x = 1

let x = 1 in ...

x(1).

!x(1)

x.set(1)

Programming Paradigms and Formal Semantics

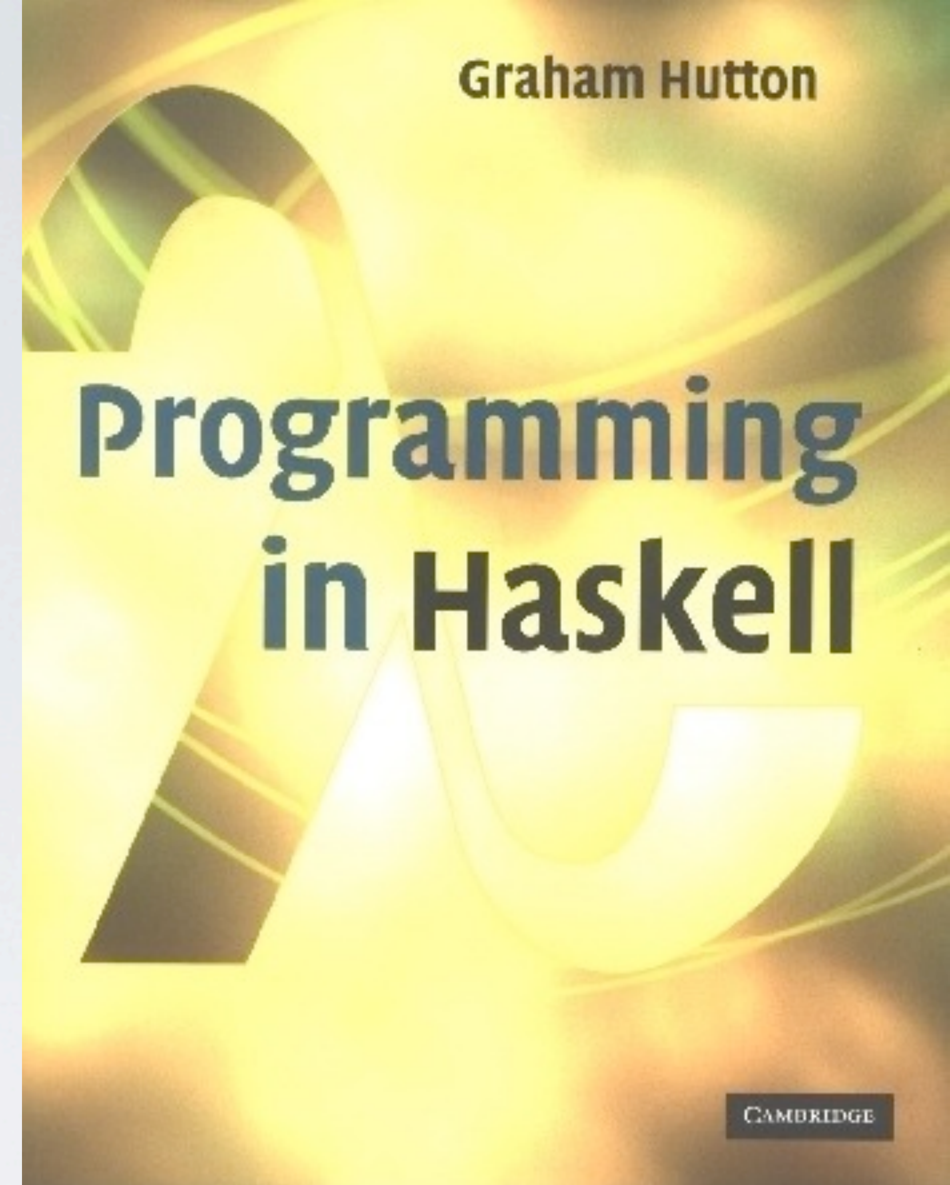
Introduction to Haskell

Ralf Lämmel

Programming in Haskell

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

A weekly series of freely available video lectures 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).

© Ralf Lämmel, 2009-2011 unless noted otherwise

What is a Functional Language?

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Example

Summing the integers 1 to 10 in Java:

```
total = 0;
for (i = 1; i ≤ 10; ++i)
   total = total+i;
```

The computation method is variable assignment.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Example

Summing the integers 1 to 10 in Haskell:

```
sum [1..10]
```

The computation method is function application.

