Chapter 15

Functional Programming

Topics

- Introduction
- Functional programs
- Mathematical functions
- Functional forms
- Lambda calculus
- Eager and lazy evaluation
- Haskell

Introduction

- Emerged in the early 1960s for Artificial Intelligence and its subfields:
  - Theorem proving
  - Symbolic computation
  - Rule-based systems
  - Natural language processing
- The original functional language was Lisp, developed by John McCarthy (1960)

Functional Programs

- A program is a description of a specific computations.
  - A program can be seen as a “black box” for obtaining outputs from inputs.
  - From this point of view, a program is equivalent to a mathematical function.

Mathematical Functions

- A ***function*** is a rule that associates to each `x` from some set `X` of values a unique `y` from another set `Y` of values.
  - In mathematical terminology, if f is the name of the function

    `y = f(X)` or

    `f: X → Y`
  - The set `X` is called the *domain* of `f`.
  - The set `Y` is called the *range* of `f`.

Mathematical Functions

- The `x` in `f(x)`, which represents any value from `X` (domain), is called *independent variable*.
- The `y` from the set `Y` (range), defined by the equation `y = f(x)` is called *dependent variable*.
- Sometimes `f` is not defined for all `x` in `X`, it is called a *partial function*. Otherwise it is a *total function*.
- Example: `square(x) = x * x`

  mapping expressions

  function name

  parameters

Mathematical Functions

- Everything is represented as a mathematical function:
  - *Program*: x represents the input and y represents the output.
  - *Procedure* or *function*: x represents the parameters and y represents the returned values.
- No distinction between a program, a procedure, and a function. However, there is a clear distinction between input an output values.

Mathematical Functions: variables

- In imperative programming languages, variables refer to memory locations as well as values.

```
x = x + 1
```

  - Means "update the program state by adding `1` to the value stored in the memory cell named x and then storing that sum back into that memory cell"
  - The name x is used to denote both a value (as in x+1), often called an *r-value*, and a memory address, called an *l-value*.

Mathematical Functions: variables

- In mathematics, variables always stand for actual values, there is no concept of memory location (l-values of variables).
  - Eliminates the concept of variable, except as a name for a value.
  - Eliminates assignment as an available operation.

Mathematical Functions: variables

- Consequences of the lack of variables and assignment
  1. No loops.
     - The effect of a loop is modeled via recursion, since there is no way to increment or decrement the value of variables.
  2. No notation of the internal state of a function.
     - The value of any function depends only on the values of its parameters, and not on any previous computations, including calls to the function itself.

Mathematical Functions: variables

- The value of a function does not depend on the order of evaluation of its parameters.
- The property of a function that its value depend only on the values of its parameters is called *referential transparency*.

3. No state.
   - There is no concept of memory locations with changing values.
   - Names are associated to values which once the value is set it never changes.

Mathematical Functions

- Functional Forms
  - Def: A higher-order function, or functional form, is one that either takes functions as parameters or yields a function as its result, or both

Functional Forms

1. Function Composition
   - A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

   Form: `h ≡ f ° g`

   which means `h (x) ≡ f ( g ( x ))`

   For `f (x) ≡ x * x * x` and
   `g (x) ≡ x + 3,`

   `h ≡ f ° g` yields `(x + 3)* (x + 3)* (x + 3)`

Functional Forms

2. Construction
   - A functional form that takes a list of functions as parameters and yields a list of the results of applying each of its parameter functions to a given parameter

   Form: `[f, g]`

   For `f (x) ≡ x * x * x` and
   `g (x) ≡ x + 3,`

   `[f, g] (4)` yields `(64, 7)`

Functional Forms

3. Apply-to-all
   - A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

   Form: α

   For `h (x) ≡ x * x * x`

   `α( h, (3, 2, 4))` yields `(27, 8, 64)`

Pure Functional Programming

- In pure functional programming there are no variables, only constants, parameters, and values.
- Most functional programming languages retain some notation of variables and assignment, and so are “impure”
  - It is still possible to program effectively using the pure approach.

Lambda Calculus

- The foundation of functional programming developed by Church (1941).
- A *lambda expression* specifies the parameters and definition of a function, but not its name.
  - Example: lambda expression that defined the function `square`:

    `(λx·x*x)`
  - The identifier `x` is a parameter for the (unnamed) function body `x*x.`

Lambda Calculus

- Application of a lambda expression to a value: `((λx·x*x)2)` which evaluates to `4`
- What is a lambda expression?
  1. Any identifier is a lambda expression.
  2. If `M` and `N` are lambda expressions, then the *application* of `M` to `N`, written `(MN)` is a lambda expression.
  3. An *abstraction*, written `(λx·M)` where `x` is an identifier and `M` is a lambda expression, is also a lambda expression.

Lambda Expressions: BNF

- A simple BNF grammar for the syntax of the lambda calculus

  LambdaExpression → ident | (M N) | (λ ident · M)
  M → LambdaExpression
  N → LambdaExpression

- Examples:

  `x`
  `(λx·x)`
  `((λx·x)(λy·y))`

Lambda Expressions: free and bound variables

- In the lambda expression `(λx·M)`
  - The identifier `x` is said to be *bound* in the subexpression `M`.
  - Any identifier not bound in `M` is said to be *free*.
  - Free variables are like globals and bound variables are like locals.
  - Free variables can be defined as:

```
free(x) = x
free(MN) = free(M) ∪ free(N)
free(λx·M) = free(M) – {x}
```


Lambda Expressions: substitution

- A substitution of an expression `N` for a variable `x` in `M`, written `M[N/x]`, is defined:
  1. If the free variable of `N` have no bound occurrences in `M`, then the term `M[N/x]` is formed by replacing all free occurrences of `x` in `M` by `N`.
  2. Otherwise, assume that the variable `y` is free in `N` and bound in `M`. Then consistently replace the binding and corresponding bound occurrences of `y` in `M` by a new variable, say `u`. Repeat this renaming of bound variables in `M` until the condition in Step 1 applies, then proceed as in Step 1.

Lambda Expressions: substitution

- Examples:

```
x[y/x] = y
(xx)[y/x] = (yy)
(zw)[y/x] = (zw)
(zx)[y/x] = (zy)
[λx·(zx))[y/x] = (λu·(zu))[y/x] = (λu·(zu))
```


Lambda Expressions: beta-reduction

- The meaning of a lambda expression is defined by the *beta-reduction* rule:

  `((λx·M)N) ⇒ M[N/x]`

- An *evaluation* of a lambda expression is a sequence `P ⇒ Q ⇒ R ⇒ …`
  - Each expression in the sequence is obtained by the application of a beta-reduction to the previous expression.

    `((λy·((λx·xyz)a))b) ⇒ ((λy·ayz)b) ⇒ (abz)`

Functional Programming vs. Lambda Calculus

- A functional programming languages is essentially an applied lambda calculus with constant values and functions build in.
  - The pure lambda expression `(xx)` can be written as `(x times x)` or `(x*x)` or `(* x x)`
  - When constants, such as numbers, are added (with their usual interpretation and definitions for functions, such as *), then *applied lambda calculi* is obtained

Eager Evaluation

- An important distinction in functional languages is usually made in the way they define function evaluation.
- *Eager Evaluation or call by value:* In languages such as Scheme, all arguments to a function are normally evaluated at the time of the call.
  - Functions such as `if` and `and` cannot be defined without potential run-time error

Eager Evaluation

```
(if ( = x 0 ) 1 ( / 1 x ))
```

- Defined the value of the function to be 1 when x is zero and 1/x otherwise.
- If all arguments to the if functions are evaluated at the time of the call, division by zero cannot be prevented.

