Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 5

MIRANDA

Patterns, functions, recursion & lists

FUNCTIONAL PROGRAMMING MIRANDA

This lecture roughly follows the online book “Programming with Miranda”, Chapters 2 and 3, which are required (essential) reading. The book can be found at the bottom of the “Resources” tab on the Moodle page, or can be downloaded from:

http://www0.cs.ucl.ac.uk/teaching/3C11/book/book.html

Here we cover a broad range of pragmatic issues relating to programming with Miranda, culminating in the use of recursive functions operating on lists

CONTENTS

- Offside rule / where blocks / evaluation
- Partial / polymorphic functions
- Patterns and pattern-matching
- Recursive functions
- Type synonyms
- Lists
  - Functions using lists
  - Recursive functions using lists

FUNCTIONAL PROGRAMMING MIRANDA

FUNCTIONS

Normal Order reduction with sharing (lazy evaluation)

Functions have names (there is no explicit λ character)

Formal parameters can be "patterns" that mirror data structure components. E.g.:

```
f :: (num, num) -> num
f z = 23

g :: (num, num) -> num
g (x,y) = x
```

The **offside rule enforces layout** for multi-line definitions:

"**where**" blocks hold local definitions. E.g:

```
h x = x + z
      where
      z = x – 1
```

The scope of "z" is the entire function body including other definitions in the where block

Functions

Miranda functions use Normal Order reduction. The implementation shares pointers to arguments so that multiple use doesn't entail multiple copies of an argument into the function body

Functions have names (there is no explicit λ character), and formal parameter names can be "patterns" that mirror data structure components. So a function operating on a 2-tuple can give it a single name or give names to its components:

```
f :: (num, num) -> num
f z = 23

g :: (num, num) -> num
g (x,y) = x
```

Note that lazy evaluation means that for "f" the argument z is never evaluated, and for "g" the item y is never evaluated. Thus g (34, (23/0)) evaluates to 34 and the divide-by-zero is never attempted

The **offside rule enforces layout** for multi-line definitions: *"every token of the object [must] lie either directly below or to the right of its first token. A token which breaks this rule is said to be 'offside' with respect to that object"*

"**where**" blocks hold local definitions. E.g:

```
h x = x + z
      where
      z = x – 1
```

The scope of "z" is the entire function body including other definitions in the where block [like "letrec" in other languages]

FUNCTIONAL PROGRAMMING MIRANDA

FUNCTIONS continued

Partial functions have no result (i.e. return an error) for some valid values of valid input type. E.g.

```
f :: (num, num) -> num
f (x,y) = x / y
```

Polymorphic functions do not evaluate some of their input. For example:

```
three :: * -> num
three x = 3
```

For many different polymorphic types, use *, **, ***,****:

```
f :: (num, *, **, ***) -> num
f (a, b, c, d) = a
```

For linked polymorphic types:

```
fst :: (*, **) -> *
fst (x, y) = x

snd :: (*, **) -> **
snd (x,y) = y

g :: (num, *) -> (*, num)
g (x, y) = (y, -x)

h :: (*, num, *) -> num
h (x,y,z) = y, if (x=z)
          = -y, otherwise
```

Functions continued

Partial functions have no result (i.e. return an error) for some valid values of valid input type. E.g. the following function gives an error whenever the second item of the 2-tuple is 0:

```
f :: (num, num) -> num
f (x,y) = x / y
```

Polymorphic functions do not evaluate some of their input (and therefore the type of that part of the input need not be constrained — though some constraints can be specified). A function argument whose type is polymorphic has a type denoted by a star. For example:

```
three :: * -> num
three x = 3
```

If there is more than one polymorphic argument and each can be a different type, this is signified by using *, **, *** etc. For example:

```
f :: (num, *, **, ***) -> num
f (a, b, c, d) = a
```

If two polymorphic types are linked (must be the same), just use * or ** twice. E.g.:

```
fst :: (*, **) -> *
fst (x, y) = x
snd :: (*, **) -> **
snd (x,y) = y
g :: (num, *) -> (*,num)
g (x, y) = (y, -x)
h :: (*, num, *) -> num
h (x,y,z) = y, if (x=z)
          = -y, otherwise
```

FUNCTIONAL PROGRAMMING MIRANDA

FUNCTIONS continued

Tuple patterns can be used either in a definition or in a test:

(*x*, *y*, *z*) = (3, "*hi*", (34, *True*, [3]))

f (x,y) = 23, if (*x*, *y*) = (3, "*hi*")
= 24, otherwise

A function can be defined by a series of alternative equations, where using different patterns on the left signifies case analysis. Patterns can contain values, names and "constructors" (e.g. tuple brackets):

```
f :: (bool, num) -> num
f (True, x)  = x
f (False, x) = -x
```

