



Government	Percentage
Current government	85%
Previous government	15%

# Programming Paradigms

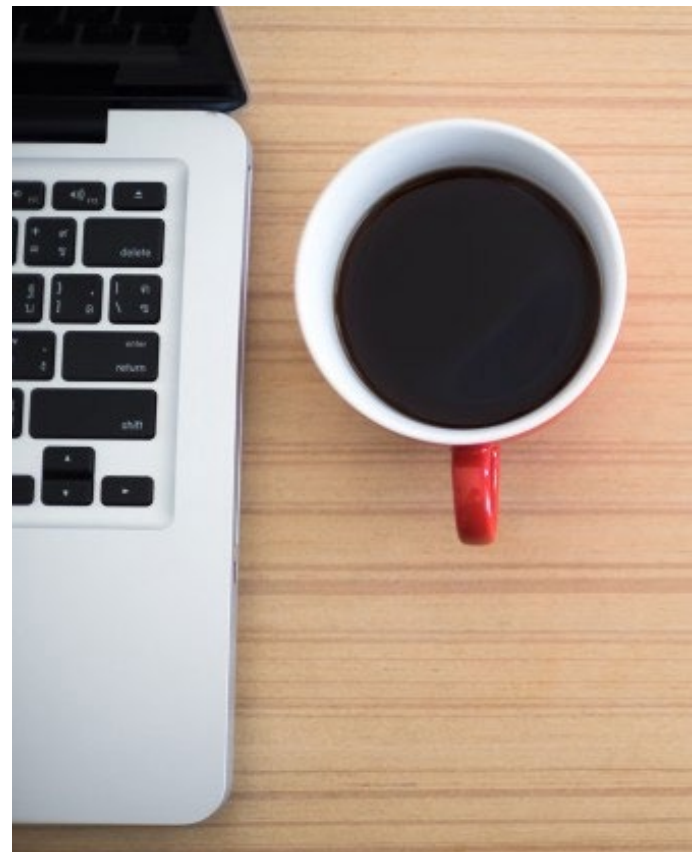
## Functional Programming

# Course Topics

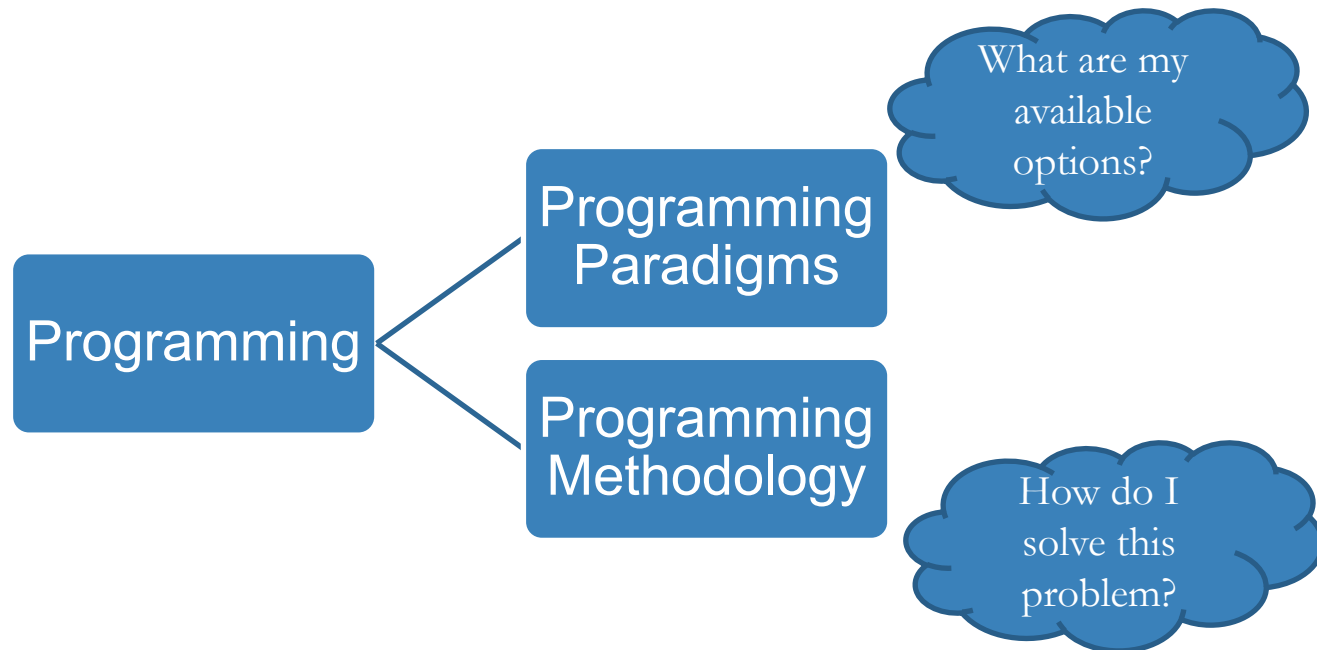
- ~~Introduction~~
- ~~Software Process Models~~
- ~~Requirements Engineering~~
- ~~Modeling~~
- Software Construction Techniques
- Testing
- Project Management
- Refactoring
- Ethical Issues

