Syntax in Functions

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Pattern Matching

- **Pattern matching** is used to specify patterns to which some data should conform and to deconstruct that data according to those pattern
- Format:

```
fnName pattern_1 = expression_1
fnName pattern_2 = expression_2
    :
fnName pattern_n = expression_n
```

- Same function name
- All expressions after = must have the same type
- Patterns will be checked from top to bottom
  - **Order Does Matter**
  - Some compiler versions may issue a warning

Order Does Matter

- Consider the following functions, `isZero1` and `isZero2`:

```
isZero1 :: Int -> [Char]
isZero1 0 = "Zero"
isZero1 n = "Not Zero"
isZero2 :: Int -> [Char]
isZero2 n = "Not Zero"
isZero2 0 = "Zero"
```

- If we load this into `GHCi`, we may get a **warning**

```
ghci> :l Pattern01.hs
[1 of 1] Compiling Main             ( Pattern01.hs, interpreted )

Pattern01.hs:8:1: warning: [-Woverlapping-patterns]
    Pattern match is redundant
    In an equation for 'isZero2': isZero2 0 = ...
  |
8 | isZero2 0 = "Zero"
  | ^^^^^^^^^^^^^^^^^^
Ok, one module loaded.
```

- Since it is just a warning, the file is loaded successfully

Order Does Matter

- However, it will fail a test:

```
ghci> isZero1 0
"Zero"
ghci> isZero2 0
"Not Zero"
```

- Since `n` can be any number including 0, `isZero2 0` is matched with `isZero2 n` first
- When defining a function using patter matching, make sure to put more general cases later

Complete vs Partial

- We generally do not partially define a function unless it is our intention:
- Consider the following function:

```
aName :: Char -> [Char]
aName 'a' = "Alice"
aName 'b' = "Bob"
aName 'c' = "Carol"
```

- The above function is fine if we absolutely sure that `aName` will not be called with arguments other than `'a'`, `'b'`, or `'c'`

```
ghci> aName 'b'
"Bob"
ghci> aName 'd'
"*** Exception: Pattern01.hs:(11,1)-(13,19): Non-exhaustive
patterns in function aName
```

- Otherwise, make sure it is complete with a catch all:

```
aName :: Char -> [Char]
aName 'a' = "Alice"
aName 'b' = "Bob"
aName 'c' = "Carol"
aName _   = "You know who"
```

Pattern Matching with Tuples

- We can pattern matching with tuples
- Suppose a vector is represented by a pair `(x, y)`
- Suppose we want to create a function to add two vectors
- First Version:

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors v1 v2 = (fst v1 + fst v2, snd v1 + snd v2)
```

- Second Version:

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

- How about functions to extract component from triples?

```
first :: (a, b, c) -> a
first  (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

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

Pattern Matching with Lists and List Comprehensions

- Pattern matching can be used with list comprehensions
- Consider the following expression:

```
ghci> [a + b | (a, b) <- [(1,2),(3,4),(5,6)]]
[3,7,11]
```

- The pattern matching is at `(a, b) <- [...]`
- If a matching fails, it moves on to the element in the test
- Here is another example:

```
ghci> [a | (a, 4) <- [(1,2),(3,4),(5,6)]]
[3]
```

Pattern Matching with Lists and List Comprehensions

- Pattern matching with lists mostly focus on `[]` and `:`
- Here is an example of `myHead` function:

```
myHead :: [a] -> a
myHead [] = error "myHead: The list is empty"
myHead (x:_) = x
```

- And some tests:

```
ghci> myHead [1,2,3]
1
ghci> myHead "Hello"
'H'
ghci> myHead []
*** Exception: myHead: The list is empty
   :
```

- The `error` function takes a string and generates a run-time error with the string

```
ghci> :t error
error :: [Char] -> a
```

- Good for debugging but not for the production

Pattern Matching with Lists and List Comprehensions

- One more example on list:

```
listInfo []     = "The list is empty"
listInfo [x]    = "The list has one element " ++ show x
listInfo [x,y]  = "The list has two elements " ++ show x ++
                  " and " ++ show y
listInfo (_:_)  = "The list has three or more elements"
```

- We can use
  - `(x:[])` instead of `[x]`
  - `(x:y:[])` instead of `[x,y]`
- However, we **cannot** use `[x,y,_]` to match a list with three **or more** elements
- `[x,y,_]` is a list with **exactly** three elements
- The `show` function is pretty much the same as `toString()` in Java
- **Note**: You cannot use the ++ operator in pattern matching
  - There are multiple way to match `xs ++ ys` with `[1,2,3,4]`

As-Pattern

- **As-pattern** allow you to break up an item according to a pattern, while keeping a reference to the entire item
- Consider the following function:

```
firstItem [] = "[] has no first item"
firstItem (x:xs) = "The first item of " ++ show (x:xs) ++
                   " is " ++ show x
```

- We use the whole `(x:xs)` in the expression
- An as-pattern precedes a regular pattern with a name an an `@` characterpause

```
firstItem2 [] = "[] has no first item"
firstItem2 lst@(x:xs) = "The first item of " ++ show lst ++
                        " is " ++ show x
```

- The expression can refer to the item `(x:xs)` by the name `lst`
- Similar to the local binding