Lazy Evaluation

- An alternative to the eager evaluation strategy is *lazy evaluation or call by name*, in which an argument to a function is not evaluated (it is deferred) until it is needed.
  - It is the default mechanism of Haskell.

Eager vs. Lazy Evaluation

- An advantage of eager evaluation is efficiency in that each argument passed to a function is only evaluated once,
  - In lazy evaluation, an argument to a function is reevaluated each time it is used, which can be more than once.
- An advantage of lazy evaluation is that it permits certain interesting functions to be defined that cannot be implemented as eager languages

Haskell

Haskell

- The interactive use of a functional language is provided by the HUGS (Haskell Users Gofer System) environment developed by Mark Jones of Nottingham University.
- HUGS is available from http://www.haskell.org/hugs/
- The Haskell web page is http://www.haskell.org/

Haskell: sessions

- Expressions can be typed directly into the Hugs/Haskell screen.
  - The computer will respond by displaying the result of evaluating the expression, followed by a new prompt on a new line, indicating that the process can begin again with another expression

```
? 6 * 7
42
```

- This sequence of interactions between user and computer is called a *session*.

Haskell: scripts

- Scripts are collections of definitions supplied by the programmer.

```
square  :: Integer → Integer
square x = x * x
smaller :: (Integer,Integer) → Integer
smaller (x,y)= if x ≤ y then x else y
```

- Given the previous script, the following session is now possible:

```
? square 3768       ? square(smaller(5,3+4))
14197824            25
```


Haskell: scripts

- The purpose of a definition of a function is to introduce a *binding* associating a given name with a given definition.
  - A set of bindings is called an *environment* or *context*.
    - Expressions are always evaluated in some context and can contain occurrences of the names found in that context.
    - The Haskell evaluator uses the definitions associated with those names as rules for simplifying expressions.

Haskell: scripts

- Some expressions can be evaluated without having to provide a context.
  - Those operations are called *primitives* (the rules of simplification are build into the evaluator).
    - Basic operations of arithmetic.
    - Other libraries can be loaded.
- At any point, a script can be modified and resubmitted to the evaluator.

Haskell: first things to remember

- Scripts are collections of definitions supplied by the programmer.
- Definitions are expressed as equations between certain kinds of expressions and describe mathematical functions.
  - Definitions are accompanied by type signatures.
- During a session, expressions are submitted for evaluation
  - These expressions can contain references to the functions defined in the script, as well as references to other functions defined in libraries.

Haskell: evaluation

- The computer evaluates an expression by reducing it to its simplest equivalent form and displaying the result.
  - This process is called *evaluation*, *simplification*, or *reduction*.
  - Example: `square(3+4)`
  - An expression is *canonical* or in *normal form* If it cannot be further reduced.

Haskell: evaluation

- A characteristic feature of functional programming is that if two different reduction sequences terminate, they lead to the same result.
  - For some expressions some ways of simplification will terminate while other do not.
  - Example: `three infinity`
  - Lazy evaluation guarantees termination whenever termination is possible

Getting Started with Hugs

```
% hugs
Type : ? for help
Prelude> 6*7
42
Prelude> square(smaller(6,9))
ERROR – Undefined variable “smaller”
Prelude> sqrt(16)
4.0
Prelude> :load example1.hs
Reading file “example1.hs”
Main> square(smaller(6,9))
36
```


Getting Started with Hugs

Typing `:?` In Hugs will produce a list of possible commands.
Typing `:quit` will exit Hugs
Typing `:reload` will repeat last load command
Typing `:load` will clear all files

Topics

- Values
- Functions
- Extensionality
- Currying
- Definitions

Values

- An expression is used to describe (or denote) a value.
  - Among the kinds of value are: numbers of various kinds, truth values, characters, tuples, functions, and lists.
  - New kinds of value can be introduced.
- The evaluator prints a value by printing its canonical representation.
  - Some values have no canonical representation (i.e. function values).
  - Other values are not finite (i.e. Π)

Values

  - For some expressions the process of reduction never stops and never produces any result (i.e. the expression `infinity`).
  - Some expressions do not denote well-defined values in the normal mathematical sense (i.e. the expression `1/0`).
- Every syntactically well-formed expression denotes a value.
  - A special symbol $\perp$ (bottom) stands for the undefined value of a particular type

Values

- The value of infinity is the undefined value ⊥ or type `Integer`.
- `1/0` is the undefined value ⊥ or type `Float`
  - `1/0 = ⊥`
- The computer is not able to produce the value ⊥.
  - It generates an error message or it remains perpetually silent.
- ⊥ is a special value that can be added to the universe of values only if its properties and its relationship with other values are precisely stated.

Values

- If `f ⊥ = ⊥`, then `f` is *strict*, otherwise it is *nonstrict*.
- `square` is a strict function because the evaluation of the undefined value goes into an infinite reduction (i.e. `? square infinity`)
- `three` is nonstrict because the evaluation of the undefined value is `3` (i.e. `? three infinity`)

Functions

- A function `f` is a rule of correspondence that associates each element of given type `A` (domain) with a unique element of a second type `B` (range).
  - The result of applying function `f` to an element `x` of the domain is written as `f(x)` or `f x` (when the parentheses are not necessary).
    - Parentheses are necessary when the argument is not a simple constant or variable.

Functions

- Examples
  - `square(3+4)` vs. `square3+4`
    - `square3+4` means `(square3)+4`
  - `square(square3)` vs. `square square 3`
    - square square 3 means (square square) 3

Extensionality

- Two functions are equal if they give equal results for equal arguments.
  - `f = g` if an only if `f x = g x` for all `x`
  - This is called the principle of *extensionality*.
  - Example:

```
double, double' :: Integer → Integer
double x = x + x
double' x = 2 * x
```

  - `double` and `double'` defines the same functional value, `double = double'`

Currying

- Replacing a structure argument by a sequence of simpler ones is a way to reduce the number of parentheses in an expression.

```
smaller :: (Integer,Integer) → Integer
smaller (x,y) = if x ≤ y then x else y

smallerc :: Integer → (Integer → Integer)
smallerc x y = if x ≤ y then x else y
```

Curying: advantages

1. Currying can help to reduce the number of parentheses that have to be written in expressions.

2. Curried function can be applied to one argument only, giving another function that may be useful in its own right

Definitions

- There are definitions for different kinds of values:
  - Definitions of fixed values.

```
pi :: Float
pi = 3.14159
```

  - Some definitions of functions use *conditional expressions*

```
smaller :: (Integer, Integer) → Integer
smaller(x,y) = if x ≤ y then x else y
```


Definitions

- The same expressions can be defined using *guarded equations.*

```
smaller :: (Integer, Integer) → Integer
smaller(x,y)
      | x ≤ y = x
      | X > y = y
```

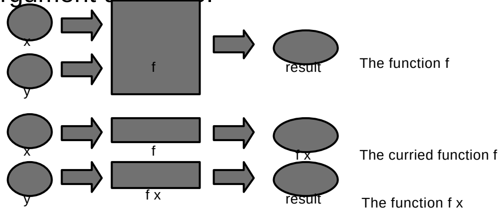
  - Each clause consists of a condition, or guard, and an expression, which is separated from the guard by an = sign.
  - The main advantage of guarded expressions is when there are three or more clauses in a definition.