© Ralf Lämmel, 2009-2011 unless noted otherwise

A Taste of Haskell

```
f []     = []
f (x:xs) = f ys ++ [x] ++ f zs
           where
             ys = [a | a ← xs, a ≤ x]
             zs = [b | b ← xs, b > x]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Historical Background

© Ralf Lämmel, 2009-2011 unless noted otherwise

Historical Background

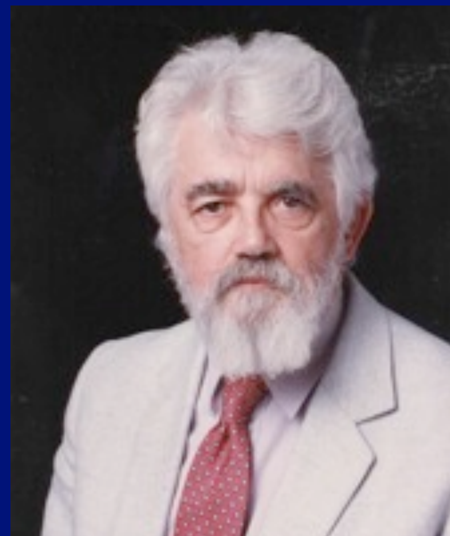
1930s:

Alonzo Church develops the lambda calculus, a simple but powerful theory of functions.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Historical Background

1950s:



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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Historical Background

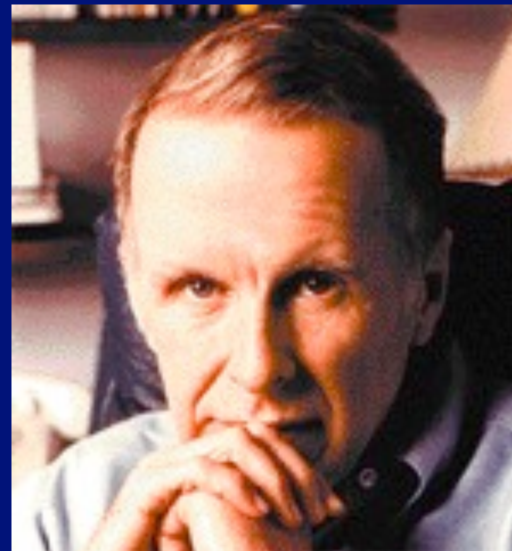
1960s:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Historical Background

1970s:



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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Historical Background

1970s:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Historical Background

1970s - 1980s:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Historical Background

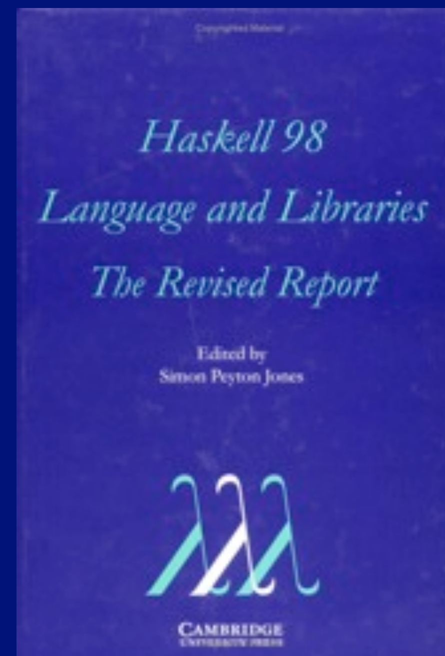
1987:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Historical Background

2003:



The committee publishes the Haskell 98 report, defining a stable version of the language.

© Ralf Lämmel, 2009-2011 unless noted otherwise

First Steps in Haskell

© Ralf Lämmel, 2009-2011 unless noted otherwise

Haskell systems

- http://www.haskell.org/
- Major option: GHC

  http://haskell.org/ghc/download.html

© Ralf Lämmel, 2009-2011 unless noted otherwise

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>
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

The Standard Prelude

The library file Prelude.hs 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 lists.

Select the first element of a list:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Function Application

In mathematics, 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

In Haskell, function application is denoted using space, and multiplication is denoted using *.

```
f a b + c*d
```

As previously, but in Haskell syntax.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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).

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Haskell Scripts

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

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

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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:

- div is enclosed in back quotes, not forward;
- x `f` y is just syntactic sugar for f x y.

© Ralf Lämmel, 2009-2011 unless noted otherwise

The interpreter does not automatically detect that the script has been changed, so a reload 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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Naming Requirements

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

```
myFun    fun1    arg_2    x'
```

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

```
xs    ns    nss
```

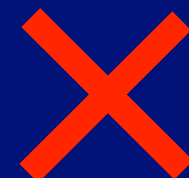
© Ralf Lämmel, 2009-2011 unless noted otherwise

The Layout Rule

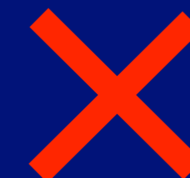
In a sequence of definitions, each definition must begin in precisely the same column:

```
a = 10

b = 20

c = 30
```

```
a = 10

 b = 20

c = 30
```



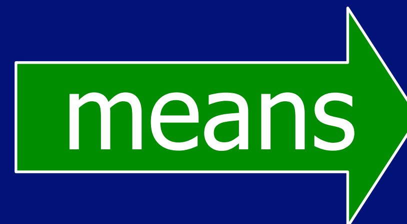
```
  a = 10

b = 20

  c = 30
```



© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
a = b + c
  where
    b = 1
    c = 2
d = a * 2
```




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

implicit grouping

explicit grouping

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

(3) Show how the library function last 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 init that removes the last element from a list can be defined in two different ways.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Common Types

© Ralf Lämmel, 2009-2011 unless noted otherwise

What is a Type?

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

```
Bool
```

contains the two logical values:

```
False        True
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Type Errors

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

```
> 1 + False
Error
```

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Types in Haskell

- If evaluating an expression e would produce a value of type t, then e has type t, written

```
e :: t
```

- Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference.

© Ralf Lämmel, 2009-2011 unless noted otherwise

- All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.

- In the Haskell interpreter, the :type command calculates the type of an expression, without evaluating it:

```
> not False
True

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Basic Types

Haskell has a number of basic types, including:

Bool	- logical values
Char	- single characters
String	- strings of characters
Int	- fixed-precision integers
Integer	- arbitrary-precision integers
Float	- floating-point numbers

© Ralf Lämmel, 2009-2011 unless noted otherwise

List Types

A list is sequence of values of the same type:

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

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

In general:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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]]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Tuple Types

A tuple is a sequence of values of different 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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])
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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;

© Ralf Lämmel, 2009-2011 unless noted otherwise

Exercises

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

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

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

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

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Functions types

© Ralf Lämmel, 2009-2011 unless noted otherwise

Function Types

A function is a mapping from values of one type to values of another type:

```
not     :: Bool → Bool

isDigit :: Char → Bool
```

In general:

t1 → t2 is the type of functions that map values of type t1 to values to type t2.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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       :: (Int,Int) → Int
add (x,y)  = x+y

zeroto    :: Int → [Int]
zeroto n   = [0..n]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

```
add'    :: Int → (Int → Int)
add' x y = x+y
```

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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:

```
add  :: (Int,Int) → Int

add' :: Int → (Int → 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Why is Currying Useful?

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

For example:

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Currying Conventions

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

The arrow → associates to the right.

Int → Int → Int → Int

Means Int → (Int → (Int → Int)).

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
mult x y z
```

Means ((mult x) y) z.

Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Polymorphic Functions

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

```
length :: [a] → Int
```

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Note:

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

```
> length [False,True]
2

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

a = Bool

a = Int

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
fst  :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip  :: [a] → [b] → [(a,b)]

id   :: a → a
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Overloaded Functions

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

$$sum :: Num\ a \Rightarrow [a] \rightarrow a$$

for any numeric type a, sum takes a list of values of type a and returns a value of type a.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Note:

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

```
> sum [1,2,3]
6

> sum [1.1,2.2,3.3]
6.6

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

a = Int

a = Float

Char is not a numeric type

© Ralf Lämmel, 2009-2011 unless noted otherwise

This slide deck is derived from G. Hutton's deck for his book "Programming in Haskell",

Haskell has a number of type classes, including:

- **Num** - Numeric types
- **Eq** - Equality types
- **Ord** - Ordered types

For example:

```
(+)  :: Num a ⇒ a → a → a
(==) :: Eq a  ⇒ a → a → Bool
(<)  :: Ord a ⇒ a → a → Bool
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Hints and Tips

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Exercises

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

```
[tail,init,reverse]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

(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)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Defining Functions

© Ralf Lämmel, 2009-2011 unless noted otherwise

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.

```
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.

© Ralf Lämmel, 2009-2011 unless noted 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 always have an else branch, which avoids any possible ambiguity problems with nested conditionals.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

```
abs n | n ≥ 0     = n
      | otherwise = -n
```

As previously, but using guarded equations.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
signum n | n < 0      = -1
         | n == 0     = 0
         | otherwise  = 1
```

Note:

The catch all condition otherwise is defined in the prelude by otherwise = True.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.

