Lecture 1 — Functional Programming

Roy Crole
Department of Computer Science
University of Leicester
October 6, 2005

Overview of Lecture 1

• From Imperative to Functional Programming:
  – What is imperative programming?
  – What is functional programming?

• Key Ideas in Functional Programming:
  – Types: Provide the data for our programs
  – Functions: These are our programs!

• Advantages:
  – Haskell code is typically short
  – Haskell code is close to the algorithms used

What is Imperative Program — Adding up square numbers

• Problem: Add up the first \( n \) square numbers
  \[ \text{ssquares } n = 0^2 + 1^2 + ... + + (n-1)^2 + n^2 \]

• Program: We could write the following in Java
  
  ```java
  public int ssquares(int n){
    private int s,i;
    s=0; i=0;
    while (i<n) {i:=i+1;s:=s+i*i;}
  }
  ```

• Execution: We may visualize running the program as follows
  
<table>
<thead>
<tr>
<th>(Stack) Memory</th>
<th>(Stack) Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = ??</td>
<td>s = 30</td>
</tr>
<tr>
<td>i = ??</td>
<td>i = 4</td>
</tr>
</tbody>
</table>

• Key Idea: Imperative programs transform the memory

The Two Aspects of Imperative Programs

• Functional Content: What the program achieves
  – Programs take some input values and return an output value
  – \( \text{ssquares} \) takes a number and returns the sum of the squares up to and including that number

• Implementational Content: How the program does it
  – Imperative programs transform the memory using variable declarations and assignment statements
  – \( \text{ssquares} \) uses variables \( i \) and \( s \) to represent locations in memory. The program transforms the memory until \( s \) contains the correct number.
What is Functional Programming?

- **Motivation:** Problems arise as programs contain two aspects:
  - High-level algorithms and low-level implementational features
  - Humans are good at the former but not the latter

- **Idea:** The idea of functional programming is to
  - Concentrate on the functional (I/O) behaviour of programs
  - Leave memory management to the language implementation

- **Summary:** Functional languages are more abstract and avoid low level detail.

Roy Crole
Leicester, October 6, 2005

A Functional Program — Summing squares in Haskell

- **Types:** First we give the type of summing-squares
  \[ \text{hssquares :: Int -> Int} \]

- **Functions:** Our program is a function
  \[
  \begin{align*}
  \text{hssquares 0} &= 0 \\
  \text{hssquares n} &= n*n + \text{hssquares}(n-1)
  \end{align*}
  \]

- **Evaluation:** Run the program by expanding definitions
  \[
  \begin{align*}
  \text{hssquares 2} &\Rightarrow 2*2 + \text{hssquares 1} \\
  &\Rightarrow 4 + (1*1 + \text{hssquares 0}) \\
  &\Rightarrow 4 + (1 + 0) \Rightarrow 5
  \end{align*}
  \]

- **Comment:** No mention of memory in the code.

Roy Crole
Leicester, October 6, 2005

Key Ideas in Functional Programming I — Types

- **Motivation:** Recall from CO1003/4 that types model data.

- **Integers:** \( \text{Int} \) is the Haskell type \( \{\ldots,-2,-1,0,1,2,\ldots\} \)

- **String:** \( \text{String} \) is the Haskell type of lists of characters.

- **Complex Datatypes:** Can be made from the basic types, eg lists of integers.

- **Built in Operations ("Functions on types"):**
  - Arithmetic Operations: \(+\) \(-\) \(\text{div}\) \(\text{mod}\) \(\text{abs}\)
  - Ordering Operations: \(>\) \(>=\) \(/=\) \(<=\) \(<\)

Roy Crole
Leicester, October 6, 2005

Key Ideas in Functional Programming II — Functions

- **Intuition:** Recall from CO1011, a function \( f:a \rightarrow b \) between sets associates to every input-value a unique output-value

  \[
  x \in a \rightarrow \quad \text{Function } f \quad \rightarrow \quad y \in b
  \]

- **Example:** The square and cube functions are written
  \[
  \begin{align*}
  \text{square :: Int -> Int} &\quad \text{cube :: Int -> Int} \\
  \text{square x} &= x * x & \text{cube x} &= x * \text{square x}
  \end{align*}
  \]

- **In General:** In Haskell, functions are defined as follows
  \[
  \langle\text{function-name}\rangle :: \langle\text{input type}\rangle\rightarrow\langle\text{output type}\rangle \\
  \langle\text{function-name}\rangle \langle\text{variable}\rangle = \langle\text{expression}\rangle
  \]

Roy Crole
Leicester, October 6, 2005
Functions with Multiple Arguments

- **Intuition:** A function $f$ with $n$ inputs is written $f: :a_1 \rightarrow \ldots \rightarrow a_n \rightarrow a$

  $x_1 \in a_1 \rightarrow \ldots \rightarrow x_n \in a_n \rightarrow y \in a$

- **Example:** The "distance" between two integers

  ```
  diff :: Int -> Int -> Int
  diff x y = abs (x - y)
  ```

- **In General:**

  $\langle$function-name$\rangle$ $\langle$type 1$\rangle$-$\ldots$-$\langle$type n$\rangle$-$\langle$output-type$\rangle$

  $\langle$function-name$\rangle$ $\langle$variable 1$\rangle$-$\ldots$-$\langle$variable n$\rangle$ $=$ $\langle$expression$\rangle$

Key Idea III — Expressions

- **Motivation:** Get the result/output of a function by applying it to an argument/input

  - Write the function name followed by the input

- **In General:** Application is governed by the typing rule

  - If $f$ is a function of type $a \rightarrow b$, and $e$ is an expression of type $a$,

  - then $f e$ is the result of applying $f$ to $e$ and has type $b$

- **Key Idea:** Expressions are fragments of code built by applying functions to arguments.

Key Ideas in Functional Programming IV — Evaluating Expressions

- **More Expressions:** Use quotes to turn functions into infix operations and brackets to turn infix operations into functions

  ```
  5 * 4  (+) 5 4  mod 13 4  13 'mod' 4  5-(-(3*4)  (5-3)*4  7 >= (3*3)  5 * (-1)
  ```

- **Precedence:** Usual rules of precedence and bracketing apply

- **Example of Evaluation:**

  ```
  cube(square3) ⇒ (square 3) * square (square 3)
  ⇒ (3*3) * ((square 3) * (square 3))
  ⇒ 9 * ((3*3) * (3*3))
  ⇒ (9 * (9*9)
  ⇒ 729
  ```

- The final outcome of an evaluation is called a value

Summary — Comparing Functional and Imperative Programs

- **Difference 1:** Level of Abstraction

  - Imperative Programs include low level memory details
  - Functional Programs describe only high-level algorithms

- **Difference 2:** How execution works

  - Imperative Programming based upon memory transformation
  - Functional Programming based upon expression evaluation

- **Difference 3:** Type systems

  - Type systems play a key role in functional programming
Today You Should Have Learned ...

- **Types:** A type is a collection of data values
- **Functions:** Transform inputs to outputs
  - We build complex expressions by defining functions and applying them to other expressions
  - The simplest (evaluated) expressions are (data) values
- **Evaluation:** Calculates the result of applying a function to an input
  - Expressions can be evaluated by hand or by HUGS to values
- **Now:** Go and look at the first practical!

