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
  \[ \text{ssquares } n = 0^2 + 1^2 + 2^2 + \cdots + (n-1)^2 + n^2 \]

• 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 += i*i;}
  }
  ```

• Execution:
  We may visualize running the program as follows
  ```plaintext
  (Stack) Memory
  s = ??
  i = ??
  ssquares 4
  (Stack) Memory
  s = 30
  i = 4
  ```

• Key Idea:
  Imperative programs transform the memory

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.

A Functional Program — Summing squares in Haskell

• Types:
  First we give the type of summing-squares
  `hssquares :: Int -> Int`

• Functions:
  Our program is a function
  ```haskell
  hssquares 0 = 0
  hssquares n = n*n + hssquares(n-1)
  ```

• Evaluation:
  Run the program by expanding definitions
  ```haskell
  hssquares 2 => 2*2 + hssquares 1
  => 4 + (1*1 + hssquares 0)
  => 4 + (1 + 0)
  => 5
  ```

• Comment:
  No mention of memory in the code.

Key Ideas in Functional Programming I — Types

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

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

• Strings: `String` is the Haskell type of lists of characters.

• Complex Datatypes: Can be made from the basic types, e.g.
  - Tuples: `(1, 2, 3)`
  - Lists: `[1, 2, 3]`

• Built-in Operations: Can be made from the basic types, e.g.

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:
  \[ x \in a \rightarrow \exists ! y \in b : f(x) = y \]

• Example:
  The square and cube functions are written
  ```haskell
  square :: Int -> Int
  cube :: Int -> Int
  square x = x * x
  cube x = x * square x
  ```

• In General:
  In Haskell, functions are defined as follows
  ```haskell
  \langle function-name \rangle :: \langle input type \rangle \rightarrow \langle output type \rangle
  \langle function-name \rangle \langle variable \rangle = \langle expression \rangle
  ```

What is Functional Programming — Two Aspects

• Key Idea: Imperative programs transform the memory
  - Functions are programs that change memory
  ```plaintext
<table>
<thead>
<tr>
<th>Memory</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>s=0</td>
<td>i=0</td>
</tr>
<tr>
<td>s=0</td>
<td>i=1</td>
</tr>
<tr>
<td>s=1</td>
<td>i=2</td>
</tr>
<tr>
<td>s=5</td>
<td>i=3</td>
</tr>
</tbody>
</table>
  ```

• Key Idea: Imperative Programs
  - Functional Content: What the program achieves
  - Implementational Content: How the program does it
  ```plaintext
  ssquares
<table>
<thead>
<tr>
<th>Function</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>summing squares</td>
<td>Int, Int</td>
</tr>
</tbody>
</table>
  ```

The Two Aspects of Imperative Programs

- Functional Content
- Implementational Content

- Programs take some input values and return an output value
- Programs transform the memory using variable declarations and assignment statements
Functions with Multiple Arguments

Intuition:

A function \( f \) with \( n \) inputs is written

\[
 f :: a_1 \rightarrow \ldots \rightarrow a_n \rightarrow a
\]

- \( x_1 \in a_1 \rightarrow \ldots \rightarrow x_n \in a_n \rightarrow \).

Example:
The "distance" between two integers

\[
 \text{diff} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]

\[
 \text{diff} \ x \ y = \text{abs} (x - y)
\]

In General:

\[
 \langle \text{function-name} \rangle :: \langle \text{type 1} \rangle \rightarrow \ldots \rightarrow \langle \text{type n} \rangle \rightarrow \langle \text{output-type} \rangle
\]

\[
 \langle \text{function-name} \rangle \ \langle \text{variable 1} \rangle \ldots \langle \text{variable n} \rangle = \langle \text{expression} \rangle
\]

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.

\[
\text{square} \ 4 \ \text{square} \ (3 + 1) \ \text{square} \ 3 + 1
\]

\[
\text{cube} \ (\text{square} \ 2) \ \text{diff} \ 6
\]

\[
\text{square} \ 2.2
\]

