LSPU Self-paced Learning Module (SLM)

Course	**Bachelor of Science in Computer Science**
Sem/AY	**First Semester/2023-2024**
Module No.	**2**
Lesson Title	**Functional Programming, Imperative Programming**
Week Duration	**4**
Date	September 25-29, 2023 October 2-6, 2023 October 9-13 2023 October 16-20, 2023
Description of the Lesson	This lesson will discuss the fundamental features and concepts to different programming languages. Topics include overview of programming languages, Introduction to language translation, type systems, data and execution control, declaration and modularity, and syntax and semantics. Laboratory will be used to demonstrate each of the concepts using different programming languages.

Learning Outcomes

Intended Learning Outcomes	Students should be able to meet the following intended learning outcomes: • Classify appropriate functional languages into an appropriate commercial or academic field. • Apply higher-order procedures to an application set. • Explain the characteristics of pure functional functions in functional programming. • Identify the factors and commands that affect the programming state. • Illustrate how execution ordering affects programming. • Analyze how function structure affects programming.
Targets/ Objectives	At the end of the lesson, students should be able to learn, practice and apply the ff: • *The Functional Programming Types* • *Call by Value and Reference* • *Function Overloading and Overriding* • *Recursion* • *Higher Order Functions* • *Imperative Programming and its Pointers*

Student Learning Strategies

Online Activities (Synchronous/ Asynchronous)

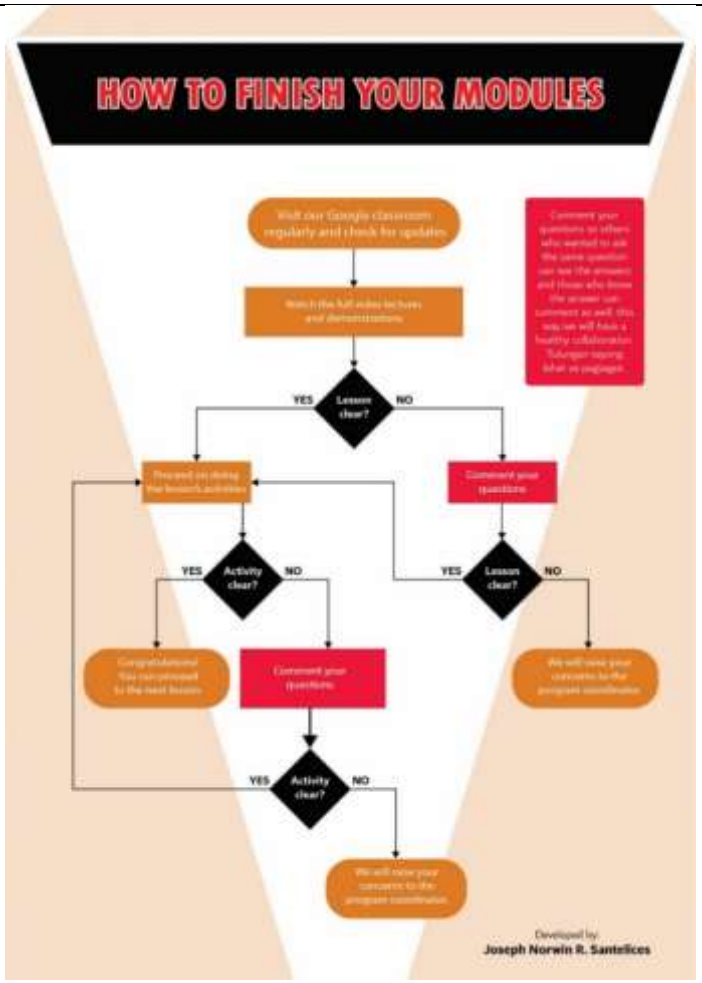
A. Online Discussion via Google Classroom.
You will be directed to join in the online classroom where you will have access to the video lectures, course materials, and evaluation tools 24/7. To have access on the Online Lectures, refer to this link: _____.

