# EECS 368 Programming Language Paradigms

David O. Johnson Fall 2022

### Reminders

- Assignment 5 due (today): 11:59 PM, Monday, October 31
  - One of our SIs, Soujanya, has come up with a great way for you to create, test, and submit Assignment 5.
  - Please review the StudentVideoDemo at the following link for more information:
  - https://drive.google.com/drive/folders/1n1R5b3YihQcbCVyQwUjVBD1IM GMdEvBE
  - If you have any question, please contact Soujanya Ambati at: saisoujanyaambati@ku.edu
  - Soujanya will also be grading Assignment 5.
- Guest Lecture (Nick Smith): Wednesday, November 2
  - Attendance Required
  - Submit a written report on the guest lecture in place of the In-Class Problem (see instructions and rubric in Canvas Lectures module)
- Assignment 6 due: 11:59 PM, Monday, November 14

### In-Class Problem Solution

• 27-(10-28) In-Class Problem Solution.pptx

### Haskell Functions

 We write Haskell functions using expressions to do evaluation.

```
isDigit :: Char -> Bool
isDigit c = c >= '0' && c <= '9'
even :: Integer a => a -> Bool
even n = n `mod` 2 == 0
```

### **Conditional Expressions**

As in most programming languages, functions can also be defined using conditional expressions.

```
abs :: Int \rightarrow Int abs n = if n \ge 0 then n = lse -n
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.

### **Conditional Expressions**

Conditional expressions can be nested:

```
signum :: Int \rightarrow Int signum n = if n < 0 then -1 else if n == 0 then 0 else 1
```

#### Note:

In Haskell, conditional expressions must <u>always</u> have an else branch, which avoids any possible ambiguity problems with nested conditionals.

### **Guarded Equations**

As an alternative to conditionals, functions can also be defined using guarded equations.

With conditionals ...

```
abs :: Int \rightarrow Int abs n = if n \ge 0 then n = lse -n
```

With guarded equations ...

```
abs :: Int \rightarrow Int
abs n | n \geq 0 = n
| otherwise = -n
```

A similar programming paradigm in C++ and JavaScript is switch-case and in Python is match-case.

### **Guarded Equations**

Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
signum n | n < 0 = -1
| n == 0 = 0
| otherwise = 1
```

#### Note:

The catch-all condition <u>otherwise</u> is defined in the Prelude by: <u>otherwise</u> = <u>True</u>.

- Many functions have a particularly clear definition using <u>pattern</u> <u>matching</u> on their arguments.
- Pattern matching uses a sequence of patterns to choose between a sequence of results.
- If the first pattern is matched, then the first result is chosen.
- If the second one is matched, then the second result is chosen.
- And so on ...

```
not :: Bool -> Bool
not False = True
not True = False

not maps False to True, and True to False.
```

Functions can often be defined in many different ways using pattern matching. For example

```
(&&) :: Bool -> Bool -> Bool
True && True = True
True && False = False
False && True = False
False && False = False
```

can be defined more compactly by

```
True && True = True
_ && _ = False
```

Note: The underscore symbol \_ is a <u>wildcard</u> pattern that matches any argument value.

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

```
True && b = b
False && _ = False
```

Patterns are matched <u>in order</u>. For example, the following definition always returns False:

Patterns may not <u>repeat</u> variables. For example, the following definition gives an error:

### Pattern Matching Numbers

- Pattern matching works on numbers, too.
- For example, this is a classical recursive definition of the Fibonacci sequence:

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

### Pattern Matching Tuples

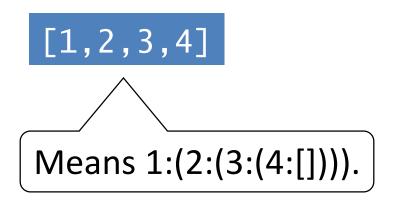
- A tuple of patterns is itself a pattern.
- It matches any tuple of the same arity whose components all match the corresponding patterns in order.
- For example, the library functions fst and snd that respectively select the first and second components of a pair are defined as follows:

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

snd :: (a,b) -> b
snd (_,y) = y
```

### Pattern Matching and Lists

Internally, every non-empty list is constructed by repeated use of an operator (:) called "cons" that adds an element to the start of a list.



### **List Patterns**

Functions on lists can be defined using cons (:) patterns.

```
head :: [a] -> a
head (x:_) = x
tail :: [a] -> [a]
tail (_:xs) = xs
```

head and tail map any non-empty list to its first and remaining elements.

### More on Cons (:) Patterns

Cons (:) patterns only match non-empty lists:

```
> head []
*** Exception: empty list
```

Cons (:) patterns must be <u>parenthesized</u>, because application has priority over (:). For example, the following definition gives an error:

Correct

**Incorrect** 

head 
$$(x:\_) = x$$

head  $x: \underline{\hspace{0.2cm}} = x$ 

### Lambda Expressions

#### **Recall arrow functions from JavaScript:**

Instead of the function keyword, it uses an arrow (=>) made up of an equal sign and a greater-than character.

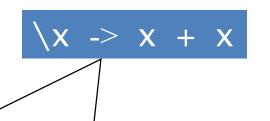
```
const power = (base, exponent) => {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};</pre>
```

- The arrow comes after the list of parameters and is followed by the function's body.
- It expresses something like "this input (the parameters) produces this result (the body)".

Haskell has a similar programming paradigm called lambda expressions.

### Lambda Expressions

Functions can be constructed without naming the functions by using <u>lambda expressions</u>.



the nameless function that takes a number x and returns the result x + x.

- The symbol  $\lambda$  is the Greek letter <u>lambda</u>, and is typed at the keyboard as a backslash \.
- In mathematics, nameless functions are usually denoted using the → symbol, as in x → x + x.
- In Haskell, the use of the  $\lambda$  symbol for nameless functions comes from the <u>lambda</u> <u>calculus</u>, the theory of functions on which Haskell is based.

### Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using <u>currying</u>.

### For example:

```
add :: Int \rightarrow Int \rightarrow Int add x y = x + y
```

#### means

```
add :: Int -> (Int -> Int)
add = \x -> (\y -> \x + \y)
```

### Why Are Lambda's Useful?

Lambda expressions can be used to avoid naming functions that are only <u>referenced once</u>.

### For example:

```
odds n = map f [0..n-1]
where
f x = x*2 + 1
```

can be simplified to

```
odds n = map (\x -> x*2 + 1) [0..n-1]
```

### **Operator Sections**

An operator written <u>between</u> its two arguments can be converted into a curried function written <u>before</u> its two arguments by using parentheses (prefix operator).

#### For example:

### **Operator Sections**

This convention also allows one of the arguments of the operator to be included in the parentheses (postfix operator).

### For example:

In general, if  $\oplus$  is an operator then functions of the form  $(\oplus)$ ,  $(x\oplus)$  and  $(\oplus y)$  are called <u>sections</u>.

### Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:



(1+) - successor function



(1/) - reciprocation function



(\*2) - doubling function

(/2) - halving function

### **In-Class Problem**

- Consider a function safetail that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case.
- Define safetail using:
- 1. A conditional expression
- 2. Guarded equations
- 3. Pattern matching
- Hint: the library function null :: [a] -> Bool can be used to test if a list is empty.