Overview of Lecture 2

- **New Types:** Today we shall learn about the following types
  - The type of booleans: `Bool`
  - The type of characters: `Char`
  - The type of strings: `String`
  - The type of fractions: `Float`
- **New Functions and Expressions:** And also about the following functions
  - Conditional expressions and guarded functions
  - Error handling and local declarations

Booleans and Logical Operators

- **Values of** `Bool`: Contains two values — `True`, `False`
- **Logical Operations:** Various built-in functions
  
  \[
  \&\& \qquad : \quad \text{Bool} \to \text{Bool} \to \text{Bool} \\
  || \qquad : \quad \text{Bool} \to \text{Bool} \to \text{Bool} \\
  \text{not} \quad : \quad \text{Bool} \to \text{Bool}
  \]
- **Example:** Define the exclusive-OR function which takes as input two booleans and returns `True` just in case they are different

\[
\text{exOr} \quad : \quad \text{Bool} \to \text{Bool} \to \text{Bool}
\]
Conditionals — If statements

- **Example:** Maximum of two numbers
  \[
  \text{maxi} :: \text{Int} \to \text{Int} \to \text{Int} \\
  \text{maxi} \ n \ m = \text{if} \ n \geq m \text{ then } n \text{ else } m
  \]

- **Example:** Testing if an integer is 0
  \[
  \text{isZero} :: \text{Int} \to \text{Bool} \\
  \text{isZero} \ x = \text{if} \ (x == 0) \text{ then } \text{True} \text{ else } \text{False}
  \]

- **Conditionals:** A *conditional expression* has the form
  \[
  \text{if } b \text{ then } e_1 \text{ else } e_2
  \]
  where
  - \(b\) is an expression of type \(\text{Bool}\)
  - \(e_1\) and \(e_2\) are expressions with the same type

Guarded functions — An alternative to if-statements

- **Example:** `doubleMax` returns double the maximum of its inputs
  \[
  \text{doubleMax} :: \text{Int} \to \text{Int} \to \text{Int} \\
  \text{doubleMax} \ x \ y = \begin{cases} 
  2 \times x & \text{if } x \geq y \\
  2 \times y & \text{if } x < y 
  \end{cases}
  \]

- **Definition:** A guarded function is of the form
  \[
  \langle \text{function-name} \rangle :: \langle \text{type 1} \rangle \to \langle \text{type n} \rangle \to \langle \text{output type} \rangle
  \]
  \[
  \langle \text{function-name} \rangle (\var 1)...(\var n) | \langle \text{guard 1} \rangle = \langle \text{expression 1} \rangle \\
  | ... = ... \\
  | \langle \text{guard m} \rangle = \langle \text{expression m} \rangle
  \]
  where \(\langle \text{guard 1} \rangle, ..., \langle \text{guard m} \rangle :: \text{Bool}\)

The Char type

- **Elements of** \(\text{Char}\): Letters, digits and special characters
- **Forming elements of** \(\text{Char}\): Single quotes form characters:
  \[
  'd' :: \text{Char} \quad '3' :: \text{Char}
  \]
- **Functions:** Characters have codes and conversion functions
  \[
  \text{chr} :: \text{Int} \to \text{Char} \quad \text{ord} :: \text{Char} \to \text{Int}
  \]
- **Examples:** Try them out!
  \[
  \text{offset} :: \text{Int} \\
  \text{offset} = \text{ord} 'A' - \text{ord} 'a' \\
  \text{capitalize} :: \text{Char} \to \text{Char} \\
  \text{capitalize} \ \text{ch} = \text{chr} (\text{ord} \ \text{ch} + \text{offset}) \\
  \text{isLower} :: \text{Char} \to \text{Bool} \\
  \text{isLower} \ x = ('a' \leq x) \&\& (x \leq 'z')
  \]

The String type

- **Elements of** \(\text{String}\): Lists of characters
- **Forming elements of** \(\text{String}\): Double quotes form strings
  \[
  "\text{Newcastle Utd}" \quad "\text{1a}"
  \]
- **Special Strings:** Newline and Tab characters
  \[
  "\text{Super \n Alan}" \quad "\text{1\t2\t3}"
  \]
- **Combining Strings:** Strings can be combined by `++`
  \[
  "\text{Super }" ++ "\text{Alan }" ++ "\text{Shearer}" = "\text{Super Alan Shearer}"
  \]
- **Example:** `duplicate` gives two copies of a string
The type of Fractions Float

- **Elements of Float**: Contains decimals, eg -21.3, 23.1e-2

- **Built in Functions**: Arithmetic, Ordering, Trigonometric

- **Conversions**: Functions between Int and String
  - ceiling, floor, round :: Float -> Int
  - fromIntegral :: Int -> Float
  - show :: Float -> String
  - read :: String -> Float

- **Overloading**: Overloading is when values/functions belong to several types
  - 2 :: Int show :: Int -> String
  - 2 :: Float show :: Float -> String

Error-Handling

- **Motivation**: Informative error messages for run-time errors

- **Example**: Dividing by zero will cause a run-time error
  - `myDiv :: Float -> Float -> Float
  - myDiv x y = x/y`

- **Solution**: Use an error message in a guarded definition
  - `myDiv :: Float -> Float -> Float
  - myDiv x y
  - | y /= 0 = x/y
  - | otherwise = error "Attempt to divide by 0"

- **Execution**: If we try to divide by 0 we get
  - Prelude> mydiv 5 0
  - Program error: Attempt to divide by 0

Local Declarations — where

- **Motivation**: Functions will often depend on other functions

- **Example**: Summing the squares of two numbers
  - `sq :: Int -> Int
  - sq x = x * x`
  - `sumSquares :: Int -> Int -> Int
  - sumSquares x y = sq x + sq y`

- **Problem**: Such definitions clutter the top-level environment

- **Answer**: Local definitions allow auxiliary functions
  - `sumSquares2 :: Int -> Int -> Int
  - sumSquares2 x y = sq x + sq y
  - where sq z = z * z`

Extended Example

- **Quadratic Equations**: The solutions of $ax^2 + bx + c = 0$ are
  - $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

- **Types**: Our program will have type
  - `roots :: Float -> Float -> Float -> String`

- **Guards**: There are 3 cases to check so use a guarded definition
  - `roots a b c
  - | a == 0 = ....
  - | b*b-4*a*c == 0 = ....
  - | otherwise = ....`
The function roots — Stage II

- **Code:** Now we can add in the answers

  ```haskell
define roots a b c as
  a == 0
  error "Not a quadratic eqn"
  b*b-4*a*c == 0
  One root: \(\frac{-b}{2*a}\)
  otherwise
  Two roots:
  \(\frac{-b + \sqrt{b^2-4ac}}{2*a}\)
  and
  \(\frac{-b - \sqrt{b^2-4ac}}{2*a}\)
```

- **Problem:** This program uses several expressions repeatedly
  - Being cluttered, the program is hard to read
  - Similarly the program is hard to understand
  - Repeated evaluation of the same expression is inefficient

---

The final version of roots

- **Local decs:** Expressions used repeatedly are made local

  ```haskell
define roots a b c as
  a == 0
  error "Not a quadratic eqn"
  disc == 0
  One root: \(\frac{-b}{2*a}\)
  otherwise
  Two roots:
  \(\frac{-b + \sqrt{b^2-4ac}}{2*a}\)
  and
  \(\frac{-b - \sqrt{b^2-4ac}}{2*a}\)
  where
  disc = b*b-4*a*c
  offset = (sqrt disc) / 2*a
  centre = -b/2*a
```

---

Today You Should Have Learned

- **Types:** We have learned about Haskell’s basic types. For each type we learned
  - Its basic values (elements)
  - Its built in functions

- **Expressions:** How to write expressions involving
  - Conditional expressions and Guarded functions
  - Error Handling and Local Declarations
Overview of Lecture 3

- **Building New Types:** Today we will learn about the following compound types
  - Pairs
  - Tuples
  - Type Synonyms

- **Describing Types:** As with basic types, for each type we want to know
  - What are the values of the type
  - What expressions can we write and how to evaluate them

---

New Types from Old I — Pair Types and Expressions

- **Examples:** For instance
  - The expression (5,3) has type (Int, Int)
  - The name ("Alan","Shearer") has type (String, String)
  - The performance ("Newcastle", 22) has type (String,Int)

- **Question:** What are the values of a pair type?

