x = 1

let x = 1 in ...

x(1).

!x(1)

x.set(1)

**Programming Paradigms and Formal Semantics**

Introduction to Haskell
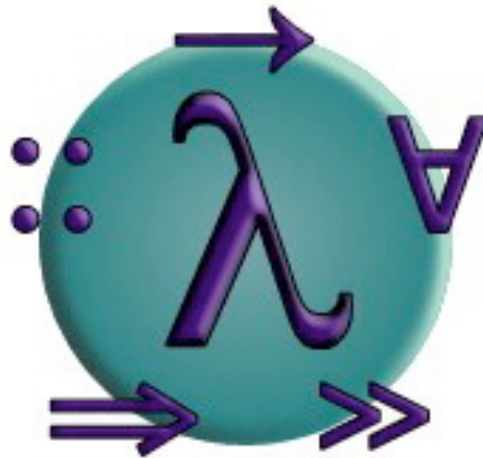
Ralf Lämmel

Programming in Haskell

Graham Hutton, *University of Nottingham*
Cambridge University Press, 2007

A weekly series of freely available <u>video lectures</u> on the book is being given by Erik Meijer on Microsoft's Channel 9 starting in October 2009. These lectures are proving amazingly popular. Pick up a copy of the book and join in the fun with Erik's great lectures!
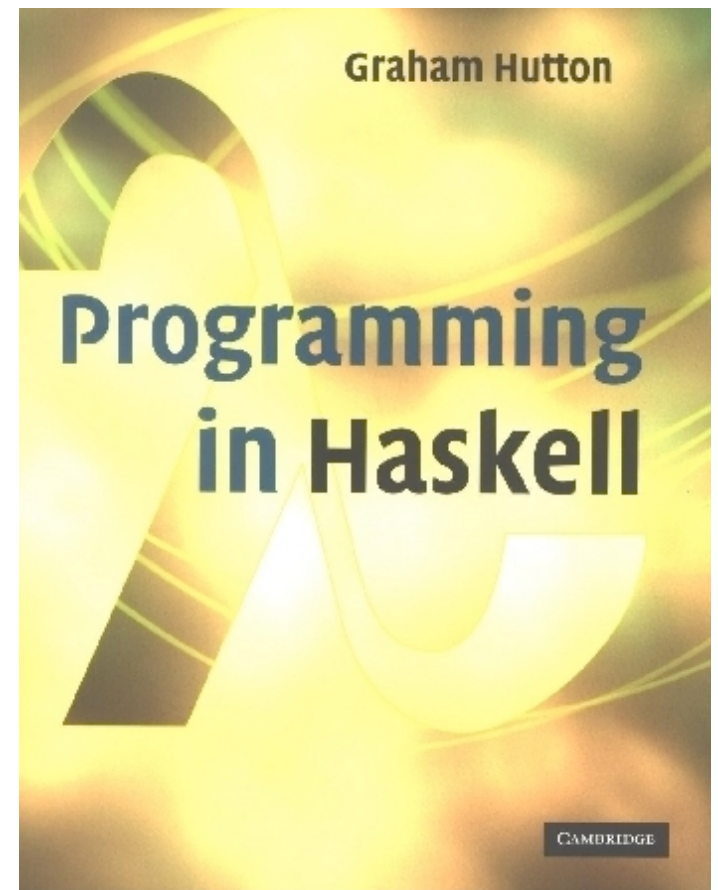
Acknowledgement:
Hutton's slides for his book are used
in this lecture on
introducing Haskell
(modulo a few adaptations).

What is a Functional Language?

3

What is a Functional Language?

Opinions differ, and it is difficult to give a precise definition, but generally speaking:

- Functional programming is <u>style</u> of programming in which the basic method of computation is the application of functions to arguments;

- A functional language is one that <u>supports</u> and <u>encourages</u> the functional style.

Example

Summing the integers 1 to 10 in Java:

total = 0;

for (i = 1; i ≤ 10; ++i)

total = total+i;

The computation method is <u>variable assignment</u>.

Example

Summing the integers 1 to 10 in Haskell:

$$\text{sum } [1..10]$$

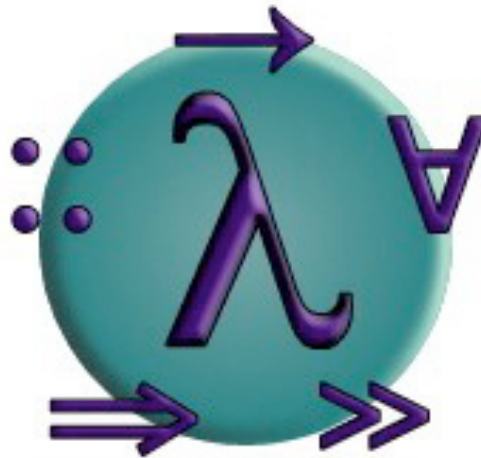The computation method is <u>function application</u>.

A Taste of Haskell

$$f\ [] \quad = []$$

$$f\ (x{:}xs) = f\ ys\ \texttt{++}\ [x]\ \texttt{++}\ f\ zs$$

where

$$ys = [a \mid a \leftarrow xs,\ a \leq x]$$

$$zs = [b \mid b \leftarrow xs,\ b > x]$$

?

Historical Background

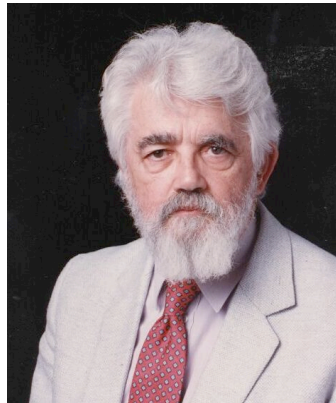# Historical Background

1930s:

Alonzo Church develops the <u>lambda calculus</u>, a simple but powerful theory of functions.

Historical Background

1950s:

John McCarthy develops <u>Lisp</u>, the first functional language, with some influences from the lambda calculus, but retaining variable assignments.

Historical Background

1960s:

Peter Landin develops <u>ISWIM</u>, the first pure functional language, based strongly on the lambda calculus, with no assignments.

Historical Background

1970s:

John Backus develops <u>FP</u>, a functional language that emphasizes higher-order functions and reasoning about programs.

Historical Background

1970s:

Robin Milner and others develop <u>ML</u>, the first modern functional language, which introduced type inference and polymorphic types.

Historical Background

1970s - 1980s:

David Turner develops a number of lazy functional languages, culminating in the Miranda system.

14

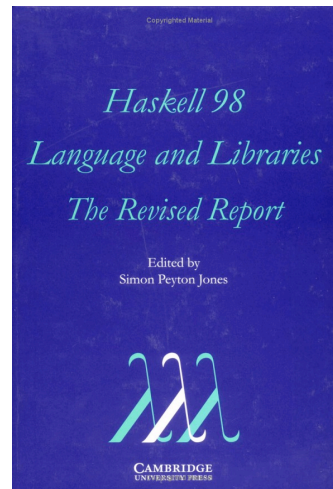# Historical Background

1987:

An international committee of researchers initiates the development of <u>Haskell</u>, a standard lazy functional language.

Historical Background
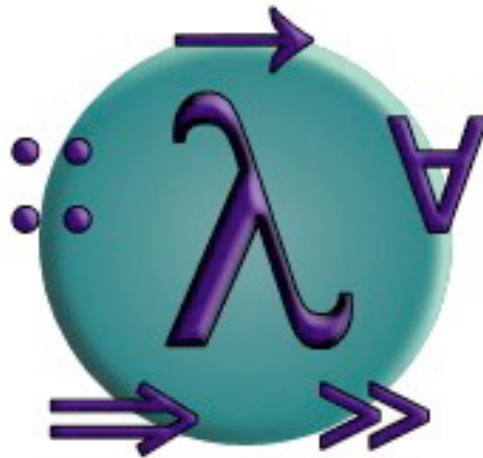
2003:

The committee publishes the <u>Haskell 98</u> report, defining a stable version of the language.

16

First Steps in Haskell

Haskell systems

★http://www.haskell.org/

★Major option: GHC

　http://haskell.org/ghc/download.html

18

Starting Haskell

Use command line.
Start the Haskell shell.

```
$ ghci
GHCi, version 6.10.4: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

The > prompt means that the Haskell system is ready to evaluate an expression.

For example:

```
> 2+3*4
14


> (2+3)*4
20


> sqrt (3^2 + 4^2)
5.0
```

The Standard Prelude

The library file <u>Prelude.hs</u> provides a large number of standard functions. In addition to the familiar numeric functions such as + and *, the library also provides many useful functions on <u>lists</u>.

Select the first element of a list:

```
> head [1,2,3,4,5]
1
```

Remove the first element from a list:

> ```
> tail [1,2,3,4,5]
[2,3,4,5]
> ```

Select the nth element of a list:

> ```
> [1,2,3,4,5] !! 2
3
> ```

Select the first n elements of a list:

> ```
> take 3 [1,2,3,4,5]
[1,2,3]
> ```

Remove the first n elements from a list:

> drop 3 [1,2,3,4,5]
[4,5]

Calculate the length of a list:

> length [1,2,3,4,5]
5

Calculate the sum of a list of numbers:

> sum [1,2,3,4,5]
15

Calculate the product of a list of numbers:

```
> product [1,2,3,4,5]
120
```

Append two lists:

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```
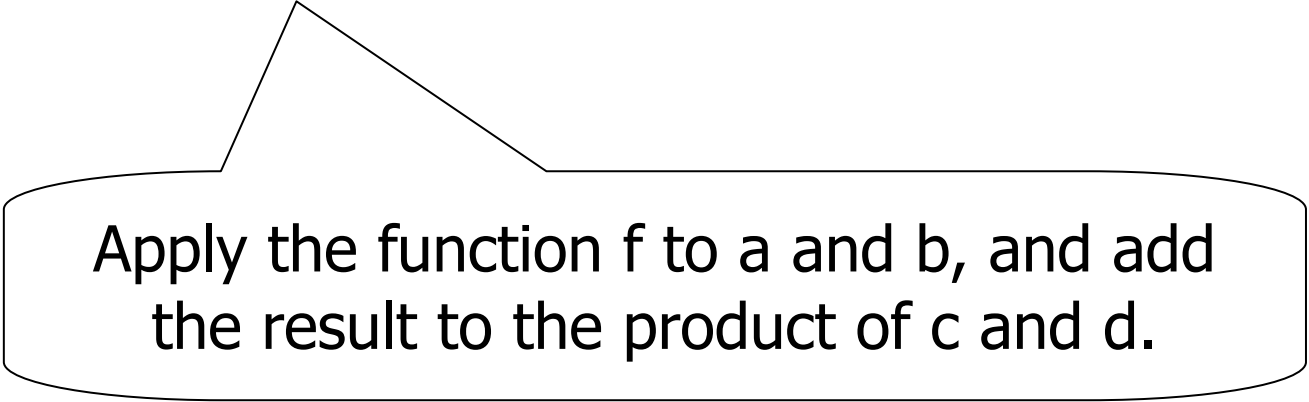
Reverse a list:

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

Function Application

In <u>mathematics</u>, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.

f(a,b) + c d

Apply the function f to a and b, and add the result to the product of c and d.

25

In <u>Haskell</u>, function application is denoted using space, and multiplication is denoted using *.

f a b + c*d

As previously, but in Haskell syntax.

Moreover, function application is assumed to have <u>higher priority</u> than all other operators.

f a + b

Means (f a) + b, rather than f (a + b).

Examples

Mathematics	Haskell
f(x)	f x
f(x,y)	f x y
f(g(x))	f (g x)
f(x,g(y))	f x (g y)
f(x)g(y)	f x * g y

Haskell Scripts

★As well as the functions in the standard prelude, you can also define your own functions;

★New functions are defined within a <u>script</u>, a text file comprising a sequence of definitions;

★By convention, Haskell scripts usually have a <u>.hs</u> suffix on their filename. This is not mandatory, but is useful for identification purposes.

My First Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running Hugs.

Start an editor, type in the following two function definitions, and save the script as test.hs:

double x   = x + x

quadruple x = double (double x)

Leaving the editor open, in another window start up the Haskell interpreter with the new script:

% ghci test.hs

Now both Prelude.hs and test.hs are loaded, and functions from both scripts can be used:

> quadruple 10
40

> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]

31

Leaving the interpreter open, return to the editor, add the following two definitions, and resave:

factorial n = product [1..n]

average ns = sum ns `div` length ns

Note:

z div is enclosed in <u>back</u> quotes, not forward;

z x `f` y is just <u>syntactic sugar</u> for f x y.

The interpreter does not automatically detect that the script has been changed, so a <u>reload</u> command must be executed before the new definitions can be used:

```
> :reload
Reading file "test.hs"

> factorial 10
3628800

> average [1,2,3,4,5]
3
```

Naming Requirements

★Function and argument names must begin with a lower-case letter. For example:

$$myFun \qquad fun1 \qquad arg\_2 \qquad x'$$

★By convention, list arguments usually have an s̲ suffix on their name. For example:

$$xs \qquad ns \qquad nss$$

34

The Layout Rule

In a sequence of definitions, each definition must begin in precisely the same column:

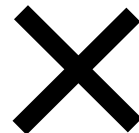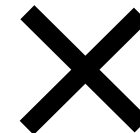|		
a = 10	a = 10	a = 10
b = 20	b = 20	b = 20
c = 30	c = 30	c = 30
✓	✗	✗

The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.