Topics

- Reduction and Currying
- Recursive definitions
- Local definitions
- Type Systems
  - Strict typing
  - Polymorphism
- Types Classes
- Types
  - Booleans
  - Characters
  - Enumerations
  - Tuples
  - Strings

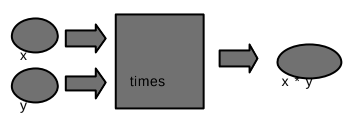
Currying

- Viewing a function with two or more arguments as a function that takes one argument at a time.



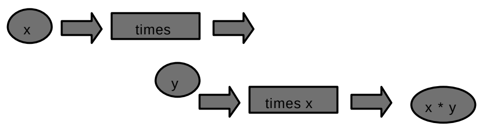
Currying: example

- The uncurried function `times` takes two numbers as inputs and return their multiplication.



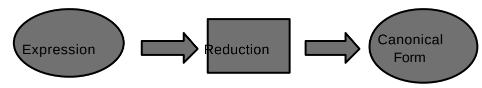
Currying: example

- The curried function `times` takes a number `x` and return the function (`times x`).
- (`times x`) takes a number `y` and returns the number (`x * y`).



Reduction

- Reduction is the process of converting a functional expression to its canonical form by repeatedly applying reduction rules



Reduction Rules

- There are two kinds of reduction rules:
  - Build-in definitions
    - For example the arithmetic operations
  - User supplied definitions

Recursive Definitions

- Definitions can also be recursive.
- Example:

```
fact :: Integer → Integer
fact n = if n==0 then 1 else n*fact(n-1)
```

  - This definition of `fact` is not completely satisfactory: if it is applied to a negative integer, then the computation never terminates.
  - For negative numbers, `fact x = ⊥`.
    - It is better if the computation terminated with a suitable error message rather than proceeding indefinitely with a futile computation.

Recursive Definitions

```
fact :: Integer → Integer
fact n
      | n < 0  =  error “negative argument”
      | n == 0 = 1
      | n > 0  = n * fact(n-1)
```

- The predefined function error takes a string as argument; when evaluated it causes immediate termination of the evaluator and displays the given error message.

```
? fact (-1)
Program error: negative argument
```


Local Definitions

- In mathematical descriptions there are expressions qualified by a phrase of the form “where …”.
  - `f(x,y) = (a+1)(a+2)`, where `a = (x+y)/2`
- Example:

```
f :: (Float,Float) → Float
f(x,y) = (a+1) * (a+2) where a = (x+y)/2
```

  - The special word `where` is used to introduce a local definition whose context (or scope) is the expression on the RHS of the definition of `f`.

Local Definitions

- When there are two or more local definitions, there are two styles:

```
f :: (Float,Float) → Float
f(x,y) = (a+1) * (b+2)
         where a = (x+y)/2
               b = (x+y)/3

f :: (Float,Float) → Float
f(x,y) = (a+1) * (b+2)
         where a = (x+y)/2; b = (x+y)/3
```


Local Definitions

- A local definition can be used in conjunction with a definition that relies on guarded equations.:

```
f :: Integer → Integer → Integer
f x y =
      | x ≤ 10 = x + a
      | x > 10 = x-a
        where a = square(y+a)
```

  - The `where` clause qualifies both guarded equations.

Type Systems

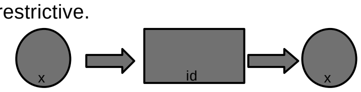
- Programming languages have either:
  - No type systems
    - Lisp, Prolog, Basic, etc
  - A strict type system
    - Pascal, Modula2
  - A polymorphic type systems
    - ML, Mirada, Haskell, Java, C++

Strong Typing Principle

- Every expression must have a type
  - 3 has type `Int`
  - `'A'` has type `Char`
- The type of a compound expression can be deduced from its constituents alone.
  - `('A',1+2)` has type `(Char, Int)`
- An expression which does not have a sensible type is illegal.
  - `'A'+3` is illegal

Strict Typing

- Every expression has a unique concrete type.
  - Although this system is good for trapping errors, it is too restrictive.



- What type should be given to `id`?
  - Is it `Int→Int`?, `Char→Char`?, `(Int,Bool)→(Int,Bool)`
- With strict typing we have to define separate versions of `id` for each type.

Polymorphism

- Polymorphism allows the definition of certain functions to be used with different types.
- Without polymorphism we would have to write different versions of the function for each possible type (type declaration is different but the body is the same).
- Polymorphism results in simpler, more general, reusable and concise programs.

Type Classes

- A curried multiplication can be used with two different type signatures:

  (x) :: Integer → Integer → Integer
  (x) :: Float → Float → Float

- So, it can be assigned a polymorphic type:

  (x) :: α → α → α

  - This type is too general (two characters or two booleans should not be multiplied).

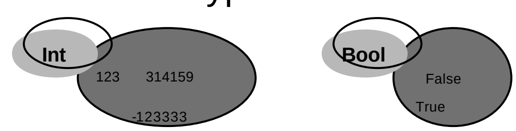
Type Classes

- Group together kindred types into *type classes*.
  - `Integer` and `Float` belong to the same class, the class of numbers.
    (x) :: Num α ⇒ α → α → α
- There are other kindred types apart from numbers.
  - The types whose value can be displayed, the types whose value can be compared for equality, the type whose value can be enumerated, etc.

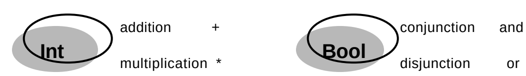
Types

- In addition to defining functions and constants, functional languages allows to define types to build new and useful types from existing ones.
- The universe of values is divided into organized collections, called *types*.
  - Integer, Float, Double, booleans, characters, lists, trees, etc.
  - An infinity variety of other types can be put together: Integer → Float, (Float, Float), etc.

Types



- Each type has associated with it certain operations which are not meaningful for other types.



Type Declaration

- The type of an expression is declared using the following convention:

  `expression :: type`

  - Example: `e :: t`
    - Reads: "the expression e has the type `t`"

    - `pi :: Double`
    - `Square :: Integer → Integer`

Types

- *Strong typing*: the value of an expression depends only on the values of its component expressions, so does its type.
- Consequence of strong typing
  - Any expression which cannot be assigned a sensible type is not well formed and is rejected by the computer before evaluation (illegal expressions).

Types

```
quad :: Integer → Integer
quad x = square square x
```

- Advantage of strong typing
  - Enables a range of errors to be detected before evaluation.
- There are two stages of analysis when a expression is submitted for evaluation.

