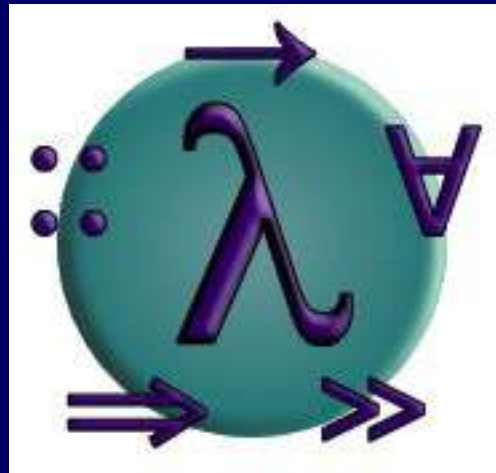


PROGRAMMING IN HASKELL



Chapter 1 - Introduction

The Software Crisis

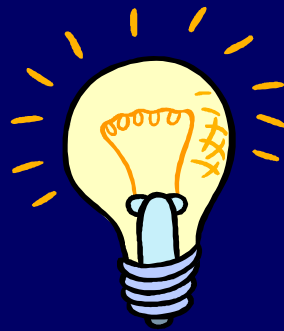
- How can we cope with the size and complexity of modern computer programs?
- How can we reduce the time and cost of program development?
- How can we increase our confidence that the finished programs work correctly?

Programming Languages

One approach to the software crisis is to design new programming languages that:

- Allow programs to be written clearly, concisely, and at a high-level of abstraction;
- Support reusable software components;
- Encourage the use of formal verification;

- Permit rapid prototyping;
- Provide powerful problem-solving tools.



Functional languages provide a particularly elegant framework in which to address these goals.

What is a Functional Language?

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

- Functional programming is style of programming in which the basic method of computation is the application of functions to arguments;
- A functional language is one that supports and encourages 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 variable assignment.

Example

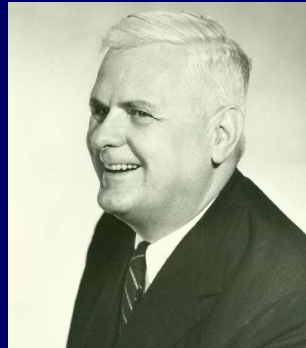
Summing the integers 1 to 10 in Haskell:

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

The computation method is function application.

Historical Background

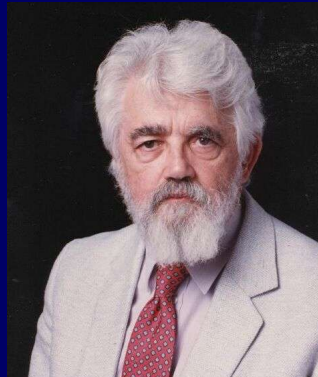
1930s:



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

Historical Background

1950s:



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

Historical Background

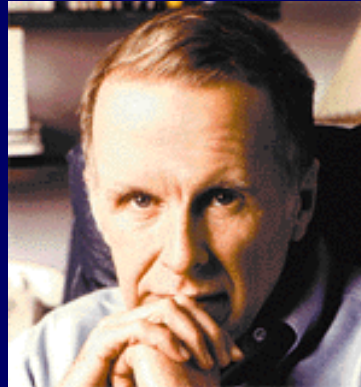
1960s:



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

Historical Background

1970s:



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

Historical Background

1970s:



Robin Milner and others develop ML, 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.

Historical Background

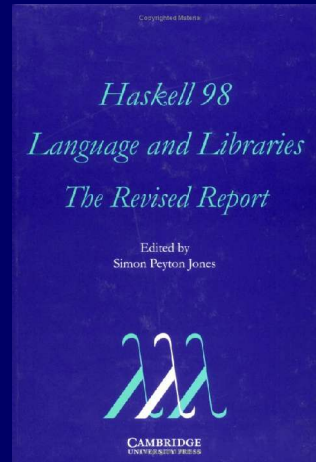
1987:



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

Historical Background

2003:



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

A Taste of Haskell

```
f [] = []
```

```
f (x:xs) = f ys ++ [x] ++ f zs
```

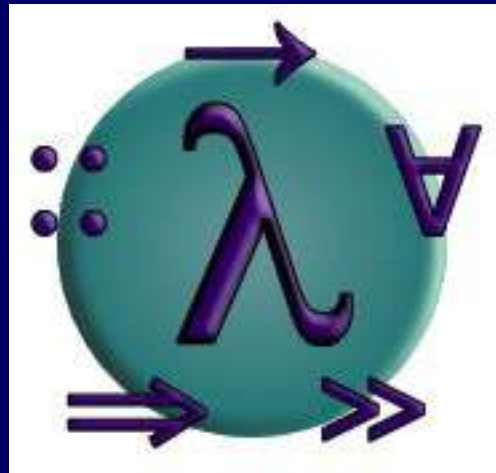
where

```
ys = [a | a ← xs, a ≤ x]
```

```
zs = [b | b ← xs, b > x]
```