Guards

- A **Guard**, indicated by a pipe (|), is used to check if some property of passed values is true or false
  - Pretty much is `if-then-else` expression
  - A lot easier to read
- Consider the dice game High-Low with two dices

```
highLow d1 d2
    | d1 + d2 < 7  = "Low"
    | d1 + d2 == 7 = "Seven"
    | d1 + d2 > 7  = "High"
```

- Multiple conditions and catch all (otherwise) are supported:

```
highLow2 d1 d2
    | d1 + d2 < 7 && d1 + d2 >= 2  = "Low"
    | d1 + d2 == 7                 = "Seven"
    | d1 + d2 > 7 && d1 + d2 <= 12 = "High"
    | otherwise    = "Impossible!!!"
```

- In the above example, the computer may need to calculate `d1 + d2` multiple times

More about `where` keyword

- To avoid calculating the same value multiple times, we can use the `where` keyword
- Haskell's `where` store the results of intermediate computations
- Here is another High-Low:

```
highLow3 d1 d2
    | total < 7 && total >= 2  = "Low"
    | total == 7               = "Seven"
    | total > 7 && total <= 12 = "High"
    | otherwise    = "Impossible!!!"
    where total = d1 + d2
```

- Put the `where` keyword after the guards
- Names are visible across all the guards

The Scope of `where`

- The variables defined in the `where` section of a function are visible only in that function
- `where` will not take the namespace of other functions

```
highLow4 d1 d2
    | total < 7 && total >= 2  = "Low"
    | total == 7               = "Seven"
    | total > 7 && total <= 12 = "High"
    | otherwise     = "Impossible!!!"
   where total = d1 + d2
total = 10
```

  - Outside `highLow4`, `total` is `10`
- Just like Java and C, local bindings have higher priority:

```
sum = 5
highLow5 d1 d2
    | sum < 7 && sum >= 2  = "Low"
    | sum == 7             = "Seven"
    | sum > 7 && sum <= 12 = "High"
    | otherwise     = "Impossible!!!"
   where sum = d1 + d2
```

  - Inside `highLow5`, `sum` is `d1 + d2`

The Scope of `where`

- `where` bindings are not shared across function bodies of different patterns
- The following code snippet will cause a compilation error:

```
call [] = "No name???"
call ('A':_) = greeting ++ " Alice"
call _       = greeting ++ " You Know Who!!!"
             where greeting = "Hi"
```

- The variable `greeting` is visible in the last pattern only

Pattern Matching in `where`

- We can use `where` bindings to patter match
- Turn first name and last name to initials:

```
toInitial first last = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = first
          (l:_) = last
```

- Here is a run:

```
ghci> toInitial "John" "Smith"
"J. S."
```

- Note that inside the definition, `f` and l are characters.
- `[f]` and `[l]` are strings

The `let` Expression

- `let` and `where` are pretty similar
  - `where` binds variables at the end of a function
  - `let` binds variables anywhere
  - Both of them are very local (to the function)
- Syntax: `let <bindings> in <expression>`
- `let` is an expression but `where` is a keyword

```
ghci> let x = 5 in x + 1
6
ghci> x + 1 where x = 5

<interactive>:47:7: error: parse error on input 'where'
```

- Since `let` is an expression, it can be anywhere where expressions are allowed

The `let` Expression

- Recall a syntax of a function:

```
functionName <arguments> = <expression>
```

- We see this kind of definition before:

```
rectVolume w h d = let area = w * h
                   in area * d
```

- How about just a simple expression:

```
ghci> 1 + 2 + 3
6
```

- `1`, `2`, and `3` are all expressions in Haskell
- We can replace one of them with `let` if we want to:

```
ghci> 1 + (let x = 1 in x + 1) + 3
6
ghci> [1, let x = 1 in x + 1, 3]
[1,2,3]
```

The `let` Expression

- `let` can be used to generate a list:

```
ghci> let square x = x * x in [square 2, square 3, square 4]
[4,9,16]
```

- `let` can be used inside a list:

```
ghci> [1, 2, 3, let square x = x * x in square 2]
[1,2,3,4]
```

- Pattern matches in `let` are allowed:

```
ghci> let (w, h) = (2, 3) in w * h * 10
6
```

- Multiple bindings in one line can be done by separating them with semicolon

```
ghci> let w = 2; h = 3 in w * h * 10
60
```

List Comprehension with `let`

- We can also use the `let` binding in list comprehensions

```
calcRecAreas :: [(Double, Double)] -> [Double]
calcRecAreas xs = [area | (w, h) <- xs, let area = w * h]
```

- Note that there is no `in` keyword
- The bindings are visible to the output (the part before the `|`)
- The bindings are also visible after the `let`

```
onlyLargeRecAreas :: [(Double, Double)] -> [Double]
onlyLargeRecAreas xs = [area | (w, h) <- xs, let area = w * h, area > 60]
```

- With the `in` keyword, it will just be a local binding of the expression

```
ghci> [y | x <- [1,2,3], let y = 2 * x]
[2,4,6]
ghci> [y | x <- [1,2,3], let y = 2 * x in y < 3]

<interactive>:4:2: error: Variable not in scope: y
ghci> [x | x <- [1,2,3], let y = 2 * x in y < 3, y > 1]

<interactive>:3:44: error: Variable not in scope: y :: Integer
```