CS 152

Programming Paradigms

More Haskell

Pattern Matching

Bindings

User Defined Types

Today

- Pattern Matching
- Bindings with *let* and *where*
- Defining Types in Haskell

Haskell vs Java

"Finally, in the specific comparison of Haskell versus Java, Haskell, though not perfect, is of a quality that is several orders of magnitude higher than Java, which is a mess (and needed an extensive advertizing campaign and aggressive salesmanship for its commercial acceptance)...
It is not only the violin that shapes the violinist, we are all shaped by the tools we train ourselves to use, and in this respect programming languages have a devious influence: they shape our thinking habits. "

Edsger W.Dijkstra, 12 April 2001

https://www.cs.utexas.edu/users/EWD/transcriptions/OtherDocs/Haskell.html

Pattern Matching in Functions

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Reminder: Lists

- Just like in Scheme, the list notation in Haskell is syntactic sugar for nested cons.

iClicker:

The list [1, 2, 3] can be written as

A. 1:2:3
B. 1:2:3:[]
C. [1:2:3]
D. [1:2:3:[]]

Pattern Matching with Lists

```
>xs = [1, 2, 3, 4]
>x:y:z = xs
> x
1
> y
2
> z
```

iClicker:

What is z?

A. 3
B. 4
C. [4]
D. [3]
E. [3, 4]

Pattern Matching with Lists

```
>xs = [1, 2, 3, 4]
>x:y:z = xs
> x
1
> y
2
> z
[3, 4]
```

Pattern Matching with Lists

```
xs = [1, 2, 3, 4]
> x:_:z = xs -- underscore is a wildcard
> x
1
> z
[3, 4]
```

Pattern Matching with Lists & Functions

Task: Write a function *second* that takes in a list of any type and returns the second element of that list.

second :: [a] -> a

What cases need special consideration?

Pattern Matching with Lists & Functions

Task: Write a function *second* that takes in a list and returns the second element of that list.

```
second :: [a] -> a
second [] = error "The list is too s
second (_:[]) = error "The list is to
second (_:x:_) = x
```

```
> second [1, 2, 3, 4]
2
> second ["Hello"]
"*** Exception: The list is too short
> second []
*** Exception: The list is too short
```

Pattern Matching with Lists & Functions

Task: Write a function *swap* that takes in a list of any type and returns another list with the first and second elements swapped.

```
swap :: [a] -> [a]
```

What cases need special consideration?

Pattern Matching with Lists & Functions

Task: Write a function *swap* that takes in a list of any type and returns another list with the first and second elements swapped.

```
swap :: [a] -> [a]
swap [] = []
swap (only:[]) = [only]
swap (first:second:rest) = second
```

```
> swap [1, 2, 3]
[2,1,3]
> swap ["Hello"]
["Hello"]
> swap []
[]
```

Strings are Lists of Characters

```
second :: [a] -> a
second [] = error "The list is too short"
second (_:[]) = error "The list is too short"
second (_:x:_) = x
```

This function takes a list of any type *a*.
a can be *Char*

Strings are Lists of Characters

```
> :t "Hello"
"Hello" :: [Char]
> :t second
second :: [a] -> a
> :t swap
swap :: [a] -> [a]
> second "Hello"
'e'
> swap "Hello"
"eHllo"
```

You'll build a parser in Haskell using pattern matching (no regular expressions)!

Pattern Matching in Functions

Task: Write a function *monthname* that takes in an integer representing a month number and returns the monthname.

```
monthname :: Int -> String
monthname 1 = "January"
monthname 2 = "February"
monthname 3 = "March"
monthname 4 = "April"
monthname 5 = "May"
monthname 6 = "June"
monthname 7 = "July"
monthname 8 = "August"
monthname 9 = "September"
monthname 10 = "October"
monthname 11 = "November"
```

> \> monthname 4
> "April"
> \> monthname 12
> "*** Exception: lecture9.hs:(17,1)-(27,25): **Non-exhaustive** patterns in function monthname

Pattern Matching in Functions

```
monthname ::  Int -> String
monthname 1 = "January"
monthname 2 = "February"
monthname 3 = "March"
monthname 4 = "April"
monthname 5 = "May"
monthname 6 = "June"
monthname 7 = "July"
monthname 8 = "August"
monthname 9 = "September"
monthname 10 = "October"
monthname 11 = "November"
monthname 12 = "December"
```

```
> monthname 12
"December"
> monthname 0
"*** Exception: lecture9.hs:(17,1)-(28,25): Non-exhaustive patterns in function monthname
```

Exhaustive Pattern Matching in Functions

```
monthname :: Int -> String
monthname 1 = "January"
monthname 2 = "February"
monthname 3 = "March"
monthname 4 = "April"
monthname 5 = "May"
monthname 6 = "June"
monthname 7 = "July"
monthname 8 = "August"
monthname 9 = "September"
monthname 10 = "October"
monthname 11 = "November"
monthname 12 = "December"
monthname _ = "Invalid Month"
```

```
> monthname 0
"Invalid Month"
> monthname 54
"Invalid Month"
```

Guards

Guards allow us to define functions based on conditions.