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

not maps False to True, and True to False.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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 wildcard pattern that matches any argument value.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

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

Patterns may not repeat variables. For example, the following definition gives an error:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

List Patterns

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

[1,2,3,4]

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Functions on lists can be defined using x:xs 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Note:

x:xs patterns only match non-empty lists:

```
> head []
Error
```

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

```
head x:_ = x
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Integer Patterns

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

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

Banned in Haskell 2010

pred maps any positive integer to its predecessor.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Note:

n+k patterns only match integers ≥ k.

```
> pred 0
Error
```

Banned in Haskell 2010

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

```
pred n+1 = n
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Lambda Expressions

Functions can be constructed without naming the functions by using lambda expressions.

$\lambda x \rightarrow x+x$

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Note:

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

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:

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

means

$$add = \lambda x \rightarrow (\lambda y \rightarrow x+y)$$

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Lambda expressions can be used to avoid naming functions that are only referenced once.

For example:

```
odds n = map f [0..n-1]
      where
        f x = x*2 + 1
```

can be simplified to

```
odds n = map (λx → x*2 + 1) [0..n-1]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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 ⊕ is an operator then functions of the form (⊕), (x⊕) and (⊕y) are called <u>sections</u>.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Exercises

(1) Consider a function safetail 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] → Bool can be used to test if a list is empty.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

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

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

(4) Do the same for the following version:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

List Comprehensions

© Ralf Lämmel, 2009-2011 unless noted otherwise

Set Comprehensions

In mathematics, the comprehension 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}.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Lists Comprehensions

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

$$[x\text{^}2 \mid x \leftarrow [1..5]]$$

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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) | x ← [1,2,3], y ← [4,5]]

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

```
> [(x,y) | y ← [4,5], x ← [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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Dependant Generators

Later generators can depend 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
concat     :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← 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:

```
> concat [[1,2,3],[4,5],[6]]

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Guards

List comprehensions can use guards 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
factors  :: Int → [Int]
factors n =
  [x | x ← [1..n], n `mod` x == 0]
```

For example:

```
> factors 15

[1,3,5,15]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

The Zip Function

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

```
zip :: [a] → [b] → [(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)]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
pairs    :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

For example:

```
> pairs [1,2,3,4]

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
sorted   :: Ord a ⇒ [a] → Bool
sorted xs =
  and [x ≤ y | (x,y) ← pairs xs]
```

For example:

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

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
positions :: Eq a ⇒ a → [a] → [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]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

String Comprehensions

A string 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].

© Ralf Lämmel, 2009-2011 unless noted otherwise

Because strings are just special kinds of lists, any polymorphic 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)]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
pyths :: Int → [(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)]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
perfects :: Int → [Int]
```

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

```
> perfects 500

[6,28,496]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

(3) The scalar product 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Recursive Functions

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Recursive Functions

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

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

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

For example:

```
  factorial 3
=
  3 * factorial 2
=
  3 * (2 * factorial 1)
=
  3 * (2 * (1 * factorial 0))
=
  3 * (2 * (1 * 1))
=
  3 * (2 * 1)
=
  3 * 2
=
  6
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Note:

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

- The recursive definition diverges on integers < 0 because the base case is never reached:

```
> factorial (−1)

Error: Control stack overflow
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Why is Recursion Useful?

- Some functions, such as factorial, are simpler to define in terms of other functions.

- As we shall see, however, many functions can naturally be defined in terms of themselves.

- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Recursion on Lists

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

```
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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

For example:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
length          :: [a] → Int
length []       = 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

For example:

```
  length [1,2,3]
=
  1 + length [2,3]
=
  1 + (1 + length [3])
=
  1 + (1 + (1 + length []))
=
  1 + (1 + (1 + 0))
=
  3
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Using a similar pattern of recursion we can define the reverse function on lists.