To ensure a healthy collaboration, you may post your inquiries on a particular topic and the instructor including all the members of the class can acknowledge the inquiry.

Monday	BSCS 3IS	Lecture	8:00-10:00
Tuesday	BSCS 3GV	Lecture	8:00-10:00
Wednesday	BSCS 3IS	Laboratory	7:00-10:00
Thursday	BSCS 3GV	Laboratory	10:00-1:00

(For further instructions, refer to your Google Classroom and see the schedule of activities for this module)

B. Learning Guide Questions:
Refer to the following diagram as your guide through the course.



Note: *The insight that you will post on online discussion forum using Learning Management System (LMS) will receive additional scores in class participation*

Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.

Functional programming languages are categorized into two groups, i.e. –

- *Pure Functional Languages* – These types of functional languages support only the functional paradigms. For example – Haskell.
- *Impure Functional Languages* – These types of functional languages support the functional paradigms and imperative style programming. For example – LISP.

Offline Activities (e-Learning/Self-Paced)

Functional Programming – Characteristics

The most prominent characteristics of functional programming are as follows –

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports *higher-order functions* and *lazy evaluation* features.
- Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

Functional Programming – Advantages

Functional programming offers the following advantages –

- Bugs-Free Code – Functional programming does not support *state*, so there are no side-effect results and we can write error-free codes.
- Efficient Parallel Programming – Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as

Functional Programming	OOP
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus is on: “What you are doing”	Focus is on “How you are doing”
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side effects.
Flow Control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements.
It uses "Recursion" concept to iterate Collection Data.	It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java
Execution order of statements is not so important.	Execution order of statements is very important.
Supports both "Abstraction over Data" and "Abstraction over Behavior".	Supports only "Abstraction over Data".

- "instructions". Such codes support easy reusability and testability.
- Efficiency – Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.
- Supports Nested Functions – Functional programming supports Nested Functions.
- Lazy Evaluation – Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

As a downside, functional programming requires a large memory space. As it does not have state, you need to create new objects every time to perform actions.

Functional Programming is used in situations where we have to perform lots of different operations on the same set of data.

- Lisp is used for artificial intelligence applications like Machine learning, language processing, Modeling of speech and vision, etc.
- Embedded Lisp interpreters add programmability to some systems like Emacs.

Functional Programming vs. Object-oriented Programming

The following table highlights the major differences between functional programming and object-oriented programming −

Efficiency of a Program Code

The efficiency of a programming code is directly proportional to the algorithmic efficiency and the execution speed. Good efficiency ensures higher performance.

The factors that affect the efficiency of a program includes −

- The speed of the machine
- Compiler speed
- Operating system
- Choosing right Programming language
- The way of data in a program is organized
- Algorithm used to solve the problem

The efficiency of a programming language can be improved by performing the following tasks −

- By removing unnecessary code or the code that goes to redundant processing.
- By making use of optimal memory and nonvolatile storage
- By making the use of reusable components wherever applicable.
- By making the use of error & exception handling at all layers of program.
- By creating programming code that ensures data integrity and consistency.
- By developing the program code that's compliant with the design logic and flow.

An efficient programming code can reduce resource consumption and completion time as much as possible with minimum risk to the operating environment.

****

In programming terms, a **function** is a block of statements that performs a specific task. Functions accept data, process it, and return a result. Functions are written primarily to support the concept of reusability. Once a function is written, it can be called easily, without having to write the same code again and again.

Different functional languages use different syntax to write a function.

Prerequisites to Writing a Function

Before writing a function, a programmer must know the following points –

- Purpose of function should be known to the programmer.
- Algorithm of the function should be known to the programmer.
- Functions data variables & their goal should be known to the programmer.
- Function's data should be known to the programmer that is called by the user.

