of `==` by default in the class definition but we could *override* it here if we wanted)

2 Implicitly using the keyword `deriving` in the `data` definition:

```
data Switch = On | Off
              deriving ( Eq )
```

- The use of `deriving` saves us a lot of work—the system builds the definition of `==` over `Switch` values automatically
- To really appreciate the benefits, try writing `==` for type `Day`, defined earlier!

Puzzle: Given the `Eq` class definition:

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

and the following data type and instance declaration:

```
data D = C Int

instance Eq D
```

(i.e. `D` uses `Eq`'s default definitions for `==` and `/=`), what happens if we try to compute `C 1 == C 2`?

Contexts

- Some function types need to be *restricted* to reflect the operations that they perform on their arguments
- Here is a valid definition

```
equals :: Int -> Int -> Bool
equals x y = x == y
```

- However, if we try to make this polymorphic, as in

```
equals :: t -> t -> Bool
equals x y = x == y
```

we get an error

- A type variable `t` in a type means (literally) "for all `t`", but `equals` will only work if values of type `t` are *comparable*

- To make the type of `equals` as *general* as possible, we need to give `t` a *context* thus

```
equals :: Eq t => t -> t -> Bool
equals x y = x == y
```

- `Eq t => ...` now means "any `t` that is a member of `Eq`" rather than "for *all* `t`"
- Example: Haskell provides a built-in function `elem` for testing membership of a list, e.g.

```
elem 1 [ 2, 4, 9 ]     -> False
elem 'a' "Harry"       -> True
elem True []           -> False
```

- So, what is the type of `elem`?
- The basic type structure is clearly of the form `a -> [a] -> Bool`
- However, the list elements (the things of type `a`) *must* be comparable, i.e. `a` must be an instance of `Eq`
- In the Haskell standard prelude, we find:

```
elem :: Eq a => a -> [ a ] -> Bool
```

Q: What is the most general type of the following function?

```
getAll a [] = []
getAll a ( ( x, y ) : rest )
  = if a == x
    then y : getAll a rest
    else getAll a rest
```


Derived Classes

- Some classes may restrict their instance types to belong to certain other classes
- The simplest example is another built-in class called `Ord` representing the *ordered* types
- For a type to be a member of `Ord` it must also be a member of the *superclass* `Eq`
- Given the data type:

```
data Ordering = LT | EQ | GT
```

`Ord` can be defined using a context thus (see over):


```
class (Eq a) => Ord a where
  compare              :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a
  compare x y
    | x == y    =  EQ
    | x <= y    =  LT
    | otherwise =  GT
  x <= y = compare x y /= GT
  x <  y = compare x y == LT
  x >= y = compare x y /= LT
  x >  y = compare x y == GT
```

- We say that `Ord` *inherits* the operations of `Eq`

- Note that we can only compute `x <= y` if we can also compute `x == y`
- The basic types `Int, Float, Bool, Char` are all instances of `Ord`
- This enables us to write, e.g. `4 <= 9`, `'d' > 't'`, `max True False`
- If necessary, we can add new types to `Ord` in the same way that we added new types to `Eq`, for example

```
data Day =  Mon | Tue | Wed | Thu |
            Fri | Sat | Sun
            deriving ( Eq, Ord )
```

- Note that we cannot derive `Ord` without `Eq`; we must list them both

- The automatically-generated definitions of `<, <= , >, ...` assume the constructors to be ordered as they are written
- Thus

```
More> Tue <  Mon
False
More> Thu >= Mon
True
More> Sun <= Sun
True
More> Fri == Sat
False
```


- Recursive data types can also be added to classes `Eq` and `Ord`:

```
data Stack a = Base | Above a ( Stack a )
                deriving ( Eq, Ord )
```

- We can compare two `Stack a` values *provided* that `a` is also an instance of `Ord`, e.g.

```
More> Above 8 Base   > Above 7 Base
True
More> Above Mon Base >= Base
True
More> Above False ( Above True Base ) < Base
False
```

- However, `Above Off Base > Base` is an error because the type `Switch` is not an instance of `Ord`

- Another important class is called `Show` which includes a function for converting an object of an instance type into a string (i.e. a `String`)
- This enables Haskell to print the result of an arbitrary expression evaluation
- For example, without making `Day` an instance of `Show` the Haskell system cannot "display" values of type `Day`, e.g.

```
Main> Mon

ERROR: Cannot find "show" function for:
*** expression : Mon
*** of type    : Day
```


- Let's fix it:

```
data Day =  Mon | Tue | Wed | Thu |
            Fri | Sat | Sun
            deriving ( Eq, Ord, Show )
```

- Now (if `Stack` also derives `Show`)...

```
Main> Mon
Mon
Main> Above Fri Base
Above Fri Base
```

- The built-in function `show ::  Show a => a -> String` uses the member functions of `Show` to convert objects into strings

- Alternatively, we might want to display values of type `Day` differently:

```
instance Show Day where
  show Mon = "Monday"
  show Tue = "Tuesday"
  show Wed = "Wednesday"
  show Thu = "Thursday"
  show Fri = "Friday"
  show Sat = "Saturday"
  show Sun = "Sunday"
```

- For example,

```
Main> ( Tue, Wed )
(Tuesday,Wednesday)
```


Multiple Constraints

- Contexts can include an arbitrary number of constraints, for example

```
showSmaller x y = if x < y then show x else show y
```

- Both `x` and `y` must be comparable by `<` and valid arguments to `show`, i.e. instances of *both* `Ord` and `Show`

```
Hugs.Base> :t showSmaller
showSmaller :: ( Ord a, Show a ) => a -> a -> String
```


- Finally, note that multiple constraints can occur in instance declarations
- For example, the pair type `( t, u )` is already defined to be an instance of `Eq`
- For two pairs to be comparable using `==` their components must also be comparable
- Hence this in Haskell's standard prelude:

```
instance ( Eq t, Eq u ) => Eq ( t, u ) where
  ( a, b ) == ( c, d )  =  if a == c then b == d
                                     else False
```

- As an exercise, look up the details of class `Eval` and work out how the `Num` class works