PROGRAMMING IN HASKELL



Chapter 2 - First Steps

The Hugs System

- Hugs is an implementation of Haskell 98, and is the most widely used Haskell system;
- The interactive nature of Hugs makes it well suited for teaching and prototyping purposes;
- Hugs is available on the web from:

www.haskell.org/hugs

Starting Hugs

On a Unix system, Hugs can be started from the % prompt by simply typing hugs:

```
% hugs
```

```
  _   _   _   _   _   _  
  ||   ||  ||  ||  ||  ||  ||  
  ||___||  ||___||  ||___||  ||  
  ||---||      ___||  
  ||   ||  
  ||   ||
```

```
Hugs 98: Based on the Haskell 98 standard  
Copyright (c) 1994-2005  
World Wide Web: http://haskell.org/hugs  
Report bugs to: hugs-bugs@haskell.org
```

```
>
```

The Hugs `>` prompt means that the Hugs 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 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
```

- 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 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 .

In Haskell, 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 higher priority than all other operators.

$f\ a\ +\ b$

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

Examples

Mathematics

$f(x)$

$f(x, y)$

$f(g(x))$

$f(x, g(y))$

$f(x)g(y)$

Haskell

`f x`

`f x y`

`f (g x)`

`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 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.

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 Hugs with the new script:

```
% hugs 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]
```

Leaving Hugs 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`.

Hugs 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
```

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

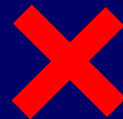
The Layout Rule

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

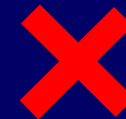
```
a = 10
b = 20
c = 30
```



```
a = 10
  b = 20
c = 30
```

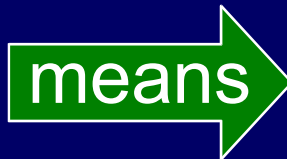


```
a = 10
b = 20
  c = 30
```



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

Useful Hugs Commands

<u>Command</u>	<u>Meaning</u>
:load <i>name</i>	load script <i>name</i>
:reload	reload current script
:edit <i>name</i>	edit script <i>name</i>
:edit	edit current script
:type <i>expr</i>	show type of <i>expr</i>
:?	show all commands
:quit	quit Hugs

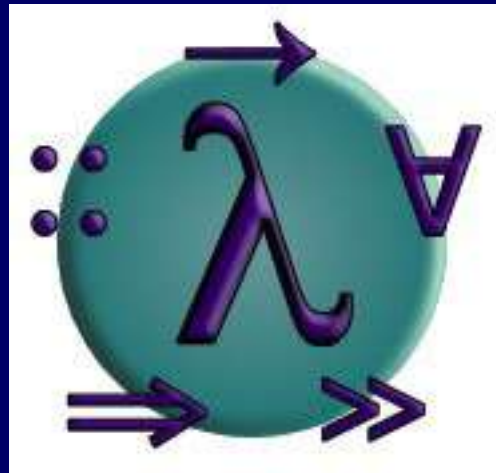
Exercises

- (1) Try out slides 2-8 and 14-17 using Hugs.
- (2) Fix the syntax errors in the program below, and test your solution using Hugs.

```
N = a 'div' length xs
  where
    a = 10
    xs = [1,2,3,4,5]
```

- (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.

PROGRAMMING IN HASKELL



Chapter 3 - Types and Classes

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

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.

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.

- 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 Hugs, the :type command calculates the type of an expression, without evaluating it:

```
> not False
True

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

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

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.

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 tuple is a sequence of values of different types:

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

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

In general:

(t_1, t_2, \dots, t_n) is the type of n -tuples whose i th components have type t_i for any i in $1 \dots n$.

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])
```

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:

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

Note:

- The arrow \rightarrow 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]
```

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 $x+y$.

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 curried functions, celebrating the work of Haskell Curry on such functions.

- Functions with more than two arguments can be carried 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 $x*y*z$.

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]
```

Currying Conventions

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

- The arrow \rightarrow associates to the right.

$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

Means $\text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))$.

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

```
mult x y z
```

Means $((\text{mult } x) y) z$.

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

Polymorphic Functions

A function is called 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.

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:

```
fst  :: (a,b) → a
```

```
head :: [a] → a
```

```
take :: Int → [a] → [a]
```

```
zip  :: [a] → [b] → [(a,b)]
```

```
id   :: a → a
```

Overloaded Functions

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

```
sum :: Num a => [a] -> 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
```

```
> sum [1.1,2.2,3.3]  
6.6
```

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

a = Int

a = Float

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:

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

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

```
(<) :: Ord a => a -> a -> Bool
```