Flow Control of a Function

When a function is "called", the program "transfers" the control to execute the function and its "flow of control" is as below –

- The program reaches to the statement containing a "function call".
- The first line inside the function is executed.
- All the statements inside the function are executed from top to bottom.
- When the function is executed successfully, the control goes back to the statement where it started from.
- Any data computed and returned by the function is used in place of the function in the original line of code.

Syntax of a Function

The general syntax of a function looks as follows –

```
returnType functionName(type1 argument1, type2 argument2, . . . ) {
  // function body
}
```

Defining a Function in C++

Let's take an example to understand how a function can be defined in C++ which is an object-oriented programming language. The following code has a function that adds two numbers and provides its result as the output.

```
#include <stdio.h>

int addNum(int a, int b);    // function prototype

int main() {
  int sum;
  sum = addNum(5,6);       // function call
```

```
  printf("sum = %d",sum);
  return 0;
}
int addNum (int a,int b) {    // function definition
  int result;
  result = a + b;
  return result;          // return statement
}
```

It will produce the following output –

Sum = 11

Defining a Function in Erlang

Let's see how the same function can be defined in Erlang, which is a functional programming language.

```
-module(helloworld).
-export([add/2,start/0]).

add(A,B) ->
  C = A + B,
  io:fwrite("~w~n",[C]).
start() ->
  add(5,6).
```

It will produce the following output –

11

Function Prototype

A function prototype is a declaration of the function that includes return-type, function-name & arguments-list. It is similar to function definition without function-body.

For Example – Some programming languages supports function prototyping & some are not.

In C++, we can make function prototype of function 'sum' like this –

int sum(int a, int b)

Note – Programming languages like Python, Erlang, etc doesn't supports function prototyping, we need to declare the complete function.

What is the use of function prototype?

The function prototype is used by the compiler when the function is called. Compiler uses it to ensure correct return-type, proper arguments list are passed-in, & their return-type is correct.

Function Signature

A function signature is similar to function prototype in which number of parameters, datatype of parameters & order of appearance should be in similar order. For Example –

```
void Sum(int a, int b, int c);       // function 1
void Sum(float a, float b, float c);  // function 2
void Sum(float a, float b, float c);  // function 3
```

Function1 and Function2 have different signatures. Function2 and Function3 have same signatures.

Note – Function overloading and Function overriding which we will discuss in the subsequent chapters are based on the concept of function signatures.

- Function overloading is possible when a class has multiple functions with the same name but different signatures.
- Function overriding is possible when a derived class function has the same name and signature as its base class.

****

Functions are of two types –

- Predefined functions
- User-defined functions

In this chapter, we will discuss in detail about functions.

Predefined Functions

These are the functions that are built into Language to perform operations & are stored in the Standard Function Library.

For Example – 'Strcat' in C++ & 'concat' in Haskell are used to append the two strings, 'strlen' in C++ & 'len' in Python are used to calculate the string length.

Program to print string length in C++

The following program shows how you can print the length of a string using C++ –

```
#include <iostream>
#include <string.h>
#include <stdio.h>
using namespace std;

int main() {
  char str[20] = "Hello World";
```

```
  int len;
  len = strlen(str);
  cout<<"String length is: "<<len;
  return 0;
}
```

It will produce the following output –

String length is: 11

Program to print string length in Python

The following program shows how to print the length of a string using Python, which is a functional programming language –

```
str = "Hello World";
print("String length is: ", len(str))
```

It will produce the following output –

('String length is: ', 11)

User-defined Functions

User-defined functions are defined by the user to perform specific tasks. There are four different patterns to define a function –

- Functions with no argument and no return value
- Functions with no argument but a return value
- Functions with argument but no return value
- Functions with argument and a return value

Functions with no argument and no return value

The following program shows how to define a function with no argument and no return value in **C++** –

```
#include <iostream>
using namespace std;

void function1() {
  cout <<"Hello World";
}
int main() {
  function1();
  return 0;
}
```