Syntax:

```
f parameters
  | condition 1 = expression1
  | condition 2 = expression2
  | condition 3 = expression3
  …
  | otherwise = default
```

```haskell
letterGrade ::  Int -> String
letterGrade x
  | x >= 90 = "A"
  | x >= 80 = "B"
  | x >= 70 = "C"
  | x >= 60 = "D"
  | otherwise = "F"
```

Case Expressions and Pattern Matching

Syntax:

```
case expression of pattern1 -> result1
                   pattern2 -> result2
                   pattern3 -> result3
                   …
```

Case Expressions and Pattern Matching

Task: Write a function *takeonly* that takes in an integer and a list and returns a list containing the first n elements of the input list.

```
takeonly :: Integer -> [a] -> [a]
takeonly n xs = case (n, xs) of
  (0 ,_)     ->  []
  (_, [])    ->  []
  (n, x:xs')  ->  x : takeonly (n-1) xs'
```

```
> takeonly 3 [2,  3,  4, 5, 6]
[2,3, 4]
> takeonly 3 "Spartan"
"Spa"
> takeonly 0 "Spartan"
""
> takeonly 10 "Spartan"
"Spartan"
```

Case Expressions and Pattern Matching

The *takeonly* function can be rewritten as:

```
takeonly :: Integer -> [a] -> [a]
takeonly  0 xs = []
takeonly n [] = []
takeonly n (x:xs) = x : takeonly (n-1) xs
```

This is notation is just syntactic sugar.
There are cases where we must use case expressions: in the middle of an expression.
We'll see an example with *let* in a few slides.

Bindings with where …

Syntax:

block

- where
  - var1 = expression1
  - var2 = expression

Example:

```
volume :: (Num a) => a -> a -> a -> a
volume width length height = area * height
  where
    area = width * length
```

Bindings with where …

Task: Write a function *lowest* that takes as its argument a list of elements that can be ordered . The function returns the lowest element in the list.

```
lowest :: (Ord a) => [a] -> a
lowest [] = error "The list is empty"
lowest [x] = x
lowest (x:xs)
  | x < lowestRest = x
  | otherwise = lowestRest
  where
    lowestRest = lowest xs
```

let … in … Expressions

Syntax:

let *var1 = expression1*
    var2 = expression 2
    …
 in *expression*

Example 1:

let result = 5 * 3 in result + 10 -- this is an expression, its value is 25

Example 2:

```
volume :: (Num a) => a -> a -> a -> a
volume width length height = let area = width * length
  in area * height
```

let … in … Expressions

Task: Write a function, *index,* that takes an element and a list as arguments. The function returns the index of the first occurrence of the given element in the list. If the element is not in the list, the function returns -1

```
index :: Eq a => a -> [a] -> Int
index _ [] = -1
index y (x:xs)
  | y == x = 0
  | otherwise = let indexRest = index y xs
     in
       case indexRest of
         -1 -> -1
         _  -> 1 + indexRest
```

Base case?

let … in … Expressions

Task: Write a function, *index,* that takes an element and a list as arguments. The function returns the index of the first occurrence of the given element in the list. If the element is not in the list, the function returns -1

```
index :: Eq a => a -> [a] -> Int
index _ [] = -1
index y (x:xs)
  | y == x = 0
  | otherwise = let indexRest = index y xs
     in
       case indexRest of
         -1 -> -1
         _  -> 1 + indexRest
```

```
> index 8 [1, 2, 9, 4]
-1
> index 9 [1, 2, 9, 4]
2
> index 1 [1, 2, 9, 4]
0
> index 'S' "Go Spartans!"
3
```

Built-in Simple Types

- Char : 'A'
- Bool: True, False
- Int: Bounded integers
- Integer: Unbounded integers
- Float: floating point with single precision
- Double: floating point with double precision

More Types

- Lists: [some type]
- Functions: some type -> some other type
- …

Type Classes

- A type class is a family of similar types that provide implementations for some common functionality.

```
> :type (+)
(+) :: Num a => a -> a -> a
```

a is a type variable.

Num a is the context

Type Classes

- Eq: types that support == and != (Char, Bool, Int, Integer, Float, Double, Lists, etc…)
- Ord: types that have an ordering. They support <, <=, etc…)
- Show: types that can be represented as strings.
- Enum: enumerable types, they can be used as a range.
- Num: numeric types, they support addition, multiplication, etc.
- Integral: Int and Integer
- Floating: Float and Double

User Defined Types

Syntax:

data TypeName = Constructor …

Example:

```
data SJSUColor = Blue | Yellow
> color = Blue
> :t color
color :: SJSUColor
> color
```

iCkicker: What do you get?

A. Blue
B. SJSUColor
C. "Blue"
D. Error

User Defined Types

Syntax:
data TypeName = Constructor …

Example:
```
data SJSUColor = Blue | Yellow
> color = Blue
> :t color
color :: SJSUColor
> color
error:
   No instance for (Show SJSUColor)
```

User Defined Types

Syntax:
data TypeName = Constructor … deriving

Example:
data SJSUColor = Blue | Yellow deriving (Show, Eq)

```
> color = Blue
> :t color
color :: SJSUColor
> color
Blue
> anotherColor = Yellow
> color == anotherColor
False
```

Reminders

- Homework 4: install and use Haskell
- Exam 1: September 23
  - Good luck!