Types

- The expression is checked to see whether it conforms to the correct syntax laid down for constructing expressions.
  - No: the computer signals a *syntax error*
  - Yes: perform the second stage of evaluation
- The expression is analysed to see if it posses a sensible type
  - Fails: the computer signals a *type error*.
  - Yes: the expression is evaluated.

Classification of Types

- Basic/Simple Types
  - Contain primitive values
- User-defined Types
  - Contain user-defined values
- Derived Types
  - Contain more complex values

Simple Data Types: booleans

- Used to define the truth value of a conditional expression.
  - There are two truth values, `True` and `False`.
  - These two values comprise the datatype `Bool` of boolean values.
  - `True`, `False` and `Bool` begin with a capital letter.
  - The datatype `Bool` can be introduce with a *datatype declaration:*

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


Simple Data Types: booleans

- Having introduce `Bool`, it is possible to define functions that take boolean arguments by *pattern matching*.
  - Example: the negation function

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

  - To simplify expressions of the form `not e`: first `e` is reduced to normal form.
    - If e cannot be reduced to normal form then the value of `not e` is undefined
    - `not ⊥ = ⊥` then `not` is strict.

Simple Data Types: booleans

- There are not two but thee boolean values: `True`, `False`, and ⊥.
- Every datatype declaration introduces an extra anonymous value, the undefined value of the datatype.
- More examples: conjunction, disjunction.

Simple Data Types: booleans

- This is how pattern matching works:

```
⊥ ∧ True = ⊥
⊥ ∧ False = ⊥
False ∧ ⊥ = False
True ∧ ⊥ = ⊥
```

  - ∧ is strict in its LHS, but nonstrict in its RHS argument.

Booleans: equality operators

- There are two equality operators = = and ≠

```
(==) :: Bool → Bool → Bool
x == y = (x∧y) ∨ (not x ∧ not y)
(≠) :: Bool → Bool → Bool
x ≠ y = not(x == y)
```

- The symbol == is used to denote a computable test for equality.
- The symbol = is used both in definitions and its normal mathematical sense.

Booleans: equality operators

- The main purpose of introducing an equality test is to be able to use it with a range of different types.
  - (==) and (≠) are *overloaded operations*.
- The proper way to introduce them is first to declare a type class `Eq` consisting of all those types for which (==) and (≠) are to be defined.

Booleans: equality operators

```
class Eq α where
  (=),(≠) :: α → α → Bool
```

- To declare that a certain type is an instance of the type class `Eq`, an *instance declaration* is needed.

```
instance Eq Bool where
(x == y) = (x ∧ y) ∨ (not x ∧ not y)
(x ≠ y) = not(x == y)
```


Booleans: comparison operators

- Booleans can also be compared.
  - Comparison operations are also overloaded and make sense with elements from a number of different types.

```
 class (Eq α) ⇒ Ord α where
(<),(≤),(≥),(>) :: α → α → Bool
(x ≤ y) = (x < y) ∨ (x == y)
(x ≥ y) = (x > y) ∨ (x == y)
(x > y) = not(x ≤ y)
```


Booleans: comparison operators

- `Bool` could be an instance of `Ord`:

```
instance Ord Bool where
      False ≤  False = False
      False ≤  True = True
      True ≤  False = False
      True ≤  True = False
```

Example: leap years

- Define a function to determine whether a year is a leap year or not.
  - A leap year is divisible by 4, except that if it is divisible by 100, then it must also be divisible by 400.

```
leapyear :: Int → Bool
leapyear y = (y mode 4 == 0) ∧
             (y mode 100 ≠ 0 ∨ (y mode 400 == 0)
```

  - Using conditional expressions:

```
leapyear y = if (y mode 100==0)
             then (y mode 400 ==0)
             else (y mode 4 == 0)
```


Characters

- Characters are denoted by enclosing them in single quotation marks.
  - Remember: the character `'7'` is different from the decimal number `7`.
- Two primitive functions are provided for processing characters, `ord` and `chr`.
  - Their types are:

```
ord :: Char → Int
chr :: Int → Char
```


Characters

- The function `ord` converts a character `c` to an integer `ord c` in the range `0 ≤ ord c ≤ 256`
- The function `chr` does the reverse, converting an integer back into the character it represents.
- Thus `chr (ord c) = c` for all characters `c`.

```
? ord'b'                    ? chr98
  98                          'b'
? chr(ord'b'+1)
  'c'
```


Characters

- Characters can be compared and tested for equality.

```
instance Eq Char where
  (x == y) = (ord x == ord y)

instance Ord Char where
  (x < y) = (ord x < ord y)

? '0' < '9'                ? 'A' < 'Z'
True                  True
```


Characters: simple functions

- Three functions for determining whether a character is a digit, lower-case letter, or upper-case letter:

```
isDigit,isLower,isUpper :: Char → Bool
isDigit c = ('0' ≤ c) ∧ (c ≤ '9')
isLower c = ('a' ≤ c) ∧ (c ≤ 'z')
isUpper c = ('A' ≤ c) ∧ (c ≤ 'Z')
```

- A function for converting lower-case letter to upper-case:

```
capitalise :: Char → Char
capitalise c = if isLower c then
               chr(offset+ord c) else c
              where offset = ord 'A' – ord 'a'
```


Enumerations

- They are user-defined types.
- Example:

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

  - This definition binds the name `Day` to a new type that consists of eight distinct values, seven of which are represented by the given constants and the eight by the undefined value ⊥
    - The seven new constants are called the constructors of the datatype Day.
    - By convention, constructor names and the new name begin with an upper-case letter.

Enumerations

- It is possible to compare elements of type `Day`, so `Day` can be declared as an instance of the type classes `Eq` and `Ord`.
  - A definition of (==) and (<) based on pattern matching would involve a large number of equations.
- Better idea. Code elements of `Day` as integers, and use integer comparison instead.

Enumerations

- Since the same idea can be employed with other enumerated types, a new type class `Enum` is declared
  - `Enum` describes types whose elements can be enumerated.

```
class Enum α where
   fromEnum :: α → Int
   toEnum   :: Int → α
```

  - A type is declared an instance of `Enum` by giving definition of `toEnum` and `fromEnum`, functions that convert between elements of the type and `Int`.

Enumerations: example

- Day is a member of `Enum`:

```
instance Enum Day where
     fromEnum Sun = 0
     fromEnum Mon = 1
     fromEnum Tue = 2
     fromEnum Wed = 3
     fromEnum Thu = 4
     fromEnum Fri = 5
     fromEnum Sat = 6
```


Enumerations: example

- Given fromEnum on Day:

```
instance Eq Day where
(x == y) = (fromEnum x == fromEnum y)

instance Ord Day where
(x < y) = (fromEnum x < fromEnum y)
```


Enumerations: example

```
workday   :: Day → Bool
workday d = (Mon ≤ d) ∧ (d ≤ Fri)

restday   :: Day → Bool
restday d = (d==Sat) ∨ (d==Sun)

dayafter   :: Day → Day
dayafter d = toEnum((fromEnum d+1) mod 7)
```


Automatic instance declarations

- Haskell provides a mechanism for declaring a type as an instance of `Eq`, `Ord`, and `Enum` in one declaration.

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

  - The deriving clause causes the evaluator to generate instance declarations of the named type classes automatically.

