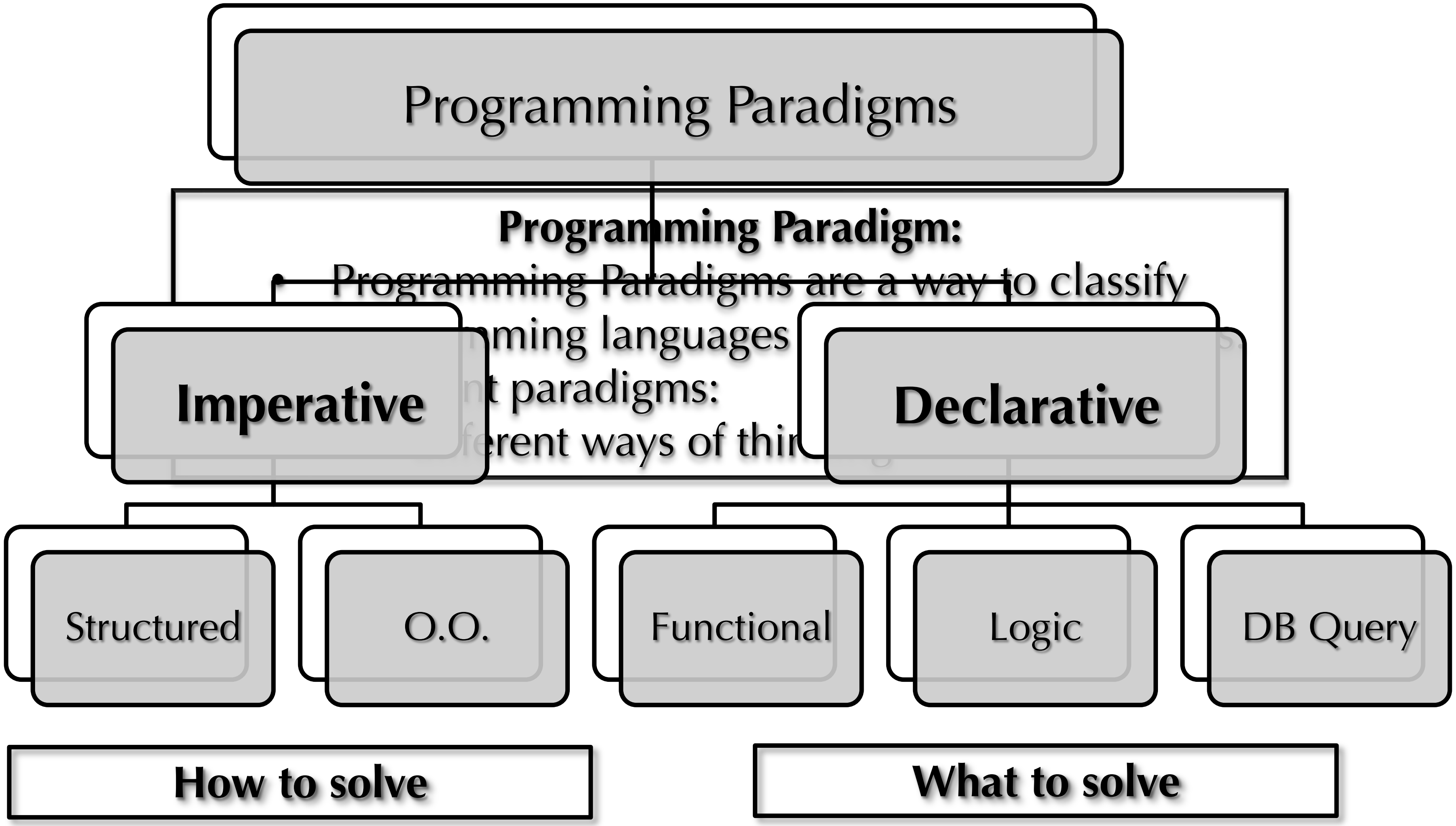
Programming Languages 3

Lect 2: Functional Programming Paradigms

Dr. Islam Gamal

Programming Paradigm:

- Programming Paradigms are a way to classify
  mming languages
  t paradigms:
  ferent ways of thi



Programming Paradigms

Imperative:

- We understand programs by understanding how the computer sequentially executes the program.
- The code describes in exacting detail the steps that the computer must take to accomplish the goal.