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

Exercises

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

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

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

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

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

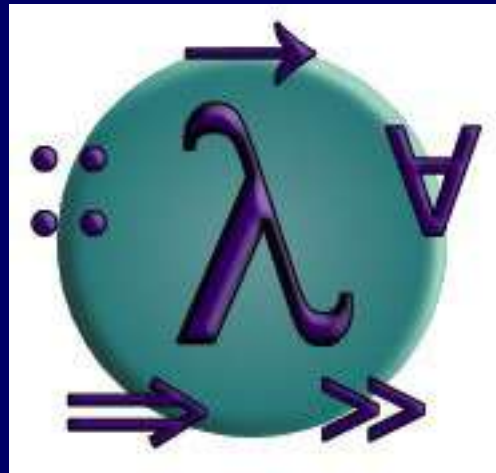
```
[tail, init, reverse]
```

(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)
```

(3) Check your answers using Hugs.

PROGRAMMING IN HASKELL



Chapter 4 - Defining Functions

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.

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.

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.

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`.

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.

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

```
( $\&\&$ )      :: Bool  $\rightarrow$  Bool  $\rightarrow$  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 wildcard pattern that matches any argument value.

- 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
```

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:[])))`.

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.

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
```

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
```

pred maps any positive integer to its predecessor.

Note:

- $n+k$ patterns only match integers $\geq k$.

```
> pred 0  
Error
```

- $n+k$ patterns must be parenthesised, 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 lambda expressions.

$\lambda x \rightarrow x+x$

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

Note:

- The symbol λ is the Greek letter lambda, and is typed at the keyboard as a backslash `\`.
- In mathematics, nameless functions are usually denoted using the λ symbol, as in $x \rightarrow x+x$.
- 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.

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

Lambda expressions are also useful when defining functions that return functions as results.

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 referenced once.

For example:

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

can be simplified to

```
odds n = map ( $\lambda x \rightarrow x*2 + 1$ ) [0..n-1]
```

Sections

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

For example:

```
> 1+2
```

```
3
```

```
> (+) 1 2
```

```
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses.

For example:

$$\begin{array}{l} > (1+) 2 \\ 3 \\ \\ > (+2) 1 \\ 3 \end{array}$$

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

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

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.

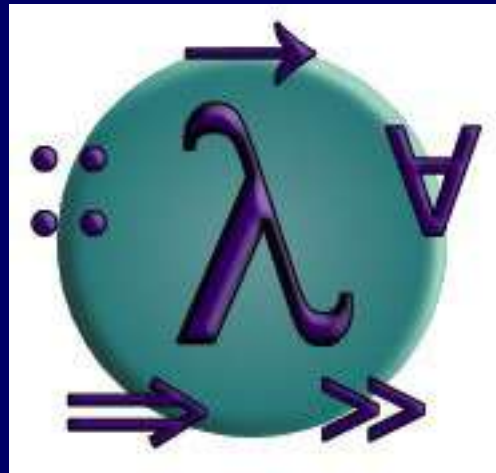
- (2) Give three possible definitions for the logical or operator (||) using pattern matching.
- (3) Redefine the following version of (&&) using conditionals rather than patterns:

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

- (4) Do the same for the following version:

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

PROGRAMMING IN HASKELL



Chapter 5 - List Comprehensions

Set Comprehensions

In mathematics, the comprehension notation can be used to construct new sets from old sets.

$$\{x^2 \mid x \in \{1\dots 5\}\}$$

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

Lists Comprehensions

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

```
[x^2 | x ← [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 $x \leftarrow [1..5]$ is called a generator, as it states how to generate values for x .
- Comprehensions can have multiple 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)]
```

- Changing the order of the generators changes the order of the elements in the final list:

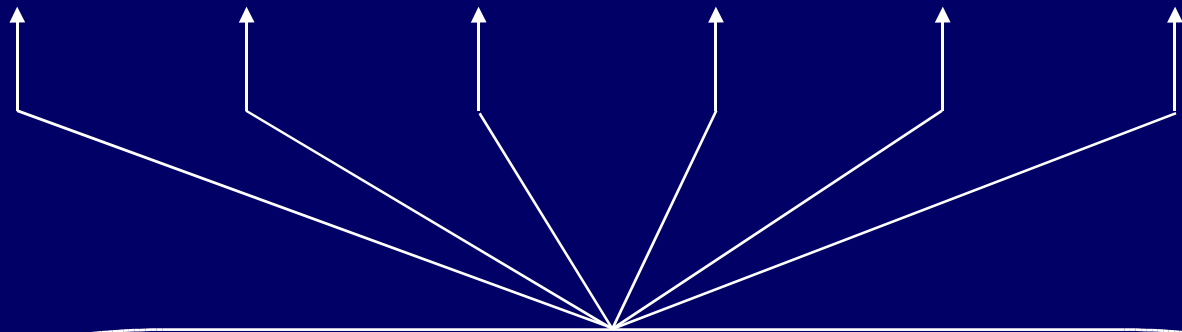
```
> [(x,y) | y ← [4,5], x ← [1,2,3]]  
[(1,4), (2,4), (3,4), (1,5), (2,5), (3,5)]
```

- Multiple generators are like nested loops, with later generators as more deeply nested loops whose variables change value more frequently.