It will produce the following output –

Hello World

The following program shows how you can define a similar function (no argument and no return value) in **Python** –

```
def function1():
  print ("Hello World")

function1()
```

It will produce the following output –

Hello World

Functions with no argument but a return value

The following program shows how to define a function with no argument but a return value in **C++** –

```
#include <iostream>
using namespace std;
string function1() {
  return("Hello World");
}

int main() {
  cout<<function1();
  return 0;
}
```

It will produce the following output –

Hello World

The following program shows how you can define a similar function (with no argument but a return value) in **Python** –

```
def function1():
  return "Hello World"
res = function1()
print(res)
```

It will produce the following output –

Hello World

Functions with argument but no return value

The following program shows how to define a function with argument but no return value in **C++** –

```
#include <iostream>
using namespace std;
void function1(int x, int y) {
  int c;
  c = x+y;
  cout<<"Sum is: "<<c;
}

int main() {
  function1(4,5);
  return 0;
}
```

It will produce the following output –

```
Sum is: 9
```

The following program shows how you can define a similar function in **Python** –

```
def function1(x,y):
  c = x + y
  print("Sum is:",c)
function1(4,5)
```

It will produce the following output –

```
('Sum is:', 9)
```

Functions with argument and a return value

The following program shows how to define a function in C++ with no argument but a return value –

```
#include <iostream>
using namespace std;
int function1(int x, int y) {
  int c;
  c = x + y;
  return c;
}

int main() {
  int res;
  res = function1(4,5);
  cout<<"Sum is: "<<res;
  return 0;
}
```

It will produce the following output –

```
Sum is: 9
```

The following program shows how to define a similar function (with argument and a return value) in **Python** –

```
def function1(x,y):
  c = x + y
  return c

res = function1(4,5)
print("Sum is ",res)
```

It will produce the following output –

```
('Sum is ', 9)
```

****

After defining a function, we need pass arguments into it to get desired output. Most programming languages support call by value and call by reference methods for passing arguments into functions.

In this chapter, we will learn "call by value" works in an object-oriented programming language like C++ and a functional programming language like Python.

In Call by Value method, the original value cannot be changed. When we pass an argument to a function, it is stored locally by the function parameter in stack memory. Hence, the values are changed inside the function only and it will not have an effect outside the function.

Call by Value in C++

The following program shows how Call by Value works in C++ –

```
#include <iostream>
using namespace std;

void swap(int a, int b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
  cout<<"\n"<<"value of a inside the function: "<<a;
  cout<<"\n"<<"value of b inside the function: "<<b;
}
int main() {
  int a = 50, b = 70;
  cout<<"value of a before sending to function: "<<a;
  cout<<"\n"<<"value of b before sending to function: "<<b;
  swap(a, b);  // passing value to function
  cout<<"\n"<<"value of a after sending to function: "<<a;
  cout<<"\n"<<"value of b after sending to function: "<<b;
```

```
  return 0;
}
```

It will produce the following output –

```
value of a before sending to function:  50
value of b before sending to function:  70
value of a inside the function:  70
value of b inside the function:  50
value of a after sending to function:  50
value of b after sending to function:  70
```

Call by Value in Python

The following program shows how Call by Value works in Python –

```
def swap(a,b):
  t = a;
  a = b;
  b = t;
  print "value of a inside the function: :",a
  print "value of b inside the function: ",b

# Now we can call the swap function
a = 50
b = 75
print "value of a before sending to function: ",a
print "value of b before sending to function: ",b
swap(a,b)
print "value of a after sending to function: ", a
print "value of b after sending to function: ",b
```

It will produce the following output –

```
value of a before sending to function:  50
value of b before sending to function:  75
value of a inside the function: : 75
value of b inside the function:  50
value of a after sending to function:  50
value of b after sending to function:  75
```

****

In Call by Reference, the original value is changed because we pass reference address of arguments. The actual and formal arguments share the same address space, so any change of value inside the function is reflected inside as well as outside the function.