- **Answer:** A pair type contains pairs of values, ie
  - If \( e_1 \) has type \( a \) and \( e_2 \) has type \( b \)
  - Then \( (e_1,e_2) \) has type \( (a,b) \)

---

From simple data values to complex data values

- **Motivation:** Data for programs modelled by values of a type

- **Problem:** Single values in basic types too simple for real data

- **Example:** A point on a plane can be specified by
  - A number for the x-coordinate and another for the y-coordinate

- **Example:** A person’s complete name could be specified by
  - A string for the first name and another for the second name

- **Example:** The performance of a football team could be
  - A string for the team and a number for the points

---

Functions using Pairs

- **Types:** Pair types can be used as input and/or output types

- **Examples:** The built in functions \( \text{fst} \) and \( \text{snd} \) are vital
  
  \[
  \text{fst} :: (a,b) \rightarrow a \\
  \text{fst} (x,y) = x \\
  \text{winUpdate} :: (\text{String},\text{Int}) \rightarrow (\text{String},\text{Int}) \\
  \text{winUpdate} (x,y) = (x,y+3) \\
  \text{movePoint} :: \text{Int} \rightarrow \text{Int} \rightarrow (\text{Int},\text{Int}) \rightarrow (\text{Int},\text{Int}) \\
  \text{movePoint} m n (x,y) = (x+m,y+n)
  \]

- **Key Idea:** If input is a pair-type, use \( (\langle \text{var1} \rangle,\langle \text{var2} \rangle) \) in definition

- **Key Idea:** If output is a pair-type, result is often \( (\langle \text{exp1} \rangle,\langle \text{exp2} \rangle) \)
New Types from Old II — Tuple Types and Expressions

• **Motivation:** Some data consists of more than two parts

• **Example:** Person on a mailing list
  - Specified by name, telephone number, and age
  - A person $p$ on the list can have type $(\text{String, Int, Int})$

• **Idea:** Generalise pairs of types to collections of types

• **Type Rule:** Given types $a_1, \ldots, a_n$, then $(a_1, \ldots, a_n)$ is a type

• **Expression Formation:** Given expressions $e_1::a_1, \ldots, e_n::a_n$, then
  \[(e_1, \ldots, e_n)::(a_1, \ldots, a_n)\]

---

Functions using Tuples

• **Example 1:** Write a function to test if a customer is an adult
  \[
  \text{isAdult} :: (\text{String, Int, Int}) \rightarrow \text{Bool}
  \]
  \[
  \text{isAdult} (\text{name, tel, age}) = (\text{age} \geq 18)
  \]

• **Example 2:** Write a function to update the telephone number
  \[
  \text{updateMove} :: (\text{String, Int, Int}) \rightarrow \text{Int} \rightarrow (\text{String, Int, Int})
  \]

• **Example 3:** Write a function to update age after a birthday
  \[
  \text{updateAge} :: (\text{String, Int, Int}) \rightarrow (\text{String, Int, Int})
  \]

---

General Definition of a Function: Patterns with Tuples

• **Definition:** Functions now have the form
  \[
  \langle\text{function-name}\rangle :: \langle\text{type 1}\rangle \rightarrow \ldots \rightarrow \langle\text{type n}\rangle \rightarrow \langle\text{out-type}\rangle
  \]
  \[
  \langle\text{function-name}\rangle \langle\text{pat 1}\rangle \ldots \langle\text{pat n}\rangle = \langle\text{exp n}\rangle
  \]

• **Patterns:** Patterns are
  - Variables $x$: Use for any type
  - Constants 0, True, ‘cherry’: Definition by cases
  - Tuples $(x, \ldots, z)$: If the argument has a tuple-type
  - Wildcards $\_\_\$: If the output doesn’t use the input

• **In general:** Use several lines and mix patterns.

---

More Examples

• **Example:** Using values and wildcards
  \[
  \text{isZero} :: \text{Int} \rightarrow \text{Bool}
  \]
  \[
  \text{isZero} \ 0 = \text{True}
  \]
  \[
  \text{isZero} \ _ = \text{False}
  \]

• **Example:** Using tuples and multiple arguments
  \[
  \text{expand} :: \text{Int} \rightarrow (\text{Int,Int}) \rightarrow (\text{Int,Int,Int})
  \]
  \[
  \text{expand} \ n \ (x,y) = (n, n*x, n*y)
  \]

• **Example:** Days in the month
  \[
  \text{days} :: \text{String} \rightarrow \text{Int} \rightarrow \text{Int}
  \]
  \[
  \text{days} \ \text{‘January’} \ x = 31
  \]
  \[
  \text{days} \ \text{‘February’} \ x = \text{if \ isLeap} \ x \ \text{then} \ 29 \ \text{else} \ 28
  \]
  \[
  \text{days} \ \text{‘March’} \ x = 31
  \]
  \[
  \text{.....}
  \]
New Types from Old III — Type Synonyms

• Motivation: More descriptive names for particular types.

• Definition: Type synonyms are declared with the keyword type.

```haskell
type Team = String
type Goals = Int
type Match = ((Team, Goals), (Team, Goals))
```

```haskell
numu :: Match
numu = (("Newcastle", 4), ("Manchester Utd", 3))
```

• Functions: Types of functions are more descriptive, same code

```haskell
homeTeam :: Match -> Team
homeTeam = 

totalGoals :: Match -> Goals
totalGoals = 
```

Today You Should Have Learned

• Tuples: Collections of data from other types

• Pairs: Pairs, triples etc are examples of tuples

• Type synonyms: Make programs easier to understand

• Pattern Matching: General description of functions covering definition by cases, tuples etc.

