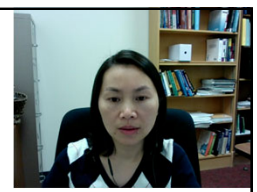
CSCI-C311 Programming Languages

Functional Programming Features

Dr. Hang Dinh

1

Outline and Reading



- After this lecture, you will learn
  - First-class objects and first-class functions
  - Lambda expressions and Lambda Calculus
  - Higher-order functions
- Reading
  - Scott 4e - Section 3.6
  - Scott 4e – Sections 11.6, 11.7
  - Scott 4e – Section C-11.7 (Supplementary section, available on companion website)

2

Classification of Values (or Objects)

- A value (or object) in a programming language can have one of three statuses: *first-class*, *second-class*, or *third-class*.

	Can be passed as a parameter	Can be returned from a subroutine	Can be assigned into a variable
First-class values	✓	✓	✓
Second-class values	✓		
Third-class values	✗		

- A value can have different statuses in different programming languages

3

Classification of Values: Examples

- Values of simple types are first-class in most programming languages.
- Labels (in languages that have them)
  - Usually third-class values
  - Second-class values in Algol.
- Subroutines or functions
  - First-class in all functional languages and most scripting languages.
  - First-class in C#, and with some restrictions, in several other imperative languages (Fortran, Modula-2 and -3, Ada 95, C, and C++)
  - Second-class in most other imperative languages
  - Third-class in Ada 93.

4

First-Class Objects/Values

- Can be used in programs without restriction (when compared to other kinds of objects in the same language).
- Support all the operations generally available to other objects
  - being passed as an argument,
  - being returned from a function,
  - being assigned to (storable in) a variable (in language with side effects)
  - being expressible as an anonymous literal value
  - being constructible at runtime
  - being storable in data structures
  - being comparable for equality with other entities
  - having an intrinsic identity (independent of any given name)

5

First Class Objects: Example and Counter-Example

- Example: strings are first-class in Java (and C++)
- Counter-example: arrays are second-class in C++
- Both can be passed as an argument to a function

```
System.out.println("Hello world");

void init(int a[], int n, int val) {
    for (int i = 0; i < n; i++)
        a[i] = val;
}
```

6

6

First Class Objects: Example & Counter-Example

- Returnable as the result of a function

```
String message() {
    return "Hello";
}
```

```
int[] returnarray() {
    int a[] ={1, 2, 3};
    return a; //error
}//array can't be returned
```

- Storable in variables

```
String text = "Have a nice day.";

int a[] = {1, 2, 3, 4, 5}, b[5];
```

7

First Class Objects: Example & Counter-Example

- Expressible as an *anonymous* literal value (in other expressions)
  `"Have a nice day."` ➔ can be embedded in other expressions
  `{1, 2, 3, 4, 5}` ➔ can only be used in array declarations

```
String text;
text = "Have a nice day.";

int a[]= {1, 2, 3, 4, 5}, b[5];
b = {1, 2, 3, 4, 5}; // error!
```

- Constructible at runtime

```
String s = "Hello world";
int *a = new int[10]; //create dynamic array using pointer
```

8

First Class Objects: Example & Counter-Example

- Storable in data structures (e.g., arrays, structs, classes in Java or C++)

```
String[] words = {"Have", "a", "nice", "day."};
int[3][] array2D = {{1, 2, 3}, {4, 5, 6}};
```

- Comparable for equality with other entities

```
if (text == "Hello again") //compares the references
if (a == b) //compares the references; a and b are arrays
if (a == {1, 2, 3, 4, 5}) … // error!
```

- Having an intrinsic identity (independent of any given name)

H	e	l	l	o	'\0'

9

9

First Class Functions

- A programming language supports first class functions if it allows functions to be first class objects.
- Supported by
  - all functional languages: Lisp, Scheme, ML,…
  - C#
  - many scripting languages: Python, Perl, JavaScript
- Partially supported by other imperative languages
  - Fortran, C, C++, …
  - Many don't support anonymous function definitions