```
a = b + c          a = b + c
   where              where
     b = 1    means     {b = 1;
     c = 2               c = 2}
d = a * 2          d = a * 2
```

means

implicit grouping

explicit grouping

Useful Interpreter Commands

Command	Meaning
:load name	load script name
:reload	reload current script
:edit name	edit script name
:edit	edit current script
:type expr	show type of expr
:?	show all commands
:quit	quit interpreter

Exercises

(1) Try out all previous examples using the Haskell interpreter.

(2) Fix the syntax errors in the program below, and test your solution using the interpreter.
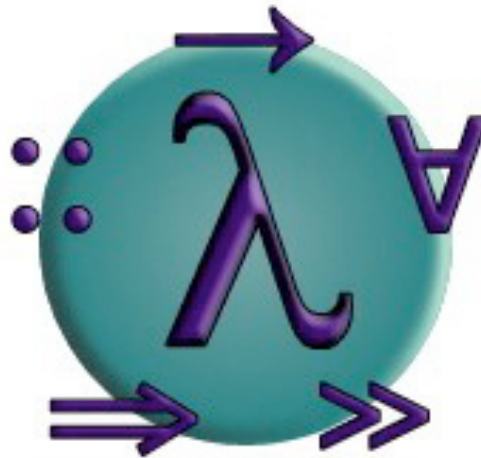
```
N = a 'div' length xs
    where
      a = 10
      xs = [1,2,3,4,5]
```

(3) Show how the library function <u>last</u> that selects the last element of a list can be defined using the functions introduced in this lecture.

(4) Can you think of another possible definition?

(5) Similarly, show how the library function <u>init</u> that removes the last element from a list can be defined in two different ways.

Common Types

What is a Type?

A <u>type</u> is a name for a collection of related values. For example, in Haskell the basic type

Bool

contains the two logical values:

False   True

41

Type Errors

Applying a function to one or more arguments of the wrong type is called a <u>type error</u>.

> 1 + False
Error

1 is a number and False is a logical value, but + requires two numbers.

Types in Haskell

★If evaluating an expression e would produce a value of type t, then e <u>has type</u> t, written

$$e :: t$$

★Every well formed expression has a type, which can be automatically calculated at compile time using a process called <u>type inference</u>.

★All type errors are found at compile time, which makes programs <u>safer and faster</u> by removing the need for type checks at run time.

★In the Haskell interpreter, the <u>:type</u> command calculates the type of an expression, without evaluating it:

```
> not False
True

> :type not False
not False :: Bool
```

44

Basic Types

Haskell has a number of <u>basic types</u>, including:

Bool   - logical values

Char   - single characters

String   - strings of characters

Int   - fixed-precision integers

Integer   - arbitrary-precision integers

Float   - floating-point numbers

45

List Types

A <u>list</u> is sequence of values of the <u>same</u> type:

$$[False,True,False] :: [Bool]$$

$$['a','b','c','d'] :: [Char]$$

In general:

[t] is the type of lists with elements of type t.

Note:

The type of a list says nothing about its length:

[False,True]   :: [Bool]

[False,True,False] :: [Bool]

The type of the elements is unrestricted. For example, we can have lists of lists:

[['a'],['b','c']] :: [[Char]]

Tuple Types

A <u>tuple</u> is a sequence of values of <u>different</u> types:

(False,True)   :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)

In general:

(t1,t2,…,tn) is the type of n-tuples whose ith components have type ti for any i in 1…n.

48

Note:

The type of a tuple encodes its size:

(False,True)   :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)

The type of the components is unrestricted:

('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b']) :: (Bool,[Char])

Hints and Tips

★When defining a new function in Haskell, it is useful to begin by writing down its type;

★Within a script, it is good practice to state the type of every new function defined;

50

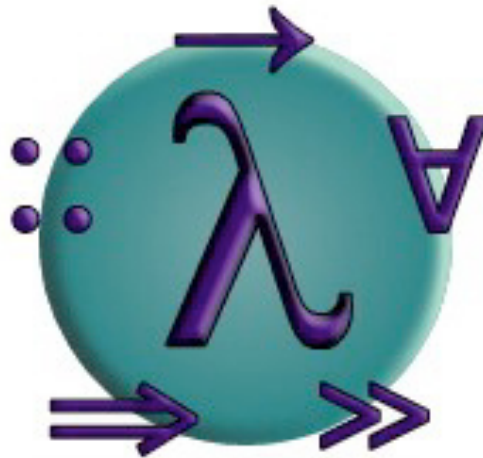# Exercises

(1) What are the types of the following values?

['a','b','c']

('a','b','c')

[(False,'0'),(True,'1')]

([False,True],['0','1'])

Functions types

Function Types

A <u>function</u> is a mapping from values of one type to values of another type:

> not   :: Bool $\rightarrow$ Bool

> isDigit :: Char $\rightarrow$ Bool

In general:

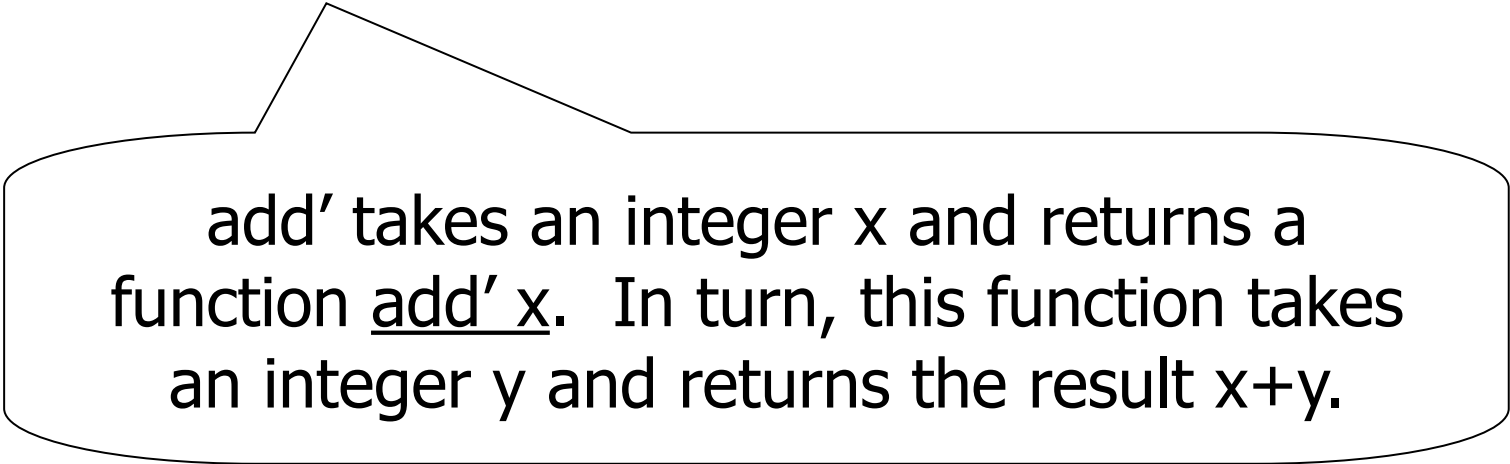> t1 $\rightarrow$ t2 is the type of functions that map values of type t1 to values to type t2.

53

Note:

★The arrow → is typed at the keyboard as ->.

★The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

$$add \quad :: (Int,Int) \rightarrow Int$$
$$add\ (x,y)\ = x+y$$

$$zeroto \quad :: Int \rightarrow [Int]$$
$$zeroto\ n\ = [0..n]$$

Curried Functions

Functions with multiple arguments are also possible by returning <u>functions as results</u>:

add'   :: Int → (Int → Int)

add' x y = x+y

add' takes an integer x and returns a function <u>add' x</u>. In turn, this function takes an integer y and returns the result x+y.

Note:

add and add' produce the same final result,
but add takes its two arguments at the same
time, whereas add' takes them one at a time:

$$\text{add} :: (\text{Int},\text{Int}) \rightarrow \text{Int}$$

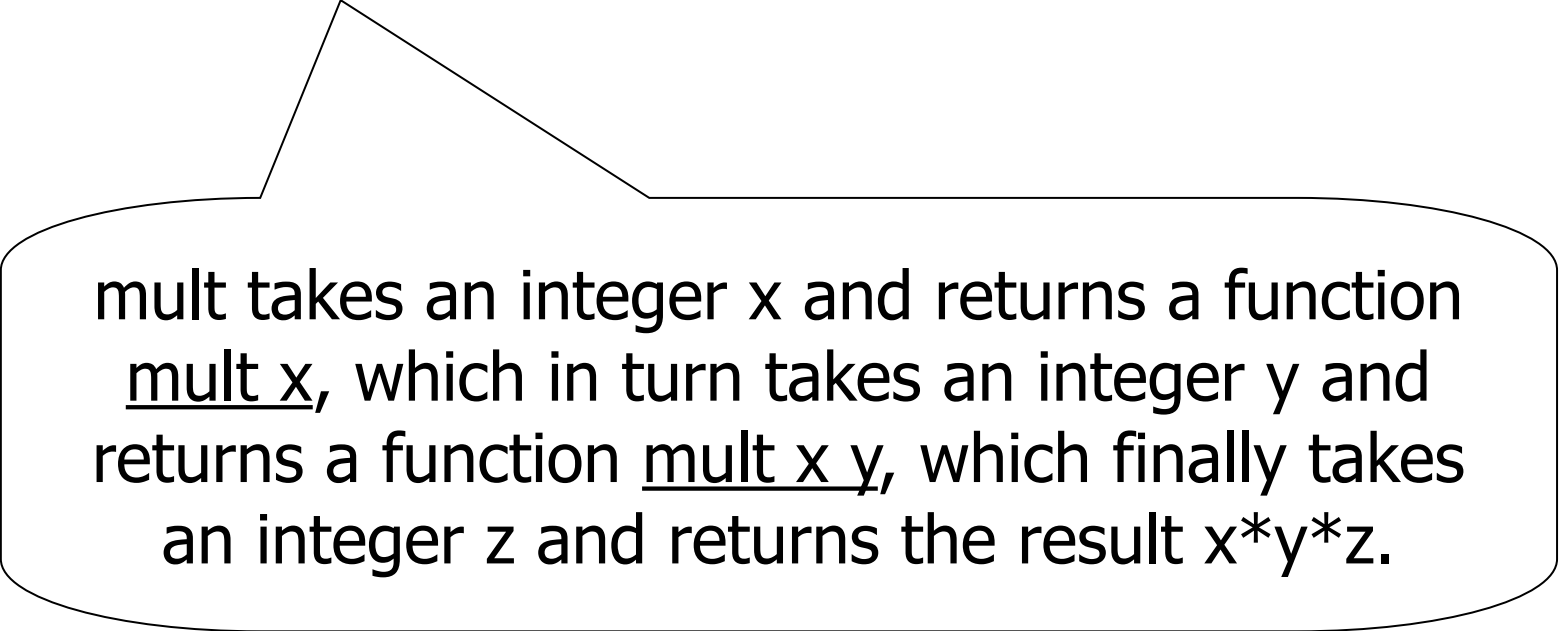$$\text{add'} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$$

Functions that take their arguments one at a
time are called <u>curried</u> functions, celebrating
the work of Haskell Curry on such functions.

Functions with more than two arguments can be curried by returning nested functions:

mult   :: Int → (Int → (Int → Int))

mult x y z = x*y*z

mult takes an integer x and returns a function <u>mult x</u>, which in turn takes an integer y and returns a function <u>mult x y</u>, which finally takes an integer z and returns the result x*y*z.

Why is Currying Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by <u>partially applying</u> a curried function.

For example:

add' 1 :: Int $\rightarrow$ Int

take 5 :: [Int] $\rightarrow$ [Int]

drop 5 :: [Int] $\rightarrow$ [Int]

Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

The arrow → associates to the <u>right</u>.

$$Int \rightarrow Int \rightarrow Int \rightarrow Int$$

Means Int → (Int → (Int → Int)).

As a consequence, it is then natural for function application to associate to the <u>left</u>.

mult x y z

Means ((mult x) y) z.
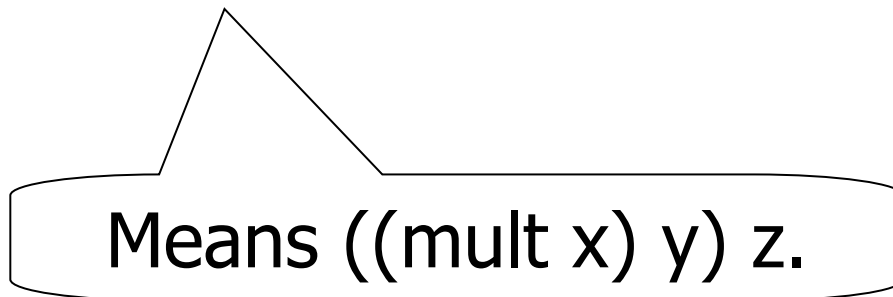
Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Polymorphic Functions

A function is called <u>polymorphic</u> ("of many forms") if its type contains one or more type variables.

$$\text{length} :: [a] \rightarrow \text{Int}$$

for any type a, length takes a list of values of type a and returns an integer.

Note:
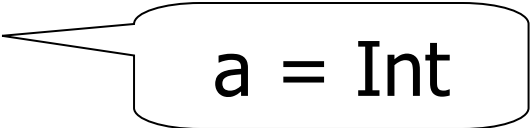
Type variables can be instantiated to different types in different circumstances:

> length [False,True]
2

a = Bool

> length [1,2,3,4]
4

a = Int

Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.

Many of the functions defined in the standard prelude are polymorphic. For example:

$$\text{fst} :: (a,b) \rightarrow a$$

$$\text{head} :: [a] \rightarrow a$$
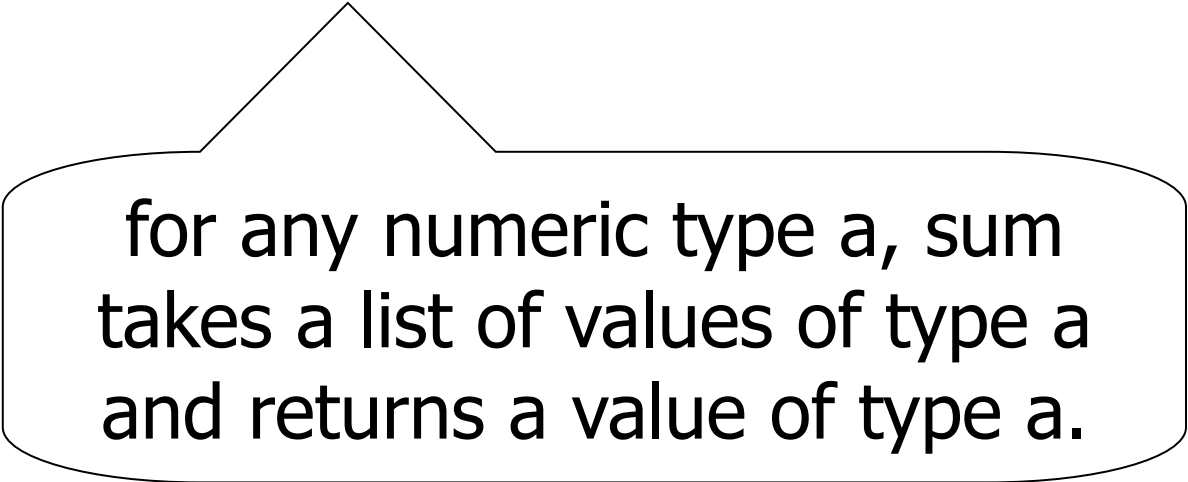
$$\text{take} :: \text{Int} \rightarrow [a] \rightarrow [a]$$

$$\text{zip} :: [a] \rightarrow [b] \rightarrow [(a,b)]$$

$$\text{id} :: a \rightarrow a$$

Overloaded Functions

A polymorphic function is called <u>overloaded</u> if its type contains one or more class constraints.

$$\text{sum} :: \text{Num } a \Rightarrow [a] \rightarrow a$$
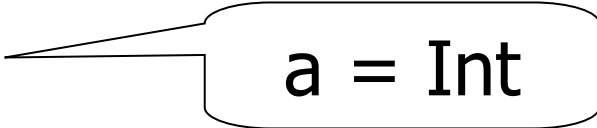
for any numeric type a, sum takes a list of values of type a and returns a value of type a.

Note:

Constrained type variables can be instantiated to any types that satisfy the constraints:

> sum [1,2,3]
6
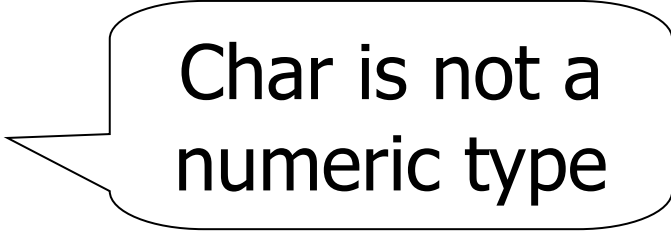
a = Int

> sum [1.1,2.2,3.3]
6.6

a = Float

> sum ['a','b','c']
ERROR

Char is not a numeric type

Haskell has a number of type classes, including:

Num - Numeric types

Eq   - Equality types

Ord   - Ordered types

For example:

$$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$$

$$(==) :: \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$$

$$(<) :: \text{Ord } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$$

Hints and Tips

★When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints.

Exercises

(1) What is the type of the following value?

[tail,init,reverse]

(2) What are the types of the following functions?

second xs   = head (tail xs)

swap (x,y)   = (y,x)

pair x y   = (x,y)

double x   = x*2

palindrome xs = reverse xs == xs

twice f x   = f (f x)

Defining Functions
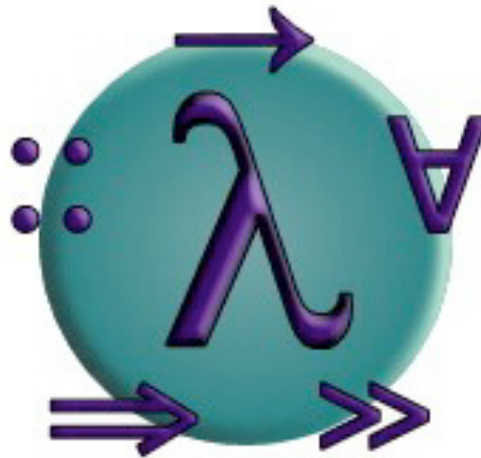
Conditional Expressions

As in most programming languages, functions can be defined using <u>conditional expressions</u>.

abs :: Int → Int
abs n = if n ≥ 0 then n else −n

abs takes an integer n and returns n if it is non-negative and -n otherwise.

Conditional expressions can be nested:

```
signum  :: Int → Int
signum n = if n < 0 then −1 else
                if n == 0 then 0 else 1
```

Note:

In Haskell, conditional expressions must <u>always</u> have an else branch, which avoids any possible ambiguity problems with nested conditionals.

Guarded Equations

As an alternative to conditionals, functions can also be defined using <u>guarded equations</u>.

$$
\begin{array}{lll}
\text{abs n} & |\ n \geq 0 & = \text{n} \\
& |\ \text{otherwise} & = -\text{n}
\end{array}
$$

As previously, but using guarded equations.

Guarded equations can be used to make definitions involving multiple conditions easier to read:

$$
\text{signum n} \quad
\begin{cases}
\text{n} < 0 & = -1 \\
\text{n} == 0 & = 0 \\
\text{otherwise} & = 1
\end{cases}
$$

Note:

The catch all condition <u>otherwise</u> is defined
in the prelude by otherwise = True.

Pattern Matching

Many functions have a particularly clear definition using <u>pattern matching</u> on their arguments.

```
not     :: Bool → Bool
not False = True
not True  = False
```

not maps False to True, and True to False.

Functions can often be defined in many different ways using pattern matching. For example

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

can be defined more compactly by

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

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

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

Note:

The underscore symbol _ is a <u>wildcard</u> pattern that matches any argument value.

Patterns are matched <u>in order</u>. For example, the following definition always returns False:

$$
\begin{aligned}
\_ \quad &\&\& \ \_ \qquad = \text{False} \\
\text{True} \ &\&\& \ \text{True} \qquad = \text{True}
\end{aligned}
$$
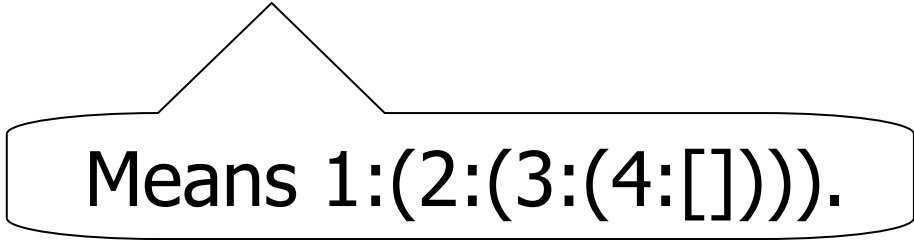
Patterns may not <u>repeat</u> variables. For example, the following definition gives an error:

$$
\begin{aligned}
b \ \&\& \ b &= b \\
\_ \ \&\& \ \_ &= \text{False}
\end{aligned}
$$

78

List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called "<u>cons</u>" that adds an element to the start of a list.

[1,2,3,4]

Means 1:(2:(3:(4:[]))).

Functions on lists can be defined using <u>x:xs</u> patterns.

head :: [a] → a
head (x:_) = x

tail :: [a] → [a]
tail (_:xs) = xs

head and tail map any non-empty list to
its first and remaining elements.

Note:

x:xs patterns only match <u>non-empty</u> lists:

> head []
Error

x:xs patterns must be <u>parenthesised</u>, because application has priority over (:). For example, the following definition gives an error:

head x:_ = x

Integer Patterns

As in mathematics, functions on integers can be defined using <u>n+k</u> patterns, where n is an integer variable and k>0 is an integer constant.

Banned in Haskell 2010

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

pred maps any positive integer to its predecessor.

Note:

n+k patterns only match integers ≥ k.

> pred 0
Error

Banned in Haskell 2010

n+k patterns must be <u>parenthesised</u>, because application has priority over +. For example, the following definition gives an error:

pred n+1 = n

Lambda Expressions

Functions can be constructed without naming the functions by using <u>lambda expressions</u>.

$$\lambda x \rightarrow x+x$$

the nameless function that takes a number x and returns the result x+x.

Note:

- The symbol λ is the Greek letter <u>lambda</u>, and is typed at the keyboard as a backslash \.

- In mathematics, nameless functions are usually denoted using the a symbol, as in x a x+x.

- In Haskell, the use of the λ symbol for nameless functions comes from the <u>lambda calculus</u>, the theory of functions on which Haskell is based.

Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using <u>currying</u>.

For example:

$$\text{add } x \; y = x+y$$

means

$$\text{add} = \lambda x \to (\lambda y \to x+y)$$

Lambda expressions are also useful when defining functions that return <u>functions as results</u>.

For example:

```
const :: a → b → a
const x _ = x
```

is more naturally defined by

```
const  :: a → (b → a)
const x = λ_ → x
```

Lambda expressions can be used to avoid naming functions that are only <u>referenced once</u>.

For example:

$$\text{odds } n = \text{map } f\,[0..n-1]$$
$$\quad \text{where}$$
$$\quad\quad f\,x = x{*}2 + 1$$

can be simplified to

$$\text{odds } n = \text{map } (\lambda x \to x{*}2 + 1)\,[0..n-1]$$

Infix vs. prefix

An operator written <u>between</u> its two arguments can be converted into a curried function written <u>before</u> its two arguments by using parentheses.

For example:

```
> 1+2
3


> (+) 1 2
3
```

Sections

We are also allowed to include one of the arguments of the operator in the parentheses.

For example:

> (1+) 2
3

> (+2) 1
3

In general, if $\oplus$ is an operator then functions of the form $(\oplus)$, $(x\oplus)$ and $(\oplus y)$ are called <u>sections</u>.

Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:

$(1+)$ - successor function

$(1/)$ - reciprocation function

$(*2)$ - doubling function

$(/2)$ - halving function

91

Exercises

(1) Consider a function <u>safetail</u> that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case. Define safetail using:

- (a) a conditional expression;
- (b) guarded equations;
- (c) pattern matching.

Hint: the library function null :: [a] $\rightarrow$ Bool can be used to test if a list is empty.

(2) Give three possible definitions for the logical or operator (||) using pattern matching.

(3) Redefine the following version of (&&) using conditionals rather than patterns:

$$\text{True } \&\& \text{ True } = \text{True}$$
$$\_ \quad \&\& \_ \quad = \text{False}$$

(4) Do the same for the following version:

$$\text{True } \&\& \text{ b } = \text{b}$$
$$\text{False } \&\& \_ = \text{False}$$

List Comprehensions
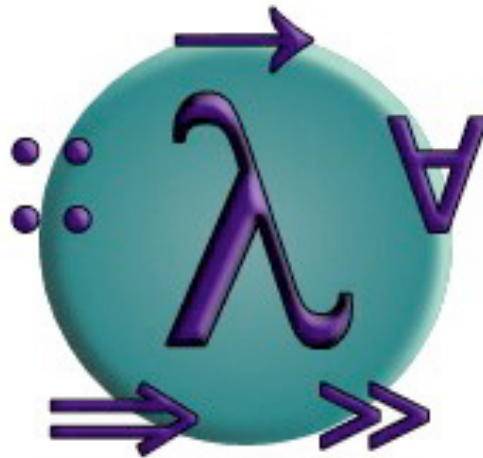
Set Comprehensions

In mathematics, the <u>comprehension</u> notation can be used to construct new sets from old sets.
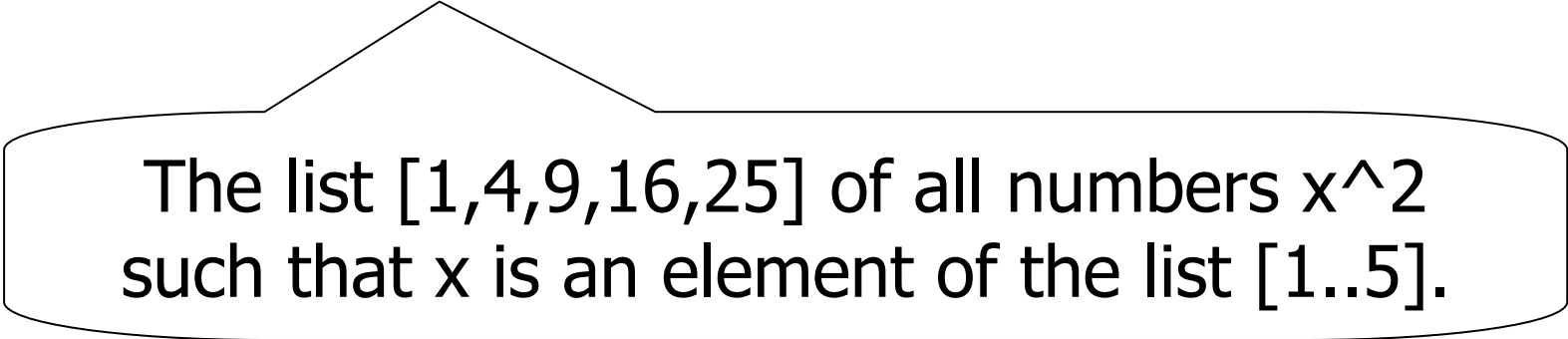
$$\{x^2 \mid x \in \{1...5\}\}$$

The set {1,4,9,16,25} of all numbers x^2 such that x is an element of the set {1...5}.

Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new <u>lists</u> from old lists.

$$[x\verb|^|2 \mid x \leftarrow [1..5]]$$

The list [1,4,9,16,25] of all numbers x^2 such that x is an element of the list [1..5].

Note:

⦿ The expression x ← [1..5] is called a <u>generator</u>, as it states how to generate values for x.

⦿ Comprehensions can have <u>multiple</u> generators, separated by commas. For example:

$$> [(x,y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5]]$$

$$[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]$$

Changing the <u>order</u> of the generators changes the order of the elements in the final list:

$$> [(x,y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5]]$$

$$[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]$$

$$> [(x,y) \mid y \leftarrow [4,5], x \leftarrow [1,2,3]]$$

$$[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]$$

Multiple generators are like <u>nested loops</u>, with later generators as more deeply nested loops whose variables change value more frequently.

Dependant Generators

Later generators can <u>depend</u> on the variables that are introduced by earlier generators.

$$[(x,y) \mid x \leftarrow [1..3], y \leftarrow [x..3]]$$

The list [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] of all pairs of numbers (x,y) such that x,y are elements of the list [1..3] and y ≥ x.

Using a dependant generator we can define the library function that <u>concatenates</u> a list of lists:

$$\text{concat} \quad :: [[a]] \rightarrow [a]$$
$$\text{concat xss} = [x \mid xs \leftarrow xss, x \leftarrow xs]$$

> We iterate over the lists of lists, and then over the elements of each list in turn, and finally we append all those elements.

For example:

$$> \text{concat } [[1,2,3],[4,5],[6]]$$

$$[1,2,3,4,5,6]$$

Guards

List comprehensions can use <u>guards</u> to restrict the values produced by earlier generators.

$$[x \mid x \leftarrow [1..10], \text{even } x]$$

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

101

Using a guard we can define a function that maps a positive integer to its list of <u>factors</u>:

$$\text{factors} :: \text{Int} \rightarrow [\text{Int}]$$
$$\text{factors } n =$$
$$[x \mid x \leftarrow [1..n], n \text{ `mod` } x == 0]$$

For example:

> factors 15

$[1,3,5,15]$

A positive integer is <u>prime</u> if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

```
prime :: Int → Bool
prime n = factors n == [1,n]
```

For example:

```
> prime 15
False

> prime 7
True
```

Using a guard we can now define a function that returns the list of all <u>primes</u> up to a given limit:

```
primes :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

For example:

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

The Zip Function

A useful library function is <u>zip</u>, which maps two lists to a list of pairs of their corresponding elements.

$$zip :: [a] \rightarrow [b] \rightarrow [(a,b)]$$

> We do not show the definition of zip at this point.

