A Level Computer Science

Introduction to Functional Programming

William Marsh
School of Electronic Engineering and Computer Science
Queen Mary University of London

Aims and Claims

- Flavour of Functional Programming
- …. how it differs from Imperative Programming (e.g. Python)

- Claim that:
  - It is possible to program using functions
  - It is useful!

Only simple examples

I hope this is convincing

- Better understanding of programming

How This Session Works

1. Talk
2. Do
3. Reflect
4. Repeat
5. …
6. Stop when times up

Outline

FP Topics

- A first functions
- Composing function
- Lists
- *If time (probably not)*
  - Recursion
  - Map, Filter and Fold

Challenge problems

Reflections

- Expressions, statements and variables
- Sequence versus composition
- How functions work
- The best language

Functional Languages?

- Many programming languages now have functional features

Haskell

First Function

A Simple Function

- This function gives the larger of two numbers

Function name

Argument

```
bigger a b = if a > b then a else b
```

Is defined as …

Layout

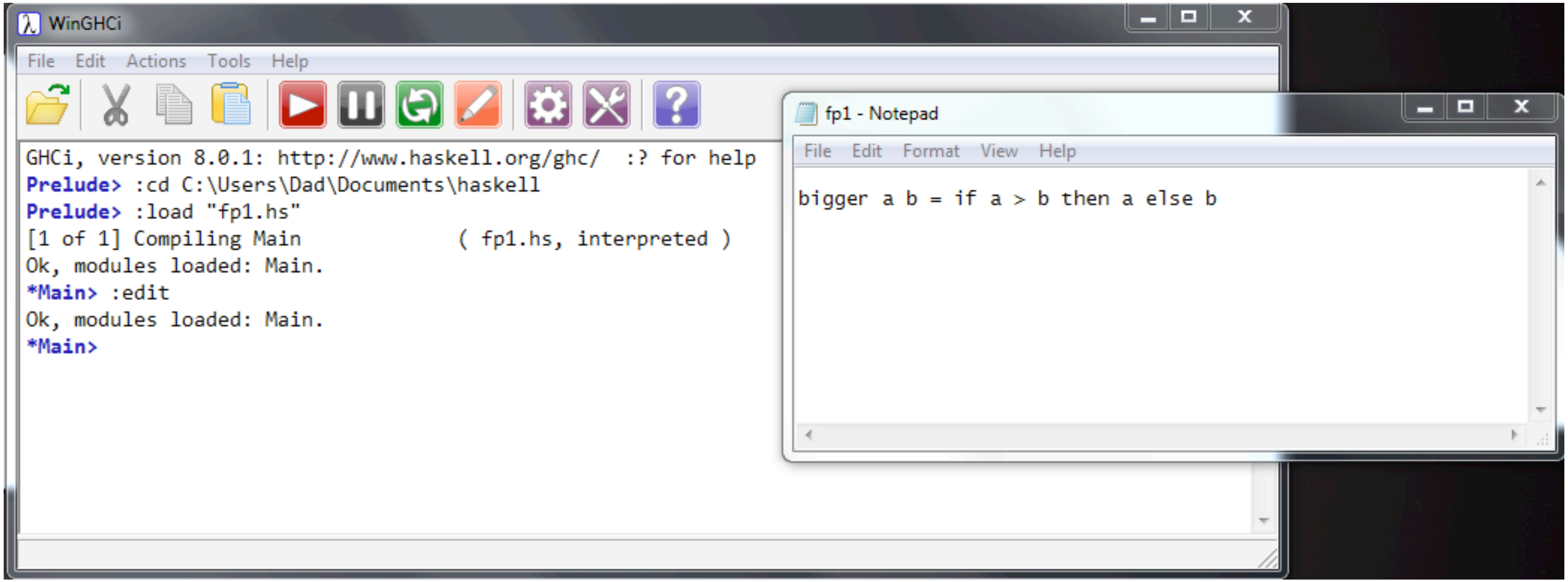
- Like Python, Haskell is layout sensitive
- The following all work

```
bigger a b =
  if a > b then a else b
```

```
bigger a b =
  if a > b
     then a
     else b
```