Patterns in function definitions are evaluated top-down:

```
f :: num -> num
f     3 = 45
f  489 = 3
f      x = 345*219
```

Patterns

Tuple patterns can be used either in a definition or in a test. The following provides a simultaneous definition of three variables:

(*x*, *y*, *z*) = (3, "*hi*", (34, *True*, [3]))

This is equivalent to the following three lines of code:

```
x = 3
y = "hi"
z = (34, True, [3])
```

The following function definition uses a tuple pattern in a test:

f (x,y) = 23, if (*x*, *y*) = (3, "*hi*")
= 24, otherwise

Argument patterns for function definitions

A function can be defined by giving a series of alternative equations, in which the use of different patterns on the left expresses case analysis on the arguments. Patterns can contain values, names and "constructors" (e.g. tuple brackets):

```
f :: (bool, num) -> num
f (True, x) = x
f (False, x) = -x
```

Patterns in function definitions are evaluated top-down: the first to match returns the associated function body.

```
f :: num -> num
f     3 = 45
f  489 = 3
f      x = 345*219  ||a name matches any value
```

FUNCTIONAL PROGRAMMING MIRANDA

FUNCTIONS continued

Non-exhaustive patterns:

```
f True = False
main = f False
```

"program error : missing case in definition of f"

Patterns can destroy laziness: the following function "notlazy_fst" when applied to (34, (67/0)) would give a runtime error *"program error : attempt to divide by zero") :*

```
notlazy_fst (x,0) = x
notlazy_fst (x,y) = x
```

Duplicate parameter names create an implicit equality test:

```
bothequal (x, x) = True
bothequal (x, y) = False
```

Argument patterns in function definitions

Non-exhaustive patterns: if none of the alternative definitions for a function match the argument data, a runtime error can occur. E.g. the following function f when applied to False will give *"program error : missing case in definition of f":*

```
f True = False
```

Patterns can destroy laziness: the following function "notlazy_fst" when applied to (34, (67/0)) would give a runtime error *"program error : attempt to divide by zero") :*

```
notlazy_fst (x,0) = x
notlazy_fst (x,y) = x
```

The pattern match on the value 0 in the first equation forces the second item of the tuple to be evaluated, forcing the error to occur

Note: patterns cannot contain function applications (Miranda has a special exception "(n + k)" where n is a variable and k is a literal integer and only matches an integer but its use is not recommended for this module)

Duplicate parameter names imply an equality test in Miranda. For example, in the following function bothequal the name "x" appears twice in the first pattern, and this pattern only matches input where both values x are the same:

```
bothequal (x, x) = True
bothequal (x, y) = False
```

FUNCTIONAL PROGRAMMING MIRANDA

RECURSIVE FUNCTION DEFINITIONS

Loops can be created using :

- Iterative constructs such as [1..] and list comprehensions
- Recursion, where a function calls itself inside its own body

Beware writing a function that loops forever:

```
loopforever x = loopforever x
```

To avoid infinite loops, a recursive function must have:

- A terminating condition
- A changing argument
- … that converges on the terminating condition

For example:

```
f :: num -> [char]
f 0 = ""
f n = "X" ++ (f (n-1)), if n > 0
    = "X" ++ (f (n+1)), otherwise
```

Recursive function definitions

Loops can be created using :

- Iterative constructs such as [1..] and list comprehensions [see later slide]
- Recursion, where a function calls itself inside its own body

Recursion is powerful and flexible, and the implementation of recursion is highly optimised in modern functional languages

Beware however that, as with most programming languages, it is easy to write a function that loops forever:

```
loopforever x = loopforever x
```

To avoid looping forever, a recursive function must have three things:

- A terminating condition
- A changing argument
- … that converges on the terminating condition

For example:

```
f :: num -> [char]
f 0 = ""
f n = "X" ++ (f (n-1)), if n > 0
    = "X" ++ (f (n+1)), otherwise
```

Note that "++" is Miranda's "append" operator for concatenating two lists. For example:

"hello" ++ " world" = "hello world"

FUNCTIONAL PROGRAMMING MIRANDA

RECURSIVE FUNCTION DEFINITIONS

Stack recursion occurs when the recursive call is an argument to another function or operator:

```
f :: num -> [char]
f 0 = ""
f n = "X" ++ (f (n-1))
```

The evaluation of (f 3) proceeds as follows:

⟶ "X" ++ (f (3-1))
⟶ "X" ++ (f 2)
⟶ "X" ++ ("X" ++ (f (2-1)))
⟶ "X" ++ ("X" ++ (f 1))
⟶ "X" ++ ("X" ++ ("X" ++ (f (1-1))))
⟶ "X" ++ ("X" ++ ("X" ++ (f 0)))
⟶ "X" ++ ("X" ++ ("X" ++ ""))
⟶ "X" ++ ("X" ++ "X")
⟶ "X" ++ "XX"
⟶ "XXX"

Stack recursion

There are different ways to write a recursive function. "Stack recursion" occurs when the recursive call is an argument to another function or operator. For example as an argument to the ++ operator:

```
f :: num -> [char]
f 0 = ""
f n = "X" ++ (f (n-1))
```

The evaluation of (f 3) proceeds as follows:

⟶ "X" ++ (f (3-1))
⟶ "X" ++ (f 2)
⟶ "X" ++ ("X" ++ (f (2-1)))
⟶ "X" ++ ("X" ++ (f 1))
⟶ "X" ++ ("X" ++ ("X" ++ (f (1-1))))
⟶ "X" ++ ("X" ++ ("X" ++ (f 0)))
⟶ "X" ++ ("X" ++ ("X" ++ ""))
⟶ "X" ++ ("X" ++ "X")
⟶ "X" ++ "XX"
⟶ "XXX"

Notice how the expression grows and then reduces, with the recursive call embedded inside the expression

FUNCTIONAL PROGRAMMING MIRANDA

RECURSIVE FUNCTION DEFINITIONS

Accumulative recursion occurs when the recursive call is **not** an argument to another function or operator. For example:

```
plus : : (num, num) -> num
plus (x, 0) = x
plus (x, y) = plus (x+1, y-1)
```

The evaluation of (plus (1,2)) proceeds as follows:

⟶ plus (1+1, 2-1)
⟶ plus (1+1, 1)
⟶ plus (1+1+1, 1-1)
⟶ plus (1+1+1, 0)
⟶ 1+1+1
⟶ 2 + 1
⟶ 3

Accumulative recursion

"Accumulative recursion" occurs when the recursive call is **not** an argument to another function or operator. For example:

```
plus : : (num, num) -> num
plus (x, 0) = x
plus (x, y) = plus (x+1, y-1)
```

The evaluation of (plus (1,2)) proceeds as follows:

⟶ plus (1+1, 2-1)
⟶ plus (1+1, 1)
⟶ plus (1+1+1, 1-1)
⟶ plus (1+1+1, 0)
⟶ 1+1+1
⟶ 2 + 1
⟶ 3

Notice how the recursive call is not embedded inside any expression: the function keeps calling itself, with the eventual result being constructed (lazily) as one of the arguments – this is called the "accumulating parameter"

FUNCTIONAL PROGRAMMING MIRANDA

TYPE SYNONYMS

- A type synonym acts like a type macro
- Used as a shorthand, does not create a new type
- Type error messages don't use type synonyms

Example:

- Without type synonyms we might have a function with the following type:

```
f : : (([char],num,[char]),(num,num,num)) -> bool
```

- Compare this with an equivalent type definition using two type synonyms "str" and "coord"

```
str      = = [char]
coord = = (num, num, num)

f : : ((str,num,str), coord) -> bool
```

Type synonyms

As a digression, we note that as function definitions become more complex the type of each such function can become complex. Sometimes it can help when writing a program with complex types to use "type synonyms" as a shorthand:

- A type synonym acts like a type macro
- It is used as a shorthand, but it does not create a new type
- The type system never sees the type synonyms (they are expanded before the type system is run) and so unfortunately type error messages don't use type synonyms

Here is an example where the operator "==" is used twice to create two type synonyms:

- Without type synonyms we might have a function with the following type:

```
f : : (([char],num,[char]),(num,num,num))
-> bool
```

- Compare this with an equivalent type definition using two type synonyms "str" and "coord"

```
str      == [char]
coord == (num, num, num)

f : : ((str,num,str), coord) -> bool
```

FUNCTIONAL PROGRAMMING MIRANDA

LISTS

A list of items, each of type **a,** is either :

- Empty, or
- An item of type **a** together with a list of items of the same type

Examples:

```
x1 :: [num]
x1 = (34 : (13 : []))        || “constructive” format
x2 :: [num]
x2 = [34,13]                 || “aggregate” format
```

Iterative constructs

```
[1..]                        || the infinite list of integers
                             || starting at 1
[ n*n | n <- [1,2,3] ]       || the list of all n*n such that n is
                             || drawn from the list [1,2,3]
```