• Pitfall! What is the difference between

```haskell
addPair :: (Int,Int) -> Int
addPair (x,y) = x + y

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

Overview of Lecture 4 — List Types

• Lists: What are lists?
  - Forming list types
  - Forming elements of list types

• Functions over lists: Some old friends, some new friends
  - Functions from CO1003/4: cons, append, head, tail
  - Some new functions: map, filter

• Clarity: Unlike Java, Haskell treatment of lists is clear
  - No list iterators!
List Types and Expressions

- **Example 1:** [3, 5, 14] :: [Int] and [3, 4+1, double 7] :: [Int]

- **Example 3:** ['d', 't', 'g'] :: [Char]

- **Example 4:** [['d'], ['d', 't'], ['d', 't', 'g']] :: [[Char]]

- **Example 5:** [double, square, cube] :: [Int -> Int]

**Empty List:** The empty list is [] and belongs to all list types

**List Expressions:** Lists are written using square brackets [...]  
- If e1, ..., en are expressions of type a  
- Then [e1, ..., en] is an expression of type [a]

Some built in functions - Two Infix Operators

- **Cons:** The cons function: adds an element to a list  
  : :: a -> [a] -> [a]

- **Example:**
  1    : [2,3,4] = [1,2,3,4]
  addone : [square] = [addone, square]
  'a'    : ['b', 'z'] = ['a', 'b', 'z']

- **Append:** Append joins two lists together  
  ++ :: [a] -> [a] -> [a]

  [True, True] ++ [False] = [True, True, False]
  [1,2] ++ (3 ++ [4,5]) = [1,2,3,4,5]
  ([1,2] ++ [3]) ++ [4,5] = [1,2,3,4,5]
  [] ++ [54.6, 67.5] = [54.6, 67.5]
  [6,5] ++ (4 : [7,3]) = [6,5,4,7,3]

More Built In Functions

- **Head and Tail:** Head gives the first element of a list, tail the remainder
  
  head [double, square] = double
  head ([5,6]++[6,7]) = 5
  
  tail [double, square] = [square]
  tail ([5,6]++[6,7]) = [6,6,7]

- **Length and Sum:** The length of a list and the sum of a list of integers
  
  length (tail [1,2,3]) = 2
  sum [1+4,8,45] = 58

- **Sequences:** The list of integers from 1 to n is written
  
  [1 .. n]

Two New Functions — Map And Filter

- **Map:** Map is a function which has two inputs.  
  - The first input is a function eg f  
  - The second is a list eg [e1, e1, e3]

  The output is the list obtained by applying the function to every element of the input list eg [f e1, f e2, f e3]

- **Filter:** Filter is a function which has two inputs.  
  - The first is a test, ie a function returning a Bool.  
  - The second is a list

  The output is the list of elements of the input list which the function maps to True, ie those elements which pass the test.
Using Map and Filter

- **Even Numbers**: The even numbers less than or equal to \( n \)
  
  \[ \text{evens} :: \text{Int} \rightarrow [\text{Int}] \]

- **Solution 1** — Using \textsf{filter}.
  
  \[
  \text{evens2} :: \text{Int} \rightarrow [\text{Int}]
  \text{evens2} \ n = \text{filter isEven} \ [1 .. \ n]
  \text{where isEven} \ x = (x \mod 2 == 0)
  \]

- **Solution 2** — Using \textsf{map}

Today You Should Have Learned

- **Types**: We have looked at list types
  
  - What list types and list expressions looks like
  
  - What built in functions are available

- **New Functions**:
  
  - Map: Apply a function to every member of a list
  
  - Filter: Delete those that don't satisfy a property or test

- **Algorithms**: Develop an algorithm by asking
  
  - Can I solve this problem by applying a function to every member of a list or by deleting certain elements.

Overview of Lecture 5

- **Recall Map**: Map is a function which has two inputs.
  
  \[
  \text{map add2} \ [2, 5, 6] = [4, 7, 8]
  \]

- **Recall Filter**: Filter is a function which has two inputs.
  
  \[
  \text{filter isEven} \ [2, 3, 4, 5, 6, 7] = [2, 4, 6]
  \]

- **List comprehension**: An alternative way of writing lists
  
  - Definition of list comprehension
  
  - Comparison with \textsf{map} and \textsf{filter}
**List Comprehension — An alternative to map and filter**

- **Example 1:** If \( ex = [2,4,7] \) then
  \[
  [ 2*e \mid e \leftarrow xs ] = [4,8,14]
  \]

- **Example 2:** If \( isEven :: \text{Int} \rightarrow \text{Bool} \) tests for even-ness
  \[
  [ \text{isEven} e \mid e \leftarrow xs ] = [\text{True}, \text{True}, \text{False}]
  \]

- **In General:** (Simple) list comprehensions are of the form
  \[
  [ \langle \text{exp} \rangle \mid \langle \text{variable} \rangle \leftarrow \langle \text{list-exp} \rangle ]
  \]

- **Evaluation:** The meaning of a list comprehension is
  - Take each element of \( \text{list-exp} \), evaluate the expression \( \text{exp} \) for each element and return the results in a list.

---

**Using List Comprehensions Instead of map**

- **Example 1:** A function which doubles a list’s elements
  \[
  \text{double} :: [\text{Int}] \rightarrow [\text{Int}]
  \]

- **Example 2:** A function which tags an integer with its evenness
  \[
  \text{isEvenList} :: [\text{Int}] \rightarrow [(\text{Int}, \text{Bool})]
  \]

- **Example 3:** A function to add pairs of numbers
  \[
  \text{addpairs} :: [(\text{Int}, \text{Int})] \rightarrow [\text{Int}]
  \]

- **In general:** map \( f \ l = [f x \mid x \leftarrow l] \)

---

**General Form of List Comprehension**

- **In General:** These list comprehensions are of the form
  \[
  [ \langle \text{exp} \rangle \mid \langle \text{variable} \rangle \leftarrow \langle \text{list-exp} \rangle , \langle \text{test} \rangle ]
  \]

- **Example:** Infact, we can use several tests — if \( l = [2,5,8,10] \)
  \[
  [ 2*e \mid e \leftarrow l , \text{isEven} e , e>3 ] = [16,20] \]

- **Key Example:** Cartesian product of two lists is a list of all pairs, such that for each pair, the first component comes from the first list and the second component from the second list.
  \[
  [ (x,y) \mid x\leftarrow[1,2,3], y\leftarrow[\text{‘a’, ‘b’, ‘c’}] ]