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

\[
5 \times 4 \ (*) \ 5 \ 4 \ \
\text{mod} \ 13 \ 4 \ 13 \ 'mod' \ 4
\]

\[
5 - (3 \times 4) \ (5 - 3) \times 4 \ 7 \ \geq (3 \times 3) \ 5 \ \times (-1)
\]

Precedence:

Usual rules of precedence and bracketing apply

Example of Evaluation:

\[
\text{cube} \ (\text{square} \ 3) \Rightarrow \ (\text{square} \ 3) \ \ast \ \text{square} \ (\text{square} \ 3)
\]

\[
\Rightarrow \ (3 \times 3) \ \ast \ ((\text{square} \ 3) \ \ast \ (\text{square} \ 3))
\]

\[
\Rightarrow \ (9 \ \ast \ (9 \ \times 9))
\]

\[
\Rightarrow \ 729
\]

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!

- Functions, Transformers, and Guards

  - Expression evaluation can be controlled by guard

  - The simplest (evaluated) expressions are (data) values

  - The simplest (guarded) expressions are (data) values

  - We build complex expressions by defining functions and app-
Local Declarations — The type of Fractions

- Float
- Char

Conditionals — If statements

Overloading:

- Conditionals:

Example:

Motivation:

Built in Functions:

Conversions:

Elements of:

Examples:

Functions:

Forming elements of:

Extended Example OP

Error-Handling

Guards: There are 3 cases to check so use a guarded definition

Example: if b then e1 else e2

Solution: One problem will have value

Conditional Equation: The solutions of ax + bx + c = 0 are

Extended Example OP

Execution:

Example: duplicate gives two copies of a string

Combining Strings: Strings can be combined by

Special Strings: Numbers and the characters

Forming elements of String: Compose indices from strings

Elements of String: Lists of characters

The String Type

Examples: Strings are written with double quotes

where (expr1::string) (expr2::string)

... :: (expr1::string)

Defining a guarded function is the form

where (guard0 ... guardn)

Example: duplicate returns double the maximum of its inputs

Answer: Local definitions allow auxiliary functions

Problem: Split definition without the local environment

Example: Summing the squares of two numbers

Local Definitions — An alternative to

Several Pairs: Overloading is means values/functions belong to

Guards: Overloading is means values/functions belong to

Built in Functions: Arithmetic, Ordering, Trigonometric

Elements of Types: Composing daughter types

The type of Fraction

Example: mydiv x y = x/y gives two copies of a string

Prelude> mydiv 5 0

Program error: Attempt to divide by 0

...=

χ

where (expr0::expr)...

...::...::...::...::...

(1 p expr1) = (1 p expr2)

 Disequal expression of the form

1 x ≤ x 2 x ≤ x 3 x ≤ x

Examples: Expressions with the same type

where t is the type of

Conditionals: A conditional expression has the form

Example: Testing if an integer is 0

maxi x y = if x≥y then x else y

maxi :: Int -> Int -> Int

sumSquares x y = sq x + sq y

sumSquares :: Int -> Int -> Int

sq x = x * x

sq :: Int -> Int

Example: Forming elements of numbers

Examples: Arithmetic, Ordering, Trigonometric
The function roots—Stage II OP

• Code:

Now we can add in the answers roots a b c
| a == 0 = error "Not a quadratic eqn"
| b*b - 4*a*c == 0 = "One root: " ++ show (-b/2*a)
| otherwise = "Two roots: " ++ show ((-b + sqrt (b*b - 4*a*c))/2*a) ++ "and" ++ show ((-b - sqrt (b*b - 4*a*c))/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 OP

• Local decs:

Expressions used repeatedly are made local

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

New Types from Old — 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, i.e.
– If e1 has type a and e2 has type b
– Then (e1, e2) has type (a, b)

If we add a and add b

Examples:

Functions using Pairs

December 6, 2005

Lecture 3 — New Types from Old

• Error Handling and Local Declarations

• Conditional expressions and Guarded functions

Expressions: how to write expressions involving