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

For example:

```
   reverse [1,2,3]
=
   reverse [2,3] ++ [1]
=
   (reverse [3] ++ [2]) ++ [1]
=
   ((reverse [] ++ [3]) ++ [2]) ++ [1]
=
   (([] ++ [3]) ++ [2]) ++ [1]
=
   [3,2,1]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Quicksort

The quicksort 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

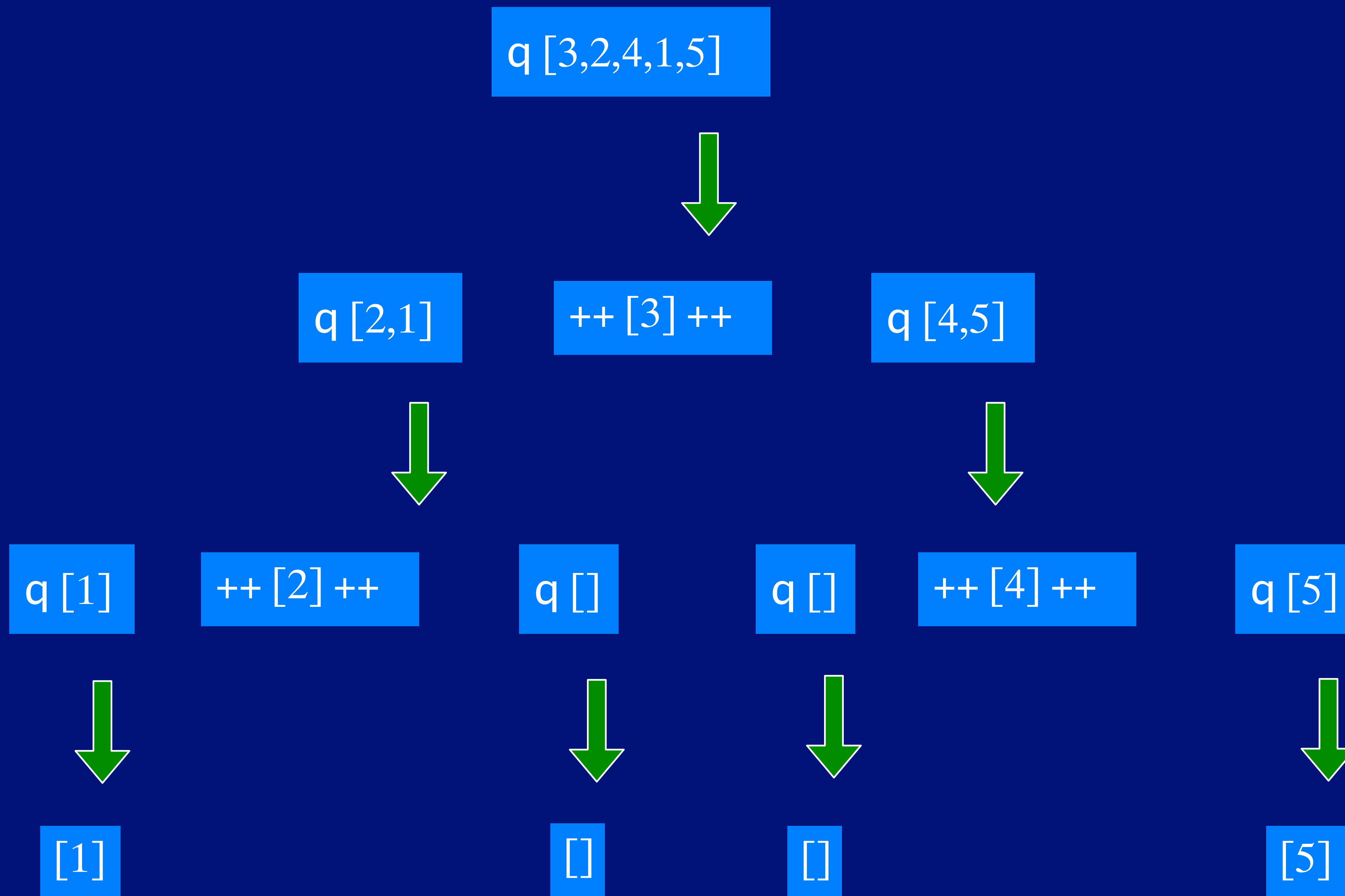
```
qsort        :: [Int] → [Int]
qsort []     = []
qsort (x:xs) =
  qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a ← xs, a ≤ x]
    larger  = [b | b ← xs, b > x]
```

Note:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

For example (abbreviating qsort as q):



© Ralf Lämmel, 2009-2011 unless noted otherwise

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] → Bool
```

Concatenate a list of lists:

```
concat :: [[a]] → [a]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Produce a list with n identical elements:

```
replicate :: Int → a → [a]
```

Select the nth element of a list:

```
(!!) :: [a] → Int → a
```

Decide if a value is an element of a list:

```
elem :: Eq a ⇒ a → [a] → Bool
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

(2) Define a recursive function

```
merge :: [Int] → [Int] → [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]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

(3) Define a recursive function

```
msort :: [Int] → [Int]
```

that implements merge sort, 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Higher-Order Functions

© Ralf Lämmel, 2009-2011 unless noted otherwise

Introduction

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

```
twice :: (a → a) → a → a
twice f x = f (f x)
```

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Why Are They Useful?

- Common programming idioms can be encoded as functions within the language itself.
- Domain specific languages can be defined as collections of higher-order functions.
- Algebraic properties of higher-order functions can be used to reason about programs.

© Ralf Lämmel, 2009-2011 unless noted otherwise

The Map Function

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

```
map :: (a → b) → [a] → [b]
```

For example:

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

[2,4,6,8]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
map f xs = [f x | x ← xs]
```

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

```
map f []     = []
map f (x:xs) = f x : map f xs
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

The Filter Function

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

```
filter :: (a → Bool) → [a] → [a]
```

For example:

```
> filter even [1..10]

[2,4,6,8,10]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Filter can be defined using a list comprehension:

```
filter p xs = [x | x ← xs, 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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

The Foldr Function

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

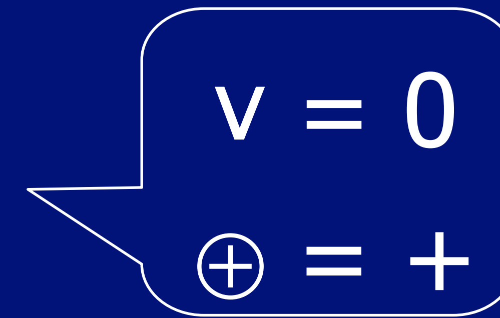
```
f []     = v
f (x:xs) = x ⊕ f xs
```

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

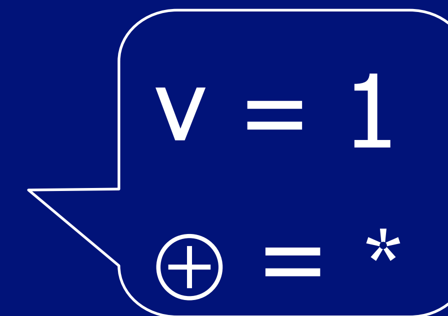
© Ralf Lämmel, 2009-2011 unless noted otherwise

For example:

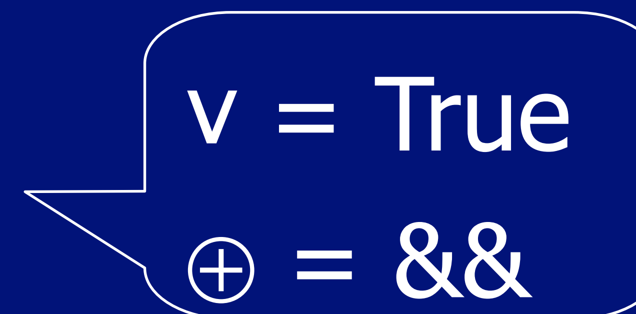
```
sum [] = 0
sum (x:xs) = x + sum xs
```




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




```
and [] = True
and (x:xs) = x && and xs
```



© Ralf Lämmel, 2009-2011 unless noted otherwise

The higher-order library function foldr (fold right) encapsulates this simple pattern of recursion, with the function ⊕ and the value v as arguments.

For example:

```
sum      = foldr (+) 0

product  = foldr (*) 1

or       = foldr (||) False

and      = foldr (&&) True
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Foldr itself can be defined using recursion:

```
foldr :: (a → b → b) → b → [a] → b

foldr f v [] = v

foldr f v (x:xs) = f x (foldr f v xs)
```

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

For example:

```
  sum [1,2,3]
=
  foldr (+) 0 [1,2,3]
=
  foldr (+) 0 (1:(2:(3:[])))
=
  1+(2+(3+0))
=
  6
```

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

For example:

```
    length [1,2,3]
=
    length (1:(2:(3:[])))
=
    1+(1+(1+0))
=
    3
```

Replace each (:) by λ_ n → 1+n and [] by 0.