Getting Started with WinGHCi

- WinGHCi is a shell
  - Use functions interactively
- Use a text editor to edit the program
  - Notepad++ is better than notepad if you have it



Practical break

Refection 1: Expressions, Statements and Variables

Expressions and Statement

- Expression → value
- Statement → command

- Python: statements and expressions
- Haskell: only expressions

The Assignment Statement

- The most important statement:

```
x = x + 1  # This is python
```

- *Update the memory location ‘x’ with its current value plus 1*
- ‘x’ is a variable

Python program is a sequence of assignments

- Function may assign, so …
- Expressions are not just values

Haskell has no statements

- No assignment
- No variables

Is it possible to program without variables?

No Variables?

- My Haskell program seems to have variables

```
bigger a b =
  if a > b then a else b
```

- ‘a’ and ‘b’ a names for values
- Not memory locations

Functions

Maths (and Haskell)

- Result of a function depends only on its arguments
- Calling a function does not change anything
- Calling a function with the same arguments always gives the same result

Python

- Result of a function *may* depend on other variables
- Calling a function *may* change variables
- Calling a function a second time with the same arguments *may* give a different result

Function Composition

Composing Functions

- One way to write bigger3

```
bigger3 a b c = bigger (bigger a b) c
```

Pass results to …

Composing Functions

- Given a functions

```
double a = 2 * a
square a = a * a
```

- Predict the results of

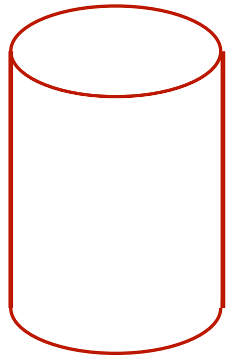
```
> double (double 5)
> double (square 3)
> square (double 3)
```

Composing Functions – Example

- Surface area of a cylinder

```
circleArea r = pi * r * r
circleCircum r = 2 * pi * r
rectArea l h = l * h

cylinderArea r h =
   2 * circleArea r +
     rectArea (circleCircum r) h
```



Practical break

Refection 2: Sequence versus Composition

Python's Invisible Statement

- Sequence of assignments

… then

… then

```
x = x + 1   # This is python
y = x * 2
x = 12
```

- Next statements on a new line
- Many languages: S1 ; S2

Haskell's Invisible Operator

- Function application

```
circleArea r = pi * r * r
circleCircum r = 2 * pi * r
rectArea l h = l * h

cylinderArea r h =
   2 * circleArea r +
     rectArea (circleCircum r) h
```

Decomposition

Python

- Sequence of statements
- … with names (functions)
- Order of memory updates

Haskell

- Expressions
- … with names (functions)
- Argument and results

Functional composition ≠ sequencing of statements

Python's Other Invisible Operator

- Function call (application)

```
def circleArea(r): return math.pi * r * r
def circleCircum(r): return 2 * math.pi * r
def rectArea(l, h): return l * h

def cylinderArea(r, h):
    return 2 * circleArea(r) + \
    rectArea(circleCircum(r), h)
```