Tuples

- One way of combining types to form new ones is by pairing them.
  - Example: `(Integer, Char)` consists of all pairs of values `(x,c)` for which `x` is an arbitrary-precision integer, and `c` is a character.
- Like other types, the type (α,β) contains an additional value $\perp$

Tuples: practical example

- A function returns a pair of numbers, the two real roots of a quadratic equation with coefficients (a,b,c):

```
roots :: (Float, Float, Float) → (Float,Float)
roots (a,b,c)
      | a == 0    = error “not quadratic”
      | e < 0     = error “complex roots”
      | otherwise = ((-b-r)/d,(-b+r)/d)
      where r = sqrt e
            d = 2*a
            e = b*b-4*a*c
```


Other Types

- A type can be declared by typing its constants or with values that depend on those of other types.

```
data Either = Left Bool | Right Char
```

  - This declares a type `Either` whose values are denoted by expressions of the form `Left b`, where b is a boolean, and `Right c`, where c is a character.
  - There are 3 boolean values (including $\perp$) and 257 characters (including $\perp$), so there are 261 distinct values of the type `Either`; these include `Left ⊥`, `Right ⊥`, and $\perp$

Other Types

- In general:

```
data Either α β = Left α | Right β
```

- The names `Left` and `Right` introduces two constructors for building values of type `Either`, these constructors are nonstrict functions with types:

```
Left  :: α → Either α β
Right :: β → Either α β
```


Other Types

- Assuming that values of types α and β can be compared, comparison on that type `Either α β` can be added as an instance declaration:

```
instance (Eq α,Eq β ) ⇒ Eq(Either α β) where
      Left x == Left y     = (x==y)
      Left x == Right y    = False
      Right x == Left y    = False
      Right x == Right y  = (x==y)
instance (Ord α,Ord β ) ⇒ Ord(Either α β) where
      Left x < Left y      = (x<y)
      Left x < Right y     = True
      Right x < Left y     = False
      Right x < Right y    = (x<y)
```


Type Synonyms

- *Type synonym declaration*: a simple notation for giving alternative names to types.
- Example:

```
roots :: (Float, Float, Float) → (Float,Float)
```

  - As an alternative, two type synonyms could be used

```
type Coeffs = (Float, Float, Float)
type Roots  = (Float,Float)
```

Type Synonyms

- This declarations do not introduce new types but merely alternative names for existing types.

```
roots :: Coeffs → Roots
```

- This new description is shorter and more informative.

- Type synonyms can be general.

```
type Pairs α     = (α,α)
type Automorph α = α → α
type Flag α      = (α,Bool)
```


Type Synonyms

- Type synonyms cannot be declared in terms of each other since every synonym must be expressible in terms of existing types.
- Synonyms can be declared in terms of another synonym.

```
type Bools = PairBool
```

- Synonyms and declarations can be mixed

```
data OneTwo α = One α | Two(Pairs α)
```


Strings

- A list of characters is called a *string*.
- The type `String` is a synonym type:

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

- Syntax: the characters of a string are enclosed in double quotation marks.
- `'a'` vs. `"a"`
  - the former is a character
  - the latter is a list of characters that happens to contain only one element.

Strings

- Strings cannot be declared separately as instances of `Eq` and `Ord` because they are just synonyms.
  - They inherit whatever instances are declared for general lists.
- Comparison on strings follow the normal lexicographic ordering.

```
? "hello" < "hallo"
False
? "Jo" < "Joanna"
True
```


Strings

- Haskell provides a primitive command for printing strings.

```
putStr :: String → IO()
```

  - Evaluating the command `putStr` causes the string to be printed literally.

```
? putStr "Hello World"
Hello World
? putStr "This sentence contains \n a newline"
This sentence contains
a newline
```


The type class Show

- Haskell provides a special type class `Show` to display information of different kinds and formats.

```
class Show α where
  showsPrec :: Int → α → String → String
```

  - The function `showsPrec` is provided for displaying values of type α
  - Using `showsPrec` it is possible to define a simpler function that takes a value and converts it to a string.

```
show :: Show α ⇒ α → String
```

The type class Show

- Example: if `Bool` is declares to be a member of `Show` and `show` is defined for booleans as

```
show False = "False"
show True  = "True"
? putStr(show True)
True
```

- Some instances of `Show` are provided as primitive.

```
? putStr("The year is "++ show(3*667))
The year is 2001
```


Topics

- Numbers
  - Natural numbers
  - Haskell numbers
- Lists
  - List notation
  - Lists as a data type
  - List operations

Numbers

- Haskell provides a sophisticated hierarchy of type classes for describing various kinds of numbers.
- Although (some) numbers are provided as primitives data types, it is theoretically possible to introduce them through suitable data type declarations.

Natural Numbers

- The natural numbers are the numbers 0, 1, 2, and so on, used for counting.
- Introduced by the declaration

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

  - The constructor `Succ` (short for ‘successor’) has type `Nat → Nat`.
  - Example: as an element of `Nat` the number `7` would be represented by

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


Natural Numbers

- Every natural number is represented by a unique value of `Nat`.
- On the other hand, not every value of `Nat` represents a well-defined natural number.
  - Example: `⊥, Succ ⊥, Succ(Succ ⊥)`
- Addition ca be defined by

```
(+)        :: Nat → Nat → Nat
m + Zero   = m
m + Succ n = Succ(m + n)
```


Natural Numbers

- Multiplication ca be defined by

```
(x)        :: Nat → Nat → Nat
m x Zero   = Zero
m x Succ n = (m x n) + m
```

- `Nat` can be a member of the type class `Eq`

```
instance Eq Nat where
  Zero == Zero     = True
  Zero == Succ n   = False
  Succ m == Zero   = False
  Succ m == Succ n  = (m == n)
```

Natural Numbers

- `Nat` can be a member of the type class `Ord`

```
instance Ord Nat where
  Zero < Zero        = False
  Zero < Succ n      = True
  Succ m < Zero      = False
  Succ m < Succ n    = (m < n)
```

- Elements of `Nat` can be printed by

```
showNat                  :: Nat → String
showNatZero              = "Zero"
showNat (Succ Zero)      = "Succ Zero"
showNat (Succ(Succ n)) = "Succ (" ++
                showNat (Succ n) ++ ")"
```

Haskell Numbers

- Haskell provide, as primitives, the following types:
  - `Int` single-precision integers
  - `Integer` arbitrary-precision integers
  - `Float` singe-precision floating-point numbers
  - `Double` double-precision gloating-point numbers
  - `Rational` rational number

The Numeric Type Classes

- The same symbols, +, x, and so on, are used for arithmetic on each numeric type.
  - Overloaded functions.
- All Haskell number types are instances of the type class `Num` defined by

```
class (Eq α, Show α) ⇒ Num α where
  (+), (-), (x)      :: α → α → α
  negate             :: α → α
  fromInteger        :: Integer → α
  …
  x – y              = x + negate y
```