  = [(1,’a’), (1,’b’) ... ]
  \]

- **Example:**
  - `league :: [Team] games = [ (t1,t2) | t1 <- league, t2 <- league, t1 /= t2]`
Removing Duplicates

- **Problem:** Given a list remove all duplicate entries

- **Algorithm:** Given a list,
  - Keep first element
  - Delete all occurrences of the first element
  - Repeat the process on the tail

- **Code:**

Today You Should Have Learned

- **List Types:** We have looked at list types
  - What list types and list expressions looks like
  - What built in functions are available

- **List comprehensions:** Like `filter` and `map`. They allow us to
  - Select elements of a list
  - Delete those that don’t satisfy certain properties
  - Apply a function to each element of the remainder

Overview of Lecture 6

- **Recursion:** General features of recursion
  - What is a recursive function?
  - How do we write recursive functions?
  - How do we evaluate recursive functions?

- **Recursion over Natural Numbers:** Special features
  - How can we guarantee evaluation works?
  - Recursion using patterns.
  - Avoiding negative input.
What is recursion?

- **Example:** Adding up the first \( n \) squares
  \[
  \text{hssquares } n = 0^2 + 1^2 + \ldots + (n-1)^2 + n^2
  \]

- **Types:** First we give the type of summing-squares
  \[
  \text{hssquares} : \text{Int} \rightarrow \text{Int}
  \]

- **Definitions:** Our program is a function
  \[
  \begin{align*}
  \text{hssquares } 0 & = 0 \\
  \text{hssquares } n & = n \times n + \text{hssquares}(n-1)
  \end{align*}
  \]

- **Key Idea:** \text{hssquares} is recursive as its definition contains \text{hssquares} in a right-hand side – the function name “recurs”.

General Definitions

- **Definition:** A function is recursive if the name recurs in its definition.

- **Intuition:** You will have seen recursion in action before
  - Imperative procedures which call themselves
  - Divide-and-conquer algorithms

- **Why Recursion:** Recursive definitions tend to be
  - Shorter, more understandable and easier to prove correct
  - Compare with a non-recursive solution
    \[
    \text{nrssquares } n = n \times (n+0.5) \times (n+1)/3
    \]

Examples of evaluation

- **Example 1:** Let’s calculate \text{hssquares} 4
  \[
  \begin{align*}
  \text{hssquares } 4 & \Rightarrow 4 \times 4 + \text{hssquares } 3 \\
  & \Rightarrow 4 \times 4 + (3 \times 3 + \text{hssquares } 2) \\
  & \ldots \\
  & \Rightarrow 16 + (9 + \ldots (1 + \text{hssquares } 0)) \\
  & \Rightarrow 16 + (9 + \ldots (1 + 0)) \Rightarrow 30
  \end{align*}
  \]

- **Example 2:** Here is a non-terminating function
  \[
  \begin{align*}
  \text{mydouble } n & = n + \text{mydouble } (n/2) \\
  \text{mydouble } 4 & \Rightarrow 4 + \text{mydouble } 2 \\
  & \Rightarrow 4 + 2 + \text{mydouble } 1 \\
  & \Rightarrow 4 + 2 + 1 + \text{mydouble } 0.5 \Rightarrow \ldots.
  \end{align*}
  \]

- **Question:** Will evaluation stop?

Problems with Recursion

- **Questions:** There are some outstanding problems
  1. Is \text{hssquares} defined for every number?
  2. Does an evaluation of a recursive function always terminate?
  3. What happens if \text{hssquares} is applied to a negative number?
  4. Are these recursive definitions sensible: \( f n = f n \), \( g n = g (n+1) \)

- **Answers:** Here are the answers
  1. Yes: The variable pattern matches every input.
  3. Trouble: Evaluation doesn’t terminate.
  4. No: Why not?
Primitive Recursion over Natural Numbers

- **Motivation:** Restrict definitions to get better behaviour

- **Idea:** Many functions defined by three cases
  - A non-recursive call selected by the pattern 0
  - A recursive call selected by the pattern \( n+1 \) (\textit{matches numbers} \( \geq 1 \))
  - The error case deals with negative input

- **Example** Our program now looks like

\[
\begin{align*}
hssquares2 \ 0 &= 0 \\
hssquares2 \ (n+1) &= (n+1) \cdot (n+1) + hssquares \ n \\
hssquares2 \ x &= \text{error} \ "\text{Negative input}" \\
\end{align*}
\]

Examples of recursive functions

- **Example 1:** star uses recursion over Int to return a string

  \[
  \begin{align*}
  \text{star} &:: \text{Int} \rightarrow \text{String} \\
  \text{star} \ 0 &= [] \\
  \text{star} \ (n+1) &= '*' : \text{star} \ n \\
  \text{star} \ n &= \text{error} \ "\text{Negative input}" \\
  \end{align*}
  \]

- **Example 2:** power is recursive in its second argument

  \[
  \begin{align*}
  \text{power} &:: \text{Float} \rightarrow \text{Int} \rightarrow \text{Float} \\
  \text{power} \ x \ 0 &= 1 \\
  \text{power} \ x \ (n+1) &= x \cdot \text{power} \ x \ n \\
  \text{power} \ x \ n &= \text{error} \ "\text{Negative input}" \\
  \end{align*}
  \]

Larger Example

- **Problem:** Produce a table for \( \text{perf} :: \text{Int} \rightarrow (\text{String}, \text{Int}) \) where \( \text{perf} \ 1 = ("\text{Arsenal"},4) \) etc.

- **Stage 1:** We need some headings and then the actual table

  \[
  \begin{align*}
  \text{printTable} &:: \text{Int} \rightarrow \text{IO}() \\
  \text{printTable} \ \text{numberTeams} &= \text{putStr} \ (\text{header} \ \text{++} \ \text{rows} \ \text{numberTeams}) \\
  \text{where} \\
  \text{header} &= "\text{Team\tPoints\n}" \\
  \end{align*}
  \]

- **Stage 2:** Convert each “row” to a string, recursively.

  \[
  \begin{align*}
  \text{rows} &:: \text{Int} \rightarrow \text{String} \\
  \text{rows} \ 0 &= \ldots. \\
  \text{rows} \ (n+1) &= \ldots. \\
  \text{rows} \ - &= \ldots. \\
  \end{align*}
  \]
The Function rows

- **Base Case:** If we want no entries, then just return 

  \[ \text{rows } 0 = [ ] \]

- **Recursive Case:** Convert \((n + 1)\)-rows by
  
  - recursively converting the first \(n\)-rows, and
  
  - adding on the \((n+1)\)-th row

- **Code:** Code for the recursive call

The Final Version

