Haskell: Pattern Matching & Recursion

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

Learning Group Activity

Share your work with your learning group members. Discuss:

1. Where did you make use of type variables and typeclasses?
2. Were your declarations ever too restrictive for the problem? Or too broad?

I will go around and check off your LGA. Have any questions for me? Homework questions? Also feel free to ask your group on HW questions.

Pattern Matching

In Haskell, when you write multiple function bodies, the body that matches first gets called. For example:

```
hungryNumber :: (Integral a) => a -> String
hungryNumber 7 = "Eats nine"
hungryNumber 9 = "Gets eaten by seven"
hungryNumber n = "Does not eat other numbers"
```

Recursion using Pattern Matching

One excellent use for pattern matching: recursion.

```haskell
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Pattern Matching Tuples

A tuple with variables in a pattern match will assign names to each element.

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

Pattern Matching Lists

To match the empty list, use an empty list as the pattern:

```
hasStuff :: [a] -> Bool
hasStuff [] = False
hasStuff xs = True
```

Deconstruction Pattern

Use `(x:xs)` as a pattern to:

- *Match a list with one or more element*
- *Get the head of the list as x*
- *Get the tail of the list as xs*

The names x and xs are arbitrary, but common.

Deconstruction Pattern

Use `(x:xs)` as a pattern to:

- *Match a list with one or more element*
- *Get the head of the list as x*
- *Get the tail of the list as xs*

The names x and xs are arbitrary, but common.

Example:

```
head' :: [a] -> a
head' [] = error "Can't do a head on nothing!"
head' (x:xs) = x
```

Deconstruction & Recursion: Map Example

```haskell
map' :: (a -> b) -> [a] -> [b]
map' _ [] = []
map' f (x:xs) = f x : map' f xs
```

Patterns in List Comprehensions

Patterns can be placed into the RHS of a list comprehension:

```haskell
-- uses an input like [(1, 2), (3, 4)]
addPairs :: (Num a) => [(a, a)] => [a]
addPairs xs = [a + b | (a, b) <- xs]
```

Patterns in List Comprehensions

Patterns can be placed into the RHS of a list comprehension:

```haskell
-- uses an input like [(1, 2), (3, 4)]
addPairs :: (Num a) => [(a, a)] => [a]
addPairs xs = [a + b | (a, b) <- xs]
```

Similarly, you can use an `(x:xs)` pattern:

```haskell
-- uses an input like [[1, 3], [4]]
tails :: [[a]] -> [[a]]
tails xs = [ys | (y:ys) <- xs]
```

Patterns in List Comprehensions

Patterns can be placed into the RHS of a list comprehension:

```haskell
-- uses an input like [(1, 2), (3, 4)]
addPairs :: (Num a) => [(a, a)] => [a]
addPairs xs = [a + b | (a, b) <- xs]
```

Similarly, you can use an `(x:xs)` pattern:

```haskell
-- uses an input like [[1, 3], [4]]
tails :: [[a]] -> [[a]]
tails xs = [ys | (y:ys) <- xs]
```

Of course, this could have been written much more expressive as

```haskell
tails = map tail
```

As-Patterns

When the whole value of a pattern is desired, as well as the decomposed names, the @ symbol can be used to create an alias.

```
firstLetter :: String -> String
firstLetter word@(x:_) = "The first letter of "
                         ++ word ++ " is " + [x]
```