For example:

> zip ['a','b','c'] [1,2,3,4]

[('a',1),('b',2),('c',3)]

105

Using zip we can define a function returns the list of all <u>pairs</u> of **adjacent elements** from a list:

$$pairs \quad :: [a] \rightarrow [(a,a)]$$
$$pairs \ xs = zip \ xs \ (tail \ xs)$$

For example:

$$> pairs \ [1,2,3,4]$$
$$[(1,2),(2,3),(3,4)]$$

Using pairs we can define a function that decides if the elements in a list are <u>sorted</u>:

$$\text{sorted} \quad :: \text{Ord a} \Rightarrow [a] \to \text{Bool}$$
$$\text{sorted xs} =$$
$$\text{and } [x \leq y \mid (x,y) \leftarrow \text{pairs xs}]$$

For example:

> sorted $[1,2,3,4]$
True

> sorted $[1,3,2,4]$
False

Using zip we can define a function that returns the list of all <u>positions</u> of a value in a list:

$$\text{positions} :: \text{Eq a} \Rightarrow \text{a} \rightarrow [\text{a}] \rightarrow [\text{Int}]$$

positions x xs =

   [i | (x',i) ← zip xs [0..n], x == x']

   where n = length xs − 1
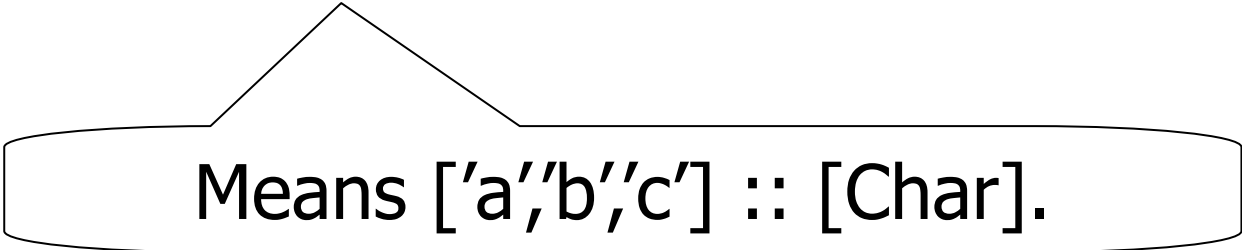
For example:

> positions 0 [1,0,0,1,0,1,1,0]

[1,2,4,7]

String Comprehensions

A <u>string</u> is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.

"abc" :: String

Means ['a','b','c'] :: [Char].

Because strings are just special kinds of lists, any <u>polymorphic</u> function that operates on lists can also be applied to strings. For example:

> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]

110

Similarly, list comprehensions can also be used to define functions on strings, such as a function that counts the lower-case letters in a string:

```
lowers   :: String → Int
lowers xs =
    length [x | x ← xs, isLower x]
```

For example:

```
> lowers "Haskell"
6
```

Exercises

(1) A triple (x,y,z) of positive integers is called
<u>pythagorean</u> if $x^2 + y^2 = z^2$. Using a list
comprehension, define a function

$$\text{pyths :: Int} \rightarrow [(\text{Int,Int,Int})]$$

that maps an integer n to all such triples with
components in [1..n]. For example:

> pyths 5
[(3,4,5),(4,3,5)]

(2) A positive integer is <u>perfect</u> if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension, define a function

$$\text{perfects :: Int} \rightarrow [\text{Int}]$$

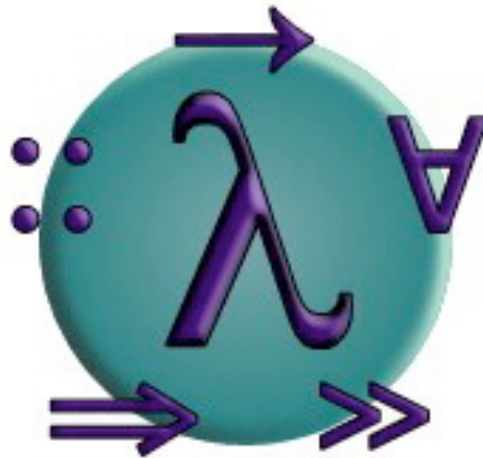that returns the list of all perfect numbers up to a given limit. For example:

> perfects 500

[6,28,496]

(3) The <u>scalar product</u> of two lists of integers xs and ys of length n is give by the sum of the products of the corresponding integers:

$$\sum_{i = 0}^{n-1} (xs_i * ys_i)$$

Using a list comprehension, define a function that returns the scalar product of two lists.

114

Recursive Functions

115

Introduction

As we have seen, many functions can naturally be defined in terms of other functions.

factorial :: Int → Int

factorial n = product [1..n]

factorial maps any integer n to the product of the integers between 1 and n.

Expressions are <u>evaluated</u> by a stepwise process of applying functions to their arguments.

For example:

factorial 4

=

product [1..4]

=

product [1,2,3,4]

=

1*2*3*4

=

24

Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called <u>recursive</u>.

$$\text{factorial } 0 \quad = 1$$
$$\text{factorial } (n+1) = (n+1) \ast \text{factorial } n$$

factorial maps 0 to 1, and any other positive integer to the product of itself and the factorial of its predecessor.

For example:

$$\text{factorial } 3$$
$$=$$
$$3 * \text{factorial } 2$$
$$=$$
$$3 * (2 * \text{factorial } 1)$$
$$=$$
$$3 * (2 * (1 * \text{factorial } 0))$$
$$=$$
$$3 * (2 * (1 * 1))$$
$$=$$
$$3 * (2 * 1)$$
$$=$$
$$3 * 2$$
$$=$$
$$6$$

119

Note:

◉ factorial 0 = 1 is appropriate because 1 is the identity for multiplication: $1*x = x = x*1$.

◉ The recursive definition <u>diverges</u> on integers < 0 because the base case is never reached:

> factorial (-1)

Error: Control stack overflow

Why is Recursion Useful?

◉Some functions, such as factorial, are <u>simpler</u> to define in terms of other functions.

◉As we shall see, however, many functions can <u>naturally</u> be defined in terms of themselves.

◉Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of <u>induction</u>.

Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on <u>lists</u>.

```
product     :: [Int] → Int
product []    = 1
product (n:ns) = n * product ns
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

For example:

$$product\ [2,3,4]$$
$$=$$
$$2 * product\ [3,4]$$
$$=$$
$$2 * (3 * product\ [4])$$
$$=$$
$$2 * (3 * (4 * product\ []))$$
$$=$$
$$2 * (3 * (4 * 1))$$
$$=$$
$$24$$

Using the same pattern of recursion as in product we can define the <u>length</u> function on lists.

$$length \quad :: [a] \rightarrow Int$$
$$length \, [] \quad = 0$$
$$length \, (\_:xs) = 1 + length \, xs$$

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.

For example:

length [1,2,3]

=

1 + length [2,3]

=

1 + (1 + length [3])

=

1 + (1 + (1 + length []))

=

1 + (1 + (1 + 0))

=

3

125

Using a similar pattern of recursion we can define the <u>reverse</u> function on lists.

$$\text{reverse} \quad :: [a] \rightarrow [a]$$
$$\text{reverse } [] \quad = []$$
$$\text{reverse } (x{:}xs) = \text{reverse } xs \; ++ \; [x]$$

reverse maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.

For example:

$$\text{reverse } [1,2,3]$$

$$=$$

$$\text{reverse } [2,3] \mathbin{++} [1]$$

$$=$$

$$(\text{reverse } [3] \mathbin{++} [2]) \mathbin{++} [1]$$

$$=$$

$$((\text{reverse } [] \mathbin{++} [3]) \mathbin{++} [2]) \mathbin{++} [1]$$

$$=$$

$$(([] \mathbin{++} [3]) \mathbin{++} [2]) \mathbin{++} [1]$$

$$=$$

$$[3,2,1]$$

Multiple Arguments

Functions with more than one argument can also be defined using recursion. For example:

Zipping the elements of two lists:

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

128

Remove the first n elements from a list:

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

Appending two lists:

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

Quicksort

The <u>quicksort</u> algorithm for sorting a list of integers can be specified by the following two rules:

The empty list is already sorted;

Non-empty lists can be sorted by sorting the tail values ≤ the head, sorting the tail values > the head, and then appending the resulting lists on either side of the head value.
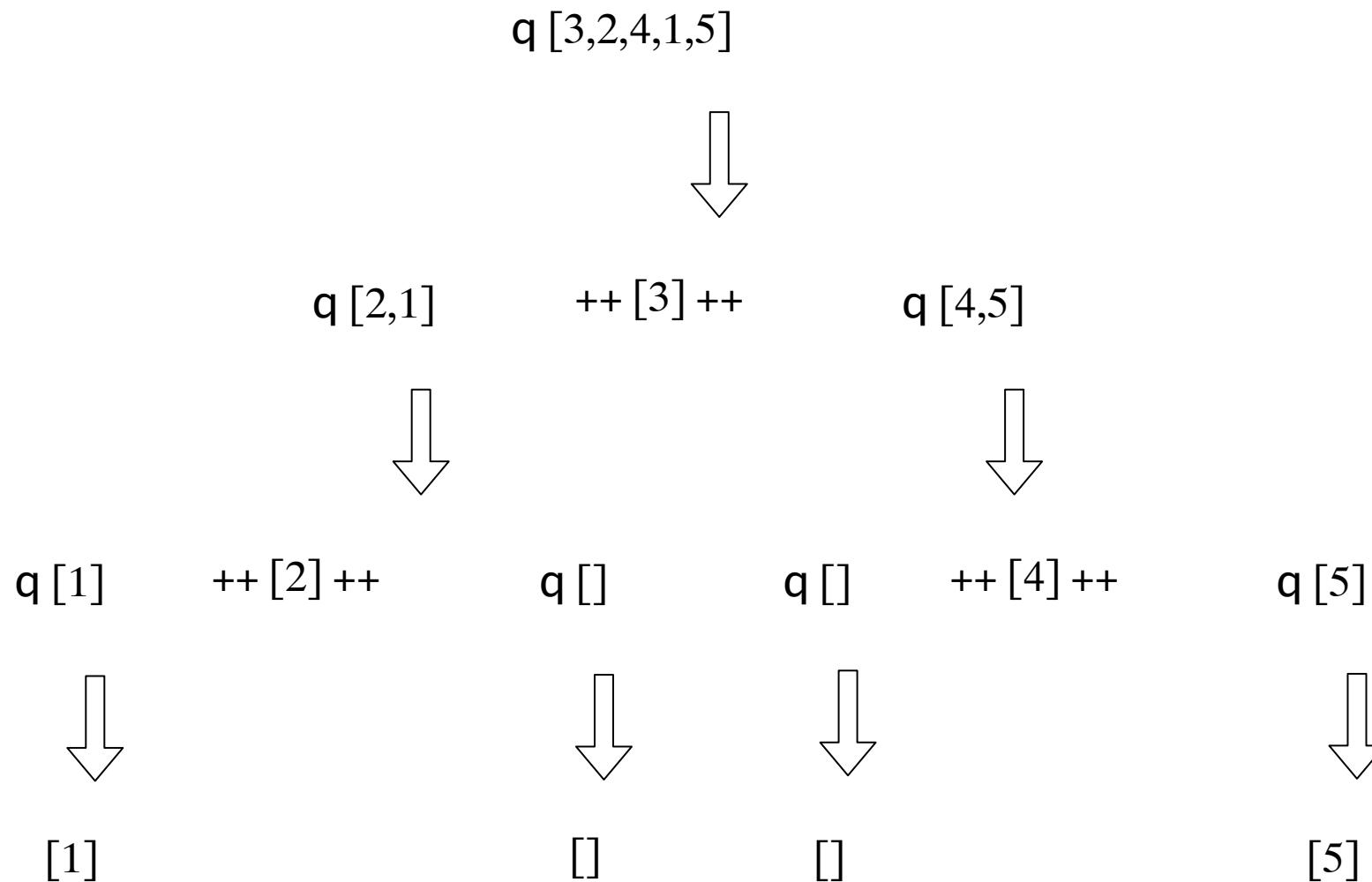
130

Using recursion, this specification can be translated directly into an implementation:

```
qsort      :: [Int] → [Int]
qsort []    = []
qsort (x:xs) =
    qsort smaller ++ [x] ++ qsort larger
    where
      smaller = [a | a ← xs, a ≤ x]
      larger  = [b | b ← xs, b > x]
```

Note:

This is probably the <u>simplest</u> implementation of quicksort in any programming language!

For example (abbreviating qsort as q):

q [3,2,4,1,5]

⇓

q [2,1]   ++ [3] ++   q [4,5]

⇓   ⇓

q [1]   ++ [2] ++   q []   q []   ++ [4] ++   q [5]

⇓   ⇓   ⇓   ⇓

[1]   []   []   [5]

Exercises

(1) Without looking at the standard prelude, define the following library functions using recursion:

Decide if all logical values in a list are true:

$$and :: [Bool] \rightarrow Bool$$

Concatenate a list of lists:

$$concat :: [[a]] \rightarrow [a]$$

133

Produce a list with n identical elements:

$$\text{replicate} :: \text{Int} \to a \to [a]$$

Select the nth element of a list:

$$(!!) :: [a] \to \text{Int} \to a$$

Decide if a value is an element of a list:

$$\text{elem} :: \text{Eq } a \Rightarrow a \to [a] \to \text{Bool}$$

(2) Define a recursive function

$$\text{merge} :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow [\text{Int}]$$

that merges two sorted lists of integers to give a single sorted list. For example:

> merge $[2,5,6]$ $[1,3,4]$

$[1,2,3,4,5,6]$
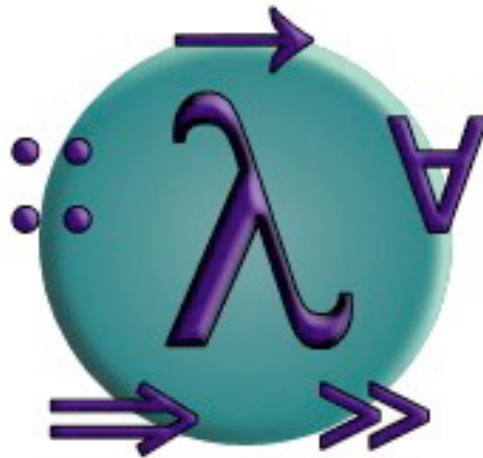
(3) Define a recursive function

$$\text{msort} :: [\text{Int}] \rightarrow [\text{Int}]$$

that implements <u>merge sort</u>, which can be specified by the following two rules:

i) Lists of length ≤ 1 are already sorted;

ii) other lists can be sorted by sorting the two halves and merging the resulting lists.

Higher-Order Functions

Introduction

A function is called <u>higher-order</u> if it takes a function as an argument or returns a function as a result.

$$\text{twice} :: (a \rightarrow a) \rightarrow a \rightarrow a$$

$$\text{twice } f\ x = f\ (f\ x)$$

twice is higher-order because it takes a function as its first argument.

138

Why Are They Useful?

◉ <u>Common programming idioms</u> can be encoded as functions within the language itself.

◉ <u>Domain specific languages</u> can be defined as collections of higher-order functions.

◉ <u>Algebraic properties</u> of higher-order functions can be used to reason about programs.

139

The Map Function

The higher-order library function called <u>map</u> applies a function to every element of a list.

$$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

For example:

> map (+1) [1,3,5,7]

[2,4,6,8]

140

The map function can be defined in a particularly simple manner using a list comprehension:

$$\text{map f xs} = [\text{f x} \mid \text{x} \leftarrow \text{xs}]$$

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

$$\text{map f } [] = []$$

$$\text{map f (x:xs)} = \text{f x : map f xs}$$

The Filter Function

The higher-order library function <u>filter</u> selects every element from a list that satisfies a predicate.

$$\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

For example:

> filter even [1..10]

[2,4,6,8,10]

142

Filter can be defined using a list comprehension:

$$\text{filter p xs} = [\text{x} \mid \text{x} \leftarrow \text{xs}, \text{p x}]$$

Alternatively, it can be defined using recursion:

```
filter p []     = []
filter p (x:xs)
        | p x        = x : filter p xs
        | otherwise  = filter p xs
```

The Foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

$$f\ [] \quad = v$$
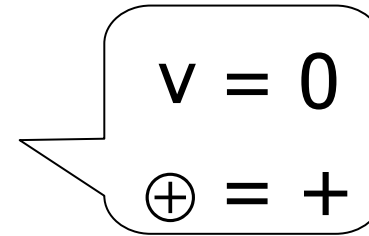$$f\ (x:xs) = x \oplus f\ xs$$

f maps the empty list to some value v, and any non-empty list to some function $\oplus$ applied to its head and f of its tail.
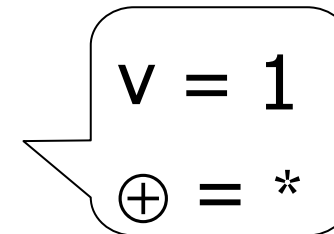
144

For example:

$$\text{sum } [] = 0$$
$$\text{sum } (x{:}xs) = x + \text{sum } xs$$
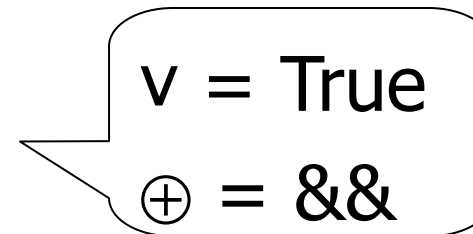
$v = 0$

$\oplus = +$

$$\text{product } [] = 1$$
$$\text{product } (x{:}xs) = x * \text{product } xs$$

$v = 1$

$\oplus = *$

$$\text{and } [] = \text{True}$$
$$\text{and } (x{:}xs) = x \text{ \&\& and } xs$$

$v = \text{True}$

$\oplus = \&\&$

The higher-order library function <u>foldr</u> (fold right) encapsulates this simple pattern of recursion, with the function $\oplus$ and the value v as arguments.

For example:

sum $= \text{foldr} (+) \, 0$

product $= \text{foldr} (*) \, 1$

or $= \text{foldr} (\|) \, \text{False}$

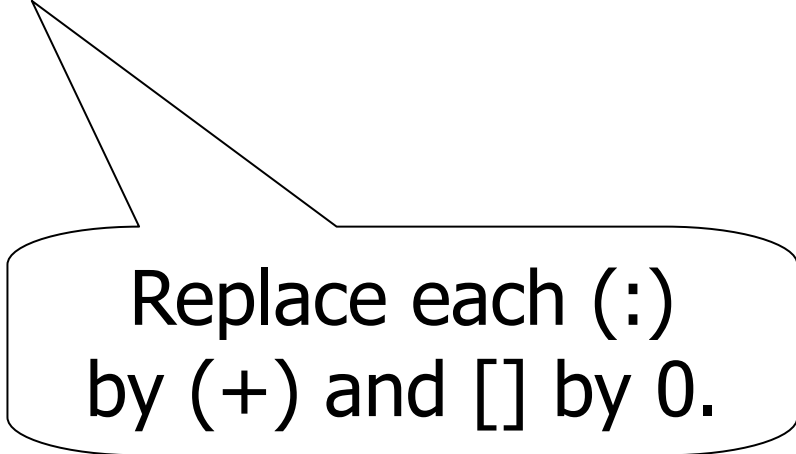and $= \text{foldr} (\&\&) \, \text{True}$

Foldr itself can be defined using recursion:

$$\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$$

$$\text{foldr f v } [] = v$$

$$\text{foldr f v } (x{:}xs) = f\ x\ (\text{foldr f v xs})$$

However, it is best to think of foldr <u>non-recursively</u>, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.

For example:
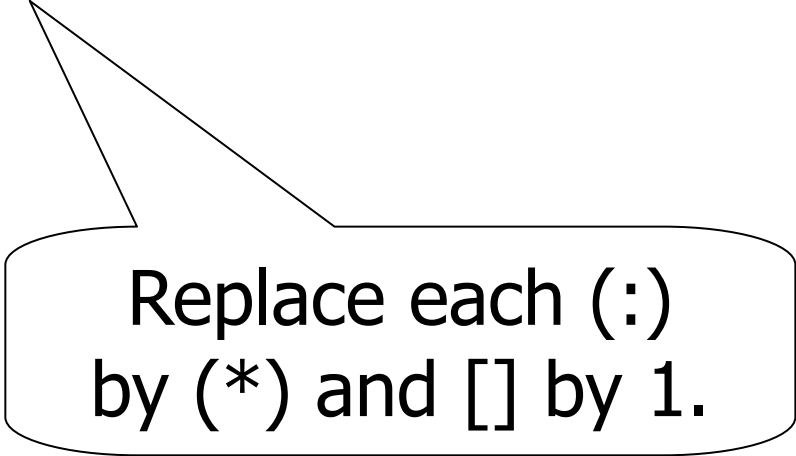
$$\text{sum } [1,2,3]$$

$$=$$

$$\text{foldr } (+) \; 0 \; [1,2,3]$$

$$=$$

$$\text{foldr } (+) \; 0 \; (1{:}(2{:}(3{:}[])))$$

$$=$$

$$1+(2+(3+0))$$

$$=$$

$$6$$

> Replace each (:) by (+) and [] by 0.

For example:

$$product\ [1,2,3]$$

=

$$foldr\ (*)\ 1\ [1,2,3]$$

=

$$foldr\ (*)\ 1\ (1:(2:(3:[])))$$

=

$$1*(2*(3*1))$$

=

$$6$$

> Replace each (:) by (*) and [] by 1.
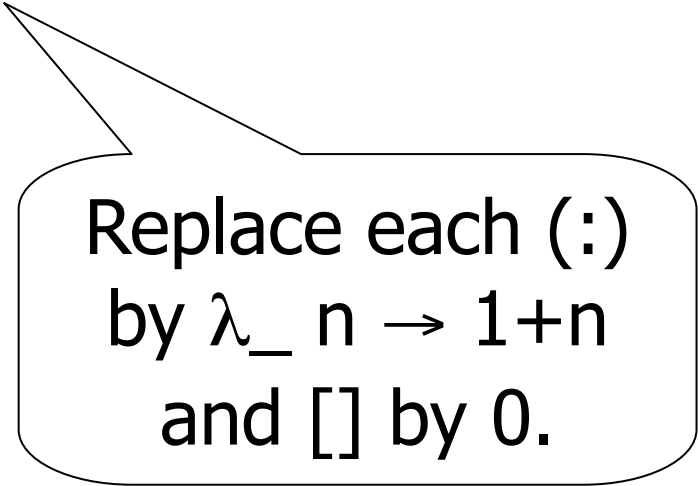
Other Foldr Examples

Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the length function:

$$\text{length} :: [a] \rightarrow \text{Int}$$

$$\text{length} \, [] = 0$$

$$\text{length} \, (\_:xs) = 1 + \text{length } xs$$

150

For example:

length [1,2,3]

=

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

=

1+(1+(1+0))

=

3

Replace each (:)
by $\lambda\_\ n \to 1+n$
and [] by 0.

Hence, we have:

length = foldr ($\lambda\_\ n \to 1+n$) 0

151

Now recall the reverse function:

$$\text{reverse } [] \quad = []$$
$$\text{reverse } (x:xs) = \text{reverse } xs ++ [x]$$

For example:

$$\text{reverse } [1,2,3]$$
$$=$$
$$\text{reverse } (1:(2:(3:[])))$$
$$=$$
$$(([] ++ [3]) ++ [2]) ++ [1]$$
$$=$$
$$[3,2,1]$$

> Replace each (:) by
> $\lambda x\ xs \rightarrow xs ++ [x]$
> and [] by [].

152
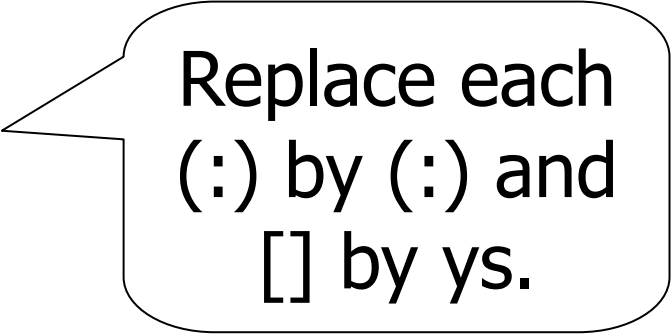
Hence, we have:

reverse =
    foldr (λx xs $\rightarrow$ xs ++ [x]) []

Finally, we note that the append function (++) has a particularly compact definition using foldr:

(++ ys) = foldr (:) ys

Replace each (:) by (:) and [] by ys.

153

Why Is Foldr Useful?

◉Some recursive functions on lists, such as sum, are <u>simpler</u> to define using foldr.

◉Properties of functions defined using foldr can be proved using algebraic properties of foldr, such as <u>fusion</u> and the <u>banana split</u> rule.

◉Advanced program <u>optimisations</u> can be simpler if foldr is used in place of explicit recursion.

154

Other Library Functions

The library function (.) returns the <u>composition</u> of two functions as a single function.

$$(.) \ :: (b \to c) \to (a \to b) \to (a \to c)$$

$$f \ . \ g \ = \lambda x \to f \ (g \ x)$$

For example:

$$odd :: Int \to Bool$$

$$odd \ = not \ . \ even$$

155

The library function <u>all</u> decides if every element of a list satisfies a given predicate.

$$\text{all} \quad :: (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool$$
$$\text{all p xs} = \text{and } [p\ x \mid x \leftarrow xs]$$

For example:

> all even [2,4,6,8,10]

True

156

Dually, the library function <u>any</u> decides if at least one element of a list satisfies a predicate.

$$\text{any} \quad :: (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool$$
$$\text{any } p \text{ } xs = or \text{ } [p \text{ } x \mid x \leftarrow xs]$$

For example:

> any isSpace "abc def"

True

157

The library function <u>takeWhile</u> selects elements from a list while a predicate holds of all the elements.

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p []    = []
takeWhile p (x:xs)
  | p x        = x : takeWhile p xs
  | otherwise    = []
```

For example:

```
> takeWhile isAlpha "abc def"

"abc"
```

Dually, the function <u>dropWhile</u> removes elements while a predicate holds of all the elements.

```
dropWhile :: (a → Bool) → [a] → [a]
dropWhile p []    = []
dropWhile p (x:xs)
  | p x        = dropWhile p xs
  | otherwise    = x:xs
```

For example:

```
> dropWhile isSpace "  abc"

"abc"
```

159

Exercises

(1) What are higher-order functions that return functions as results better known as?