The latter is an example “list comprehension”

LISTS

A list of items, each of type **a,** is either :

- Empty, or
- An item of type **a** together with a list of items of the same type

Here are two examples using two alternative syntaxes. Names x1 and x2 are both of type [num] and are bound to identical values. x1 uses what we call a “constructive” format (generally used for building lists dynamically), where “[]” denotes the empty list (called “nil”) and “:” (called “cons”) is a built-in operator that inserts an item into a list of items. By contrast x2 uses what we call an “aggregate” format (generally used when the items in the list are known statically), where the empty list is implicit and the “:” operator is not used:

```
x1 :: [num]
x1 = (34 : (13 : []))
x2 :: [num]
x2 = [34,13]
```

There are also iterative constructs to create lists. For example the syntax [1..] is the infinite list of integers starting at 1

List Comprehensions are very powerful list-building iterative constructs. For example [n*n | n <- [1,2,3]] is “the list of all n*n such that n is drawn from the list [1,2,3]” (i.e. the list [1, 4, 9]). New variables used inside a list comprehension are local to the list comprehension

FUNCTIONAL PROGRAMMING MIRANDA

FUNCTIONS ON LISTS

```
bothempty : : ([*],[**]) -> bool
bothempty ([], []) = True
bothempty anything = False

myhd  : : [*] -> *
myhd [] = error “take head of empty list”
myhd (x : rest) = x
```

Notice that “(x:rest)” is a list pattern

There are two built-in functions on lists:

```
hd :: [*] -> *    || returns the leftmost (head) item of a list
tl  :: [*] -> [*]  || returns the list without the head item
```

Example functions on lists

The following function returns True if both items in its argument (which is a 2-tuple) are empty lists:

```
bothempty : : ([*],[**]) -> bool
bothempty ([], []) = True
bothempty anything = False
```

The following function returns the head (the leftmost item) of a list of items. If the argument list is empty, it uses the built-in error function to abort the program and print an error message on the screen

```
myhd : : [*] -> *
myhd [] = error “take head of empty list”
myhd (x : rest) = x
```

Notice the use of a “list pattern” in both of the alternative equations for the above definition – in the pattern “(x : rest)” the leftmost (“head”) item of the argument list is bound to the name “x” and the remainder of the list (without the head item) is bound to the name “rest”

There are two built-in functions hd :: [*] -> * and tl :: [*] -> [*] (pronounced “head” and “tail”) that respectively return (i) the head of a list, and (ii) the list with the head item removed. In both cases an error is raised if the function is applied to an empty list

FUNCTIONAL PROGRAMMING MIRANDA

RECURSIVE FUNCTIONS ON LISTS

Using stack recursion and pattern matching to sum all of the items in a list of numbers:

```
sumlist : : [num] -> num
sumlist [] = 0
sumlist (x : rest) = x + (sumlist rest)
```

A second example finds the length of a polymorphic list:

```
length : : [*] -> num
length [] = 0
length (x : rest) = 1 + (length rest)
```

Recursive functions on lists

Recursive functions on lists very often pattern-match on the empty list (the "nil" value) as a terminating condition and use a list pattern in the alternative equation to define the function

Here are two examples. The first uses stack recursion to sum all of the items in a list of numbers:

```
sumlist : : [num] -> num
sumlist [] = 0
sumlist (x : rest) = x + (sumlist rest)
```

The second uses stack recursion to determine the length of a polymorphic list (i.e. the number of items in a list of anything):

```
length : : [*] -> num
length [] = 0
length (x : rest) = 1 + (length rest)
```

FUNCTIONAL PROGRAMMING MIRANDA

Exercise

Can you write a function “threes” which takes as input a list of numbers and produces as output a count of how many times the number 3 occurs in the input ?

EXERCISE

Can you write a function “threes” which takes as input a list of numbers and produces as output a count of how many times the number 3 occurs in the input ?

FUNCTIONAL PROGRAMMING MIRANDA

In summary, this lecture has (roughly) followed the online book “Programming with Miranda”, Chapters 2 and 3, covering a broad range of pragmatic issues relating to programming with Miranda, culminating in the use of recursive functions on lists. The final slide set an exercise that you should attempt to solve before the next lecture

SUMMARY

- Offside rule / where blocks / evaluation
- Partial / polymorphic functions
- Patterns and pattern-matching
- Recursive functions
- Lists
  - Functions using lists
  - Recursive functions using lists

FUNCTIONAL PROGRAMMING
MIRANDA