```hs
perf :: Int -> (String,Int)
perf 1 = ("Arsenal",4)
perf 2 = ("Notts",5)
perf 3 = ("Chelsea",7)
perf n = error "perf out of range"

rows :: Int -> String
rows 0 = []
rows (n+1) = rows n ++
  fst(perf(n+1)) ++ "\t\t " ++
  show(snd(perf(n+1))) ++ "\n"
rows _ = error"rows out of range"

printTable :: Int -> IO()
printTable numberTeams = putStr(header ++ rows numberTeams)
  where
    header = "Team\t\t Points\n"
```

Today You Should Have Learned

- **Recursion:** Allows new functions to be written.
  
  - Advantages: Clarity, brevity, tractability
  
  - Disadvantages: Evaluation may not stop

- **Primitive Recursion:** Avoids bad behaviour of some recursive functions
  
  - The value at 0 is non-recursive
  
  - Each recursive call uses a smaller input
  
  - An error-clause catches negative inputs

- **Algorithm:** Ask yourself, what needs to be done to the recursive call to get the answer.
Overview of Lecture 7

- **Lists:** Another look at lists
  - Lists are a recursive structure
  - Every list can be formed by [] and :

- **List Recursion:** Primitive recursion for Lists
  - How do we write primitive recursive functions
  - Examples — ++, length, head, tail, take, drop, zip

- **Avoiding Recursion?**: List comprehensions revisited

Recursion over lists

- **Question:** This lecture is about the following question
  - We know what a recursive function over Int is
  - What is a recursive function over lists?

- **Answer:** In general, the answer is the same as before
  - A recursive function mentions itself in its definition
  - Evaluating the function may reintroduce the function
  - Hopefully this will stop at the answer

Another Look at Lists

- **Recall:** The two basic operations concerning lists
  - The empty list []
  - The cons operator (:): a -> [a] -> [a]

- **Key Idea:** Every list is either empty, or of the form x:xs
  
  \[ [2,3,7] = 2:3:7:[] \]
  
  \[ [\text{True, False}] = \text{True:False:[]} \]

- **Recursion:** Define recursive functions using the scheme
  - Non-recursive call: Define the function on the empty list []
  - Recursive call: Define the function on \( x:xs \) by using the function only on \( xs \)

Examples of Recursive Functions

- **Example 1:** Doubling every element of an integer list
  
  \[
  \text{double} :: [\text{Int}] \rightarrow [\text{Int}]
  \text{double} [] = []
  \text{double} (x:xs) = (2*x) : \text{double} xs
  \]

- **Example 2:** Selecting the even members of a list
  
  \[
  \text{onlyEvens} :: [\text{Int}] \rightarrow [\text{Int}]
  \text{onlyEvens} [] = []
  \text{onlyEvens} (x:xs) = \text{if isEven x then} \; x \; \text{rest else rest}
  \text{where rest = onlyEvens xs}
  \]

- **Example 3:** Flattening some lists
  
  \[
  \text{flatten} :: [[\text{a}]] \rightarrow [\text{a}]
  \text{flatten} [] = []
  \text{flatten} (x:xs) = x ++ \text{flatten} xs
  \]
The General Pattern

- **Definition:** Primitive Recursive List Functions are given by

\[
\text{function-name}[] = \langle \text{expression 1} \rangle \\
\text{function-name}(x:xs) = \langle \text{expression 2} \rangle
\]

where

\[
\langle \text{expression 1} \rangle \text{ does not contain } \langle \text{function-name} \rangle \\
\langle \text{expression 2} \rangle \text{ may contain expressions } \langle \text{function-name} \rangle \text{ xs}
\]

- **Compare:** Very similar to recursion over Int

\[
\text{function-name}0 = \langle \text{expression 1} \rangle \\
\text{function-name}(n+1) = \langle \text{expression 2} \rangle
\]

where

\[
\langle \text{expression 1} \rangle \text{ does not contain } \langle \text{function-name} \rangle \\
\langle \text{expression 2} \rangle \text{ may contain expressions } \langle \text{function-name} \rangle \text{ n}
\]

More Examples:

- **Example 4:** Append is defined recursively

\[\text{append} :: [a] -> [a] -> [a]\]

- **Example 5:** Testing if an integer is an element of a list

\[\text{member} :: \text{Int} -> [\text{Int}] -> \text{Bool}\]

- **Example 6:** Reversing a list

\[\text{reverse} :: [a] -> [a]\]

What can we do with a list?

- **Mapping:** Applying a function to every member of the list

\[\text{double} [2,3,72,1] = [2*2, 2*3, 2*72, 2*1]\]

\[\text{isEven} [2,3,72,1] = [\text{True, False, True, False}]\]

- **Filtering:** Selecting particular elements

\[\text{onlyEvens} [2,3,72,1] = [2,72]\]

- **Taking Lists Apart:** head, tail, take, drop

- **Combining Lists:** zip

- **Folding:** Combining the elements of the list

\[\text{sumList} [2,3,7,2,1] = 2 + 3 + 7 + 2 + 1\]

List Comprehension Revisited

- **Recall:** List comprehensions look like

\[\{ \langle \text{exp} \rangle | \langle \text{variable} \rangle <- \langle \text{list-exp} \rangle, \langle \text{test} \rangle \}\]

- **Intuition:** Roughly speaking this means

- Take each element of the list \langle \text{list-exp} \rangle

- Check they satisfy \langle \text{test} \rangle

- Form a list by applying \langle \text{exp} \rangle to those that do

- **Idea:** Equivalent to filtering and then mapping. As these are recursive, so are list comprehensions although the recursion is hidden
Today You Should Have Learned

• **List Recursion:** Lists are recursive data structures
  - Hence, functions over lists tend to be recursive
  - But, as before, general recursion is badly behaved

• **Primitive List Recursion:** Similar to natural numbers
  - A non-recursive call using the pattern `[]`
  - A recursive call using the pattern `(x:xs)`

• **List comprehension:** An alternative way of doing some recursion

---

Overview of Lecture 8

• **Problem:** Our restrictions on recursive functions are too severe

• **Solution:** New definitional formats which keep termination
  - Using new patterns
  - Generalising the recursion scheme

• **Examples:** Applications to integers and lists

• **Sorting Algorithms:** What is a sorting algorithm?
  - Insertion Sort, Quicksort and Mergesort

---

Lecture 8 — More Complex Recursion

**Roy Crole**
Department of Computer Science
University of Leicester
October 6, 2005

More general forms of primitive recursion

• **Recall:** Our primitive recursive functions follow the scheme
  
  - **Base Case:** Define the function non-recursively at 0
  
  - **Inductive Case:** Define the function at (n+1) in terms of the function at n

    \[
    \begin{align*}
    &\langle \text{function-name} \rangle 0 = \langle \text{exp 1} \rangle \\
    &\langle \text{function-name} \rangle (n+1) = \langle \text{exp 2} \rangle \\
    &\langle \text{function-name} \rangle x = \text{error(message)}
    \end{align*}
    \]

    where
    - \langle expression 1 \rangle does not contain \langle function-name \rangle
    - \langle expression 2 \rangle may contain \langle function-name \rangle applied to n

• **Motivation:** But some functions do not fit this scheme, requiring more complex patterns
Fibonacci Numbers – More Complex Patterns

• Example: The first Fibonacci numbers are 0,1. For each subsequent Fibonacci number, add the previous two together
  \[0, 1, 1, 2, 3, 5, 8, 13, 21, 34\]

• Problem: The following does not terminate on input 1
  \[
  \begin{align*}
  \text{fib} \ 0 &= 0 \\
  \text{fib} \ (n+1) &= \text{fib} \ n + \text{fib} \ (n-1)
  \end{align*}
  \]

• Solution: The new pattern \((n+2)\) matches inputs \(\geq 2\)
  \[
  \begin{align*}
  \text{fib} \ 0 &= 0 \\
  \text{fib} \ 1 &= 1 \\
  \text{fib} \ (n+2) &= \text{fib} \ (n+1) + \text{fib} \ n
  \end{align*}
  \]

• In General: There are patterns \((n+1), (n+2), (n+3)\)

More general recursion on lists

• Recall: Our primitive recursive functions follow the pattern
  – Base Case: Defines the function at [] non-recursively
  – Inductive Case: Defines the function at \((x:xs)\) in terms of the function at \(xs\)
    \[
    \begin{align*}
    \text{parentheses} \ [] &= \text{expression 1} \\
    \text{parentheses} \ (x:xs) &= \text{expression 2}
    \end{align*}
    \]
  where
  \[
  \begin{align*}
  \text{expression 1} & \quad \text{does not contain} \quad \text{parentheses} \ \text{applied to} \ \text{xs} \\
  \text{expression 2} & \quad \text{may contain} \quad \text{parentheses} \ \text{applied to} \ \text{xs}
  \end{align*}
  \]

• Motivation: As with integers, some functions don’t fit this shape

More General Patterns for Lists

• Recall: With integers, we used more general patterns.

• Idea: Use \((x:(y:xs))\) pattern to access first two elements

• Example: We want a function to delete every second element
  \[
  \text{delete} \ [2,3,5,7,9,5,7] = [2,5,9,7]
  \]

• Solution: Here is the code
  \[
  \begin{align*}
  \text{delete} \ &: \ [a] \rightarrow [a] \\
  \text{delete} \ [] &= [] \\
  \text{delete} \ [x] &= [x] \\
  \text{delete} \ (x:(y:xs)) &= x : \ \text{delete} \ xs
  \end{align*}
  \]

• Example: To delete every third element use pattern \((x:(y:(z:xs)))\)

Examples of Recursion and patterns — See how the typing helps

• Example 1: Summing pairs in a list of pairs
  \[
  \text{sumPairs} \ :: \ [(\text{Int,Int})] \rightarrow \text{Int}
  \]

• Example 2: Unzipping lists \[
  \text{unZip} \ :: \ [(\text{a,b})] \rightarrow ([\text{a}],[\text{b}])
  \]
**Sorting Algorithms 1: Insertsort**

- **Problem:** A sorting algorithm rearranges a list in order
  
  sort \([2, 7, 13, 5, 0, 4]\) = \([0, 2, 4, 5, 7, 13]\)
  
- **Recursion:** Such algorithms usually recursively sort a smaller list
  
- **Insertsort Alg:** To sort a list, sort the tail recursively, and then insert the head

- **Code:**
  
  ```plaintext
  inssort :: [Int] -> [Int]
inssort [] = []
inssort (x:xs) = insert x (inssort xs)
  