Hence, we have:

```
length = foldr (λ_ n → 1+n) 0
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Now recall the reverse function:

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

For example:

```
  reverse [1,2,3]
=
  reverse (1:(2:(3:[])))
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

Replace each (:) by λx xs → xs ++ [x] and [] by [].

© Ralf Lämmel, 2009-2011 unless noted otherwise

Hence, we have:

```
reverse =
  foldr (λx xs → 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Why Is Foldr Useful?

- Some recursive functions on lists, such as sum, are simpler to define using foldr.

- Properties of functions defined using foldr can be proved using algebraic properties of foldr, such as fusion and the banana split rule.

- Advanced program optimisations can be simpler if foldr is used in place of explicit recursion.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Other Library Functions

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

```
(.)   :: (b → c) → (a → b) → (a → c)
f . g  = λx → f (g x)
```

For example:

```
odd :: Int → Bool
odd  = not . even
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
all      :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

For example:

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

True
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
any      :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

For example:

```
> any isSpace "abc def"

True
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

The library function takeWhile 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"
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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"
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Type Declarations

© Ralf Lämmel, 2009-2011 unless noted otherwise

Type Declarations

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

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

String is a synonym for the type [Char].

© Ralf Lämmel, 2009-2011 unless noted otherwise

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)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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)
```

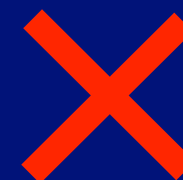
© Ralf Lämmel, 2009-2011 unless noted otherwise

Type declarations can be nested:

```
type Pos   = (Int,Int)

type Trans = Pos → Pos
```

However, they cannot be recursive:

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



© Ralf Lämmel, 2009-2011 unless noted otherwise

Data Declarations

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

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

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
data Maybe a = Nothing | Just a
```

we can define:

```
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Recursive Types

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

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

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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)

⋮
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

For example, the value

```
Succ (Succ (Succ Zero))
```

represents the natural number

$$1 + (1 + (1 + 0)) = 3$$

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
nat2int              :: Nat → Int
nat2int Zero         = 0
nat2int (Succ n)     = 1 + nat2int n

int2nat              :: Int → Nat
int2nat 0            = Zero
int2nat (n+1)        = Succ (int2nat n)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
add :: Nat → Nat → 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)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

For example:

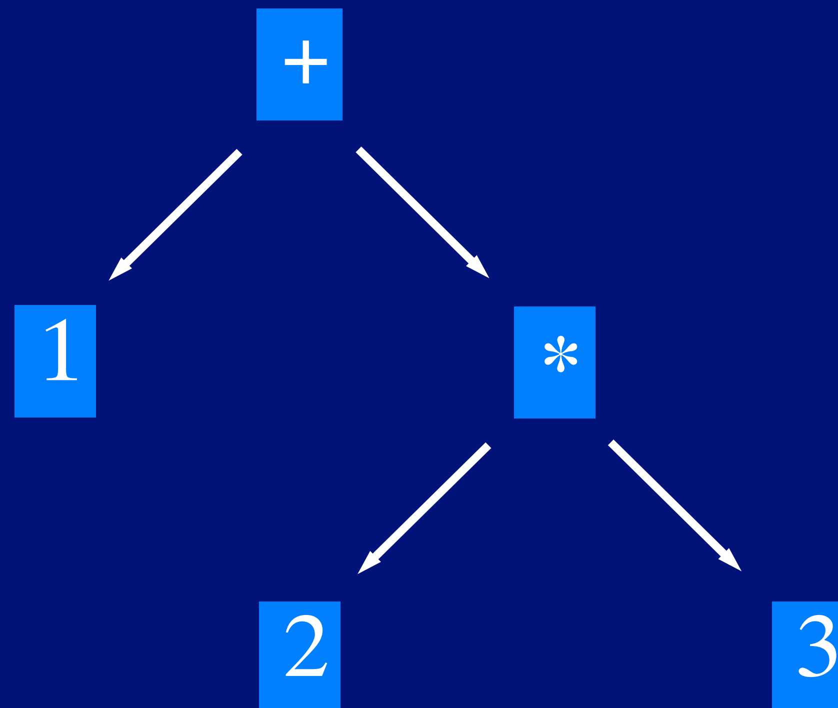
```
  add (Succ (Succ Zero)) (Succ Zero)
=
  Succ (add (Succ Zero) (Succ Zero))
=
  Succ (Succ (add Zero (Succ Zero))
=
  Succ (Succ (Succ Zero))
```

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Arithmetic Expressions

Consider a simple form of expressions built up from integers using addition and multiplication.



