Functional Programming

Pattern Matching

H. Turgut Uyar

2013-2017

License

Topics

1. Data Types
   - Tuples
   - Lists
   - Algebraic Types
2. Pattern Matching
   - Patterns
   - Parameter Patterns
   - Examples
3. Lists
   - List Expressions
   - Standard Functions
   - Examples

Tuples

- tuple: a collection of a fixed number of values
- different but fixed types

```
n :: (t1, t2, ..., tn)
n = (e1, e2, ..., en)
```

- selector functions on pairs:
  `fst`, `snd`

Tuple Example

representing a term in a polynomial: $2.4x^2$

```
t :: (Float, Integer)
t = (2.4, 2)

-- fst t ~> 2.4
-- snd t ~> 2
```


Tuple Parameters

- tuples can be sent as parameters
- not the same as multiple parameters
- tuples can be returned as result

Tuple Parameter Example

```
gcd :: Integer -> Integer -> Integer
gcd x y
  | y == 0    = x
  | otherwise = gcd y (x ‘mod‘ y)

-- gcd 9702 945
```

```
gcd’ :: (Integer, Integer) -> Integer
gcd’ a
  | snd a == 0 = fst a
  | otherwise  = gcd’ (snd a, (fst a) ‘mod‘ (snd a))

-- gcd’ (9702, 945)
```


Tuple Result Example

simplifying a fraction

```
simplify :: (Integer, Integer) -> (Integer, Integer)
simplify f = (n ‘div‘ g, d ‘div‘ g)
  where
    n = fst f
    d = snd f
    g = gcd n d
```

Type Synonyms

- type synonym: giving an existing type a new name

```
type NewName = ExistingType
```

example

```
type Term = (Float, Integer)

t :: Term
t = (2.4, 2)
```


Example: Type Synonyms

```
type Fraction = (Integer, Integer)

simplify :: Fraction -> Fraction
simplify f = (n ‘div‘ g, d ‘div‘ g)
  where
    n = fst f
    d = snd f
    g = gcd n d

x :: Fraction
x = (21, 14)

-- simplify x ~> (3, 2)
```


Example: Type Synonyms

```
type DayInYear = (Integer, Integer)

dec4 :: DayInYear
dec4 = (4, 12)

-- simplify dec4 ~> (1, 3)
```


Lists

- list: a combination of an arbitrary number of values
- all of the same type

```
n :: [t]
n = [e1, e2, ..., en]
```

List Example

second degree polynomial: $2.4x^2 + 1.8x - 4.6$

```
p1 :: (Float, Float, Float)
p1 = (-4.6, 1.8, 2.4)
```

any degree polynomial: $3.4x^3 - 7.1x + 0.5$

```
p2 :: [Float]
p2 = [0.5, -7.1, 0.0, 3.4]
```

sparse terms: $72.3x^{9558} - 5.0x^3$

```
p3 :: [Term]
p3 = [(-5.0, 3), (72.3, 9558)]
```


Lists

- a list consists of a first item (head)
  followed by a list of the remaining items (tail)
- note the recursion in the definition
- check if empty: `null`
- get the head: `head`
- get the tail: `tail`
- independent of type: `[a]`

List Operation Examples

```
null :: [a] -> Bool
-- null []           ~> True
-- null [1, 2, 3, 4] ~> False

head :: [a] -> a
-- head [1, 2, 3, 4] ~> 1
-- head []           ~> error
-- head [1]          ~> 1

tail :: [a] -> [a]
-- tail [1, 2, 3, 4] ~> [2, 3, 4]
-- tail []           ~> error
-- tail [1]          ~> []
```


List Construction

- list construction:
  `item : sublist`
- associates from the right

examples

```
(:) :: a -> [a] -> [a]
-- 1 : [2, 3]       ~> [1, 2, 3]
-- 1 : 2 : 3 : []   ~> [1, 2, 3]
```

List Size

number of elements in a list

```
length :: [a] -> Int
length xs
  | null xs   = 0
  | otherwise = 1 + length (tail xs)
```


List Example

sum of first two elements

```
firstPlusSecond :: [Integer] -> Integer
firstPlusSecond xs
  | null xs        = 0
  | null (tail xs) = head xs
  | otherwise      = head xs + head (tail xs)
```


Strings

- a string is a list of characters

```
type String = [Char]
```

examples

```
-- head "word"       ~> 'w'
-- tail "word"       ~> "ord"

-- null "word"       ~> False
-- null ""           ~> True

-- 'w' : "ord"       ~> "word"
-- length "word"     ~> 4
```