(2) Express the comprehension [f x | x ← xs, p x] using the functions map and filter.

(3) Redefine map f and filter p using foldr.

160

Type Declarations

Type Declarations

In Haskell, a new name for an existing type can be defined using a <u>type declaration</u>.

type String = [Char]

String is a synonym for the type [Char].

Type declarations can be used to make other types easier to read. For example, given

type Pos = (Int,Int)

we can define:

origin :: Pos
origin = (0,0)

left :: Pos → Pos
left (x,y) = (x−1,y)

Like function definitions, type declarations can also have <u>parameters</u>. For example, given

type Pair a = (a,a)

we can define:

mult :: Pair Int → Int
mult (m,n) = m*n

copy :: a → Pair a
copy x = (x,x)

Type declarations can be nested:

```
type Pos  = (Int,Int)

type Trans = Pos → Pos
```

✓
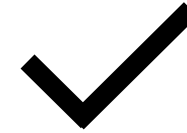
However, they cannot be recursive:

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

✗

165

Data Declarations

A completely new type can be defined by specifying its values using a <u>data declaration</u>.

data Bool = False | True

Bool is a new type, with two new values False and True.

Note:

- The two values False and True are called the <u>constructors</u> for the type Bool.

- Type and constructor names must begin with an upper-case letter.

- Data declarations are similar to context free grammars. The former specifies the values of a type, the latter the sentences of a language.

167

Values of new types can be used in the same ways as those of built in types. For example, given

data Answer = Yes | No | Unknown

we can define:

answers   :: [Answer]
answers   = [Yes,No,Unknown]

flip :: Answer → Answer
flip Yes = No
flip No   = Yes
flip Unknown = Unknown

168

The constructors in a data declaration can also have parameters. For example, given

data Shape = Circle Float
                       | Rect Float Float

we can define:

square :: Float → Shape
square n = Rect n n

area :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y

Note:

◉Shape has values of the form Circle r where r is a float, and Rect x y where x and y are floats.

◉Circle and Rect can be viewed as <u>functions</u> that construct values of type Shape:

Circle :: Float → Shape

Rect :: Float → Float → Shape

170

Not surprisingly, data declarations themselves can also have parameters. For example, given

data Maybe a = Nothing | Just a

we can define:
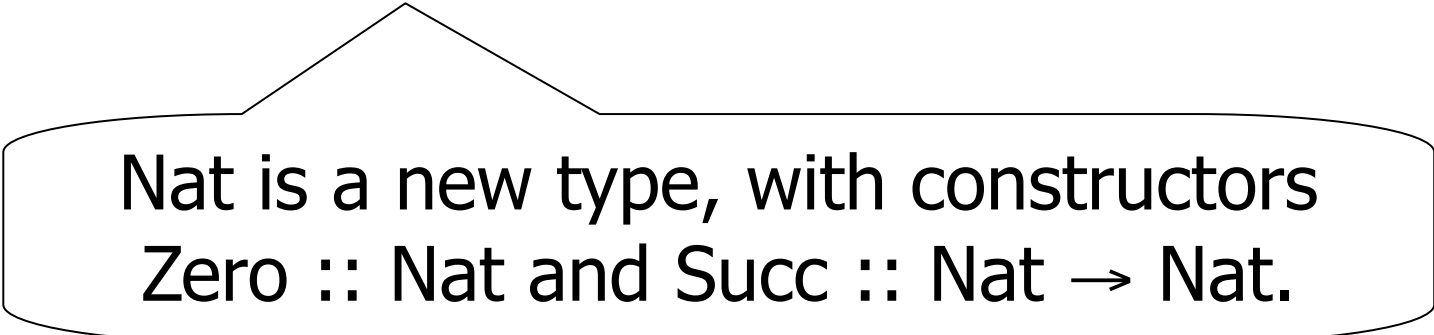
safediv :: Int $\rightarrow$ Int $\rightarrow$ Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] $\rightarrow$ Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be <u>recursive</u>.

data Nat = Zero | Succ Nat

Nat is a new type, with constructors
Zero :: Nat and Succ :: Nat → Nat.

172

Note:

A value of type Nat is either Zero, or of the form Succ n where n :: Nat. That is, Nat contains the following infinite sequence of values:

Zero

Succ Zero

Succ (Succ Zero)

.
.
.

173

We can think of values of type Nat as <u>natural numbers</u>, where Zero represents 0, and Succ represents the successor function 1+.

For example, the value

$$Succ\ (Succ\ (Succ\ Zero))$$

represents the natural number

$$1 + (1 + (1 + 0)) \qquad = \quad 3$$

Using recursion, it is easy to define functions that convert between values of type Nat and Int:

nat2int   :: Nat $\to$ Int

nat2int Zero $= 0$

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

int2nat   :: Int $\to$ Nat

int2nat 0 $=$ Zero

int2nat (n+1) $=$ Succ (int2nat n)

Two naturals can be added by converting them to integers, adding, and then converting back:

add :: Nat $\rightarrow$ Nat $\rightarrow$ Nat

add m n = int2nat (nat2int m + nat2int n)

However, using recursion the function add can be defined without the need for conversions:

add Zero n = n

add (Succ m) n = Succ (add m n)

176

For example:

$$\text{add (Succ (Succ Zero)) (Succ Zero)}$$

$$=$$

$$\text{Succ (add (Succ Zero) (Succ Zero))}$$

$$=$$

$$\text{Succ (Succ (add Zero (Succ Zero))}$$

$$=$$

$$\text{Succ (Succ (Succ Zero))}$$

Note: The recursive definition for add corresponds to the laws $0+n = n$ and $(1+m)+n = 1+(m+n)$.

Arithmetic Expressions

Consider a simple form of <u>expressions</u> built up from integers using addition and multiplication.
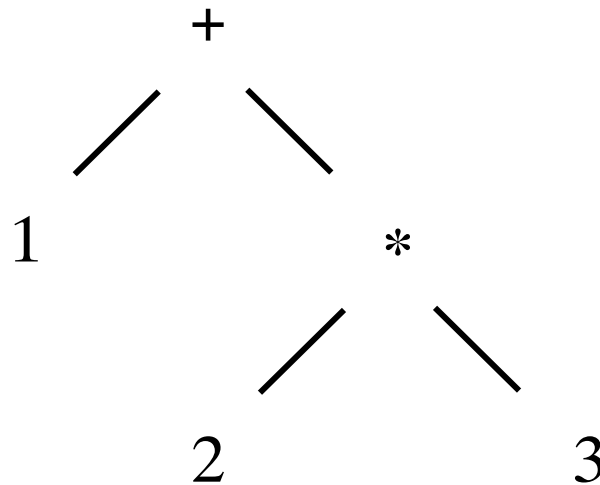
Using recursion, a suitable new type to represent such expressions can be declared by:

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

For example, the expression on the previous slide would be represented as follows:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

179

Using recursion, it is now easy to define functions that process expressions. For example:

size :: Expr → Int

size (Val n)   = 1

size (Add x y) = size x + size y

size (Mul x y) = size x + size y

eval :: Expr → Int

eval (Val n)   = n

eval (Add x y) = eval x + eval y

eval (Mul x y) = eval x * eval y

180

On the types of constructors

The three constructors have types:

Val :: Int → Expr
Add :: Expr → Expr → Expr
Mul :: Expr → Expr → Expr

A fold for expressions

Many functions on expressions can be defined by replacing the constructors by other functions using a suitable <u>fold</u> function. For example:

$$eval = fold\ id\ (+)\ (*)$$

Binary Trees

In computing, it is often useful to store data in a two-way branching structure or <u>binary tree</u>.

```
        5
      /   \
     3     7
    / \   / \
   1   4 6   9
```
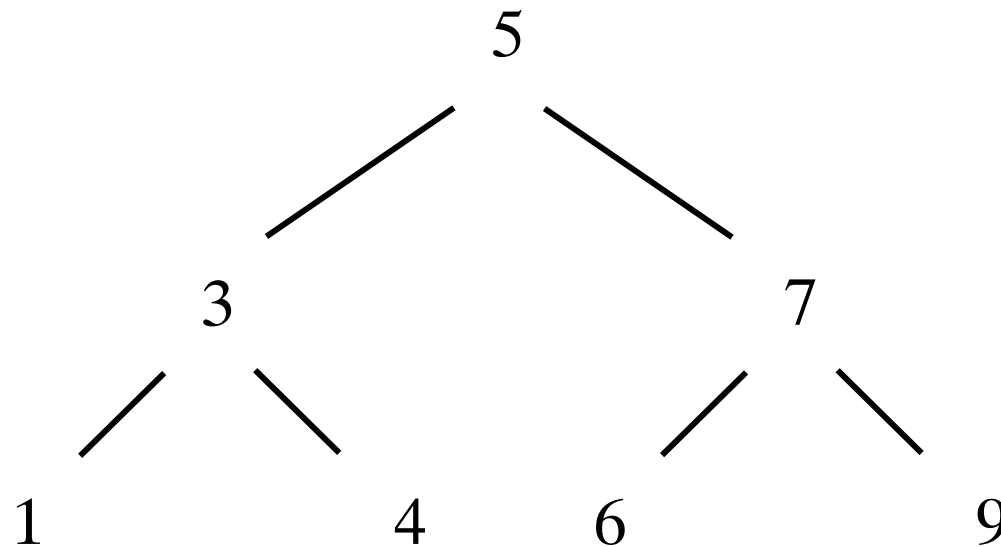
Using recursion, a suitable new type to represent such binary trees can be declared by:

```
data Tree    = Leaf Int
             | Node Tree Int Tree
```

For example, the tree on the previous slide would be represented as follows:

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

184

We can now define a function that decides if a given integer occurs in a binary tree:

occurs :: Int → Tree → Bool

occurs m (Leaf n) = m==n

occurs m (Node l n r) =   m==n

‖ occurs m l

‖ occurs m r

In the worst case, when the integer does not occur, this function traverses the entire tree.

Now consider the function <u>flatten</u> that returns the list of all the integers contained in a tree:

$$\text{flatten} :: \text{Tree} \rightarrow [\text{Int}]$$
$$\text{flatten } (\text{Leaf n}) = [\text{n}]$$
$$\text{flatten } (\text{Node l n r}) = \text{flatten l}$$
$$++ [\text{n}]$$
$$++ \text{flatten r}$$

A tree is a <u>search tree</u> if it flattens to a list that is ordered. Our example tree is a search tree, as it flattens to the ordered list [1,3,4,5,6,7,9].

Search trees have the important property that when trying to find a value in a tree we can always decide which of the two sub-trees it may occur in:

```
occurs m (Leaf n) = m==n

occurs m (Node l n r)    | m==n = True

                         | m<n  = occurs m l

                         | m>n  = occurs m r
```

This new definition is more <u>efficient</u>, because it only traverses one path down the tree.

187

Exercises

(1   Using recursion and the function add, define a function that <u>multiplies</u> two natural numbers.

(2   Define a suitable function <u>fold</u> for expressions, and give a few examples of its use.

(3   A binary tree is <u>complete</u> if the two sub-trees of every node are of equal size. Define a function that decides if a binary tree is complete.

Functional Parsers

What is a Parser?

A <u>parser</u> is a program that analyses a piece of text to determine its <u>syntactic structure</u>.

2*3+4   **means** →

```
        +
       / \
      *   4
     / \
    2   3
```

190
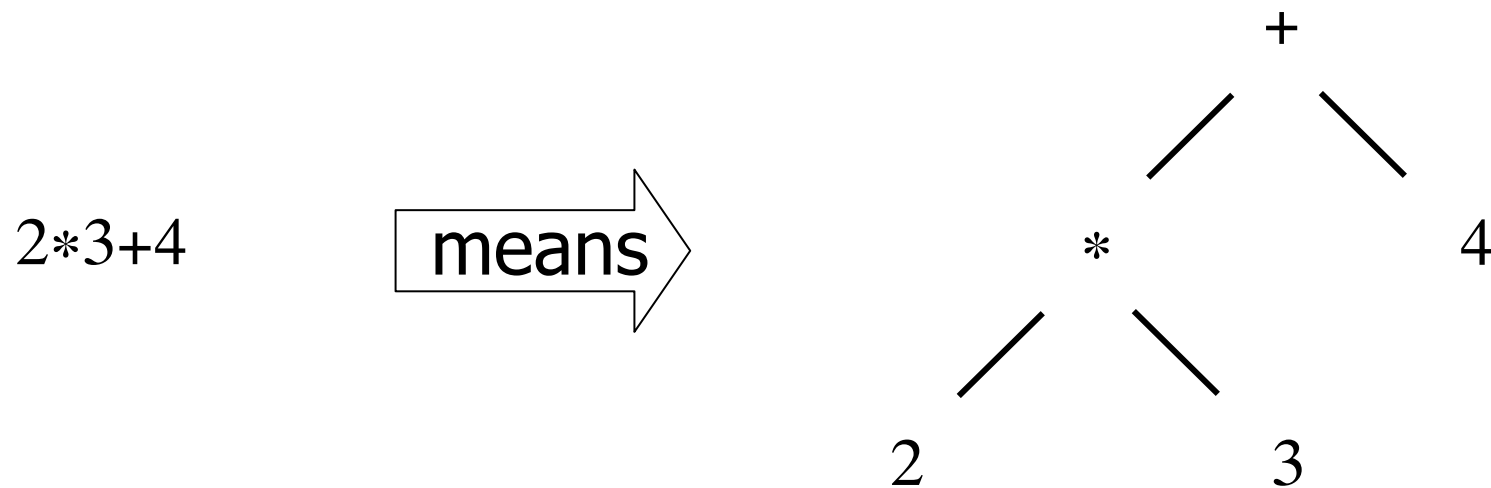
Where Are They Used?