call

call

call

call

call

Recursion

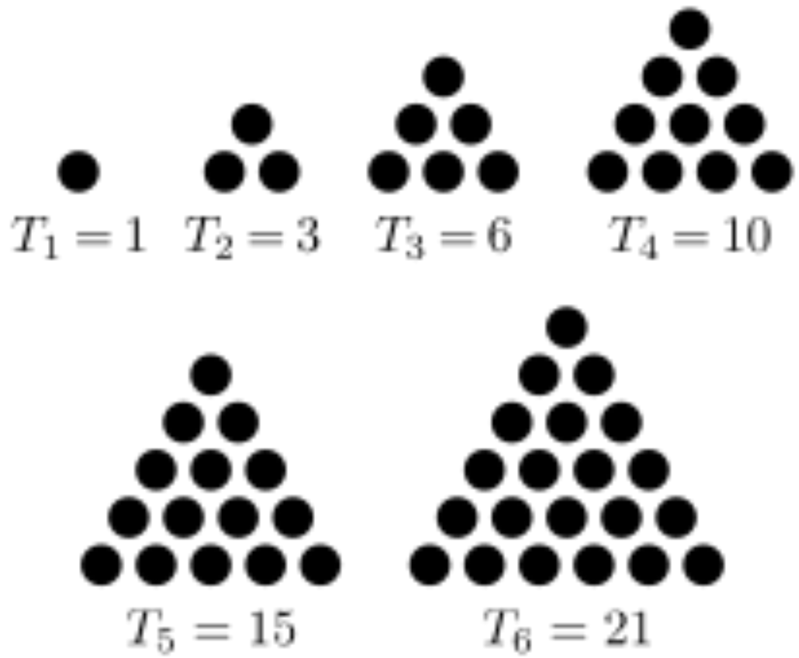
Recursion

- Can the definition of a function use the function being defined.
  - This is known as recursion

- It can if
  - There is a non-recursive base case
  - Each recursive call is nearer the base case

Recursion – Example

- A triangle number counts the number of dots in an equilateral triangle (see picture)
- We can define by:



Base case

```
trigNum 1 = 1
trigNum n = n + trigNum (n-1)
```

Recursive; smaller n

Patterns

- The argument can match a pattern

Pattern

```
trigNum 1 = 1
trigNum n = n + trigNum (n-1)
```

- Equivalent to:

```
trigNum n
   | n == 1     = 1
   | otherwise = n + trigNum (n-1)
```

Practical break

Refection 3: How Functions Work

Comparison with dry running a Python program

Example Python Program

- Variables are:
  - mark
  - total
  - min
  - average
  - grade

```
# Enter two marks
#   Save minimum
mark = int(input("Mark 1 > "))
total = mark
min = mark

mark = int(input("Mark 2 > "))
if mark < min:
    min = mark
total = total + mark

# Calculate average
average = total / 2

# Calculate grade
if min < 30 or average < 50:
   grade = "fail"
else:
   grade = "pass"
```

Dry Running a Program

- Table has column for each variable
- Row for each step

Memory

Sequence

Step	Variable				
	mark	total	min	average	grade
1	35				
2		35			
3			35		
4	45				
5		80			
6				40	
7					fail

Rewriting (Reduction)

- Replace each call to a function by its definition
- Replace arguments by expressions

```
trigNum 1 = 1
trigNum n = n + trigNum (n-1)
```

```
trigNum 3
  = 3 + trigNum 2
  = 3 + 2 + trigNum 1
  = 3 + 2 + 1
  = 6
```

Lists

Lists in Haskell

- Haskell has lists … similar to Python
- LISP
  - First functional language
  - ‘List processing’
- Example: `[1, 2, 3]`
- Equivalent to:

```
1 : 2 : 3 : []
```

Cons

Empty list

Useful List Functions

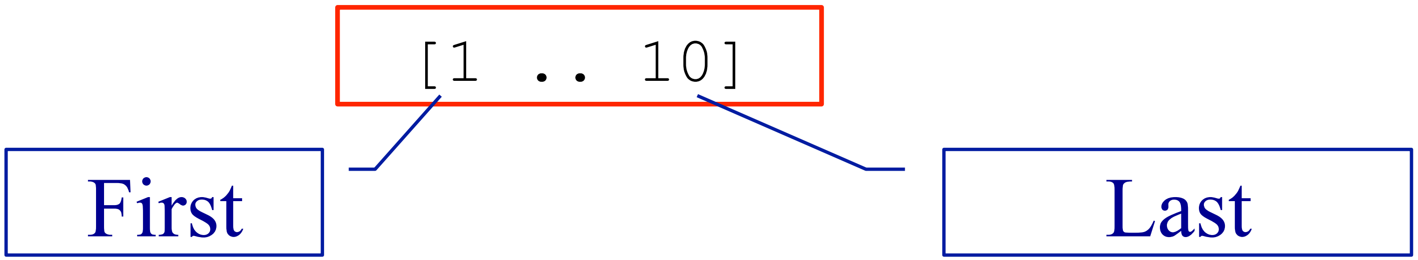
Function	Description	Example

`elem`	Member of list	`Main> elem 4 [1,2,3,4,5]`
		`True`
		`Main> elem 4 [1,3,5]`
		`False`