  where
  insert :: Int -> [Int] -> [Int]
  insert n [] = [n]
  insert n (x:xs)
    | n <= x = n:x:xs
    | otherwise = x:(insert n xs)
  ```

**The function insert**

- **Patterns:** Insert takes two arguments, number and list
  
  - The recursion for `insert` doesn’t depend on the number
  
  - The recursion for `insert` does depend on whether the list is empty or not — use the `[]` and `(x:xs)` patterns

- **Code:** Here is the final code
  
  ```plaintext
  insert :: Int -> [Int] -> [Int]
  insert n [] = [n]
  insert n (x:xs)
    | n <= x = n:x:xs
    | otherwise = x:(insert n xs)
  ```

**Sorting Algorithms 2: Quicksort**

- **Quicksort Alg:** Given a list \(l\) and a number \(n\) in the list

  sort \(l\) = sort those elements less than \(n\) ++
  
  number of occurrences of \(n\) ++
  
  sort those elements greater than \(n\)

- **Code:** The algorithm may be coded

  ```plaintext
  qsort :: [Int] -> [Int]
  qsort [] = []
  qsort (x:xs) = qsort (less x xs) ++
  
  occs x (x:xs) ++
  
  qsort (more x xs)
  
  where
  less, occs, more :: Int -> [Int] -> [Int]
  less n xs = [x | x <= xs, x < n]
  occs n xs = [x | x <= xs, x == n]
  more n xs = [x | x <= xs, x > n]
  ```

**Defining the Auxiliary Functions**

- **Problem:** The auxiliary functions can be specified

  - `less` takes a number and a list and returns those elements of the list less than the number
  
  - `occs` takes a number and a list and returns the occurrences of the number in the list
  
  - `more` takes a number and a list and returns those elements of the list more than the number

- **Code:** Using list comprehensions gives short code

  ```plaintext
  less, occs, more :: Int -> [Int] -> [Int]
  less n xs = [x | x <= xs, x < n]
  occs n xs = [x | x <= xs, x == n]
  more n xs = [x | x <= xs, x > n]
  ```
Sorting Algorithm 3: Mergesort

- **Mergesort Alg:** Split the list in half, recursively sort each half and merge the results.

- **Code:** Overall function reflects the algorithm

```haskell
msort [] = []
msort [x] = [x]
msort xs = merge (msort ys) (msort ws)
  where (ys,ws) = (take l xs, drop l xs)
  l = length xs \ div \ 2

where merge combines two sorted lists

merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) = if x<y then x : merge xs (y:ys)
  else y : merge (x:xs) ys
```

Today You Should Have Learned

- **Recursion Schemes:** We've generalised the recursion schemes to allow more functions to be written
  - More general patterns
  - Recursive calls to ANY smaller value

- **Examples:** Applied them to recursion over integers and lists

- **Sorting Algorithms:** We've put these ideas into practice by defining three sorting algorithms
  - Insertion Sort
  - QuickSort
  - MergeSort

Overview of Lecture 9

- **Motivation:** Why do we want higher order functions

- **Definition:** What is a higher order function

- **Examples:**
  - Mapping: Applying a function to every member of a list
  - Filtering: Selecting elements of a list satisfying a property

- **Application:** Higher order sorting algorithms
**Motivation**

- **Example 1:** A function to double the elements of a list
  ```haskell
doubleList :: [Int] -> [Int]
doubleList [] = []
doubleList (x:xs) = (2*x) : doubleList xs
  ```

- **Example 2:** A function to square the elements of a list
  ```haskell
  squareList :: [Int] -> [Int]
squareList [] = []
squareList (x:xs) = (x*x) : squareList xs
  ```

- **Example 3:** A function to increment the elements of a list
  ```haskell
  incList :: [Int] -> [Int]
  incList [] = []
  incList (x:xs) = (x+1) : incList xs
  ```

**The Common Pattern**

- **Problem:** Three separate definitions despite a clear pattern
  ```haskell
  function :: Int -> Int
  functionList :: [Int] -> [Int]
  functionList [] = []
  functionList (x:xs) = (function x) : functionList xs
  ```
  where in our previous examples function is
  ```haskell
  double square inc
  ```

- **Intuition:** Examples apply a function to each member of a list
  ```haskell
  doubleList xs = map double xs
  squareList xs = map square xs
  incList xs = map inc xs
  ```

- **Key Idea:** Make auxiliary function function an input

**A Higher Order Function — map**

- **The Idea Coded:**
  ```haskell
  map f [] = []
  map f (x:xs) = (fx) : map f xs
  ```

- **Advantages:** There are several advantages
  - Shortens code as previous examples are given by
    ```haskell
    doubleList xs = map double xs
    squareList xs = map square xs
    incList xs = map inc xs
    ```
  - Captures the algorithmic content and is easier to understand
  - Easier code-modification and code re-use

**A Definition of Higher Order Functions**

- **Question:** What is the type of `map`?
  - First argument is a function
  - Second argument is a list whose elements have the same type and the input of the function.
  - Result is a list whose elements are the output type of the function.

- **Answer:** So overall type is `map :: (a -> b) -> [a] -> [b]`

- **Definition:** A function is higher-order if an input is a function.

- **Another Example:** Type of `filter` is
  ```haskell
  filterInt :: (a -> Bool) -> [a] -> [a]
  ```
Quicksort Revisited

- **Idea:** Recall our implementation of `quicksort`

```haskell
def qsort :: Ord a => [a] -> [a]
def qsort [] = []
def qsort (x:xs) = qsort less ++ occs ++ qsort more
    where
        less = [e | e<xs, e<x]
        occs = x : [e | e<xs, e==x]
        more = [e | e<xs, e>x]
```

- **Polymorphism:** Quicksort requires an order on the elements:
  - The output list depends upon the order on the elements
  - This requirement is reflected in type class information `Ord a`
  - Don’t worry about type classes as they are beyond this course

Roy Crole
Leicester, October 6, 2005

97

Limitations of Quicksort

- **Example:** Games tables might have type `[(Team,Points)]`

- **Problem:** How can we order the table?

  
  Arsenal 16
  AVilla 16
  Derby 10
  Birm. 4
  ...

- **Solution:** Write a new function for this problem

```haskell
def tSort [] = []
def tSort (x:xs) = tSort less ++ [x] ++ tSort more
    where more = [e| e<xs, snd e > snd x]
        less = [e| e<xs, snd e < snd x]
```

- **What did we assume here?**

Roy Crole
Leicester, October 6, 2005

98

Higher Order Sorting

- **Motivation:** But what if we want other orders, eg
  - Sort teams in order of names, not points
  - Sort on points, but if two teams have the same points, compare names

- **Key Idea:** Make the comparison a parameter of quicksort

```haskell
def qsortCp :: (a -> a -> Bool) -> [a] -> [a]
def qsortCp ord [] = []
def qsortCp ord (x:xs) = qsortCp ord less ++ occs ++ qsortCp ord more
    where less = [e | e<xs, ord e x]
        occs = x : [e | e<xs, e==x]
        more = [e | e<xs, ord e x]
```

Examples

- **Key Idea:** To use a higher order sorting algorithm, use the required order to define the function to sort by

- **Example 1:** To sort by names