■ For example:

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

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



$x \leftarrow [1,2,3]$ is the last generator, so the value of the x component of each pair changes most frequently.

Dependant Generators

Later generators can depend on the variables that are introduced by earlier generators.

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

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

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

For example:

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

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x ← [1..10], 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.

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]
```

A positive integer is prime 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 primes 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 `zip`, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] → [b] → [(a,b)]
```

For example:

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

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)]
```


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

```
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
```

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]
```

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].

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)]
```

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:

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

For example:

```
> Towers "Haske11"
6
```

Exercises

- (1) A triple (x,y,z) of positive integers is called pythagorean 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)]
```

(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
```

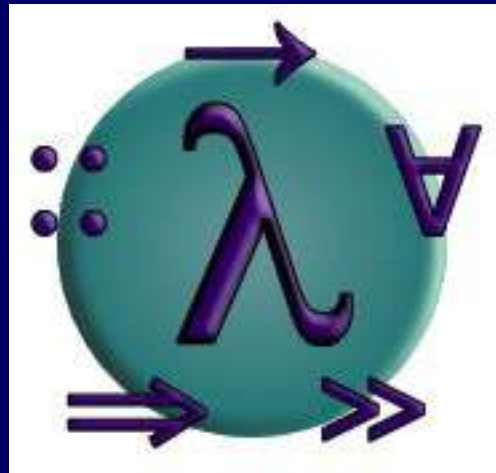
```
[6, 28, 496]
```

- (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.

PROGRAMMING IN HASKELL



Chapter 6 - Recursive Functions

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 evaluated 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 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.

For example:

$$\begin{aligned} &= \text{factorial } 3 \\ &= 3 * \text{factorial } 2 \\ &= 3 * (2 * \text{factorial } 1) \\ &= 3 * (2 * (1 * \text{factorial } 0)) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

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
```

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.

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.

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 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.

For example:

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

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.

For example:

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

■ 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 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 \leq the head, sorting the tail values $>$ the head, and then appending the resulting lists on either side of the head value.

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 simplest 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] → Bool
```

- Concatenate a list of lists:

```
concat :: [[a]] → [a]
```

- 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
```

(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]
```

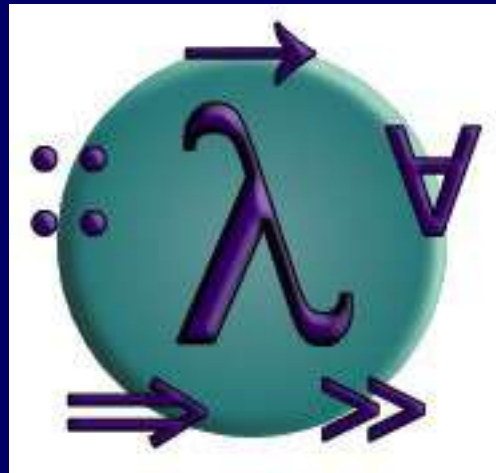
(3) Define a recursive function

```
mmerge sort :: [Int] → [Int]
```

that implements merge sort, which can be specified by the following two rules:

- Lists of length ≤ 1 are already sorted;
- Other lists can be sorted by sorting the two halves and merging the resulting lists.

PROGRAMMING IN HASKELL



Chapter 7 - Higher-Order Functions

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.

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.

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]
```

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
```

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]
```

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
```

The Foldr Function

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

$$\begin{aligned} f [] &= v \\ f (x:xs) &= x \oplus f xs \end{aligned}$$

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.

For example:

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

$v = 0$
 $\oplus = +$

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

$v = 1$
 $\oplus = *$

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

$v = \text{True}$
 $\oplus = \&\&$

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

For example:

```
sum      = foldr (+) 0
product = foldr (*) 1
or       = foldr (||) False
and      = foldr (&&) True
```


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.

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.

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:

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

For example:

$$\begin{aligned} & \text{length } [1,2,3] \\ = & \text{length } (1:(2:(3:[]))) \\ = & 1+(1+(1+0)) \\ = & 3 \end{aligned}$$

Hence, we have:

$$\text{length} = \text{foldr } (\lambda_n \rightarrow 1+n) 0$$

Replace each (:) by $\lambda_n \rightarrow 1+n$ and [] by 0.

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
 $\lambda x xs \rightarrow xs ++ [x]$
and [] by [].

Hence, we have:

```
reverse =  
  foldr (\x 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`.

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.