© Ralf Lämmel, 2009-2011 unless noted otherwise

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))
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

On the types of constructors

The three constructors have types:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

A fold for expressions

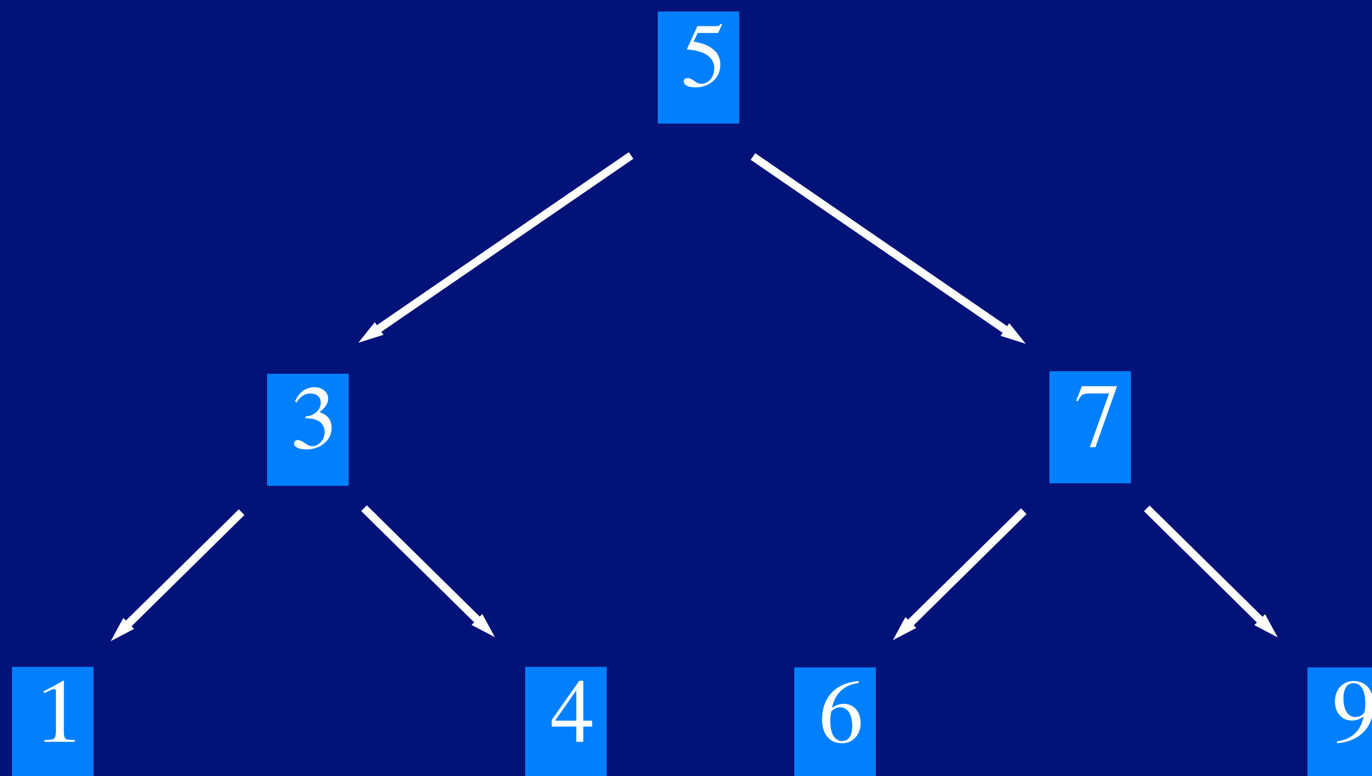
Many functions on expressions can be defined by replacing the constructors by other functions using a suitable fold function. For example:

```
eval = fold id (+) (*)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Binary Trees

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



© Ralf Lämmel, 2009-2011 unless noted otherwise

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))
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

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

A tree is a search tree 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].

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Exercises

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

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

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

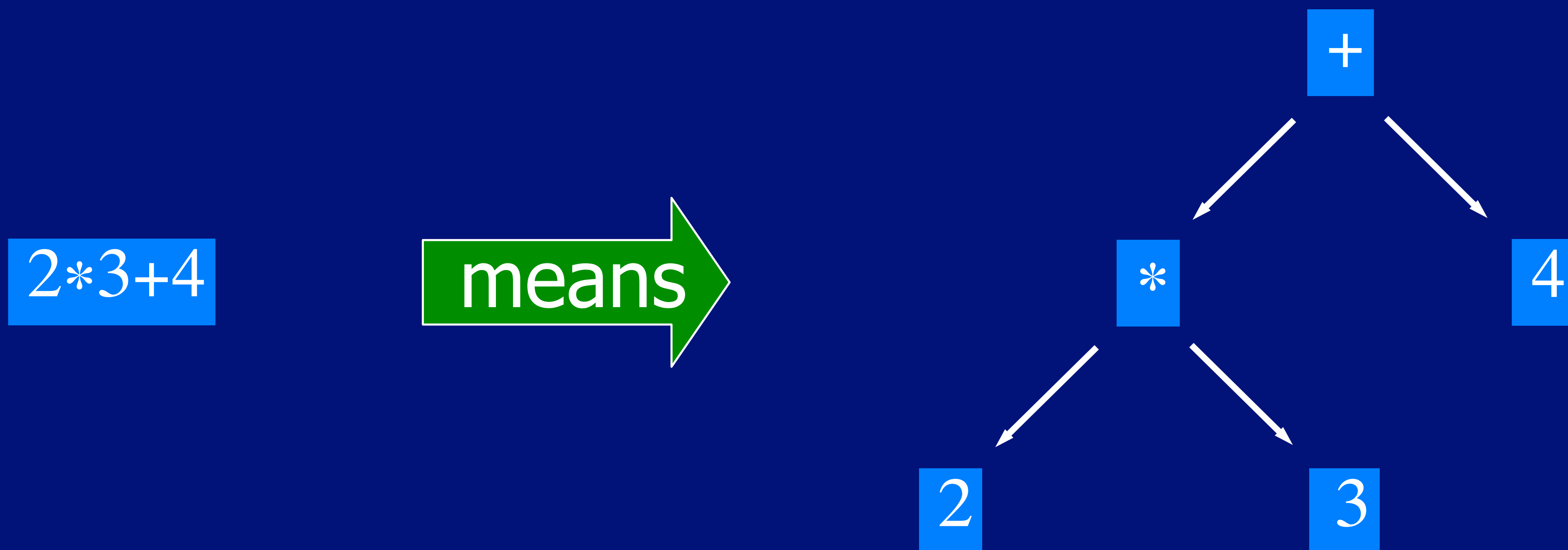
© Ralf Lämmel, 2009-2011 unless noted otherwise

Functional Parsers

© Ralf Lämmel, 2009-2011 unless noted otherwise

What is a Parser?

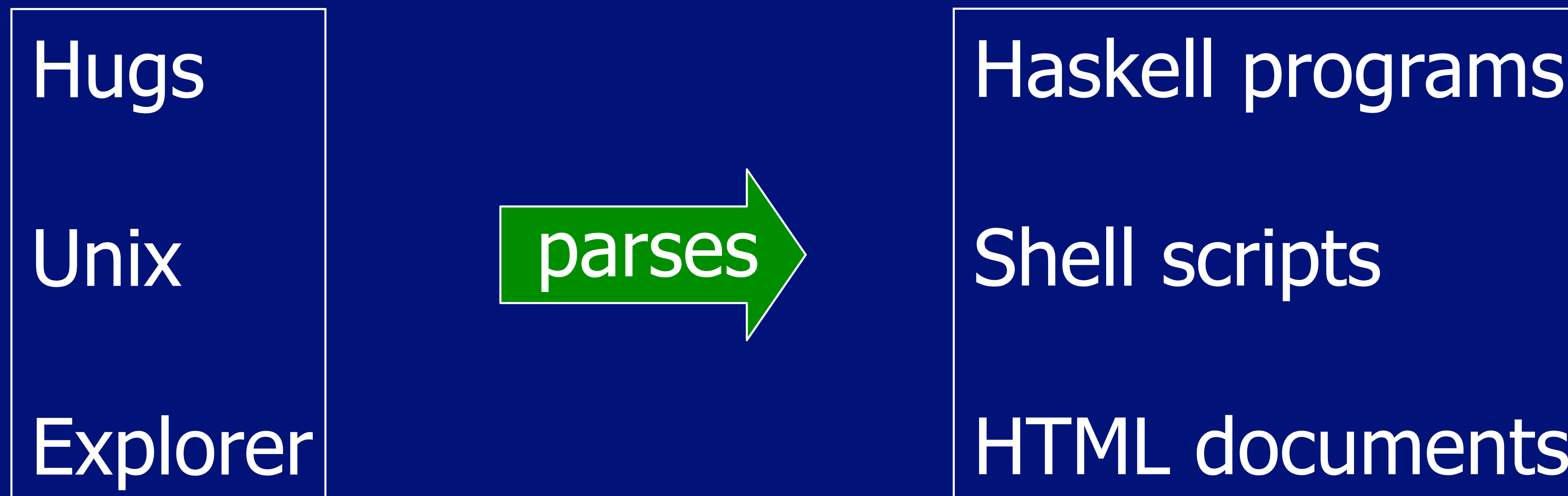
A parser is a program that analyses a piece of text to determine its syntactic structure.



© Ralf Lämmel, 2009-2011 unless noted otherwise

Where Are They Used?

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



© Ralf Lämmel, 2009-2011 unless noted otherwise

The Parser Type

In a functional language such as Haskell, parsers can naturally be viewed as functions.