– That are values of the type
to know
– Type synonyms
– Types
– Pairs

Composite types

• Today we will learn about the following
building new types
– Error handling and local declarations
– Conditional expressions and guarded functions
– From simple data values to complex data values

Motivation:
Data for programs modelled by values of a type
– 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
– A number for the x-coordinate and another for the y-coordinate
– A point on a plane can be specified by

• 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

• Example: 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, i.e.
– If e1 has type a and e2 has type b
– Then (e1, e2) has type (a, b)

If we add a and add b

Examples:

Functions using Pairs

December 6, 2005

Lecture 3 — New Types from Old

• Error Handling and Local Declarations

• Conditional expressions and Guarded functions

Expressions: how to write expressions involving

– That are values of the type
to know
– Type synonyms
– Types
– Pairs

Composite types

• Today we will learn about the following
building new types
– Error handling and local declarations
– Conditional expressions and guarded functions
– From simple data values to complex data values

Motivation:
Data for programs modelled by values of a type
– 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
– A number for the x-coordinate and another for the y-coordinate
– A point on a plane can be specified by

• 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

• Example: 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, i.e.
– If e1 has type a and e2 has type b
– Then (e1, e2) has type (a, b)

If we add a and add b

Examples:

Functions using Pairs
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 \((\text{String}, \text{Int}, \text{Int})\)

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
\[ \text{isAdult} :: (\text{String}, \text{Int}, \text{Int}) \to \text{Bool} \]
\[
isAdult \ (\text{name}, \text{tel}, \text{age}) = (\text{age} \geq 18)
\]

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

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

New Types from Old III — Type Synonyms

Motivation:
More descriptive names for particular types.

Definition:
Type synonyms are declared with the keyword \textit{type}.

\[ \textit{type} \ 	ext{Team} = \text{String} \]
\[ \textit{type} \ 	ext{Goals} = \text{Int} \]
\[ \textit{type} \ 	ext{Match} = ((\text{Team}, \text{Goals}), (\text{Team}, \text{Goals})) \]

\[ \text{numu} :: \text{Match} \]
\[ \text{numu} = ("\text{Newcastle}", 4), ("\text{Manchester Utd}", 3) \]

Functions:
Types of functions are more descriptive, same code

\[ \text{homeTeam} :: \text{Match} \to \text{Team} \]
\[ \text{totalGoals} :: \text{Match} \to \text{Goals} \]

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
\[ \text{addPair} :: (\text{Int}, \text{Int}) \to \text{Int} \]
\[ \text{addTwo} :: \text{Int} \to \text{Int} \to \text{Int} \]

Lecture 4 — List Types

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: \text{cons}, \text{append}, \text{head}, \text{tail}
      - Some new functions: \text{map}, \text{filter}
    - Clarity:
      - Unlike Java, Haskell treatment of lists is clear
        - No list iterators!

General Definition of a Function: Patterns with Tuples

Definition:
Functions now have the form \(<\text{function-name}> :: <\text{type 1}> \to \ldots \to <\text{type n}> \to <\text{out-type}>\)
\[
<\text{function-name}> \ <\text{pat 1}> \ldots <\text{pat n}> = <\text{exp n}>
\]

Patterns:
- Variables \(x\): Use for any type
- Constants \(0, \text{True}, "\text{cherry}"\): 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
    \[ \text{isZero} :: \text{Int} \to \text{Bool} \]
    \[ \text{isZero} \ 0 = \text{True} \]
    \[ \text{isZero} \ = \text{False} \]
- Example:
  - Using tuples and multiple arguments
    \[ \text{expand} :: \text{Int} \to (\text{Int},\text{Int}) \to (\text{Int},\text{Int},\text{Int}) \]
    \[ \text{expand} \ n \ (x,y) = (n, n*x, n*y) \]
- Example:
  - Days in the month
    \[ \text{days} :: \text{String} \to \text{Int} \to \text{Int} \]
    \[ \text{days} "\text{January}" \ = \text{31} \]
    \[ \text{days} "\text{February}" \ = \text{if} \ \text{isLeap} \ x \ \text{then} \text{29} \ \text{else} \text{28} \]
    \[ \ldots. \]

Today You Should Have Learned
- Lists: What are lists?
- 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
\[ \text{addPair} :: (\text{Int},\text{Int}) \to \text{Int} \]
\[ \text{addTwo} :: \text{Int} \to \text{Int} \to \text{Int} \]
List Types and Expressions

• Example 1: 
\[3, 5, 14\] :: [Int] and 
\[3, 4+1, \text{double 7}\] :: [Int]

• Example 3: 
\[\text{'d', 't', 'g'}\] :: [Char]

• Example 4: 
\[\text{[\text{'d'}, \text{'d'}, \text{'t'}]}, \text{[\text{'d'}, \text{'t'}, \text{'g'}]}\] :: [[Char]]

• Example 5: 
\[
\text{\text{double}, \text{square}, \text{cube}}\] :: [Int -> Int]

Empty List: 
The empty list is 
\[
\text{[]}
\]
and belongs to all list types

List Expressions: 
Lists are written using square brackets 
\[...\]

– If 
\[e_1, \ldots, e_n\] are expressions of type 
\[a\];

– Then 
\[e_1, \ldots, e_n\] is an expression of type 
\[a\].

Cons: 
The cons function: 
\[\text{cons} : a \rightarrow [a] \rightarrow [a]
1 : [2,3,4] = [1,2,3,4]

addone : [square] = [addone, square]

• Append: 
Append joins two lists together
\[++ : [a] \rightarrow [a] \rightarrow [a]\]
\[\text{[True, True]} ++ \text{[False]} = \text{[True, True, False]}\]
\[\text{[1,2]} ++ (\text{[3]} ++ \text{[4,5]} ) = \text{[1,2,3,4,5]}\]
\[(\text{[1,2]} ++ \text{[3]} ) ++ \text{[4,5]} = \text{[1,2,3,4,5]}\]
\[\text{[]} ++ \text{[54.6, 67.5]} = \text{[54.6, 67.5]}\]
\[\text{[6,5]} ++ (4 : \text{[7,3]} ) = \text{[6,5,4,7,3]}\]

Some built in functions - Two infix operators

• Cons: The cons function: 
\[\text{cons} : a \rightarrow [a] \rightarrow [a]
1 : [2,3,4] = [1,2,3,4]

addone : [square] = [addone, square]

• Append: 
Append joins two lists together
\[++ : [a] \rightarrow [a] \rightarrow [a]\]

Solution 2 — Using map
\[
\text{map} \quad \text{isEven} \quad \text{[1 .. n]} = \text{[True, True, False]}
\text{even\_numbers} = \text{map} \quad \text{isEven} \quad \text{[1 .. n]} = \text{[1,3,5,7,9]}
\]

• Head and Tail: 
Head gives the first element of a list, tail the remainder
\[\text{head} : \text{[1 .. n]} = \text{[1,3,5,7,9]}\]
\[\text{head} : \text{[1,2,3]} = \text{[1,2,3]}\]
\[\text{tail} : \text{[1 .. n]} = \text{[3,5,7,9]}\]
\[\text{tail} : \text{[1,2,3]} = \text{[2,3]}\]

• Length and Sum: 
The length of a list and the sum of a list of integers
\[\text{length} : \text{[1 .. n]} = \text{[1,3,5,7,9]}\]
\[\text{sum} : \text{[1 .. n]} = \text{[1,3,5,7,9]}\]

• Sequences: 
The list of integers from 1 to n is written 
\[\text{[1 .. n]}\]

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: Developed an algorithm by asking
– Can I solve this problem by applying a function to every member of a list?
– Where does the function go?
– Create a list that has two inputs.

Two New Functions — Map and Filter

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

• Solution 1
\[\text{evens2} : \text{Int} \rightarrow [\text{Int}]\]
\[\text{evens2} n = \text{filter} \quad \text{isEven} \quad \text{[1 .. n]}\]

• Solution 2
\[\text{evens2} n = \text{map} \quad \text{isEven} \quad \text{[1 .. n]}\]

Lecture 5 — List Comprehensions

Overview of Lecture 5
• Recall Map:
Map is a function which has two inputs.
\[\text{map} \quad \text{add2} \quad \text{[2, 5, 6]} = \text{[4, 7, 8]}\]

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

• List comprehension:
An alternative way of writing lists
– Definition of list comprehension