Algebraic Types

- algebraic types: constructors and components

```
data T = C1 t11 t12 ... t1m
       | C2 t21 t22 ....... t2n
       | ...
```

- value construction: `Ci ei1 ei2 ... eik`
- constructors are functions
- `Ci` may be the same as, or different from T

Algebraic Type Examples

simple product type

```
data Person = Person String Integer
              deriving Show

church :: Person
church = Person "Alonzo Church" 1903
```


Algebraic Type Examples

enumeration

```
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec
             deriving Show

m :: Month
m = Feb
```


Algebraic Type Examples

multiple options

```
type Coords = (Float, Float)
type Length = Float

data Shape = Point     Coords
           | Circle    Coords Length
           | Rectangle Coords Length Length
             deriving Show

p, c, r :: Shape
p = Point     (0.0, 0.0)
c = Circle    (0.0, 0.0) 1.0
r = Rectangle (5.9, 7.6) 5.7 2.3
```


Patterns

- expressions can be checked against patterns
- result is the expression for the first matched pattern

```
case expr of
    p1 -> e1
    p2 -> e2
    ...
    pn -> en
    _  -> e
```

- matched patterns generate bindings

Pattern Examples

literal value pattern

```
gcd :: Integer -> Integer -> Integer
gcd x y = case y of
    0 -> x
    _ -> gcd y (x `mod` y)
```


Pattern Examples

tuple pattern

```
gcd' :: (Integer, Integer) -> Integer
gcd' a = case a of
    (x, 0) -> x
    (x, y) -> gcd' (y, x `mod` y)

-- gcd' (9702, 945)
-- second pattern, bindings: x <-> 9702, y <-> 945

-- gcd' (63, 0)
-- first pattern, bindings: x <-> 63
```


Nested Patterns

- patterns can be nested

example

```
shift :: ((a, b), c) -> (a, (b, c))
shift s = case s of
    ((x, y), z) -> (x, (y, z))
```


Wildcards

- if binding not needed, use wildcard: _

example: third component of a triple

```
third :: (a, b, c) -> c
third t = case t of
    (x, y, z) -> z

-- OR:
third t = case t of
    (_, _, z) -> z
```

List Patterns

- empty list:
  `[]`
- nonempty list:
  `x:xs`
- list with exactly one element:
  `[x]`
- list with exactly two elements:
  `[x1,x2]`
- list with at least two elements:
  `x1:x2:xs`

List Pattern Examples

number of elements

```
length :: [a] -> Int
length xs = case xs of
    []     -> 0
    x:xs’ -> 1 + length xs’
```


List Pattern Examples

sum of the first and third elements

```
firstPlusThird :: [Integer] -> Integer
firstPlusThird xs = case xs of
    []         -> 0
    [x1]       -> x1
    [x1,_]     -> x1
    x1:_:x3:_  -> x1 + x3
```


List Pattern Examples

check whether a list is in nondecreasing order

```
nondecreasing :: [Integer] -> Bool
nondecreasing xs = case xs of
  []        -> True
  [_]       -> True
  x1:x2:xs -> x1 <= x2 && nondecreasing (x2 : xs)
```

- reconstructing not necessary: @

List Pattern Examples

check whether list is in nondecreasing order

```
nondecreasing :: [Integer] -> Bool
nondecreasing xs = case xs of
  []           -> True
  [_]          -> True
  x1:xs@(x2:_) -> x1 <= x2 && nondecreasing xs
```


Algebraic Type Patterns

- patterns can match algebraic types
- use pattern matching to get values out of product types

Algebraic Type Pattern Examples

get component out of product type

```
birthYear :: Person -> Integer
birthYear p = case p of
    Person _ y -> y

-- birthYear (Person "Alonzo Church" 1903) ~> 1903
-- binding: y <-> 1903
```


Algebraic Type Pattern Examples

number of days in a month

```
daysInMonth :: Month -> Integer -> Integer
daysInMonth m y = case m of
    Apr -> 30
    Jun -> 30
    Sep -> 30
    Nov -> 30
    Feb -> if y `mod` 4 == 0 then 29 else 28
    _   -> 31

-- daysInMonth Jan 2014 ~> 31
-- daysInMonth Feb 2014 ~> 28
-- daysInMonth Feb 2016 ~> 29
```

Algebraic Type Pattern Examples

area of a geometric shape

```
area :: Shape -> Float
area s = case s of
    Point     _      -> 0.0
    Circle    _ r    -> 3.14159 * r * r
    Rectangle _ h w -> h * w

-- area (Circle (0.0, 0.0) 3.0) ~> 28.274311
-- second pattern, binding: r <-> 3.0
```