Almost every real life program uses some form of parser to <u>pre-process</u> its input.

Hugs
Unix
Explorer

parses →

Haskell programs
Shell scripts
HTML documents
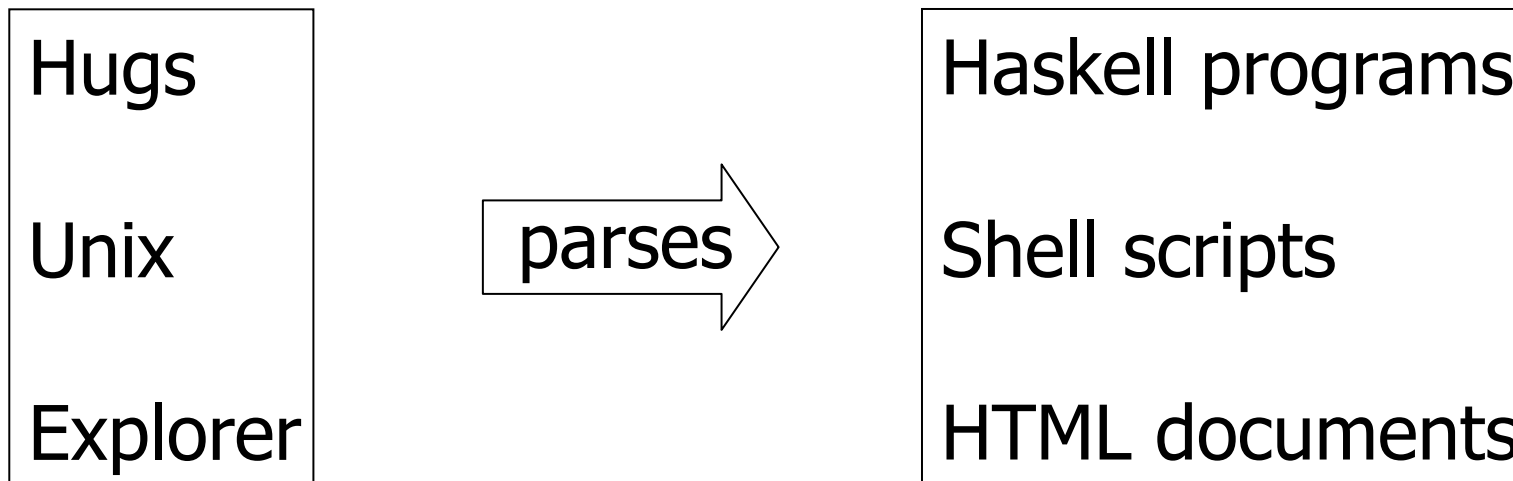
The Parser Type

In a functional language such as Haskell, parsers can naturally be viewed as <u>functions</u>.

$$\text{type Parser} = \text{String} \rightarrow \text{Tree}$$

A parser is a function that takes a string and returns some form of tree.

However, a parser might not require all of its input string, so we also return any <u>unused input</u>:

$$\text{type Parser} = \text{String} \rightarrow (\text{Tree,String})$$

A string might be parsable in many ways, including none, so we generalize to a <u>list of results</u>:

$$\text{type Parser} = \text{String} \rightarrow [(\text{Tree,String})]$$

193

Finally, a parser might not always produce a tree, so we generalize to a value of <u>any type</u>:

$$\text{type Parser a} = \text{String} \to [(a, \text{String})]$$

Note: For simplicity, we will only consider parsers that either fail and return the empty list of results, or succeed and return a <u>singleton list</u>.

Basic Parsers

The parser <u>item</u> fails if the input is empty, and consumes the first character otherwise:

item :: Parser Char

item = λinp → case inp of

$$[] \quad \rightarrow []$$

$$(x{:}xs) \rightarrow [(x,xs)]$$

z The parser <u>failure</u> always fails:

$$\text{failure} :: \text{Parser a}$$

$$\text{failure} = \lambda \text{inp} \rightarrow []$$

z The parser <u>return v</u> always succeeds, returning the value v without consuming any input:

$$\text{return} :: a \rightarrow \text{Parser a}$$

$$\text{return v} = \lambda \text{inp} \rightarrow [(v, \text{inp})]$$

196

The parser <u>p +++ q</u> behaves as the parser p if it succeeds, and as the parser q otherwise:

$$(+++) :: \text{Parser a} \to \text{Parser a} \to \text{Parser a}$$

$$p +++ q = \lambda inp \to \text{case p inp of}$$

$$[] \qquad \to \text{q inp}$$

$$[(v,out)] \to [(v,out)]$$

The function <u>parse</u> applies a parser to a string:

$$\text{parse} :: \text{Parser a} \to \text{String} \to [(a, \text{String})]$$

$$\text{parse p inp} = \text{p inp}$$

Examples

The behavior of the five parsing primitives can be illustrated with some simple <u>examples</u>:

```
% ghci Parsing

> parse item ""
[]

> parse item "abc"
[('a',"bc")]
```

> parse failure "abc"

[]

> parse (return 1) "abc"

[(1,"abc")]

> parse (item +++ return 'd') "abc"

[('a',"bc")]

> parse (failure +++ return 'd') "abc"

[('d',"abc")]

Note:

◉The library file <u>Parsing</u> is available on the web from the Programming in Haskell home page.

◉For technical reasons, the first failure example actually gives an error concerning <u>types</u>, but this does not occur in non-trivial examples.

◉The Parser type is a <u>monad</u>, a mathematical structure that has proved useful for modeling many different kinds of computations.

Sequencing

A sequence of parsers can be combined as a single composite parser using the keyword <u>do</u>.

For example:

$$p :: Parser\ (Char,Char)$$

$$p = do$$
$$\quad x \leftarrow item$$
$$\quad item$$
$$\quad y \leftarrow item$$
$$\quad return\ (x,y)$$

Note:

- Each parser must begin in precisely the same column. That is, the <u>layout rule</u> applies.

- The values returned by intermediate parsers are <u>discarded</u> by default, but if required can be named using the ← operator.

- The value returned by the <u>last</u> parser is the value returned by the sequence as a whole.

If any parser in a sequence of parsers <u>fails</u>, then the sequence as a whole fails. For example:

```
> parse p "abcdef"
[(('a','c'),"def")]

> parse p "ab"
[]
```

The do notation is not specific to the Parser type, but can be used with <u>any</u> monadic type.

204

Derived Primitives

Parsing a character that <u>satisfies</u> a predicate:

```
sat :: (Char → Bool) → Parser Char
sat p = do
          x ← item
          if p x then
            return x
          else
            failure
```

Parsing a <u>digit</u> and specific <u>characters</u>:

digit :: Parser Char

digit = sat isDigit

char :: Char → Parser Char

char x = sat (x ==)

Applying a parser <u>zero or more</u> times:

many :: Parser a → Parser [a]

many p = many1 p +++ return []

Applying a parser <u>one or more</u> times:

```
many1  :: Parser a –> Parser [a]
many1 p = do      v  ← p
                  vs ← many p
                  return (v:vs)
```

Parsing a specific <u>string</u> of characters:

```
string     :: String → Parser String
string []    = return []
string (x:xs) = do   char x
                     string xs
                     return (x:xs)
```

Example

We can now define a parser that consumes a list of one or more digits from a string:

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

For example:

> parse p "[1,2,3,4]"
[("1234","")]

> parse p "[1,2,3,4"
[]

Note: More sophisticated parsing libraries can indicate and/or recover from <u>errors</u> in the input string.

Arithmetic Expressions

Consider a simple form of <u>expressions</u> built up from single digits using the operations of addition + and multiplication *, together with parentheses.

We also assume that:

- * and + associate to the right;

- * has higher priority than +.

210

Formally, the syntax of such expressions is defined by the following context free <u>grammar</u>:

expr   → term '+' expr │ term

term   → factor '*' term │ factor

factor → digit │ '(' expr ')'

digit → '0' │ '1' │ … │ '9'

However, for reasons of efficiency, it is important to <u>factorise</u> the rules for expr and term:

$$expr \rightarrow term \ ('+' \ expr \ | \ \varepsilon)$$

$$term \rightarrow factor \ ('*' \ term \ | \ \varepsilon)$$

Note: The symbol ε denotes the empty string.

It is now easy to translate the grammar into a parser that <u>evaluates</u> expressions, by simply rewriting the grammar rules using the parsing primitives.

That is, we have:

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

```haskell
term :: Parser Int
term = do   f ← factor
            do      char '*'
                    t ← term
                    return (f * t)
            +++ return f
```

etc.

214

Finally, if we define

eval   :: String → Int
eval xs = fst (head (parse expr xs))

then we try out some examples:

> eval "2*3+4"
10

> eval "2*(3+4)"
14

Exercises

(1) Why does factorising the expression grammar make the resulting parser more efficient?

(2) Extend the expression parser to allow the use of subtraction and division, based upon the following extensions to the grammar:

$$expr \rightarrow term \ ('+' \ expr \ | \ '-' \ expr \ | \ \varepsilon)$$

$$term \rightarrow factor \ ('*' \ term \ | \ '/' \ term \ | \ \varepsilon)$$

Interactive Programs

Introduction

To date, we have seen how Haskell can be used to write <u>batch</u> programs that take all their inputs at the start and give all their outputs at the end.

inputs | batch program | outputs

218
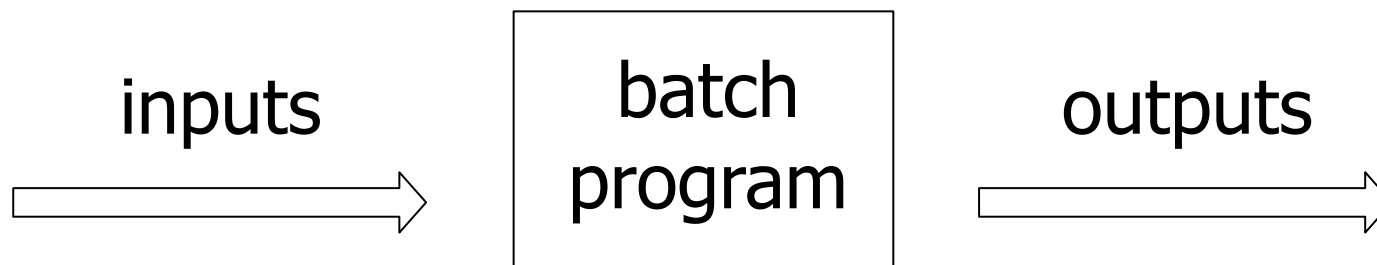
However, we would also like to use Haskell to write <u>interactive</u> programs that read from the keyboard and write to the screen, as they are running.

keyboard

inputs → interactive program → outputs
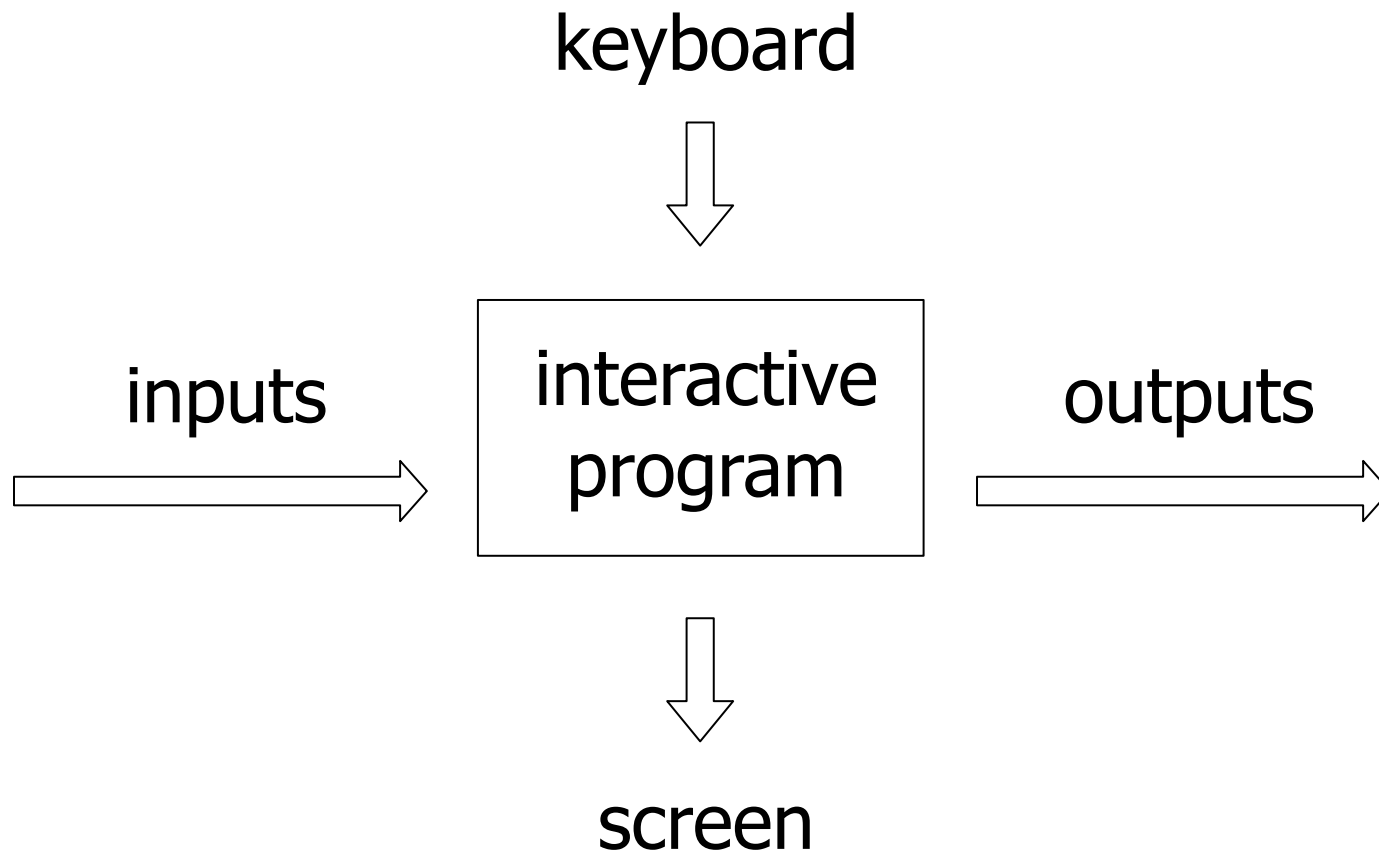
screen