– Example 1:
\[\{i + 1 | i \in [1..n] \} = [2, 3, 4, ..., n+1]\]

– Example 2:
\[\{x^2 \mid x \in [1, 2, 3] \} = [1, 4, 9]\]

– Example 3:
\[\{x \cdot y \mid x \in [1, 2, 3], y \in [4, 5, 6] \} = [4, 5, 6, 8, 10, 12, 12, 15, 18]\]

– Example 4:
\[\{x \mid x \neq 5 \} = [1, 2, 3, 4, 6, 7, 8, 9, 10]\]

– Example 5:
\[\{x \mid x \neq 5, x \neq 7 \} = [1, 2, 3, 4, 6, 8, 9, 10]\]
List Comprehension — An alternative to \texttt{map} and \texttt{filter}

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

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

- **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 \( \langle \text{list-exp} \rangle \)
    - Evaluate the expression \( \langle \text{exp} \rangle \)
    - Return the results in a list.

Using List Comprehensions Instead of \texttt{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:
    \[
    \text{map} \ f \ l = [f \ x \mid x \leftarrow l]
    \]

Using List Comprehensions Instead of \texttt{filter}

- **Intuition:**
  - List Comprehension can also select elements from a list

- **Example:**
  - We can select the even numbers in a list
    \[ e \mid e \leftarrow l, \text{isEven} e \]

- **Example:**
  - Selecting names beginning with \( A \)
    \[ \text{names} :: [\text{String}] \rightarrow [\text{String}] \]
    \[ \text{names} \ l \rightarrow \{ e \mid e \leftarrow l, \text{head} e == 'A' \} \]

- **Example:**
  - Combining selection and applying functions
    \[ \text{doubleEven} :: [\text{Int}] \rightarrow [\text{Int}] \]
    \[ \text{doubleEven} \ l \rightarrow [2*e \mid e \leftarrow l, \text{isEven} e] \]

General Form of List Comprehension

- **In General:**
  - These list comprehensions are of the form
    \[
    [\langle \text{exp} \rangle \mid \langle \text{variable} \rangle \leftarrow \langle \text{list-exp} \rangle, \langle \text{test} \rangle]
    \]
  - e.g.,
    \[ [2*e \mid e \leftarrow l, \text{isEven} e, e>3] = [16,20] \]

- **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.
    \[ [(x,y) \mid x<-[1,2,3], y<-['a','b','c']] \]
    \[ = [(1,'a'), (1,'b') ... ] \]

- **Example:**
  - \( \text{league} :: [\text{Team}] \)
  - \( \text{games} = [(t1,t2) \mid t1 \leftarrow \text{league}, t2 \leftarrow \text{league}, t1 \neq t2] \)

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 \texttt{filter} and \texttt{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

\[ \text{hssquares } n = \sum_{i=1}^{n} i^2 \]

Types:
First we give the type of summing-squares

\[ \text{hssquares} :: \text{Int} \rightarrow \text{Int} \]

Definitions:
Our program is a function

\[ \text{hssquares } 0 = 0 \]
\[ \text{hssquares } n = n^2 + \text{hssquares } (n-1) \]

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 –

Examples of evaluation

Example 1:
Let's calculate \( \text{hssquares } 4 \)

\[ \begin{align*}
\text{hssquares } 4 & \Rightarrow 4^2 + \text{hssquares } 3 \\
& \Rightarrow 16 + (3^2 + \text{hssquares } 2) \\
& \Rightarrow 16 + (9 + \ldots (1 + \text{hssquares } 0)) \\
& \Rightarrow 16 + (9 + \ldots (1 + 0)) \\
& \Rightarrow 30
\end{align*} \]

Example 2:
Here is a non-terminating function

\[ \text{mydouble } n = n + \text{mydouble } (n/2) \]

\[ \begin{align*}
\text{mydouble } 4 & \Rightarrow 4 + \text{mydouble } 2 \\
& \Rightarrow 4 + 2 + \text{mydouble } 1 \\
& \Rightarrow 4 + 2 + 1 + \text{mydouble } 0.5 \\
& \Rightarrow \ldots
\end{align*} \]

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?

Examples of recursive functions

Example 1:
\text{star} uses recursion over \text{Int} to return a string

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

Example 2:
\text{power} is recursive in its second argument

\[ \text{power} :: \text{Float} \rightarrow \text{Int} \rightarrow \text{Float} \]
\[ \text{power } x 0 = 1 \]
\[ \text{power } x (n+1) = x \times \text{power } x n \]