Call by Reference in C++

The following program shows how Call by Value works in C++ –

```
#include <iostream>
using namespace std;

void swap(int *a, int *b) {
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
  cout<<"\n"<<"value of a inside the function: "<<*a;
  cout<<"\n"<<"value of b inside the function: "<<*b;
}
int main() {
  int a = 50, b = 75;
  cout<<"\n"<<"value of a before sending to function: "<<a;
  cout<<"\n"<<"value of b before sending to function: "<<b;
  swap(&a, &b);  // passing value to function
  cout<<"\n"<<"value of a after sending to function: "<<a;
  cout<<"\n"<<"value of b after sending to function: "<<b;
  return 0;
}
```

It will produce the following output –

```
value of a before sending to function:  50
value of b before sending to function:  75
value of a inside the function:  75
value of b inside the function: 50
value of a after sending to function:  75
value of b after sending to function:  50
```

Call by Reference in Python

The following program shows how Call by Value works in Python –

```
def swap(a,b):
  t = a;
  a = b;
  b = t;
  print "value of a inside the function: :",a
  print "value of b inside the function: ",b
  return(a,b)

# Now we can call swap function
a = 50
b =75
print "value of a before sending to function: ",a
```

```
print "value of b before sending to function: ",b
x = swap(a,b)
print "value of a after sending to function: ", x[0]
print "value of b after sending to function: ",x[1]
```

It will produce the following output –

```
value of a before sending to function:  50
value of b before sending to function:  75
value of a inside the function:  75
value of b inside the function:  50
value of a after sending to function:  75
value of b after sending to function:  50
```

****

When we have multiple functions with the same name but different parameters, then they are said to be overloaded. This technique is used to enhance the readability of the program.

There are two ways to overload a function, i.e. –

- Having different number of arguments
- Having different argument types

Function overloading is normally done when we have to perform one single operation with different number or types of arguments.

Function Overloading in C++

The following example shows how function overloading is done in C++, which is an object oriented programming language –

```
#include <iostream>
using namespace std;
void addnum(int,int);
void addnum(int,int,int);

int main() {
  addnum (5,5);
  addnum (5,2,8);
  return 0;
}

void addnum (int x, int y) {
  cout<<"Integer number: "<<x+y<<endl;
}

void addnum (int x, int y, int z) {
```

```
  cout<<"Float number: "<<x+y+z<<endl;
}
```

It will produce the following output –

```
Integer number: 10
Float number: 15
```

Function Overloading in Erlang

The following example shows how to perform function overloading in Erlang, which is a functional programming language –

```
-module(helloworld).
-export([addnum/2,addnum/3,start/0]).

addnum(X,Y) ->
  Z = X+Y,
  io:fwrite("~w~n",[Z]).

addnum(X,Y,Z) ->
  A = X+Y+Z,
  io:fwrite("~w~n",[A]).

start() ->
  addnum(5,5),    addnum(5,2,8).
```

It will produce the following output –

```
10
15
```

****

When the base class and derived class have member functions with exactly the same name, same return-type, and same arguments list, then it is said to be function overriding.

Function Overriding using C++

The following example shows how function overriding is done in C++, which is an objectoriented programming language –

```
#include <iostream>
using namespace std;

class A {
  public:
  void display() {
```

```
    cout<<"Base class";
  }
};

class B:public A {
  public:
  void display() {
    cout<<"Derived Class";
  }
};

int main() {
  B obj;
  obj.display();
  return 0;
}
```

It will produce the following output

Derived Class

Function Overriding using Python

The following example shows how to perform function overriding in Python, which is a functional programming language –

```
class A(object):
  def disp(self):
    print "Base Class"
class B(A):
  def disp(self):
    print "Derived Class"
x = A()
y = B()
x.disp()
y.disp()
```

It will produce the following output –

Base Class
Derived Class

****

A function that calls itself is known as a recursive function and this technique is known as recursion. A recursion instruction continues until another instruction prevents it.