Other Library Functions

The library function `(.)` returns the composition 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
```

The library function `all` 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
```

Dually, the library function any 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
```

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"
```

Dually, the function dropWhile 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
```

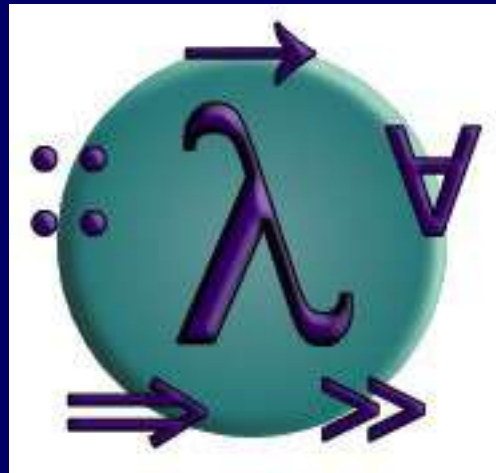
For example:

```
> dropWhile isSpace "  abc"
"abc"
```

Exercises

- (1) What are higher-order functions that return functions as results better known as?
- (2) Express the comprehension $[f\ x \mid x \leftarrow xs, p\ x]$ using the functions `map` and `filter`.
- (3) Redefine `map` `f` and `filter` `p` using `foldr`.

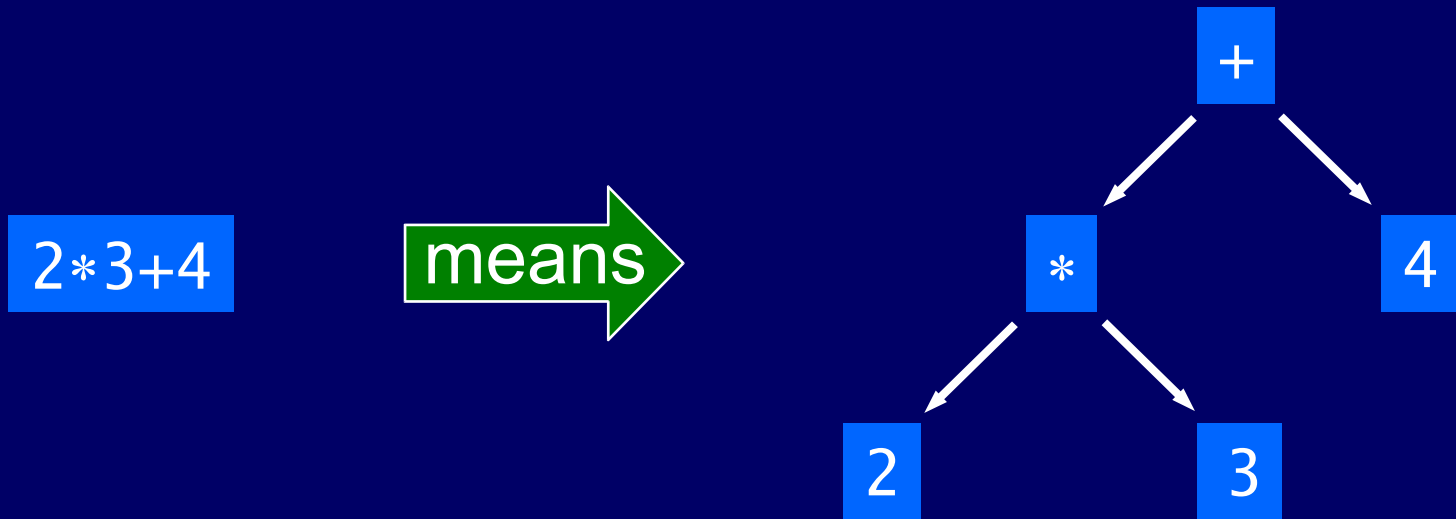
PROGRAMMING IN HASKELL



Chapter 8 - Functional Parsers

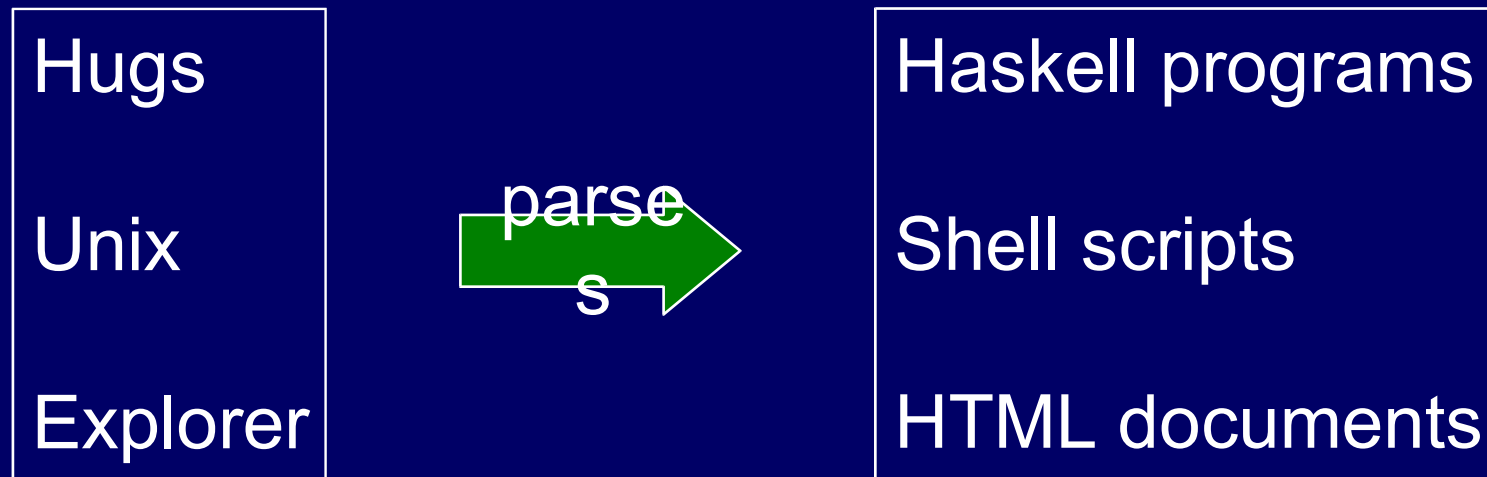
What is a Parser?

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



Where Are They Used?

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



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.

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)]
```