\[ \text{power } x n = \text{error } \text{''Negative input''} \]

Examples of primitive recursion

Motivation:
Use the following style of definition

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

where

\( \langle \text{exp 1} \rangle \) does not contain \( \langle \text{function-name} \rangle \)
\( \langle \text{exp 2} \rangle \) may contain \( \langle \text{function-name} \rangle \) applied to \( n \)

Evaluation:
Termination guaranteed!

1. If the input evaluates to \( 0 \), evaluate \( \langle \text{exp 1} \rangle \)
2. If not, if the input is greater than \( 0 \), evaluate \( \langle \text{exp 2} \rangle \)
3. If neither, return the error message

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

\[ \text{printTable } :: \text{Int} \rightarrow \text{IO() \]
\[ \text{printTable } \text{numberTeams} = \text{putStr } \langle \text{header} \rangle \langle \text{rows } \text{numberTeams} \rangle \]

where

\[ \langle \text{header} \rangle = \text{\"Team } \text{Points}\" \]

Stage 2:
Convert each "row" to a string, recursively.

\[ \text{rows } :: \text{Int} \rightarrow \text{String} \]
\[ \text{rows } 0 = \ldots \]
\[ \text{rows } (n+1) = \ldots \]

Examples of recursive functions
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

Roy Crole Leicester, October 6, 2005

The Final Version

perf :: Int \rightarrow (String, Int)

perf 1 = ("Arsenal", 4)

perf 2 = ("Notts", 5)

perf 3 = ("Chelsea", 7)

perf \_ = error "perf out of range"

rows :: Int \rightarrow 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 \rightarrow 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.

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

Examples of Recursive Functions

Example 1: Doubling every element of an integer list

\[ \text{double} :: [\text{Int}] \rightarrow [\text{Int}] \]

\[ \text{double } [] = [] \]

\[ \text{double } (x:xs) = (2\times x) : \text{double } xs \]

Example 2: Selecting the even members of a list

\[ \text{onlyEvens} :: [\text{Int}] \rightarrow [\text{Int}] \]

\[ \text{onlyEvens } [] = [] \]

\[ \text{onlyEvens } (x:xs) = \text{if } \text{isEven } x \text{ then } x: \text{rest} \text{ else rest} \]

where

rest = onlyEvens xs

Example 3: Flattening some lists

\[ \text{flatten} :: [[a]] \rightarrow [a] \]

\[ \text{flatten } [] = [] \]

\[ \text{flatten } (x:xs) = x \text{ ++ } \text{flatten } xs \]

Exercise of Recursive Functions

Every time the function is called, perhaps some new function is being created.

This leads to growth in the call stack.

Exercise 1 Review

Exercise 2 Review

List comprehensions revisited

Exercise 3 Review

Exercise 4 Review

Recursion over Lists

Recursion over Lists

Exercise 5 Review

Exercise 6 Review

Exercise 7 Review

Exercise 8 Review
The General Pattern

• Definition: Primitive Recursive List Functions are given by
  \[
  \langle \text{function-name} \rangle [] = \langle \text{expression 1} \rangle \\
  \langle \text{function-name} \rangle (x:xs) = \langle \text{expression 2} \rangle
  \]
  where
  \[
  \langle \text{expression 1} \rangle \text{ does not contain } \langle \text{function-name} \rangle \\
  \langle \text{expression 2} \rangle \text{ may contain expressions } \langle \text{function-name} \rangle
  \]

• Compare: Very similar to recursion over \(\mathbb{N}\)
  \[
  \langle \text{function-name} \rangle 0 = \langle \text{expression 1} \rangle \\
  \langle \text{function-name} \rangle (n+1) = \langle \text{expression 2} \rangle
  \]
  where
  \[
  \langle \text{expression 1} \rangle \text{ does not contain } \langle \text{function-name} \rangle \\
  \langle \text{expression 2} \rangle \text{ may contain expressions } \langle \text{function-name} \rangle \]

Example 4: Append is defined recursively
  \[
  \text{append} :: [a] \to [a] \to [a]
  \]

Example 5: Testing if an integer is an element of a list