Parameter Patterns

- formal parameters are patterns
- components of pattern matched with actual parameters
- in case of multiple patterns, first match will be selected

```
n p1 = e1
n p2 = e2
...
```


Parameter Pattern Example

```
gcd :: Integer -> Integer -> Integer
gcd x y = case y of
    0 -> x
    _ -> gcd y (x `mod` y)

-- OR:
gcd :: Integer -> Integer -> Integer
gcd x 0 = x
gcd x y = gcd y (x `mod` y)
```


Parameter Pattern Example

```
gcd' :: (Integer, Integer) -> Integer
gcd' a = case a of
    (x, 0) -> x
    (x, y) -> gcd' (y, x `mod` y)

-- OR:
gcd' :: (Integer, Integer) -> Integer
gcd' (x, 0) = x
gcd' (x, y) = gcd' (y, x `mod` y)
```

Parameter Pattern Example

```haskell
shift :: ((a, b), c) -> (a, (b, c))
shift s = case s of
    ((x, y), z) -> (x, (y, z))

-- OR:
shift :: ((a, b), c) -> (a, (b, c))
shift ((x, y), z) = (x, (y, z))
```


Parameter Pattern Example

```haskell
third :: (a, b, c) -> c
third t = case t of
    (_ , _, z) -> z

-- OR:
third :: (a, b, c) -> c
third (_, _, z) = z
```


Parameter Pattern Example

```haskell
length :: [a] -> Int
length xs = case xs of
    []    -> 0
    x:xs' -> 1 + length xs'

-- OR:
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```


Parameter Pattern Example

```haskell
birthYear :: Person -> Year
birthYear p = case p of
    Person _ y -> y

-- OR:
birthYear :: Person -> Year
birthYear (Person _ y) = y
```

Record Types

- give names to fields
- automatically creates functions to extract components

```
data T = C1 { n11 :: t11,
              n12 :: t12,
              ...,
              n1m :: t1m }
       | ...
```


Record Examples

example

```
data PersonR = PersonR { fullname :: String,
                         born :: Integer }
               deriving Show

church :: PersonR
church = PersonR "Alonzo Church" 1903
church = PersonR { born=1903, fullname="Alonzo Church" }

-- fullname church ~> "Alonzo Church"
-- born church     ~> 1903
```


Example: Quadrants

```
quadrant :: Coords -> Integer
quadrant (x, y) = case (x>=0, y>=0) of
    (True,  True)  -> 1
    (False, True)  -> 2
    (False, False) -> 3
    (True,  False) -> 4
```


Example: Fibonacci

```
fibStep :: (Integer, Integer) -> (Integer, Integer)
fibStep (u, v) = (v, u + v)

-- fibPair n ~> (fib n, fib (n + 1))
fibPair :: Integer -> (Integer, Integer)
fibPair 1 = (1, 1)
fibPair n = fibStep (fibPair (n - 1))

fastFib n = fst (fibPair n)
```

List Operators

- index: `!!`
- append: `++`

examples

```
-- [1, 2, 3] !! 0        ~> 1
-- [1, 2, 3] !! 2        ~> 3
-- [1, 2, 3] !! 3        ~> error

-- [1, 2, 3] ++ [4, 5]  ~> [1, 2, 3, 4, 5]
```


Example: Indexing Lists

```
(!!) :: [a] -> Int -> a
(!!) []      _ = error "no such element"
(!!) (x:xs) 0 = x
(!!) (x:xs) n = (!!) xs (n - 1)
```

- use the infix operator notation:

```
(!!) :: [a] -> Int -> a
[]     !! _ = error "no such element"
(x:xs) !! 0 = x
(x:xs) !! n = xs !! (n - 1)
```


Example: Appending Lists

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```


Ranges

- `[n .. m]`: range with increment 1
- `[n, p .. m]`: range with increment `p - n`

examples

```
[2 .. 7]          ~> [2, 3, 4, 5, 6, 7]
[3.1 .. 7.0]      ~> [3.1, 4.1, 5.1, 6.1, 7.1]
['a' .. 'm']      ~> "abcdefghijklm"

[7, 6 .. 3]       ~> [7, 6, 5, 4, 3]
[0.0, 0.3 .. 1.0] ~> [0.0, 0.3, 0.6, 0.8999999999999999]
['a', 'c' .. 'n'] ~> "acegikm"
```

Membership Check

- check whether an element is a member of a list
  `elem 'r' "word" ~> True`
  `elem 'x' "word" ~> False`