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

```
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 singleton list.

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)]
```

- The parser failure always fails:

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

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

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

- The parser $p \text{ +++ } q$ 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
    []      → parse q inp
    [(v,out)] → [(v,out)]
```

- The function parse applies a parser to a string:

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

Examples

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

```
% hugs 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 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.

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)
```

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

- 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.

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
```

- 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 []
```

- 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)
```


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 errors in the input string.

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 $+$.

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'
```

However, for reasons of efficiency, it is important to factorise the rules for *expr* and *term*:

$$\text{expr} \rightarrow \text{term} ('+' \text{ expr} \mid \varepsilon)$$
$$\text{term} \rightarrow \text{factor} ('*' \text{ term} \mid \varepsilon)$$

Note:

- The symbol ε denotes the empty string.

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
```

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

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

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
```

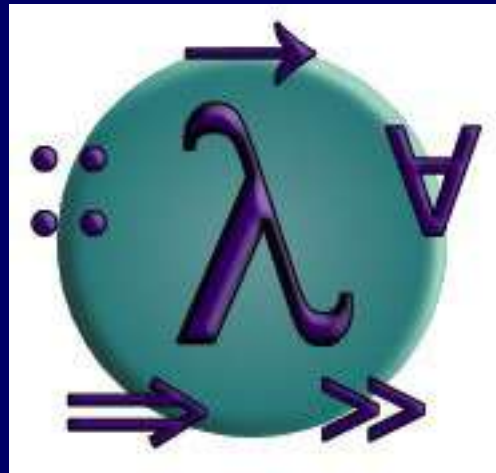

Exercises

- (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 → term ('+' expr | '-' expr | ε)
```

```
term → factor ('*' term | '/' term | ε)
```