  \[
  \text{member} :: \mathbb{N} \to [\mathbb{N}] \to \mathbb{B}
  \]

Example 6: Reversing a list
  \[
  \text{reverse} :: [a] \to [a]
  \]

What can we do with a list?

• Mapping: Applying a function to every member of the list
  \[
  \text{double } [2,3,72,1] = [2 \times 2, 2 \times 3, 2 \times 72, 2 \times 1]
  \]
  \[
  \text{isEven } [2,3,72,1] = [\text{True}, \text{False}, \text{True}, \text{False}]
  \]

• Filtering: Selecting particular elements
  \[
  \text{onlyEvens } [2,3,72,1] = [2,72]
  \]

• Taking Lists Apart:
  \[
  \text{head, tail, take, drop}
  \]

• Combining Lists:
  \[
  \text{zip}
  \]

• Folding:
  \[
  \text{Combining the elements of the list}
  \]
  \[
  \text{sumList } [2,3,7,2,1] = 2 + 3 + 7 + 2 + 1
  \]

What can we do with a list?

More general forms of primitive recursion

• 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
  – Check they satisfy the test
  – Form a list by applying the expression 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. Although the recursion is hidden, so are list comprehensions although the recursion is.

Lecture 8 — More Complex Recursion

• 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?
  • Examples: Applications to integers and lists
  • Generalising the recursion scheme
  • Using new patterns

Problem: Our restrictions on recursive functions are too severe

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?
  • Examples: Applications to integers and lists
  • Generalising the recursion scheme
  • Using new patterns

List comprehension: An alternative way of doing some recursion

• 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
  – Check they satisfy the test
  – Form a list by applying the expression 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. Although the recursion is hidden, so are list comprehensions although the recursion is.

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. Although the recursion is hidden, so are list comprehensions although the recursion is.
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

fib 0 = 0
fib (n+1) = fib n + fib (n-1)

Solution:
The new pattern (n+2) matches inputs ≥ 2

fib 0 = 0
fib 1 = 1
fib (n+2) = fib (n+1) + fib n

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 [ ] recursively

– Inductive Case: Defines the function at (x:xs) in terms of

• The function at [ ]
• The function at xs

How General Patterns Fit Our Needs

Examples of Recursion and patterns – See how the typing helps

Example 1:
Summing pairs in a list of pairs

sumPairs :: [(Int,Int)] -> Int

Example 2:
Unzipping lists

unZip :: [(a,b)] -> ([a], [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:
inssort :: [Int] -> [Int]
inssort [] = []
inssort (x:xs) = insert x (inssort xs)

where
insert puts the number x in the correct place

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

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

qsort :: [Int] -> [Int]
qsort [] = []
qsort (x:xs) = qsort (less x xs) ++
  occs x (x:xs) ++
  qsort (more x xs)

where
less, occs, more are auxiliary functions

Defining the Auxiliary Functions

Problem:
The auxiliary functions can be specified
– less takes a number and a list and returns those elements
– greater a number and a list and returns the occurrences
– check for less than the number
– check for greater than the number

Patterns:
Less takes two arguments, number and list

Code:
Using list comprehensions gives short code

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]
Another Example: Floyd of Floyd's
Definition: A function is higher-order if an input is a function.
Answer: \( \left( f \circ g \right) \)
Quicksort Revisited

Idea:
Recall our implementation of quicksort

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

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