```
type Parser = String → Tree
```

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
type Parser = String → (Tree,String)
```

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

```
type Parser = String → [(Tree,String)]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
type Parser a = String → [(a,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>.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Basic Parsers

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

```
item :: Parser Char

item  = λinp → case inp of
               []     → []
               (x:xs) → [(x,xs)]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
failure :: Parser a
failure  = λinp → []
```

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

```
return  :: a → Parser a
return v = λinp → [(v,inp)]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

The parser p +++ q behaves as the parser p if it succeeds, and as the parser q otherwise:

```
(+++)  :: Parser a → Parser a → Parser a
p +++ q = λinp → case p inp of
                   []        → q inp
                   [(v,out)] → [(v,out)]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

The function parse applies a parser to a string:

```
parse :: Parser a → String → [(a,String)]
parse p inp = p inp
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Examples

The behavior of the five parsing primitives can be illustrated with some simple examples:

```
% ghci Parsing

> parse item ""
[]

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

```
> parse failure "abc"
[]

> parse (return 1) "abc"
[(1,"abc")]

> parse (item +++ return 'd') "abc"
[('a',"bc")]

> parse (failure +++ return 'd') "abc"
[('d',"abc")]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Note:

- The library file Parsing is available on the web from the Programming in Haskell home page.

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

- The Parser type is a monad, a mathematical structure that has proved useful for modeling many different kinds of computations.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Sequencing

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

For example:

```
p :: Parser (Char,Char)
p  = do
        x ← item
        item
        y ← item
        return (x,y)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Note:

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

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

- The value returned by the last parser is the value returned by the sequence as a whole.

© Ralf Lämmel, 2009-2011 unless noted otherwise

If any parser in a sequence of parsers fails, 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 any monadic type.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Derived Primitives

Parsing a character that satisfies a predicate:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Parsing a digit and specific characters:

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

char  :: Char → Parser Char
char x = sat (x ==)
```

Applying a parser zero or more times:

```
many  :: Parser a → Parser [a]
many p = many1 p +++ return []
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Applying a parser one or more times:

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

Parsing a specific string of characters:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Arithmetic Expressions

Consider a simple form of expressions 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 +.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Formally, the syntax of such expressions is defined by the following context free grammar:

```
expr   → term '+' expr | term

term   → factor '*' term | factor

factor → digit | '(' expr ')‘

digit  → '0' | '1' | ... | '9'
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

expr → term ('+' expr | ε)

term → factor ('*' term | ε)

Note: The symbol ε denotes the empty string.

© Ralf Lämmel, 2009-2011 unless noted otherwise

It is now easy to translate the grammar into a parser that evaluates 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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

etc.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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:

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

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Interactive Programs

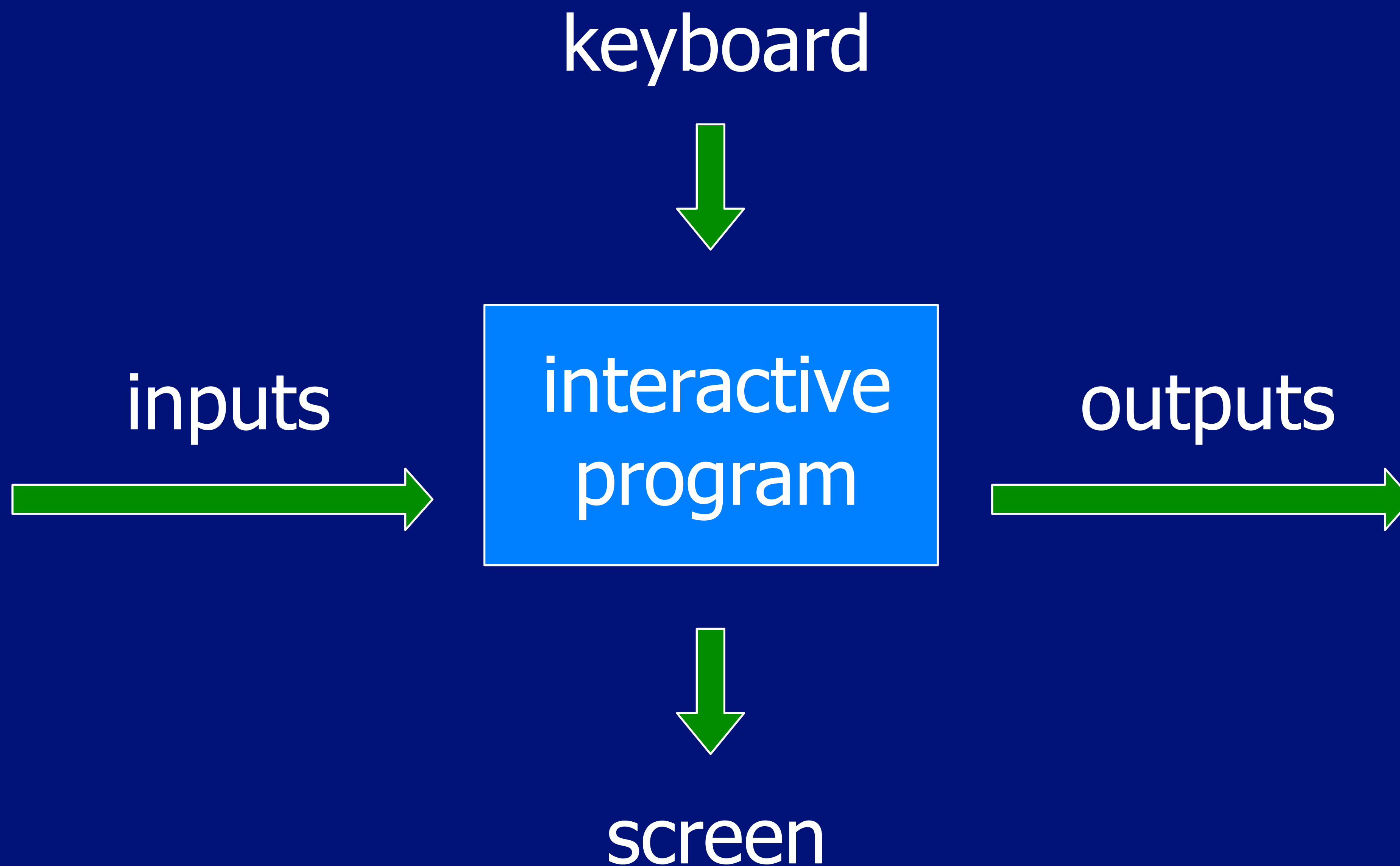
© Ralf Lämmel, 2009-2011 unless noted otherwise

Introduction

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.



© Ralf Lämmel, 2009-2011 unless noted otherwise

The Problem

Haskell programs are pure mathematical functions:

Haskell programs have no side effects.

However, reading from the keyboard and writing to the screen are side effects:

Interactive programs have side effects.

© Ralf Lämmel, 2009-2011 unless noted otherwise

The Solution

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

IO a

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

For example:

IO Char

The type of actions that return a character.

IO ()

The type of purely side effecting actions that return no result value.

Note:

() is the type of tuples with no components.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Basic Actions

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

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

```
getChar :: IO Char
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

```
putChar :: Char → IO ()
```

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

```
return :: a → IO a
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Sequencing

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

For example:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

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)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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'
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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"
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Hangman

Consider the following version of hangman:

- 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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

- The game ends when the guess is correct.

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

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

The action sgetLine 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)
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

The function guess 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 getCh reads a character from the keyboard, without echoing it to the screen.

© Ralf Lämmel, 2009-2011 unless noted otherwise

The function diff 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"
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

Exercise

Implement the game of nim in Haskell, where the rules of the game are as follows:

- The board comprises five rows of stars:

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

- 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].

© Ralf Lämmel, 2009-2011 unless noted otherwise

The Countdown Problem

© Ralf Lämmel, 2009-2011 unless noted otherwise

What Is Countdown?

- A popular quiz programme on British television that has been running since 1982.

- Based upon an original French version called "Des Chiffres et Des Lettres".

- Includes a numbers game that we shall refer to as the countdown problem.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Example

Using the numbers

1 3 7 10 25 50

and the arithmetic operators

+ – * ÷

construct an expression whose value is 765

© Ralf Lämmel, 2009-2011 unless noted otherwise

Rules

- All the numbers, including intermediate results, must be positive naturals (1,2,3,…).
- Each of the source numbers can be used at most once when constructing the expression.
- We abstract from other rules that are adopted on television for pragmatic reasons.

© Ralf Lämmel, 2009-2011 unless noted otherwise

For our example, one possible solution is

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

Notes:

- There are 780 solutions for this example.
- Changing the target number to 831 gives an example that has no solutions.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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).
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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]
  ).
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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).
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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).
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Formalising The Problem

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

```
choices :: [a] → [[a]]
```

For example:

```
> choices [1,2]

[[],[1],[2],[1,2],[2,1]]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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])]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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]]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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

© Ralf Lämmel, 2009-2011 unless noted otherwise

Can We Do Better?

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

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

- Combining generation with evaluation would allow earlier rejection of invalid expressions.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Fusing Two Functions

Valid expressions and their values:

```
type Result = (Expr,Int)
```

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

```
results    :: [Int] → [Result]
results ns = [(e,n) | e ← exprs ns
                    , n ← eval e]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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]
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

How Fast Is It Now?

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

One solution: 0.04 seconds

All solutions: 3.47 seconds

Around 10 times faster in both cases.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Can We Do Better?

- Many expressions will be essentially the same using simple arithmetic properties, such as:

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

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

- Exploiting such properties would considerably reduce the search and solution spaces.

© Ralf Lämmel, 2009-2011 unless noted otherwise

Exploiting Properties

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

```
valid          :: Op → Int → Int → Bool
valid Add x y  = True   x ≤ y
valid Sub x y  = x > y
valid Mul x y  = True   x ≤ y && x ≠ 1 && y ≠ 1
valid Div x y  = x `mod` y == 0   && y ≠ 1
```

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

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.

© Ralf Lämmel, 2009-2011 unless noted otherwise

End of Introduction to Haskell

© Ralf Lämmel, 2009-2011 unless noted otherwise