Recursion in C++

The following example shows how recursion works in C++, which is an object-oriented programming language –

```
#include <stdio.h>
long int fact(int n);

int main() {
  int n;
  printf("Enter a positive integer: ");
  scanf("%d", &n);
  printf("Factorial of %d = %ld", n, fact(n));
  return 0;
}
long int fact(int n) {
  if (n >= 1)
    return n*fact(n-1);
  else
    return 1;
}
```

It will produce the following output

```
Enter a positive integer: 5
Factorial of 5 = 120
```

Recursion in Python

The following example shows how recursion works in Python, which is a functional programming language –

```
def fact(n):
  if n == 1:
    return n
  else:
    return n* fact (n-1)

# accepts input from user
num = int(input("Enter a number: "))
# check whether number is positive or not

if num < 0:
  print("Sorry, factorial does not exist for negative numbers")
else:
  print("The factorial of " + str(num) +  " is " + str(fact(num)))
```

It will produce the following output –

```
Enter a number: 6
```

The factorial of 6 is 720

****

A higher order function (HOF) is a function that follows at least one of the following conditions –

- Takes on or more functions as argument
- Returns a function as its result

HOF in PHP

The following example shows how to write a higher order function in PHP, which is an object-oriented programming language –

```
<?php
$twice = function($f, $v) {
  return $f($f($v));
};

$f = function($v) {
  return $v + 3;
};

echo($twice($f, 7));
```

It will produce the following output –

13

HOF in Python

The following example shows how to write a higher order function in Python, which is an object-oriented programming language –

```
def twice(function):
  return lambda x: function(function(x))
def f(x):
  return x + 3
g = twice(f)
print g(7)
```

It will produce the following output –

13

LESSON 2

Introduction to Imperative Languages

Functional Languages

- All languages we have studies so far were variants of lambda Calculus
- Such languages are known as functional languages
- We have also seen that these languages allow us to design powerful type systems
- And even perform type inference

Salient Features of Functional Languages

I The functional languages we studied have a set of defining features:
I Most noticeable feature: No side effects!
I This means that evaluating an expression never changes the value of any other expression
I Example:
let x = 3+4 in let y = x+5 in x+y
I Here, evaluating the expression x+5 cannot change the value of any other expression

No Side Effects

I No side effects means no assignments and no variables!
I Recall: Let-bindings are only names for values
I The value they stand for can never change
I Example:
let x = 3 in let x = 4 in x

Impact of No Side Effects

I Question: How can we exploit the fact that evaluating expressions never changes the value of any other expression?
I Answers:
I We can evaluate expressions in parallel
I We can delay evaluation until a value is actually used
I Question: What kind of side effect can evaluating expressions still have?
I Answer: They may still trigger a run-time error

I Unfortunately, run-time errors negate all the benefits we listed!
I Question: What can we do about this?
I Solution: Type systems
I Any sound type system will guarantee no run-time errors
I Conclusion: We can only fully take advantage of functional features if we use a sound type system

The Alternative to Functional Programming

I However, there is also an alternative (and much more common) way of programming called imperative programming
I Features of imperative programming:
I Side effects
I Assignments that change the values of variables
I Programs are sequences of statements instead of one expression
I Imperative programming is the dominant model
I This style is much closer to the way hardware executes

Imperative Programming Languages

I You have all used imperative programming languages
I Imperative Languages:
I FORTRAN
I ALGOL
I C, C++
I Java
I Python
Etc

Features of Imperative Languages

I At a minimum, a language must have the following features to be considered imperative:
I Variables and assignments
I Loops and Conditionals and/or goto
I Observe that features such as pointers, recursion and arrays are optional
I For example, FORTRAN originally only had integers and floats, loops, conditionals and goto statements
Thomas

Example Compare and Contrast

I Let's look at some example imperative programs
I I will use C style since most of you should be familiar with this
I Adding all numbers from 1 to 10 in L:

```
fun add with n =
if n == 0 then 0 else n + (add (n-1)) in (n 10)
```

I Here is the same program in C:

```
int res = 0, i;
for(i=0; i < 10; i++) res += i;
return res;
```

I Question: Which style do you prefer?