```
tSort [] = []
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

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

```
qsortCp :: (a -> a -> Bool) -> [a] -> [a]
qsortCp ord [] = []
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 x e]
```

Examples

- To use a higher order sorting algorithm, use the required order to define the function to sort by
- Example 1:
  To sort by names
  `ord (t, p) (t', p') = t < t'`
- Example 2:
  To sort by points and then names
  `ord (t, p) (t', p') = (p < p') || (p == p' && t < t')`
- What should we assume about `ord`?

Roy Crole Leicester, October 6, 2005

To today you should have learned

- Higher Order Functions: Functions which take 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
  - No sorting on application of higher order functions to sorting
  - Have list functions already map, list, sort
  - Rule: code reuse and more abstract code

Roy Crole Leicester, October 6, 2005

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
- What is parametric polymorphism?

Roy Crole Leicester, October 6, 2005

Monomorphic `length`

Example:
Let us define the length of a list of integers

```
mylength :: [Int] -> Int
mylength [] = 0
mylength (x:xs) = 1 + mylength xs
```

Problem:
We want to evaluate the length of a list of characters

```
Prelude> mylength ['a', 'g']
ERROR: Type error in application
```

Solution:
Define a new length function for lists of characters

```
... but this is not very efficient!
```

Roy Crole Leicester, October 6, 2005

Lecture 10 — (Parametric) Polymorphism

Roy Crole Leicester, October 6, 2005

What should we assume about each

```
(\(a > b\) \&\& \(d < e\)) \mathbin{||} (\(d > b\) \&\& \(c < e\)) = (\(d > b\) \&\& \(c < e\)) \; (\(a > b\) \&\& \(d < e\))
```

Example:
To sort by points and then names

```
> sortByPointsAndNames :: [(Team,Points)] -> [(Team,Points)]
> sortByPointsAndNames = qsortCp (\(t, p) (t', p') -> (p < p') || (p == p' && t < t'))
```

Example: To sort by points and then names

```
> sortByPointsAndNames [["Arsenal",16], ["AVilla",16], ["Derby",10], ["Birm.",4]]
[("Arsenal",16), ("AVilla",16), ("Birm.",4), ("Derby",10)]
```

Roy Crole Leicester, October 6, 2005
Polymorphism

A function is overloaded if it can be called on different types of input, and it is implemented by a particular algorithm.

Polymorphic types are defined in the same way:
- Type synonyms:
  - Type synonyms define new type names for existing types.
    - Example: `type Point = (Int,Int)`
  - Example: `type Line = (Point,Point)`

Type synonyms are defined to the right of type synonyms:
- `type Point = (Int,Int)`
- `type Line = (Point,Point)`

Type synonyms for type-variables:
- A function is an instance of a particular type synonym if it can be called on different types of input, and it is implemented by a particular algorithm.

Instances of type synonyms:
- `mylength` :: `[Int]` -> Int
- `mylength` :: `[a]` -> Int
- `mylength` :: `[c]` -> Int

Polymorphism in Haskell's type system:
- Type synonyms have many types
- Universal types:
  - Polymorphic function types:
    - Example: `f :: [a] -> [a]`
    - Example: `f :: (Integral a) => a -> a`

Polymorphic expressions:
- `f x :: b` in `f :: a -> b` - why?
- `f x :: b` - how do we know if `f` is polymorphic?
- Type synonyms for type-variables:
  - Type synonyms define new type names for existing types.
  - Type synonyms are defined to the right of type synonyms:
    - `type Point = (Int,Int)`
    - `type Line = (Point,Point)`

Polymorphic expressions:
- `f x :: a` in `f :: (Integral a) => a -> a`
- Example: `f x :: b` in `f :: a -> b` - why?
- Polymorphic expressions:
  - Polymorphic function types:
    - Example: `f :: [a] -> [a]`
    - Example: `f :: (Integral a) => a -> a`

Polymorphic expressions:
- `f x :: a` in `f :: (Integral a) => a -> a`
- Example: `f x :: b` in `f :: a -> b` - why?
- Polymorphic expressions:
  - Polymorphic function types:
    - Example: `f :: [a] -> [a]`
    - Example: `f :: (Integral a) => a -> a`
• Assume $f$ is a function with principle type $f::\left[a\right],\left[b\right]{}\rightarrow{}\text{Int}\rightarrow\left[b,a\right]$.

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

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