Integral Types

- The members of the *Integral* type are two primitive types `Int` and `Integer`.
- The operators `div` and `mod` are provided as primitive.
  - If x and y are integers, and y is not zero, then `x div y` = $\lfloor x / y \rfloor$.
    - $\lfloor 13.8 \rfloor = 13$, $\lfloor -13.8 \rfloor = -14$
  - The value `x mod y` is defined by the equation `x = (x div y) * y + (x mod y)`

Lists

- Lists can be used to fetch and carry data from one function to another.
- Lists can be taken apart, rearranged, and combined with other lists.
- Lists can be summed and multiplied.
- Lists of characters can be read and printed.
- …

List Notation

- A finite list is denoted using square brackets and commas.
  - `[1,2,3]`
  - `["hello","goodbye"]`
- All the elements of a list must have the same type.
- The empty list is written as `[]`.
- A singleton list contains only one element
  - `[x]`
  - `[[]]` the empty list is its only member

List Notation

- If the elements of a list all have type α, then the list itself will be assigned the type `[α]`.
  - `[1,2,3]          :: [Int]`
  - `['h','e','l','l','o'] :: [Char]`
  - `[[1,2],[3]]      :: [[Int]]`
  - `[(+),(x)]        :: [Int → Int → Int]`
- A list may contain the same value more than once.
- Two lists are equal if and only if they contain the same value in the same order.

Lists as a data type

- A list can be constructed fro scratch by starting with an empty list and successively adding elements one by one.
  - Elements can be added to the front of the list, or the rear, or to somewhere in the middle.
- Data type declaration (list):

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

  - The constructor Cons (short for 'construct') add an element to the front of the list.
    - `[1,2,3]    Cons 1 (Cons 2 (Cons 3 Nil))`

Lists as a data type

- In functional programming, lists are defined as elements of `List α`.
  - The syntax `[α]` is used instead of `List α`.
  - The constructor `Nil` is written as `[]`
  - The constructor `Cons` is written as an infix operator `(:)`
    - `(:)` associates to the right
    - `[1,2,3] = 1:(2:(3:[])) = 1:2:3:[]`

Lists as a data type

- Like functions over data types, functions over lists can be defined by pattern matching.

```
instance (Eq α) ⇒ Eq [α] where
  [] == []          = True
  [] == (y:ys)      = False
  (x:xs) == []      = False
  (x:xs) == (y:ys)= (x == y) ∧ (xs == ys)
```


List Operations

- Some of the most commonly used functions and operations on lists.
- For each function: give the definition, illustrate its use, and state some of its properties.

Concatenation

- Two lists, both of the same type, can be concatenated to form one longer list.
- This function is denoted by the binary operator `++`.

```
? [1,2,3] ++ [4,5]
[1,2,3,4,5]
? [1,2] ++ [] ++ [1]
[1,2,1]
```

Concatenation

- The formal definition of ++ is

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

  - The definition of ++ is by pattern matching on the left-hand argument.
  - The two patterns are disjoint and cover all cases, apart from the undefined list ⊥.
  - It follows by case exhaustion that

    `⊥ ++ ys = ⊥`

Concatenation

- It is not the case that `xs ++ ⊥ = ⊥`

```
? [1,2,3] ++ undefined
[1,2,3{Interrupted!}
```

- The list `[1,2,3] ++ ⊥` is a *partial list*; in full form it is the list `1:2:3:⊥`.
  - The evaluator can compute the first three elements, but thereafter it goes into a nonterminating computation, so we interrupt it.
- Some properties:

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
xs ++ [] = [] ++ xs = xs
```


Reverse

- This function reverses the order of elements in a finite list.

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

- The definition is

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

- In words, to reverse a list (x:xs) one reverses xs and then adds x to the end.

Length

- The length of a list is the number of elements it contains.
- The definition is

```
length          :: [α] → Int
length []       = 0
length (x:xs)   = 1 + length(xs)
```

- The nature of the list elements is irrelevant when computing the length:

```
? length [undefined,undefined]
2
```


Length

- Not every list has a well-defined length.
  - The partial lists have an undefined length
    - `⊥, x:⊥, x:y:⊥`
  - Only finite lists have well-defined lengths.
    - The list `[⊥,⊥]` is a finite list, not a partial list because it is the list `⊥:⊥:[]`, which ends in `[]` not ⊥. The computer cannot produce the elements, but it can produce the length of the list.
- The function length satisfies a distribution property:

```
length(xs ++ ys) = length xs + length ys
```


Head and Tail

- The function *head* selects the first element of a nonempty list, and *tail* selects the rest:

```
head            :: [α] → α
head []         = error "empty list"
head (x:xs)     = x
tail            :: [α] → [α]
tail []         = error "empty list"
tail (x:xs)     = xs
```

  - These are constant-time operations, since they deliver their result in one reduction step.

Init and last

- The function *last* and *init* select the last element of a nonempty list and what remains after the last element has been removed.

```
? last [1,2,3,4,5]
5
? init [1,2,3,4,5]
[1,2,3,4]
```


Init and last

- First attempt (definition):

```
last      :: [α] → α
last      = head · reverse

init      :: [α] → α
init      = reverse · tail · reverse
```

- Problem?
  - `init xs = ⊥` for all partial and infinite lists `xs`

Init and last

- Second attempt (definition):

```
last (x:xs) = if null xs then x else last xs

init (x:xs) = if null xs then [] else x:init xs
```

- With this definition
  - `init xs = xs` for all partial and infinite lists `xs`

Init and last

- Third attempt (definition):
  - Since `[x]` is an abbreviation for `x:[]`

```
last [x]        = x
last (x:xs)     = last xs
init [x]        = []
init (x:xs)     = x:init xs
```

- Problem?
  - There is a serious danger of confusion because the patterns `[x]` and `(x:xs)` are not *disjoint*.
    - The second includes the first as a special case.

Init and last

- If the order of the equations are reversed:

```
last' (x:xs)   = last' xs
last' [x]      = x
```

- The definition of `last'` would simply be incorrect.
  - `last' xs = ⊥`
- It is not a good practice to write definition that depend critically on the order of the equations.

Init and last

- Definition

```
last           :: [α] → α
last []        = error "empty list"
last [x]       = x
last (x:y:ys)  = last(y:ys)

init           :: [α] → [α]
init []        = error "empty list"
init [x]       = []
init [x:y:xs)  = x:init(y:xs)
```

Topics

- Lists Operations
- Trees
- Lazy Evaluation

Concat

- The function `concat` concatenates a list of lists into one long list.

```
? concat [[1,2],[3,2,1]]
[1,2,3,2,1]
```

- Definition

```
concat           :: [[α]] → [α]
concat []        = []
concat (xs:xss)  = xs ++ concat xss
```

- Basic property:

```
concat (xss ++ yss) = concat xss ++ concat yss
```


Take and drop

- The function `take` and `drop` each take a nonnegative integer `n` and a list `xs` as arguments.
  - The value `take n xs` consists of the first `n` elements of `xs`
  - The value `drop n xs` is what remains

```
? take 3 "functional"   ? take 3 [1,2]
"fun"                   [1,2]
? drop 3 "functional"   ? drop 3 [1,2]
"ctional"               []
```


Take and drop

- Definitions:

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

drop              :: Int → [α] → [α]
drop 0 xs         = xs
drop n []         = []
drop (n+1)(x:xs)  = drop n xs
```


Take and drop

- These definitions use a combination of pattern matching with natural numbers and lists.
- Patterns are disjoint and cover all possible cases.
  - Every natural number is either zero (first equation) or
  - The successor of a natural number
    - Distinguish between an empty list (second equation) and
    - A nonempty list (third equation).