## Lecture Objectives

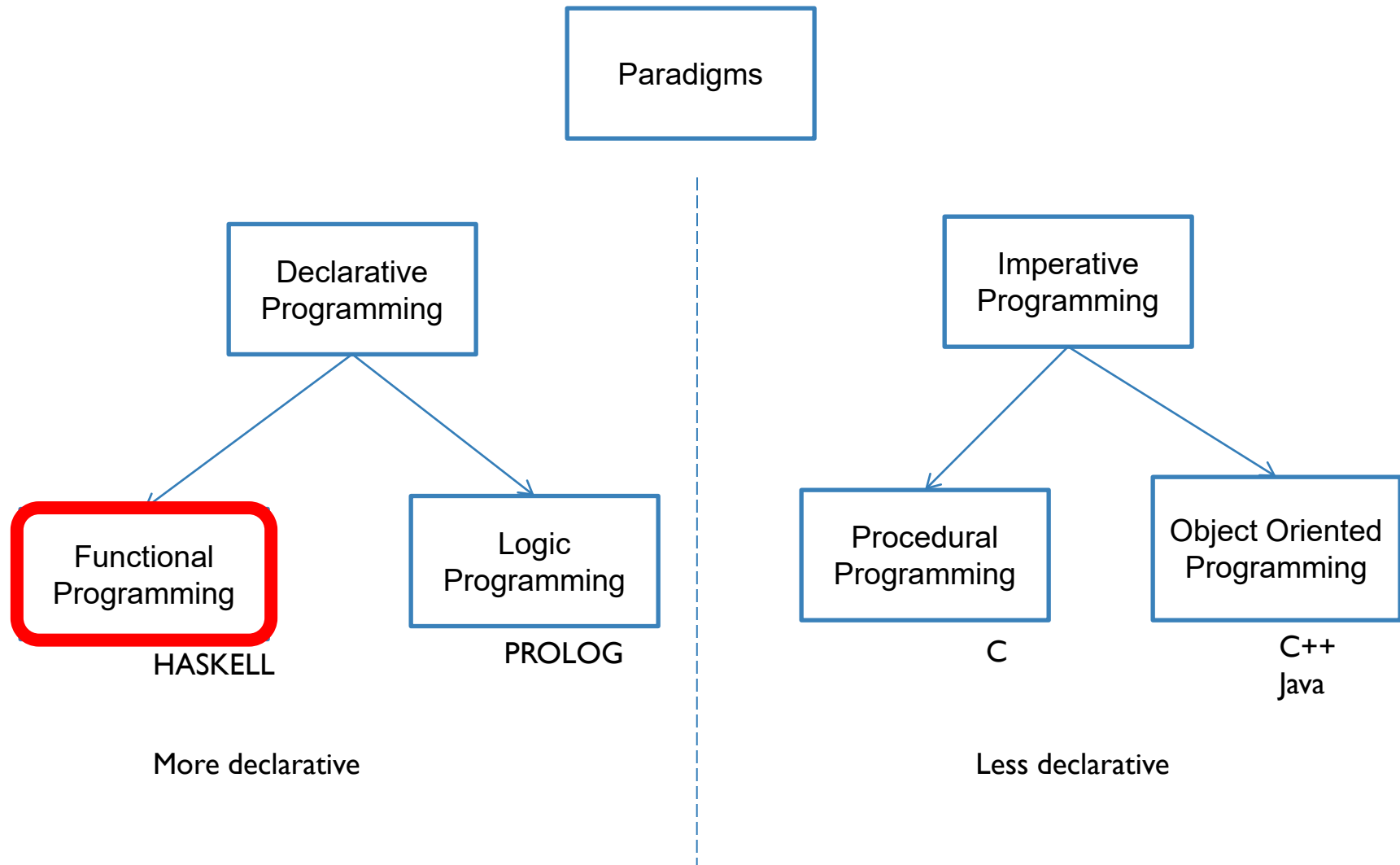
- ✓ To know what is Functional Programming
- ✓ To know about Haskell and Hugs



