patterns, functions, recurs

Contents

- Offside rule / where b
- Partial / polymorp
- Patterns and patte
- Recursive functions
- Lists
  - Functions using lists
  - Recursive functions using lists

Functions

- Function evaluation : normal order, lazy evaluation
- e.g. $fst\ (34, (23/9)) \equiv 34$ (NB fst returns the first item in a two-tuple)
- The offside rule for multi-line definitions, and "where" blocks
  - "every token o
    which breaks t

    $f\ x$

    $+\ 1$ $+\ 1$
  - where blocks for local definitions

    $f\ x \equiv x$

    wh

    $y\ =\ x - 1$
- 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 fu** rt of the input need not be c

$$fst :: (*, **) -> * \qquad snd\ :: (\ ,\ )\ \ >\ **$$

$$fst\ (x, y) = x \qquad snd$$

$$g :: (*, num) -> (num, *)$$

$$g\ (x, y) = (-y, x) \qquad three\ x = 3$$

Patterns

- Tuple patterns
  - $(x, y, z) = (3, \text{“}hello\text{''}, (34, True, [3]))$
    - as a test
    - as a definition
- Patterns for functi

- Top-down evaluation semantics of paterns in function definitio

$$
\begin{aligned}
f\ 3 &= 45 \\
f\ 489 &= 3 \\
f\ any &= 345 * 219
\end{aligned}
$$

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 destr *ogram error : attempt to divi*
  notlazy_fst (x,0)
  notlazy_fst (x,y)
- *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 a 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 create
  - Iterative cons
  - 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 for
  1. A Terminati
  2. A changing ar
  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 eval

  → “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) -> n
plus (x, 0) = x
plus (x, y) = plus (x+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 (
- Compare these alt

```
f : : (([char],num,[ch
```

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

LISTS

- Lists are another wa
- But lists are RECUR

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

LISTS

- A list of type a is either :
  - Empty, or
  - An element of t
- Examples of a list of nu

  - (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 dra [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 : : ([*],[**]) ->
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 + (sumlis
```

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

Exercise

- Can you write a funct
output a count of how many times the number 3 occurs in the input ?

Summary

- Offside rule / where b
- Partial / polymorp
- Patterns and patte
- Recursive functions
- Lists
  - Functions using lists
  - Recursive functions using lists