Functional:

- Program consists of functions.
- These functions are much like mathematical functions defined by ordinary equations.

Example

How to compute factorial of n?

Math

$$n! = \begin{cases} 1 & n=0 \\ (n-1)! * n & n>0 \end{cases}$$

Imperative (sequential)

```
int fact (int n) {
    int y = 1;
    for (int i=1; i<=n; i++)
            y = y*I;
    return y; }
```

Functional

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

Imperative (recursive)

```
int fact (int n) {
    if  (n == 0)    return 1;
    return  n* fact (n-1);
}
```

Functional Languages Characteristics

- Functional programming is a way of writing software applications using only *pure functions* and *immutable values.*

- There are *no variables or assignments* — Code is like algebra, and in algebra you never reuse variables.

- A pure function has *no side effects*, meaning that it does not read anything from the outside world or write anything to the outside world.

- The output of a *pure function* depends only on
  - its input parameters and
  - its internal algorithm.

- There are *no loops* — only recursive functions.

Referential Transparency

- A function has the property of referential transparency if *its value depends only on the values of its parameters.*
  - *Does f(x)+f(x) equal 2*f(x)? In C? In Haskell?*

- In a pure functional language, all functions are referentially transparent, and therefore *always yield the same result* no matter how often they are called.

```c
#include <stdio.h>

int f(int x) {
        return x * x;
}
int main() {
    int result1 = f(5) + f(5);
    int result2 = 2 * f(5);
    printf("Result1: %d\n", result1);
    printf("Result2: %d\n", result2);
    return 0;
}
```

```haskell
f :: Int -> Int
f x = x * x

main :: IO ()
main = do
        let result1 = f 5 + f 5
                result2 = 2 * f 5

putStrLn $ "Result1: " ++ show result1
putStrLn $ "Result2: " ++ show result2
```

Advantages of Functional Languages

- Allow programs to be written clearly, and at a *high-level of abstraction*.
- Have a *simple syntax.*
- Pure functions and immutable values are easier to *reason about*
- Immutable values make *parallel/concurrent programming easier*

The optimal solution for parallelism and concurrency && Very useful for AI application

Drawbacks of Functional Programming

- Writing pure functions is easy but combining them into a *complete application is hard*.

- *Pure functions and I/O don't really mix* - you write as much of your application's code in an FP style as you can, and then you write a thin I/O layer around the outside of the FP code.

- Using only immutable values and recursion can potentially lead to *performance problems.*

Multi-paradigm Programming Languages

- **Functional Programming Languages:** Lisp, ML, Erlang, Elixir, Haskell.
- **Imperative languages with functional features:** Python, C#, Java
- **Functional languages with object-oriented features:** F#, Scala, Ocalm, Clojure

Multi-paradigm Programming Languages

Lisp is the second-oldest high-level programming language. It quickly became favored artificial intelligence (AI) research. Today, the best-known general-purpose Lisp dialects are Common Lisp and Scheme.

Ideas from **ML** have roots in Lisp, and have influenced numerous other languages, like Haskell. It can be referred to as an *impure* functional language, because although it encourages functional programming, it allows side-effects (like Lisp)

Multi-paradigm Programming Languages

Erlang is a general-purpose, concurrent, functional programming language. It was originally a proprietary language within Ericsson but was released as open source in 1998.

The Erlang runtime system is well suited for systems with the following characteristics: Distributed, Fault-tolerant, Highly available, non-stop applications.

Elixir builds on top of Erlang and shares the same abstractions for building distributed, fault-tolerant applications. Elixir also provides a productive tooling and an extensible design.