# Paradigm Vs Methodology



# Programming Paradigms



# What is Functional Programming?



- Functional programming is style of programming in which the basic method of computation is the application of functions to arguments;
- Program viewed as a collection of functions.
- There are no assignments
- Emphasize on simple and clean semantics.
- Examples of languages: Scheme, Miranda, Haskell, ML (meta language)

# Functional Programming Paradigm



- Design of a functional program is based on mathematical functions
- Problem Solving = Evaluation of Functions
- A program consists of function calls with appropriate arguments.
- Based on  $\lambda$ -calculus with added constructs for convenience.

# Why is it Useful?

- ▶ The abstract nature of functional programming leads to *considerably* simpler programs;
- ▶ It also supports a number of powerful new ways to structure and reason about programs.
- ▶ Example:

**Summing the integers 1 to 10 in Haskell:**

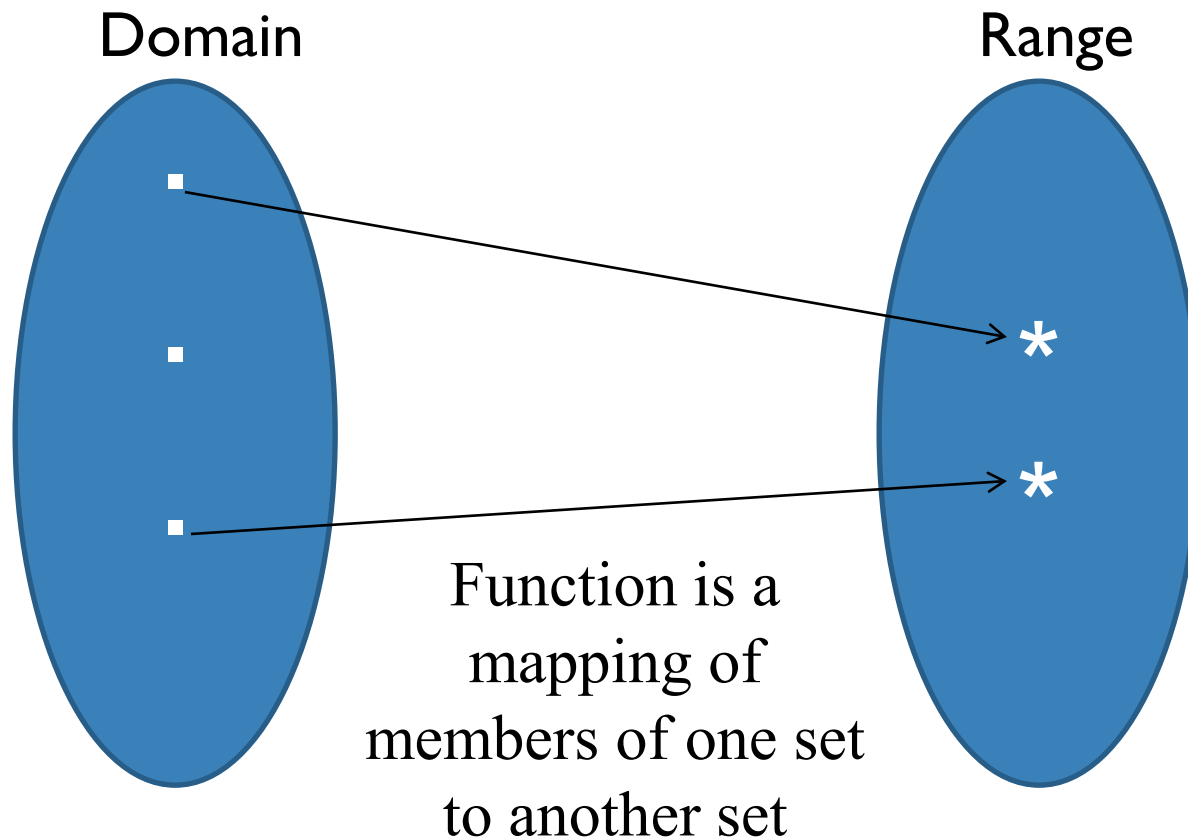
```
sum [1..10]
```

**The computation method is function application.**



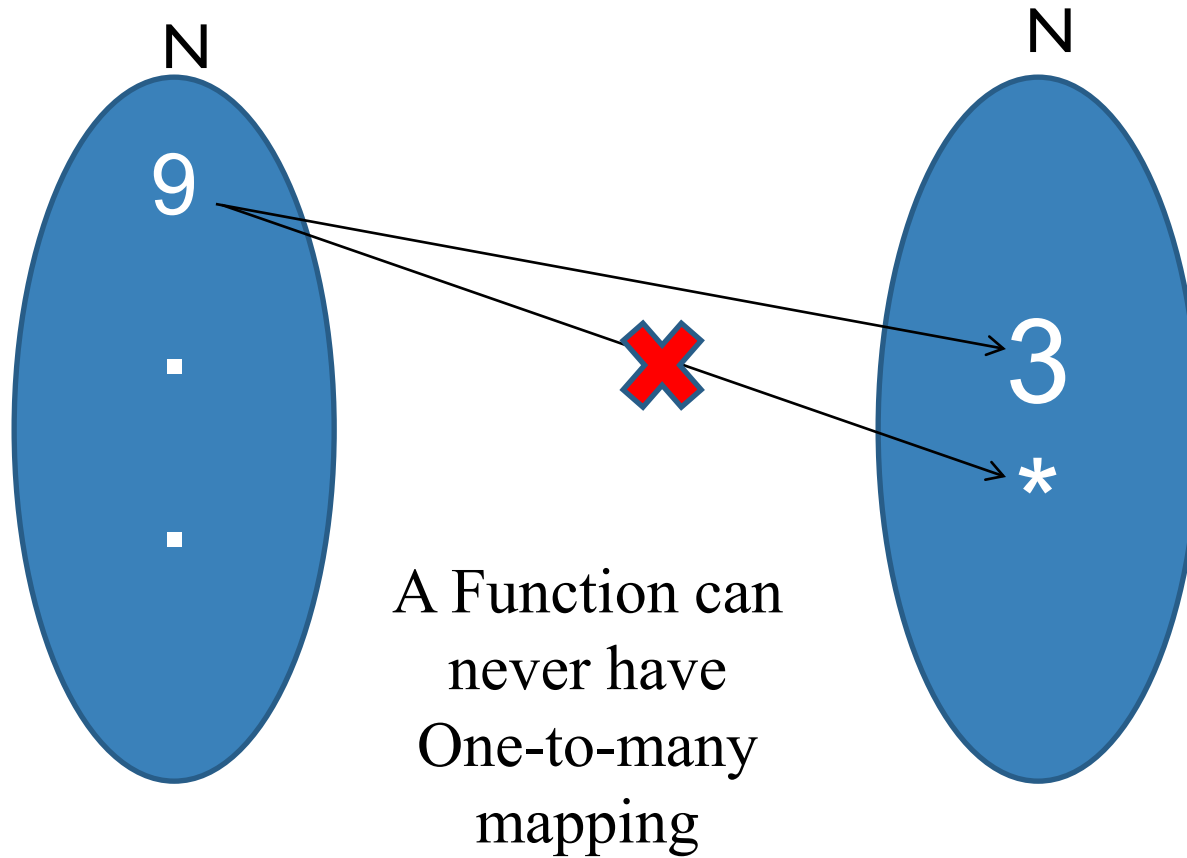
# Functional Programming Paradigm

- What is a Function?



# Functional Programming Paradigm

- Example: If Function is Square-Root



# Applications of Functional Languages



- **LISP** is used for artificial intelligence
  - Knowledge representation
  - Machine learning
  - Natural language processing