Take and drop

- There are two arguments on which pattern matching is performed
  - Pattern matching is performed on the clauses of a definition in order from the first to the last.
  - Within a clause, pattern matching is performed from left to right.

```
? take 0 ⊥
[]
? take ⊥ []
⊥
```

Take and drop

- The functions take and drop satisfy a number of useful laws:

```
take n xs ++ drop n xs = xs
```

for all (finite) natural numbers `n` and all lists `xs`.

```
take ⊥ xs ++ drop ⊥ xs = ⊥ ++ ⊥ = ⊥
```

not `xs`.

```
take m · take n = take (m min n)
drop m · drop n = drop (m + n)
take m · drop n = drop n · take(m + n)
```


List index

- A list `xs` can be indexed by a natural number `n` to find the element appearing at position `n`.
- This operation is denoted by `xs !! n`

```
? [1,2,3,4]!!2
3
? [1,2,3,4]!!0
1
```

  - Indexing begins at 0.

List index

- Definition

```
(!!)           :: [α] → Int → α
(x:xs)!!0      = x
(x:xs)!!(n+1)  = xs!!n
```

- Indexing is an expensive operation since `xs!!n` takes a number of reduction steps proportional to `n`.

Map

- The function *`map`* applies a function to each element of a list.

```
? map square [9,3]
[81,9]

? map (<3) [1,2,3]
[True,True,False]

? map nextLetter "HAL"
"IBM"
```


Map: definition

- The definition is

```
map          :: (α→β)→[α]→[β]
map f []     = []
map f (x:xs) = f x:map f xs
```

- The use of map is illustrated by the following example:
  - *"the sum of the squares of the integers from 1 up to 100"*
  - The function `sum` and `upto` can be defined by

Map: example

```
sum          :: (Num α)⇒[α]→α
sum []       = 0
sum (x:xs)   = x + sum xs
upto         :: (Integral α)⇒α→α→[α]
upto m n     = if m > n then []
                 else m:upto(m+1)n

? sum(map square(upto 1 100))
338700

[m..n] = upto m n
[m..] = from m
```

Map: laws

```
map id = id
```

- Applying the identity function to every element of a list leaves the list unchanged.
  - The two occurrences of `id` have different types; on the left `id :: α → α`, and on the right `id :: [α] → [α]`

```
map (f · g) = map f · map g
```

- Applying `g` to every element of a list, and the applying `f` to each element of the result gives the same result as applying `f · g` to the original list.

Map: laws

```
f · head = head · map f
map f · tail = tail · map f
map f · reverse = reverse · map f
map f · concat = concat · map(map f)
map f (xs ++ ys) = map f xs ++ map f ys
```

- The common theme behind each of these equations concern the types of the functions involved:

```
head      :: [α] → α
tail      :: [α] → [α]
reverse   :: [α] → [α]
concat    :: [[α]] → [α]
```


Map: laws

- Those functions do not depend in any way on the nature of the list elements.
  - They are simply combinators that shuffle, rearrange, or extract elements from lists.
  - This is why they have polymorphic types.
- We can either 'rename' the list elements (via `map f`) and then do the operation, or do the operation and then rename the elements.

Filter

- The function `filter` takes a boolean function `p` and a list `xs` and return that sublist of `xs` whose elements satisfy `p`.

```
? filter even [1,2,4,5,32]
[2,4,32]

? (sum · map square · filter even) [1..10]
220
```

- The sum of the squares of the even integers in the range 1 to 10

Filter: definition

```
filter           :: (α→Bool)→[α]→[α]
filter p []      = []
filter p (x:xs)  = if p x then x:filter p xs
                     else filter p xs
```

- Some laws

```
filter p · filter q = filter (p and q)
Filter p · concat = concat · map(filter p)
```


Zip

- The function `zip` takes two lists and returns a list of pairs of corresponding elements.

```
? zip [0..4] “hello”
[(0,’h’),(1,’e’),(2,’l’),(3,’l’),(4,’o’)]

? zip [0,1] “hello”
[(0,’h’),(1,’e’)]
```

Zip: definition

- If two lists do not have the same length, then the length of the zipped list is the shorter of the lengths of the two arguments.

```
zip               :: [α]→[β]→[(α,β)]
zip [] ys         = []
zip xs []         = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

  - What would happen if we just defined `zip [] []` instead of the two basic cases.

Unzip

- The function unzip takes a list of pairs and unzips it into two lists.

```
? unzip [(1,True),(2,True),(3,False)]
([1,2,3],[True,True,False])
```

- Definition

```
unzip  :: [(α,β)]→([α],[β])
unzip  = pair(map fst, map snd)
```


Unzip

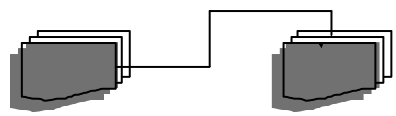
- Two basic functions on pairs are `fst` and `snd`, defined by:

```
fst         :: (α,β) → α
fst (x,y)   = x
snd         :: (α,β) → β
snd (x,y)   = y
```

- A basic function that takes pairs of functions as arguments:

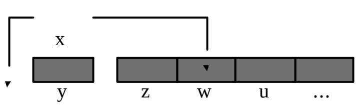
```
pair        :: (α→β,α→γ)→α→(β,γ)
pair (f,g) x  = (f x, g x)
```


Insertion Sort



```
sort [] = []
sort (x : xs) = insert x (sort xs)
```


Insertion




```
insert x (y : ys)
          | x<=y      = x : y : ys
          | x>y       = y : insert x ys
insert x [] = [x]
```


Sorting: example

sort [3,1,2]
insert 3 (sort [1,2])
insert 3 (insert 1 (sort [2]))
insert 3 (insert 1 (insert 2 (sort [])))
insert 3 (insert 1 (insert 2 []))
insert 3 (insert 1 [2])

insert 3 [1, 2]
1 : insert 3 [2]
1 : 2 : insert 3 []
1 : 2 : [3]
[1, 2, 3]

The Type of Sort

What is the type of sort?

```
sort :: [a] -> [a]
```

Can sort many different types of data.

But not all!

Consider a list of functions, for example...

The Correct Type of Sort

```
sort :: Ord a => [a] -> [a]
```

If a has an ordering...

…then sort has this type.

Sort has this type because

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

Overloaded, rather than polymorphic.

Polymorphism vs. Overloading

- A *polymorphic* function works in the same way for every type
  - Example: `length, ++`
- An *overloaded* function works in different ways for different types
  - Example: `==, <=`

A Better Way of Sorting

- Divide the list into two roughly equal halves.
- Sort each half.
- Merge the sorted halves together.

Merge Sort: definition

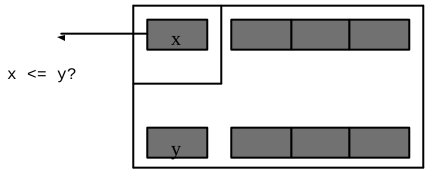
```
mergeSort xs = merge (mergeSort front)
                     (mergeSort back)
          where size = length xs `div` 2
                front = take size xs
                back = drop size xs
```

- But when are front and back smaller than `xs`?

MergeSort with Base Cases

```
mergeSort []            = []
mergeSort [x]           = [x]
mergeSort xs | size > 0  =
                    merge (mergeSort front)
                    (mergeSort back)
          where size = length xs `div` 2
                front = take size xs
                back = drop size xs
```

Merging: example




```
merge [1, 3] [2, 4] → 1 : merge [3] [2, 4]
                    → 1 : 2 : merge [3] [4]
                    → 1 : 2 : 3 : merge [] [4]
                    → 1 : 2 : 3 : [4] → [1,2,3,4]
```


Defining Merge

Requires an ordering.