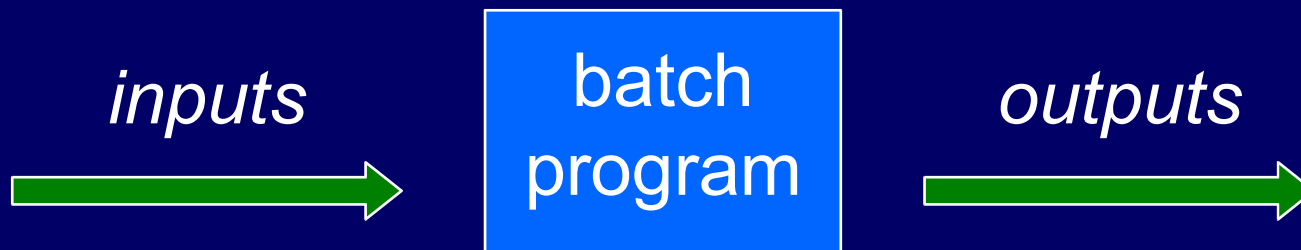
PROGRAMMING IN HASKELL



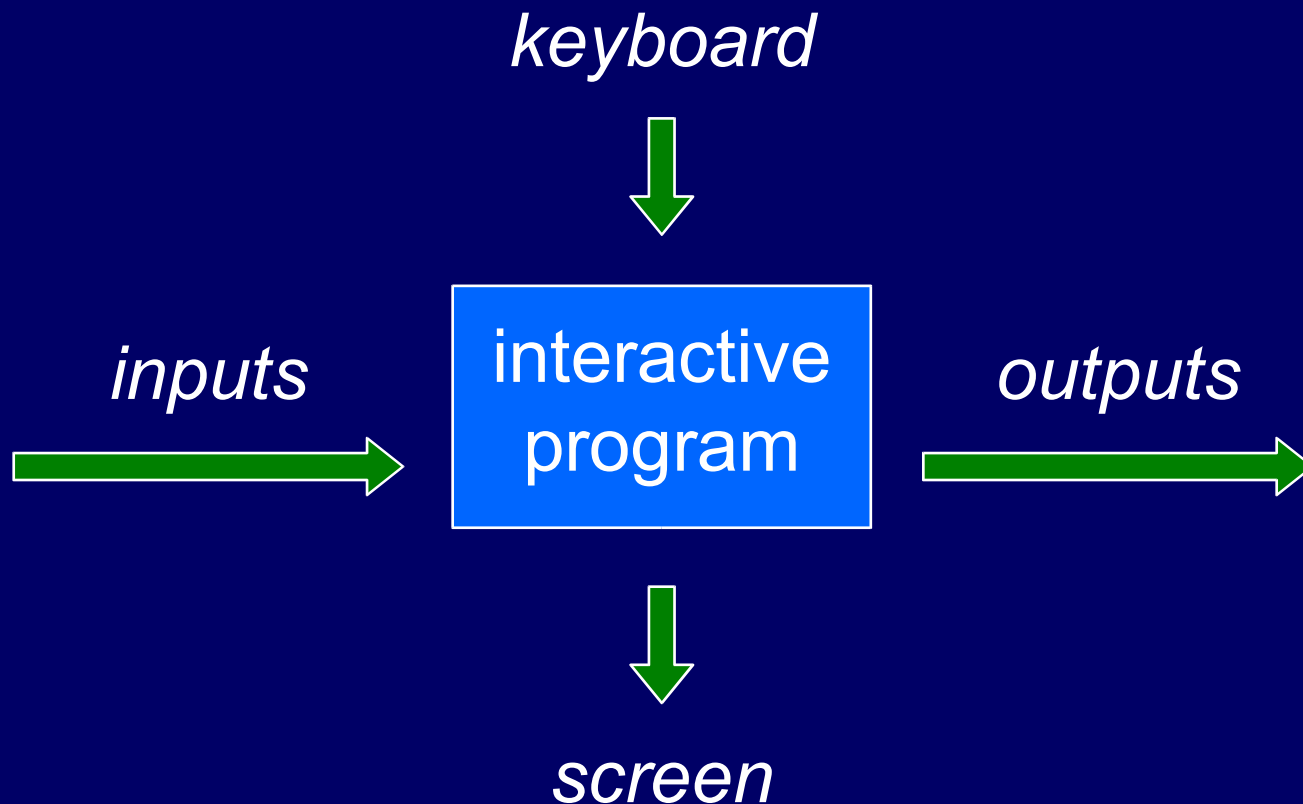
Chapter 9 - Interactive Programs

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.



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



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.

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`.

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.

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
```


- The action putChar c writes the character *c* to the screen, and returns no result value:

```
putChar :: Char → IO ()
```

- The action return v simply returns the value *v*, without performing any interaction:

```
return :: a → IO a
```

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)
```

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'
```

Example

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

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

For example:

```
> strlen
```

```
Enter a string: abcde
```

```
The string has 5 characters
```

Note:

- Evaluating an action executes its side effects, with the final result value being discarded.

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.

- 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
```


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)
```

Note:

- The action `getCh` reads a character from the keyboard, without echoing it to the screen.
- This useful action is not part of the standard library, but is a special Hugs primitive that can be imported into a script as follows:

```
primitive getCh :: IO Char
```

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 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 "haske11" "pasca1"
"-as--11"
```

Exercise

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

- The board comprises five rows of stars:

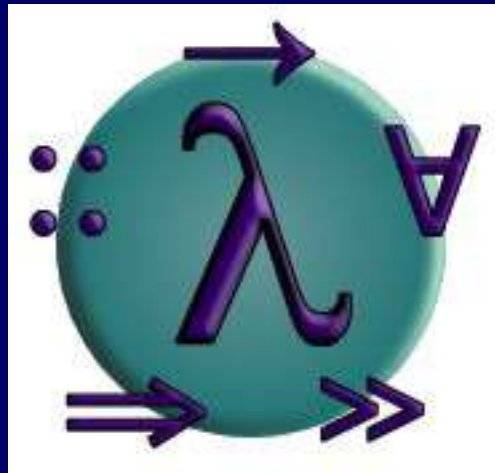
```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

- 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].

PROGRAMMING IN HASKELL



Chapter 10 - Declaring Types and Classes

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].

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 parameters. 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)
```

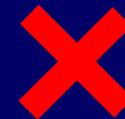
Type declarations can be nested:

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



However, they cannot be recursive:

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



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.

Note:

- The two values False and True are called the constructors 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.

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
```

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 functions that construct values of type `Shape`:

```
Circle :: Float → Shape
```

```
Rect    :: Float → Float → Shape
```


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)
```

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
 $\text{Zero} :: \text{Nat}$ and $\text{Succ} :: \text{Nat} \rightarrow \text{Nat}$.