  - Modeling of speech and vision
  
- **Scheme** is used to teach introductory programming at a significant number of universities

# Comparing Fun. and Imp. Languages

- Imperative Languages:
  - Efficient execution
  - Complex semantics
  - Complex syntax
  - Concurrency is programmer designed
  
- Functional Languages:
  - Inefficient execution
  - Simple semantics
  - Simple syntax
  - Programs can automatically be made concurrent

# Hugs : Haskell interpreter

- An interpreter for Haskell, and the most widely used implementation of the language;
- An interactive system, which is well-suited for teaching and prototyping purposes;
- Hugs is freely available from:

[www.haskell.org/hugs](http://www.haskell.org/hugs)

# Function Application

Mathematics style

$$f(a,b) + c d$$

Haskell Style

$$f\ a\ b + c*d$$

Functions have higher priority than all other operators.

$$f\ a + b$$

# Examples

Mathematics

$f(x)$

$f(x, y)$

$f(g(x))$

$f(x, g(y))$

$f(x)g(y)$

Haskell

$f \ x$

$f \ x \ y$

$f \ (g \ x)$

$f \ x \ (g \ y)$

$f \ x \ * \ g \ y$

# My First Script

- When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running Hugs.
- Start an editor, type in the following two function definitions, and save the script as test.hs:

```
double x      = x + x
```

```
quadruple x = double (double x)
```



# Example



- Exercise: Write a program to compute the sum of  $N$  numbers, where  $N$  is provided by the User?