```
merge :: Ord a => [a] -> [a] -> [a]
merge (x : xs) (y : ys)
      | x <= y    = x : merge xs (y : ys)
      | x > y     = y : merge (x : xs) ys
merge [] ys       = ys
merge xs []       = xs
```

One list gets smaller.

Two possible base cases.

The Cost of Sorting

Insertion Sort
Sorting n elements takes n*n/2 comparisons.

Merge Sort
Sorting n elements takes n*log2 n comparisons.

Num elements	Cost by insertion	Cost by merging
10	50	40
1000	500000	10000
1000000	500000000000	20000000

Summary: List Recursion

- Recursive case: expresses the results in terms of the same function on a shorter list.
  - `f (x:xs) = … f xs …`
- Base case(s): handles the shortest possible list.
  - `f [] = …`

Example: Counting Words

Input

A string representing a text containing many words. For example

`"hello clouds hello sky"`

Output

A string listing the words in order, along with how many times each word occurred.

`"clouds: 1\nhello: 2\nsky: 1"`

```
clouds: 1
hello: 2
sky: 1
```


Step 1: Breaking Input into Words

`"hello clouds hello sky"`

words

`["hello", "clouds", "hello", "sky"]`

Step 2: Sorting the Words

```
["hello", "clouds", "hello", "sky"]
```

sort

```
["clouds", "hello", "hello", "sky"]
```


The groupBy Function

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

groupBy p xs -- breaks xs into segments [x1,x2…], such that p xi is True for each xi in the segment.

```
groupBy (<) [3,2,4,1,5] = [[3], [2,4], [1,5]]
groupBy (==) "hello" = ["h", "e", "ll", "o"]
```


Step 3: Grouping Equal Words

```
["clouds", "hello", "hello", "sky"]
```

groupBy (==)

```
[["clouds"], ["hello", "hello"], ["sky"]]
```


Step 4: Counting Each Group

```
[["clouds"], ["hello", "hello"], ["sky"]]
```

map (λws -> (head ws, length ws))

```
[("clouds",1), ("hello", 2), ("sky",1)]
```


Step 5: Formatting Each Group

```
[("clouds",1), ("hello", 2), ("sky",1)]
```

map (λ(w,n) -> w++show n)

```
["clouds: 1", "hello: 2", "sky: 1"]
```


Step 6: Combining the Lines

```
["clouds: 1", "hello: 2", "sky: 1"]
```

unlines

```
"clouds: 1\nhello: 2\nsky: 1\n"
```

```
clouds: 1
hello: 2
sky: 1
```

The Complete Definition

```
countWords :: String -> String
countWords s =
      unlines .
      map (λ(w,n) -> w++show n) .
      map (λws -> (head ws, length ws)) .
      groupBy (==) .
      sort .
      words s
```


Trees

- Any recursive data type that exhibits a nonlinear structure is generically called a tree.
- The syntactic structure of arithmetic or functional expressions can also be modeled by a tree.
- There are numerous species and subspecies of tree.

Trees

- Trees can be classified according to
  - The precise form of the branching structure
  - The location of information within the tree
  - The relationship between the information stored in different parts of the tree

Binary Trees

- A binary tree is a tree with a simple two-way branching structure.

```
data Btree α = Leaf α | Fork(Btree α)(Btree α)
```

- A value of `Btree α` is either a *leaf node*, which contains a value of type α, or a *fork node*, which consists of two further trees, called the *left* and *right subtrees* of the node.
- A leaf is sometimes called an external node, or tip, and a fork node is sometimes called an internal node.

Binary Trees

- Example:

```
Fork(Leaf 1)(Fork(Leaf 2)(Leaf 3))
```

  - Consists of a node with a left subtree `Leaf 1` and a right subtree which consists of a left subtree `Leaf 2` and a right subtree `Leaf 3`.

```
Fork(Fork(Leaf 1)(Leaf 2))(Leaf 3)
```

  - Contains the same sequence of numbers in its leaves but the way the information is organized is different.
  - The two expressions denote different values.

Trees: size

- The `size` of a tree is the number of its leaf nodes.

```
size              :: Btree α → Int
size (Leaf x)     = 1
size (Fork xt yt) = size xt + size yt
```

  - The function `size` plays the same role for trees as `length` does for lists.

`size = length · flatten` , where

```
Flatten              :: Btree α → [α]
Flatten (Leaf x)     = [x]
Flatten (Fork xt yt) = flatten xt ++ flatten yt
```

Trees: height

- The `height` of a tree measures how far away the furthest leaf is.

```
height               :: Btree α → Int
height (Leaf x)     = 0
height (Fork xt yt) = 1 +
                     (height xt max height yt)
```


Reductions

- Reduction sequence: `square (3+4)`
- Two reduction policies
  - *Innermost reduction*: a reduction that contains no other reduction.
  - Outermost reduction: a reduction that is contained in no other reduction.
- Other example: `fst (square 4, square 2)`

Outermost Reduction

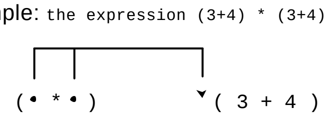
- Sometimes outermost reduction will give an answer when innermost fails to terminate.
- If both methods terminate, then they give the same result.
- Outermost reduction has the important property that if an expression has a normal form then the outermost reduction will compute it.

Outermost Reduction

- Is outermost reduction a better choice than innermost reduction?
- Problem: outermost reduction can sometimes require most steps than innermost reductions.
  - The problem arises with any function whose definition contains repeated occurrences of an argument.

Outermost Reduction

- The problem can be solved by representing expressions as graphs rather than trees.
  - Unlike trees, graphs can share subexpressions.
- Example: `the expression (3+4) * (3+4)`



  - Each occurrence of 3+4 is represented by an arrow, called a pointer, to a single instance of (3+4)

Outermost Reduction

- Using outermost graph reduction has only three steps.
  - The representation of expressions as graphs means that duplicated subexpressions can be shared and reduced at most once.
- With graph reduction, outermost reduction never takes more steps than innermost reduction.

Lazy vs. Eager Evaluation

- Outermost graph reduction is called *lazy evaluation*.
- Innermost graph reduction is called *eager evaluation*.
- Lazy evaluation is adopted by Haskell:
  1. It terminates whenever any reduction order terminates.
  2. It requires no more (and possibly fewer) steps than eager evaluation.