Multi-paradigm programming languages

- ***Closure:*** First release in 2007. It runs on the Java virtual machine and as a result integrates with Java and fully supports calling Java code from Clojure, and Clojure code can be called from Java also. It has elegant concurrency support

- ***Scala*:** First release in 2004. *One of the most recent and highly used functional programming language. Scala source code is intended to be compiled to Java bytecode, so that the resulting executable code runs on a JVM. Scala provides language interoperability with Java, so that libraries written in both languages may be referenced directly in Scala or Java code. It is used in some of the Hadoop components like Apache Spark.*

- ***F#:*** is part of VS2010. It is a functional programming language which runs on .NET. It is most often used as a cross-platform Common Language Infrastructure (CLI) language. It can generate JavaScript and graphics processing unit (GPU) code.

Haskell Language

a ***pure functional*** *programming language*

Haskell in Industry

- Many companies have used Haskell for a range of projects, including:
    - *ABN AMRO* is an international bank in Amsterdam that used Haskell to measure the risk for its investment banking activities.
    - *Bluespec,* is a company that use Haskell provides methodologies, and tools for modern integrated circuit design.
    - *Haskore* is a music composition software developed using Haskell
    - Others: ***www.haskell.org/haskellwiki/Haskell_in_industry***

Haskell Compiler

- Haskell has many implementations, two of which are widely used:
  - Hugs: is an interpreter.
  - Glasgow (GHC): has a compiler and interpreter.

- GHC is freely available from: *www.haskell.org/platform*

Hands On

Glasgow Haskell Compiler

- GHC is the leading implementation of Haskell and comprises a compiler and interpreter.

- The interactive nature of the interpreter makes it well suited for teaching and prototyping.

- GHC is freely available from: www.haskell.org/platform

- The interpreter can be started from the terminal command prompt simply typing ***ghci***

The Standard Prelude

- Haskell comes with many standard library functions.
- Standard Prelude is a module available in every language implementation and implicitly imported always into all modules

  EX: ***head [2, 5, 8, 6]***

- The Haskell 2010 Report: Chapters 5, 6 and 9 Described it as core type definitions.

Let's try

- Open your terminal and type in **ghci**

Try:

```
> 3 + 5
> 5 / 2
> product [1..5]
> 5 * -3
```

>>>> Error

```
> 5 * (-3)
> True == False
> True == 1
```

>>>> Error ***Why??***

Let’s try

Try:

```
> ord ‘D’
```

It gives you ***error***

Try:

```
> :module Data.Char
> ord ‘D’
> chr 90
```

It works now. **Why?**

Note the changing in the **prompt**

Haskell Scripts

Haskell Scripts

- Scripting language are usually interpreted, not compiled.
- Sometimes a script can be compiled before others use it. Once compiled:
  - it will run faster
  - doesn't need another application to run it.
  - this prevents end users from changing the code.

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 ***GHCi***.

- 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)
```

- in GHCi window load the new script:

```
> :load test
```

My First Script

Now try:

```
> quadruple 10
> take (double 2) [1,2,3,4,5,6]
> double 2.5
> double 'm'                >>>> Error
> double -2                 >>>> Error
```

Leaving GHCi open, return to the editor, add the following two definitions, and resave:

```
x = 5
fact n = product [1..n]
```

GHCi does not automatically detect that the script has been changed, so a reload command must be executed before the new definitions can be used:

```
> :reload
> fact 10
```

My First Script

Note that:

- Each script has a list of definitions.
- The purpose of a definition is to introduce a binding associating a given name with a given value or expression.

Recommended Textbooks

- ***Haskell - The Craft of Functional Programming***, by Simon Thompson
- ***Introduction to Functional Programming***, Richard Bird Philip Wadler
- ***Real World Haskell***, by Bryan O'Sullivan, John Goerzen, and Don Stewart

Haskell Report

- Haskell Communities and Activities Reports

  https://wiki.haskell.org/Haskell_Communities_and_Activities_Report

- Haskell Communities and Activities Report, Thirty fourth Edition — May 2018

  http://www.haskell.org/communities/05-2018/report.pdf

- Haskell 2010 -Language Report

Course Grading Policy

Mid-term Examination	20 %
Semester Work	10 %
Practical (project)	20 %
Final-Year Examination	50 %

Course Channel

Teams Channel Code: 4x7c4n6

Thank you