Note:

- A value of type `Nat` is either `Zero`, or of the form `Succ n` where $n :: \text{Nat}$. That is, `Nat` contains the following infinite sequence of values:

`Zero`

`Succ Zero`

`Succ (Succ Zero)`

⋮

- 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

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

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)
```

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)
```

For example:

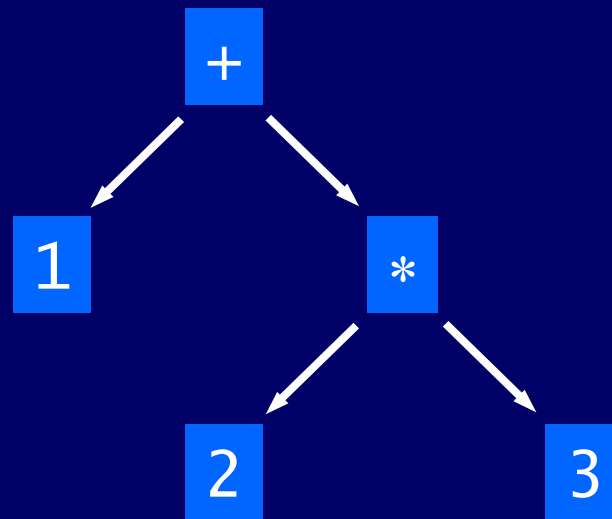
$$\begin{aligned} & \text{add (Succ (Succ Zero)) (Succ Zero)} \\ = & \text{Succ (add (Succ Zero) (Succ Zero))} \\ = & \text{Succ (Succ (add Zero (Succ Zero)))} \\ = & \text{Succ (Succ (Succ Zero))} \end{aligned}$$

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 expressions 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))
```

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
```

Note:

- The three constructors have types:

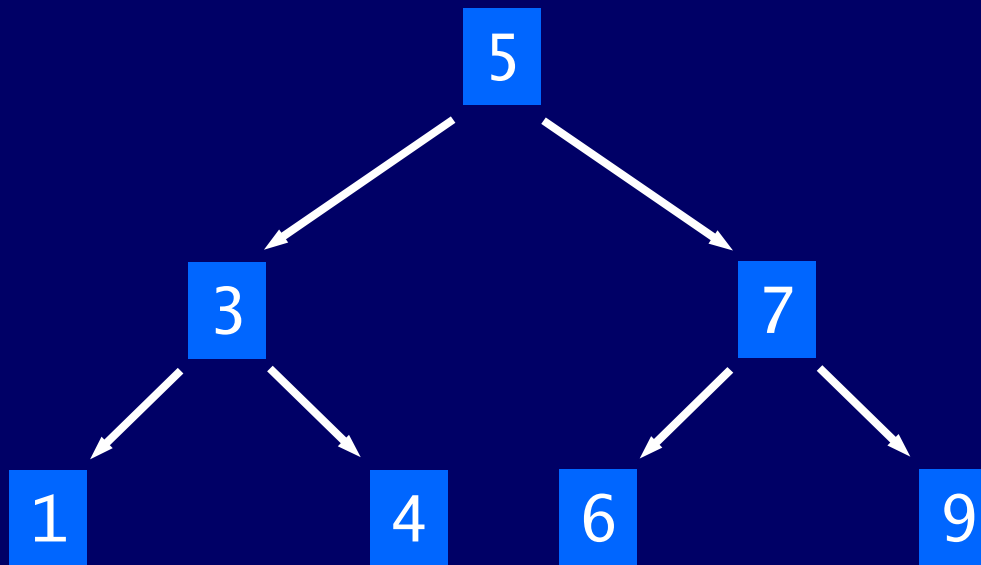
```
Va1 :: Int → Expr  
Add :: Expr → Expr → Expr  
Mul :: Expr → Expr → Expr
```

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

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

Binary Trees

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



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))
```

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
```

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

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].

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:

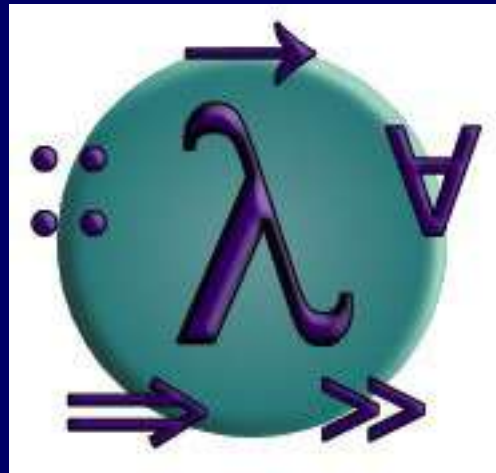
```
occurs m (Leaf n)           = m==n
occurs m (Node l n r) | m==n = True
                       | m<n  = occurs m l
                       | m>n  = occurs m r
```

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

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.

PROGRAMMING IN HASKELL



Chapter 11 - The Countdown Problem

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.

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 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.

For our example, one possible solution is

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

Notes:

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

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
```

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
```


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:

```
choices :: [a] → [[a]]
```

For example:

```
> 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]]
```

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

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.

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]
```

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

All solutions: 3.47 seconds

Around 10
times faster in
both cases.

Can We Do Better?

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

$$x * y = y * x$$

$$x * 1 = x$$

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

Exploiting Properties

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

```
valid          :: Op → Int → Int → Bool
```

```
valid Add x y  =  $x \leq y$ 
```

```
valid Sub x y  =  $x > y$ 
```

```
valid Mul x y  =  $x \leq y \ \&\& \ x \neq 1 \ \&\& \ y \neq 1$ 
```

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

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.

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.