```
elem :: Char -> [Char] -> Bool
elem _ []     = False
elem x (c:cs) = if x == c then True else elem x cs
```

- exercise: make a list of n copies of an item
  `replicate 3 'c' ~> "ccc"`

Last Element

- get the last element of a list
  `last "word" ~> 'd'`

```
last :: [a] -> a
last []     = error "empty list"
last [x]    = x
last (x:xs) = last xs
```

- exercise: get all elements but the last of a list
  `init "word" ~> "wor"`

Split

- take n elements from the front of a list
  `take 3 "Peccary" ~> "Pec"`

```
take :: Int -> [a] -> [a]
take 0 _      = []
take _ []     = []
take n (x:xs) = x : take (n - 1) xs
```

- exercise: drop n elements from the front of a list
  `drop 3 "Peccary" ~> "cary"`
- exercise: split a list at a given position
  `splitAt 3 "Peccary" ~> ("Pec", "cary")`

Reverse

- reverse a list
  `reverse "word" ~> "drow"`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = (reverse xs) ++ [x]
```

Concatenate

- convert a list of lists of items into a list of items
  `concat [[2, 3], [], [4] ~> [2, 3, 4]`

```
concat :: [[a]] -> [a]
concat []       = []
concat (xs:xss) = xs ++ concat xss
```


Zip

- convert two lists into a list of pairs
  `zip [1, 2] "ab" ~> [(1, 'a'), (2, 'b')]`

```
zip :: [a] -> [b] -> [(a, b)]
zip []     []     = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

- not all cases are covered:
  `zip [1, 2] "abc" ~> [(1, 'a'), (2, 'b')]`

Zip

```
zip :: [a] -> [b] -> [(a, b)]
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []
```

- exercise: convert three lists into a list of triples
  `zip3 [1, 2] "abc" [7, 4]`
  `~> [(1, 'a', 7), (2, 'b', 4)]`

Unzip

- convert a list of pairs into a pair of lists
  `unzip [(1, 'a'), (2, 'b')] ~> ([1, 2], "ab")`

```
unzip :: [(a, b)] -> ([a], [b])
unzip []          = ([], [])
unzip ((x, y):xys) = (x : xs, y : ys)
  where
    (xs, ys) = unzip xys
```

- exercise: convert a list of triples into three lists
  `unzip3 [(1, 'a', 7), (2, 'b', 4)]`
  `~> ([1, 2], "ab", [7, 4])`

Example: Merging Lists

merge two ordered lists

```
merge :: [Integer] -> [Integer] -> [Integer]
merge xs     []     = xs
merge []     ys     = ys
merge (x:xs) (y:ys)
  | x <= y    = x : merge xs (y : ys)
  | otherwise = y : merge (x : xs) ys
```


Merging Lists

merge two ordered lists

```
merge :: [Integer] -> [Integer] -> [Integer]
merge xs [] = xs
merge [] ys = ys
merge xs@(x':xs') ys@(y':ys')
  | x' <= y'  = x' : merge xs' ys
  | otherwise = y' : merge xs  ys'
```


Roman Numeral Conversion

convert an integer to Roman numerals

- adapted from the book “Dive into Python” by Mark Pilgrim: `http://www.diveintopython.net/`

```
romanNumerals =
  [("M", 1000), ("CM", 900), ("D", 500), ("CD", 400),
   ("C",  100), ("XC",  90), ("L",  50), ("XL",  40),
   ("X",   10), ("IX",   9), ("V",   5), ("IV",   4),
   ("I",    1)]
```


Roman Numeral Conversion

Python

```
def toRoman(n):
    result = ""
    for numeral, integer in romanNumerals:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

Roman Numeral Conversion

```
toRoman :: Integer -> String
toRoman n = tR n romanNumerals
  where
    tR :: Integer -> [(String, Integer)] -> String
    tR n []                = ""
    tR n xs@((s, k):xs')
      | n >= k    = s ++ tR (n - k) xs
      | otherwise = tR n xs'
```

- exercise: convert a Roman numeral string into an integer

Roman Numeral Conversion

Python

```
def fromRoman(s):
    result = 0
    index = 0
    for numeral, integer in romanNumerals:
        while s[index : index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```


References

Required Reading: Thompson

- Chapter 5: Data types, tuples and lists
- Chapter 6: Programming with lists
- Chapter 7: Defining functions over lists