10

10

Lambda Expressions

- Anonymous function definitions also called *lambda expressions*
  - They describe functions by directly describing their behavior.
  - Inspired from *lambda calculus*, a branch of mathematics that studies functions, recursion, computability. Invented by A. Church in 1930.
- In strict sense of the term, first class functions also require
  - lambda expressions that can be embedded in other expressions
  - nested lambda expressions with *unlimited extent* (i.e., keeping them alive even after their scopes are no longer active)
- C++11 and Java 8 provide lambda expressions but without unlimited extent

11

11

Java Lambda Expressions

- In Java, a lambda expression is a short code block that takes parameters
  - similar to method declarations,
  - but without a name and can be implemented right in the body of a method
- Syntax of Java lambda expression: *parameter_list* -> *body*
- Examples:

```
 n ->  System.out.println(n);
(x, y) -> { return x+y; }
(x, y) -> x+y
(int x, int y) -> x+y
```

12

Java Lambda Expressions in Action

- Lambda expressions are usually passed as parameters to a function

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    numbers.forEach( (n) -> { System.out.println(n); } );
  }
}
```

Use a lambda expression in the ArrayList's forEach() method to print every item in the list

Try this code at https://www.w3schools.com/java/java_lambda.asp

13

Java Lambda Expressions in Action

- Java lambda expressions can be stored in variables if
  - the variable's type is an interface which has only one method
  - the lambda expression has the same number of parameters and same return type as the interface's method.

```
interface StringFunction {
  String run(String str);
}

public class Main {
  public static void main(String[] args) {
    StringFunction exclaim = (s) -> s + "!";
    StringFunction ask = (s) -> s + "?";
    System.out.println(exclaim.run("Hello"));//print "Hello!"
    System.out.println(ask.run("Hello"));//print "Hello?"
  }
}
```

14

Lambda Expressions in Scheme/Racket

- Common syntax of lambda expression in Scheme/Racket:

    (lambda (*parameter1 parameter2 …*) *expression*)

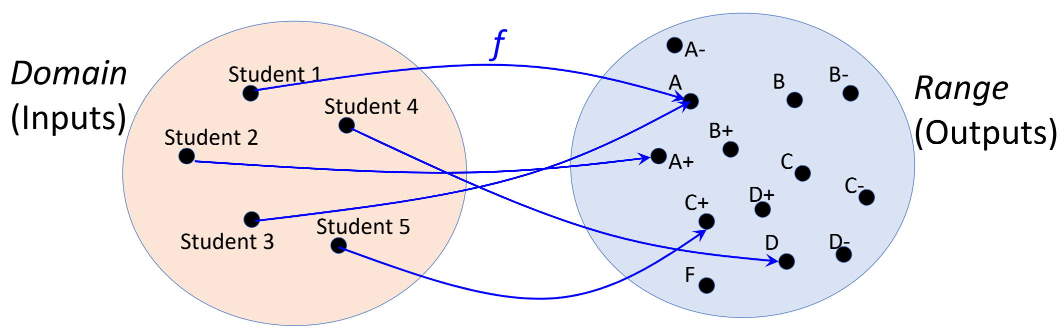
- Example:

```
> (lambda (x y) (+ x y))
#<procedure>
> ((lambda (x y) (+ x y)) 1 2)
3
```

15

Mathematical Foundations of Functions

- A function is a single-valued mapping: it consists of three parts
    - *Domain*: any non-empty set
    - *Range*: any non-empty set
    - An association that maps each element in domain to one element in the range.



16

Mathematical Foundations of Functions

- If f is a function with domain A and range B, we write

$$f: A \to B$$

  - The element in B that is associated with the element x in A is denoted $f(x)$.
  - Function f is also defined as a subset of the cross-product $A \times B$:

$$f = \{(x, y) \in A \times B \mid y = f(x)\}$$

- Example: Function $sqrt$ is defined by one of the following ways:
  - $sqrt: \mathrm{R}^+ \to \mathrm{R}$ $\qquad sqrt(x) = y\ if\ y \geq 0\ and\ y^2 = x$
  - $sqrt = \{(x, y) \in \mathrm{R}^+ \times \mathrm{R} \mid y \geq 0\ and\ y^2 = x\}$

  where R^+ denotes the set of nonnegative numbers, and R denotes the set of all real numbers.

17

Lambda Calculus

- The definition of function as a set is *nonconstructive*:
  - It doesn't tell us how to compute $f(x)$
- Church designed the lambda calculus to address this limitation
  - It also provides a *meta-language* for formal definition of functions
  - Any *computable* function can be written as a lambda expression
- A lambda expression can be defined recursively as
  1. a *name* or a *number*
  2. a lambda *abstraction* consisting of the letter λ, a name, a dot, and a lambda expression
  3. a function *application* consisting of two adjacent lambda expressions (the first one is not a number)
  4. a parenthesized lambda expression.

18

18

Lambda Calculus

- When two expressions appear adjacent to one another: sqrt *n*
  - the first is interpreted as a function to be applied to the second.
  - The application associates left-to-right: f x y means (f x) y rather than f (x y)
  - Application before abstraction: λx.A B means λx.(A B) rather than (λx.A) B
- The letter λ introduces the lambda calculus equivalent of a formal parameter.
- Examples:
  - **λ x. x** describes the identity function $f(x) = x$
  - (**λ x. x**) 7 means the identity function is applied to the constant 7 and is evaluated to 7.
  - **λ x.** 7 describes the constant function $f(x) = 7$

19

Lambda Calculus

- To accommodate arithmetic, we allow lambda expressions of the form

  op x y

  - to denote the arithmetic expression (x op y)
  - where op is the name of one of standard arithmetic functions: plus, minus, times…
- Examples:
  - **λ x. times x x** describes a function $f(x) = x^2$
  - (**λ x. times x x**) 7 means the function $f(x) = x^2$ is applied to the constant 7 and is evaluated to 49.
  - **λ x. λ y. times x y** describes the function with two variables $f(x, y) = xy$

20

Computability

- A function is called *computable* if there exists an *algorithm* to compute it.
- 1930s: Different formalizations of the notion of an algorithm
  - Turing's model of computing was *Turing machine* (learn more in CSCI-B401)
  - Church's model of computing was *lambda calculus*
- *Church-Turing* thesis
  - Intuitive notion of algorithms = Turing machine algorithms
  - A function is computable if and only if it is computed by a Turing machine.

23

23

Higher-Order Functions

- A *higher-order function* (or *functional form*) takes a function as an argument or returns a function as its result
  - Its domain or range is the set of functions from A to B, for some sets A and B.
- A *higher-order language* supports higher-order functions and allows functions to be constituents of data structures.
- Examples of built-in higher-order functions
  - In Scheme/Racket: map, apply
- Higher-order functions are great for building things

24

24

Higher-Order Functions: Common Uses

- One common use is to build new functions from existing ones

```
#lang racket
(define make-double
  (lambda (f) [lambda (x) (f x x)])
)
(define twice (make-double +))
(define square (make-double *))
```

```
> (make-double +)
#<procedure>
> (twice 1)
2
```

25

Higher Order Functions: Common Uses

- **Currying** (named for logician Haskell Curry)
  - replace a multi-argument function with a function that takes a single argument and returns a function that expects the remaining arguments.

```
#lang racket
(define curried-plus
  (lambda (a) [lambda (b) (+ a b)])
)
(define plus3 (curried-plus 3))
```

```
> (curried-plus 3)
#<procedure>
> ((curried-plus 3) 4)
7
> (plus3 4)
7
```

26

Higher-Order Functions in Imperative Languages?

- Why aren't higher-order functions more common in imperative programming languages?
- First, need function *constructor* to create new functions on the fly
  - Function *constructors* are a significant departure from syntax and semantics of traditional imperative languages.
- Second, the ability to specify functions as return values increases the cost of storage management.
  - It requires eliminate function nesting
  - Or requires that we give local variables unlimited extent.

27