  ```haskell
  ord (t, p) (t', p') = t < t'
  ```

- **Example 2:** To sort by points and then names

  ```haskell
  ord (t, p) (t', p') = (p < p') || (p == p' && t < t')
  ```

- **What should we assume about `ord`?**

Roy Crole
Leicester, October 6, 2005

99
Today You Should Have Learned

- **Higher Order Functions**: Functions which takes functions as input
  - Facilitates code reuse and more abstract code
  - Many list functions are either `map`, `filter` or `fold`

- **HO Sorting**: An application of higher order functions to sorting
  - Produces more powerful sorting
  - Order of resulting list determined by a function
  - Lexicographic order allows us to try one order and then another

---

Overview of Lecture 10

- **Motivation**: Some examples leading to polymorphism

- **Definition**: What is parametric polymorphism?
  - What is a polymorphic type?
  - What is a polymorphic function?
  - Polymorphism and higher order functions
  - Applying polymorphic functions to polymorphic expressions

---

Monomorphic length

- **Example**: Let us define the length of a list of integers
  ```haskell
  mylength :: [Int] -> Int
  mylength [] = 0
  mylength (x:xs) = 1 + mylength xs
  ```

- **Problem**: We want to evaluate the length of a list of characters
  ```haskell
  Prelude> mylength ['a', 'g']
  ERROR: Type error in application
  *** expression : mylength ['a','g']
  *** term : ['a','g']
  *** type : [Char]
  *** does not match : [Int]
  ```

- **Solution**: Define a new length function for lists of characters
  ```haskell
  mylength :: [Char] -> Int
  mylength [] = 0
  mylength (x:xs) = 1 + mylength xs
  ```
  ... but this is not very efficient!
Polymorphic length

- **Better Solution:** The algorithm's input depends on the list type, but not on the type of integers.

- **Idea:** An alternative approach to typing mylength
  - There is one input and one output: `mylength :: a -> b`
  - The output is an integer: `mylength :: a -> Int`
  - The input is a list: `mylength :: [c] -> Int`
  - There is nothing more to infer from the code of `mylength` so
  
  `mylength :: [c] -> Int`

  This is an efficient function - works at all list types!

---

Haskell's Polymorphic Type System

- **Types:** Now we will deal with the following types:
  - Basic, built in types: Int, Char, Bool, String, Float
  - Type variables representing any type: `a`, `b`, `c`, ...
  - Types built with type constructors: `[]`, `->`, `(,)`
    
    `[Int] a->a a->b a->Bool (String,a->a) [a->Bool]`
  - Type synonyms: `type <type-name> = <type-expression>`
    
    `type Point = (Int,Int)`
    `type Line = (Point,Point)`
    `type Test = a->Bool`

---

Some Definitions

- **Polymorphism** is the ability to appear in different forms

- **Definition:** A type is *parametric polymorphic* iff it contains type variables (that is, type parameters).

- **Definition:** A function is *parametric polymorphic* iff it can be called on different types of input, and it is implemented by (code for) a single algorithm

- **Definition:** A function is *overloaded* iff it can be called on different types of input, and for each type of input, the function is implemented by (code for) a particular algorithm.

- **Examples:** Of overloading are the arithmetic operators: integer and floating-point addition.

---

Polymorphic Expressions

- **Key Idea:** Expressions have many types
  - Amongst these is a *principle* type

- **Example:** What is the type of `id x = x`
  - `id` sends an integer to an integer. So `id :: Int -> Int`
  - `id` sends a list of type `a` to a list of type `a`. So `id :: [a] -> [a]`
  - `id` sends an expression of type `b` to an expression of type `b`. So `id :: b -> b`

- **Principle Type:** The last type includes the previous two – why?
  - In fact the principal type of `id` is `id :: b -> b` – why?
Examples

- **Example 1:** What is the type of \( \text{map} \)
  \[
  \text{map} \, f \, [] = [] \\
  \text{map} \, f \, (x:xs) = f \, x : \text{map} \, f \, xs
  \]

- **Example 2:** What is the type of \( \text{filter} \)
  \[
  \text{filter} \, f \, [] = [] \\
  \text{filter} \, f \, (x:xs) = \text{if} \, f \, x \, \text{then} \, x : \text{filter} \, f \, xs \, \text{else} \, \text{filter} \, f \, xs
  \]

- **Example 3:** What is the type of \( \text{iterate} \)
  \[
  \text{iterate} \, f \, 0 \, x = x \\
  \text{iterate} \, f \, (n+1) \, x = f \, (\text{iterate} \, f \, n \, x)
  \]

Applying Polymorphic Expressions to Polymorphic Functions

- **Previously:** The typing of applications of expressions:
  - If \( \exp_1 \) is an expression with type \( a \rightarrow b \)
  - And \( \exp_2 \) is an expression with type \( a \)
  - Then \( \exp_1 \, \exp_2 \) has type \( b \)

- **Problem:** How does this apply to polymorphic functions?
  \[
  \text{length} :: [c] \rightarrow \text{Int} \\
  [2,4,5] :: [\text{Int}] \\
  \text{length} \, [2,4,5] :: \text{Int}
  \]

- **Key Idea:** Argument type can be an *instance* of input type

When is a Type an Instance of Another Type

- **Recall:** Two facts about expressions containing variables
  - Variables stand for arbitrary elements of a particular type
  - *Instances* of the expression are obtained by substituting expressions for variables

- **Key Idea:** (Parametric) polymorphic types are defined in the same way:
  - Type-expressions may contain type-variables
  - *Instances* of type-expressions are obtained by substituting types for type-variables

- **Example:** \([\text{Int}]\) is an instance of \([c]\) – substitute \text{Int} for \(c\)

More formally - Unification

- **Monomorphic:** Can a function be applied to an argument?
  - If the function’s input type is the same type as its argument
  \[
  f : a \rightarrow b \\
  x : a \\
  f \, x : b
  \]

- **Polymorphically:** Can a function be applied to an argument?
  - If the function’s input type is *unifiable* with argument’s type
  \[
  f : a \rightarrow b \\
  x : c \\
  \theta \text{ unifies } a, c \\
  f \, x : \theta \, b
  \]
  where \(\theta\) maps type variables to types

- **Example:** In the \text{length} example, set \(\theta c = \text{Int}\)
\textbf{Past Paper:} Assume \( f \) is a function with principle type

\[
f::([a],[b])\rightarrow\text{Int}\rightarrow[(b,a)]
\]

Do the following expressions type check? State \textbf{Yes} or \textbf{No} with a brief reason and, if \textbf{Yes}, what is the principal type of the expression?

1. \( f (3,3) \) 2
2. \( f ([],[]) \) 5
3. \( f ([\text{tail,head}],[]) \) 3
4. \( f ([\text{True,False}],[\text{'x'}]) \)

\textbf{Today You Should Have Learned}

\begin{itemize}
  \item \textbf{Polymorphism:}
    \begin{itemize}
      \item Saves on code — one function (algorithm) has many types
      \item This implements our algorithmic intuition
    \end{itemize}
  \item \textbf{Type Checking:} Expressions and functions have many types including a principle one
    \begin{itemize}
      \item Polymorphic functions are applied to expressions whose type is an instance of the type of the input of the function
    \end{itemize}
\end{itemize}