# Imperative Solution Vs Functional

```
function sum(n:int) : int;
```

```
{
```

```
  if n == 0;
  sum = 0; then 0
```

```
  for (i = 0; i < n; i++)
```

```
  {
    else
```

```
    sum = sum + i
```

```
  end;
```

```
}
```

```
}
```

No State – so  
remove this

Now how to solve using  
Functional paradigm

In Functional programming  
Always think about value  
And their expected output

No Loops –  
replace with a  
function

**RECURSION**

# Imperative Vs Functional

```
main ()
{
    sum = 0;

    for (i = 0; i < n; i++)
    {
        sum = sum + i
    }
}
```

```
func sum(n:int) : int;
{
    if n = 0
        then 0

    else
        n + sum(n-1)

    end;
}
```

# Example: Factorial

- As we have seen, many functions can naturally be defined in terms of other functions.

```
factorial  :: Int → Int  
factorial n = product [1..n]
```

# Example: Factorial

- Expressions are evaluated by a stepwise process of applying functions to their arguments.

- For example:


$$\begin{aligned} & \text{factorial } 3 \\ &= \text{product } [1..3] \\ &= \text{product } [1, 2, 3] \\ &= 1 * 2 * 3 \\ &= 6 \end{aligned}$$

# Recursive Functions


- In Haskell, functions can also be defined in terms of themselves. Such functions are called recursive.

```
factorial 0 = 1  
factorial n = n * factorial (n-1)
```

# For example:



`factorial 3`  
=  
`3 * factorial 2`  
=  
`3 * (2 * factorial 1)`  
=  
`3 * (2 * (1 * factorial 0))`  
=  
`3 * (2 * (1 * 1))`  
=  
`3 * (2 * 1)`  
=  
`3 * 2`  
=  
`6`



# Quicksort

- The quicksort algorithm for sorting a list of integers can be specified by the following two rules:
- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail values  $\leq$  the head, sorting the tail values  $>$  the head, and then appending the resulting lists on either side of the head value.

Quicksort Algorithm:

<https://www.youtube.com/watch?v=8hHWpuAPBHo>



# Quicksort

- Using recursion, this specification can be translated directly into an implementation:

```
qsort      :: [Int] → [Int]
qsort []   = []
qsort (x:xs) = qsort [a | a ← xs, a ≤ x]
              ++ [x] ++
              qsort [b | b ← xs, b > x]
```

- ++ operator is used to concatenate two arrays/lists.
- This is probably the simplest implementation of quicksort in any programming language!

# Quicksort: How to read each line

```
qsort      :: [Int] → [Int]
```

```
qsort []   = []
```

The result of sorting an empty list (written []) is an empty list

```
qsort (x:xs) = qsort [a | a ← xs, a ≤ x]
               ++ [x] ++
               qsort [b | b ← xs, b > x]
```

To sort a list whose first element is  $x$  and the rest of which is called  $xs$ , just sort all the elements of  $xs$  which are less than  $x$ , sort all the elements of  $xs$  which are greater than  $x$ , and concatenate ( $++$ ) the results, with  $x$  sandwiched in the middle.

```
qsort [a | a ← xs, a ≤ x]
```

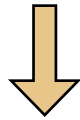
The list of all  $a$ 's such that  $a$  is drawn from the list  $xs$ , and  $a$  is less than  $x$

# For example (abbreviating qsort as q)

q [3, 2, 4, 1, 5]

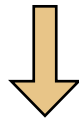


q [2, 1] ++ [3] ++ q [4, 5]



q [1] ++ [2] ++ q []

q [] ++ [4] ++ q [5]



[1]

[2]

[]

[3]

[]

[4]

[5]

# Key Points



- Programming Paradigms
  - Declarative and Imperative
- Functional Programming Paradigm
  - the basic method of computation is the application of functions to arguments
- Functional programming leads to *considerably* simpler programs

# References



- Ian Sommerville, “Software Engineering”, 10<sup>th</sup> Edition, Addison-Wesley, 2015.
  - Timothy C. Lethbridge and Robert Laganière, “Object-Oriented Software Engineering: Practical Software Development using UML and Java”, 2<sup>nd</sup> Edition, McGraw Hill, 2001.
  - R. S. Pressman, Software Engineering: A Practitioner’s Approach, 10th Edition, McGraw-Hill, 2005.
- 