COMP0020 Functional Programming

Lecture 5

Miranda

patterns, functions, recursion and lists

Contents

Functions

- Function evaluation : normal order, lazy evaluation
- e.g. $fst\ (34, (23, 21)) \equiv 34$ (NB *fst* returns the first item in a two-tuple)
- The offside rule for multi-line definitions, and "where" blocks
  - "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"

$$f\ x = x \qquad\qquad f\ x = x$$
$$+\ 1 \qquad\qquad +\ 1$$

  - where blocks for local definitions

$$x = x + y$$
$$\textit{where}$$
$$y = x - 1$$

- Application associates to the left ! $f\ 34\ 27 \equiv ((f\ 34)\ 27)$ and $f\ g\ 27 \equiv ((f\ g)\ 27)$
- Mapping from source to target type domains : $inc\ ::\ num -> num$

Functions

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

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

- **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)

$$fst :: (*, **) -> *$$
$$fst\ (x, y) = x$$

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

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

$$three :: * -> num$$
$$three\ x = 3$$

Patterns

- Tuple patterns
  - $(x, y, z) = (3, "hello", (34, True), [3])$
    - ⋆ as a test
    - ⋆ as a definition
- Patterns for function definitions

$$not\ True\ =\ False$$
$$not\ False\ =\ True$$

- Top-down evaluation semantics of paterns in function definitions

$$f\ 3\ =\ 5$$
$$f\ 489\ =\ 3$$
$$f\ any\ =\ 345 * 219$$

Patterns in function definitions

- Non-exhaustive patterns : if none of the alternative definitions for function match the argument data, a runtime error can occur (e.g. *"program error : missing case in definition of f"*).
  f True = False
- Patterns can destroy laziness ((*notlazy_fst (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

- *Can a pattern contain a function application ?*
  - *No ! A pattern must be a constant expression*
  - *Special exception — (n + k) where n is a variable and k is a literal integer and only matches an integer (in Miranda, not Amanda) (not recommended)*

- *Duplicate parameter names create an implicit equality test (Miranda only, not Amanda)*
  bothequal (x, x) = True
  bothequal (x, y) = False

Recursive Functions

- Loops can be created using :
  - Iterative constructs such as [1./] and list comprehensions (see later, under "lists")
  - Recursion
    - A function calls itself inside its own body
    - Very powerful — very flexible
    - Highly optimised in modern functional languages

Recursive Functions

- Beware !
  loopforever x = loopforever x

- To avoid looping forever, a recursive function must have three things :
  1. A Terminating Condition
  2. A changing argument
  3. — that converges on the terminating condition !

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


(NB "++" is the "append" operator — it concatenates two lists, thus "hello "++"mum" → "hello mum")

Recursive Functions

- Stack recursion

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

Therefore 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"

Recursive Functions

- Accumulative recursion

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

Therefore for plus (1, 2)
→ plus (1+1, 2-1)
→ plus (1+1, 1)
→ plus (1+1+1, 1-1)
→ plus (1+1+1, 0)
→ 1+1+1

Type Synonyms

- Used as a shorthand (acts as a type macro, does not create a new type)
- Compare these alternatives :


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

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

LISTS

- Lists are another way to collect together related data
- But lists are RECURSIVE

- (Data can be defined in a recursive way, just like functions!)

LISTS

- A list of type a is either :
  - Empty, or
  - An element of type a together with a list of elements of the same type
- Examples of a list of num (using two alternative syntaxes) :
  - (34 : []) [34]
  - (34 : (13 : [])) [34, 13]
- Iterative constructs : [1..]
- List Comprehensions : [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. it defines the list [1, 4, 9]). Note that new variables used inside a list coprehension are local to the list comprehension.

Functions using lists


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



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

Recursive functions using lists


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



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

Exercise

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

Summary

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

END OF LECTURE