The Problem

Haskell programs are pure mathematical functions:

Haskell programs <u>have no side effects</u>.
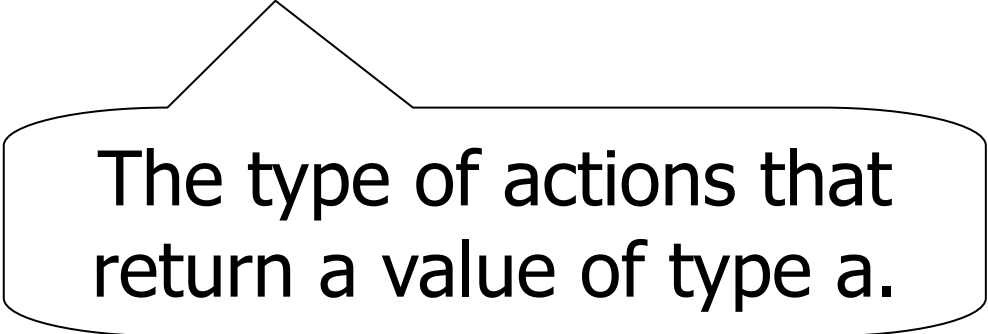
However, reading from the keyboard and writing to the screen are side effects:

Interactive programs <u>have side effects</u>.

The Solution

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure <u>actions</u> that may involve side effects.

IO a

The type of actions that return a value of type a.

For example:

IO Char

> The type of actions that return a character.

IO ()

> The type of purely side effecting actions that return <u>no</u> result value.

Note:

() is the type of tuples with no components.

Basic Actions

The standard library provides a number of actions, including the following three primitives:

The action <u>getChar</u> reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

getChar :: IO Char

The action <u>putChar c</u> writes the character c to the screen, and returns no result value:

$$putChar :: Char \to IO \ ()$$

The action <u>return v</u> simply returns the value v, without performing any interaction:

$$return :: a \to IO \ a$$

Sequencing

A sequence of actions can be combined as a single composite action using the keyword <u>do</u>.

For example:

```
a :: IO (Char,Char)

a = do
        x ← getChar
        getChar
        y ← getChar
        return (x,y)
```

Derived Primitives

Reading a string from the keyboard:

```
getLine :: IO String
getLine  = do
             x ← getChar
             if x == '\n' then
                 return []
             else
                 do xs ← getLine
                    return (x:xs)
```

Writing a string to the screen:

```
putStr    :: String → IO ()
putStr []   = return ()
putStr (x:xs) = do  putChar x
                    putStr xs
```

Writing a string and moving to a new line:

```
putStrLn  :: String → IO ()
putStrLn xs = do  putStr xs
                  putChar '\n'
```

227

Example

We can now define an action that prompts for a string to be entered and displays its length:

```
strlen :: IO ()
strlen = do  putStr "Enter a string: "
             xs ← getLine
             putStr "The string has "
             putStr (show (length xs))
             putStrLn " characters"
```

For example:

> strlen

Enter a string: abcde
The string has 5 characters

Note: Evaluating an action <u>executes</u> its side effects, with the final result value being discarded.

Hangman

Consider the following version of <u>hangman</u>:

◉ One player secretly types in a word.

◉ The other player tries to deduce the word, by entering a sequence of guesses.

◉ For each guess, the computer indicates which letters in the secret word occur in the guess.

⊙The game ends when the guess is correct.

We adopt a <u>top down</u> approach to implementing hangman in Haskell, starting as follows:

```haskell
hangman :: IO ()
hangman =
    do putStrLn "Think of a word: "
        word ← sgetLine
        putStrLn "Try to guess it:"
        guess word
```

The action <u>sgetLine</u> reads a line of text from the keyboard, echoing each character as a dash:

```
sgetLine :: IO String
sgetLine  = do
              x ← getCh
              if x == '\n' then
                do putChar x
                   return []
              else
                do putChar '–'
                   xs ← sgetLine
                   return (x:xs)
```

The function <u>guess</u> is the main loop, which requests and processes guesses until the game ends.

```
guess    :: String → IO ()
guess word =
  do   putStr "> "
       xs ← getLine
       if xs == word then
         putStrLn "You got it!"
       else
         do   putStrLn (diff word xs)
              guess word
```

The action <u>getCh</u> reads a character from the keyboard, without echoing it to the screen.

The function <u>diff</u> indicates which characters in one string occur in a second string:

```
diff    :: String → String → String
diff xs ys =
    [if elem x ys then x else '-' | x ← xs]
```

For example:

```
> diff "haskell" "pascal"
"-as--ll"
```

Exercise

Implement the game of <u>nim</u> in Haskell, where the rules of the game are as follows:

⦿ The board comprises five rows of stars:

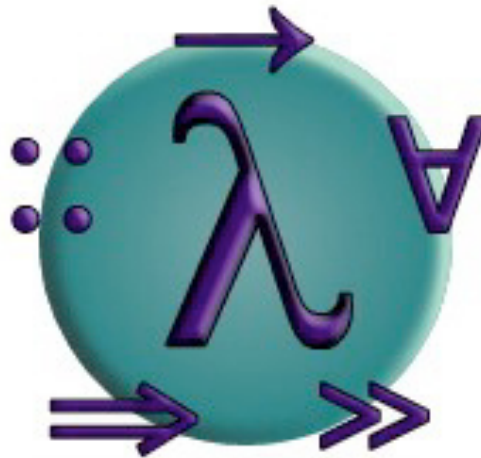1: * * * * *
2: * * * *
3: * * *
4: * *
5: *

235

⦿Two players take it turn about to remove one or more stars from the end of a single row.

⦿The winner is the player who removes the last star or stars from the board.

Hint:

Represent the board as a list of five integers that give the number of stars remaining on each row. For example, the initial board is [5,4,3,2,1].

The Countdown Problem

What Is Countdown?

◉A popular <u>quiz programme</u> on British television that has been running since 1982.

◉Based upon an original <u>French</u> version called "Des Chiffres et Des Lettres".

◉Includes a numbers game that we shall refer to as the <u>countdown problem</u>.

Example

Using the numbers

    1   3   7   10   25   50

and the arithmetic operators

    +   −   *   ÷

construct an expression whose value is 765

Rules

◉All the numbers, including intermediate results, must be <u>positive naturals</u> (1,2,3,…).

◉Each of the source numbers can be used at <u>most once</u> when constructing the expression.

◉We <u>abstract</u> from other rules that are adopted on television for pragmatic reasons.

For our example, one possible solution is

$$(25-10) * (50+1) \qquad = \qquad 765$$

Notes:

⦿ There are <u>780</u> solutions for this example.

⦿ Changing the target number to 831 gives an example that has <u>no</u> solutions.

A Prolog-based reference solution

% Solutions Ts for countdown problem with numbers Ns, result X

```
solve(Ns,X,Ts) :-
  findall(T,(
    sublist(L,Ns),
    permutation(L,P),
    compute(P,T),
    X is T
  ),Ts).
```

A Prolog-based reference solution cont'd

% Generate all sublists of a given list

```
sublist([],[]).
sublist(Z,[H|T]) :-
  sublist(Y,T),
  ( Z = Y
  ; Z = [H|Y]
  ).
```

A Prolog-based reference solution cont'd

% Generate all permutations of a given list

```
permutation([],[]).
permutation([H|T1],L) :-
  permutation(T1,T2),
  append(T2a,T2b,T2),
  append(T2a,[H|T2b],L).
```

244

A Prolog-based reference solution cont'd

% Complete sequences of numbers into arithmetic expressions

```
compute([R],R).
compute(As,T) :-
  append(As1,As2,As),
  As1 = [_|_],  As2 = [_|_],
  compute(As1,T1),
  compute(As2,T2),
  ( T = T1 + T2
  ; T = T1 - T2
  ; T = T1 * T2
  ; T = T1 / T2
  ),
  R is T,  R > 0, integer(R).
```

245

Evaluating Expressions

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
```

246

Decide if the result of applying an operator to two positive natural numbers is another such:

```
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
```

247

Return the overall value of an expression, provided that it is a positive natural number:

```
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]
```

Either succeeds and returns a singleton list, or fails and returns the empty list.

Formalising The Problem

Return a list of all possible ways of choosing zero or more elements from a list:

$$\text{choices} :: [a] \rightarrow [[a]]$$

For example:

$$> \text{choices } [1,2]$$

$$[[],[1],[2],[1,2],[2,1]]$$

Return a list of all the values in an expression:

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

Decide if an expression is a solution for a given list of source numbers and a target number:

```
solution      :: Expr → [Int] → Int → Bool
solution e ns n = elem (values e) (choices ns)
          && eval e == [n]
```

Brute Force Solution

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

split :: [a] → [([a],[a])]

For example:

> split [1,2,3,4]

[([1],[2,3,4]),([1,2],[3,4]),([1,2,3],[4])]

251

Return 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) ← split ns
             , l    ← exprs ls
             , r    ← exprs rs
             , e    ← combine l r]
```

The key function in this lecture.

Combine two expressions using each operator:

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

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

```
solutions   :: [Int] → Int → [Expr]
solutions ns n = [e | ns' ← choices ns
            , e  ← exprs ns'
            , eval e == [n]]
```

How Fast Is It?

System: 1.2GHz Pentium M laptop

Compiler: GHC version 6.4.1

Example: solutions [1,3,7,10,25,50] 765

One solution: 0.36 seconds

All solutions: 43.98 seconds

254

Can We Do Better?

- Many of the expressions that are considered will typically be <u>invalid</u> - fail to evaluate.

- For our example, only around <u>5 million</u> of the 33 million possible expressions are valid.

- Combining generation with evaluation would allow <u>earlier rejection</u> of invalid expressions.

Fusing Two Functions

Valid expressions and their values:

$$\text{type Result} = (\text{Expr,Int})$$

We seek to define a function that fuses together the generation and evaluation of expressions:

$$\text{results} :: [\text{Int}] \to [\text{Result}]$$
$$\text{results ns} = [(e,n) \mid e \leftarrow \text{exprs ns}$$
$$, n \leftarrow \text{eval e}]$$

This behaviour is achieved by defining

```
results []  = []
results [n] = [(Val n,n) | n > 0]
results ns  =
  [res | (ls,rs) ← split ns
       , lx     ← results ls
       , ry     ← results rs
       , res    ← combine' lx ry]
```

where

```
combine' :: Result → Result → [Result]
```

Combining results:

```
combine' (l,x) (r,y) =
  [(App o l r, apply o x y)
    | o ← [Add,Sub,Mul,Div]
    , valid o x y]
```
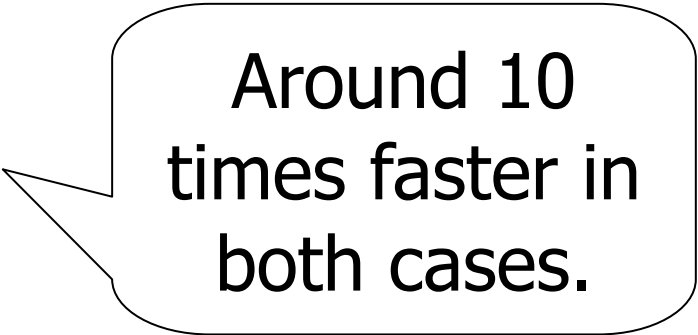
New function that solves countdown problems:

```
solutions'    :: [Int] → Int → [Expr]
solutions' ns n =
  [e | ns'  ← choices ns
    , (e,m) ← results ns'
    , m == n]
```

How Fast Is It Now?

Example: solutions' [1,3,7,10,25,50] 765

One solution: 0.04 seconds

Around 10 times faster in both cases.

All solutions: 3.47 seconds

259

Can We Do Better?

⦿ Many expressions will be <u>essentially the same</u> using simple arithmetic properties, such as:

$$x * y \quad = \quad y * x$$

$$x * 1 \quad = \quad x$$

⦿ Exploiting such properties would considerably <u>reduce</u> the search and solution spaces.

Exploiting Properties

Strengthening the valid predicate to take account of commutativity and identity properties:

valid   :: Op $\to$ Int $\to$ Int $\to$ Bool

valid Add x y   = True $\boxed{x \leq y}$

valid Sub x y   = x > y

valid Mul x y   = True $\boxed{x \leq y \;\&\&\; x \neq 1 \;\&\&\; y \neq 1}$

valid Div x y   = x `mod` y == 0 $\boxed{\&\& \; y \neq 1}$

How Fast Is It Now?

Example: solutions'' [1,3,7,10,25,50] 765

Valid: 250,000 expressions   *Around 20 times less.*

Solutions: 49 expressions   *Around 16 times less.*

262

One solution: 0.02 seconds

> Around 2 times faster.

All solutions: 0.44 seconds

> Around 7 times faster.

More generally, our program usually produces a solution to problems from the television show in an instant, and all solutions in under a second.

263

End of Introduction to Haskell