`head`	First element of list	`Main> head [2,4,6,8]` `2`
`tail`	List without first element	`Main> tail [3,5,7,9]` `[5,7,9]`
`++`	Concatenate two lists	`Main> [1,2,3] ++ [7,9]` `[1,2,3,7,9]`

Ranges

- Similar to Python



List Recursion

- Many functions on lists are defined recursively
- Base case: empty list
- Recursive case: apply to tail of list

```
-- length of a list
len []       = 0
len (x:xs) = 1 + len xs
```

Base case

Recursive call

Pattern - empty

Pattern – not empty

Practical break

Map, Filter and Fold

- Functions that abstract common ways of processing a list
- Called ‘recursive functions’

Two Similar Functions

- Two functions that create a new list from an old one
  - The new list is the same length
  - Each new element is derived from the corresponding old element

```
-- Add 1 to each entry is a list
addOne []       = []
addOne (x:xs) = x+1:addOne xs
```

```
-- Square each entry in a list
square []       = []
square (x:xs) = x*x:square xs
```

Using Map

- A function to apply a function to each element in a list

```
inc x = x + 1

-- Add 1 to each entry is a list
addOne ls = map inc ls
```

```
square x = x * x

-- Square each entry in a list
squares xs = map square xs
```

Filter

- Select items from a list

Predicate

```
moreThan a b = b > a

Main> filter (moreThan 3) [3,2,5,1,7,8]
[5,7,8]
```

How is Map Defined?

- Recursive definition of map

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

```
map inc [1,2,3]
  =
```

Fold – Reducing a list

- Combine the elements of a list

```
-- length of a list
len []       = 0
len (x:xs) = 1 + len xs
```

```
-- sum of a list
addUp []       = 0
addUp (x:xs) = x + addUp xs
```

Using Fold – Reducing a list

- Combine the elements of a list

```
count x y = y + 1

-- length of a list
len xs = foldr count 0 xs
```

```
add x y = x + y

-- sum of a list
addUp xs = foldr add 0 xs
```

How is Foldr Defined?

- Recursive definition of foldr

```
foldr f a []     = a
foldr f a x:xs = f x (foldr f a xs)
```

```
foldr add 0 [1,2,3]
  = add 1 (foldr add 0 [2,3])
  = add 1 (add 2 (foldr add 0 [3]))
  = add 1 (add 2 (add 3 (foldr add 0 [])))
  = add 1 (add 2 (add 3 0))
  = add 1 (add 2 3)
  = add 1 5
  = 6
```

Map, Foldr, Filter – Summary

Function	Description
`map`	Apply function to each list element
`filter`	Select elements satisfying a predicate
`foldr`	Combine elements using a function

- These are called recursive function
- foldr is more general – *it can be used to define the other two*

Google Map Reduce

- Very large datasets can be processed using the Map Reduce framework
  - Divide the list of input
  - Map function to each list (separate computers)
  - Reduce list of results (from the separate computers)

Refection 4: The Best Language?

Programming Language

- Between machine and users

Machine | C | Java | Haskell | User

- More abstract
- Haskell is ‘declarative’
- Performance

Functional Programming in Practice

- Functional languages
  - LISP – the original one
  - Haskell
  - Scala – compiles to JVM
  - F♯ – compiles to .NET

- Influences
  - Java, Python, C♯
  - Python has versions of map and fold

Further Haskell Topics

- Map, folder, filter
- List comprehension
- Anonymous functions – lambda
- Types
- Polymorphism
- Input and output

Summary – Functional Programming

- Programming with expressions
- No statements
  - No assignment → no variables
  - No sequence → no loops
- Composition of functions
- Possible and practical
  - Programs can be shorter