Very basic imperative programming

I Now, let's get even more basic and only use conditionals and goto statements to write the same program:

```
int res = 0, i;
again:
res +=i;
i++;
if(i<10) goto again;
return res;
```

l Which style do you prefer?

GOTOs in Programming

l All early imperative languages include goto statements
l Rational: 1) Hardware supports only compare and jump instructions 2) GOTOs allow for more expressive control flow
l Example of GOTO use:

```
int i = 0;
int sum;
again:
i++;
int z = get_input();
if(z < 0) goto error:
n+=z;
if(i < 5) goto again:
return n;
error:
return -1;
```

l Not so long ago, it was universally accepted that GOTO statements are necessary for expressive programs
l However, as software became larger, GOTO statements started becoming problematic
l Central Problem of GOTO: "Spagetti Code"
l This means that thread of execution is very hard to follow in program text
l Jumps to a label could come from almost anyplace (in extreme cases even from other functions!)

l In much early (and also more recent) code, GOTO not only implemented loops but was also used for code reuse
l Real Comment from numerical analyst: "Why bother writing a function if I can just jump to the label?"
l In 1968, Dijkstra wrote a very influential essay called"GOTO Statement Considered Harmful" in which he argued that GOTO statements facilitate unreadable code and should be removed from programming languages

The End of GOTO

l At first, this article was very controversial

I But over time, most programmers started to agree that GOTO constructs should be avoided
I Imperative programming without GOTOs is known as structural programming
I But not everyone was on board...

Side Trip: GOTO and COBOL

I COBOL stands for COmmon Business Oriented Language
I In addition to GOTO, COBOL also includes the ALTER keyword
I After executing ALTER X TO PROCEED TO Y, any future GOTO X means GOTO Y instead
I Can change control flow structures at runtime!
I This was marketed as allowing polymorphism

Side Trip: GOTO and COBOL

I Dijkstra's comment: "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense."

Structured Programming

I Today there is a consensus that GOTOs are not a good idea
I Instead, imperative languages include many kinds of loops and branching constructs
I Examples in C++: while, do-while, for, if, switch
I One legitimate use of GOTO: Error-handling code
I This popularized exceptions in most modern languages

A Simple Imperative Language

I Let's start by looking at at a very basic imperative language we will call IMP1:
P ! _ | S1; S2
S ! if(C) then S1 else S2 fi | id = e
| while(C) do S od
e ! id | e1 + e2 | e1 − e2 | int
C ! e1 _ e2 | e1 = e2 | not C | C1 and C2
I This language has variables, declarations, conditionals and loops
I But no pointers, functions, ...
I What are some example programs in IMP1?

Semantics of IMP1

I Let's try to give operational semantics for this language
I First, we will again use an environment E to map variables to their values
I Start with the semantics of expressions
I Question: What do expressions evaluate to?
I Answer: Integers
I Therefore, the result (value after colon) in operational semantics rules for expression is an integer

Semantics of IMP1

I Here are operational semantics for expressions in IMP1 (first cut)
I Variable:
E ` v : E(id)
I Plus
E ` e1 : v1
E ` e2 : v2
E ` e1 + e2 : v1 + v2
I Minus
E ` e1 : v1
E ` e2 : v2
E ` e1 − e2 : v1 − v2

I On to the semantics of Predicates:
I Question: What do predicates evaluate to?
I Answer: True and False
I Therefore, the result (value after colon) in operation semantics rules for predicates is a Boolean

I Here are operational semantics for predicates in IMP1
I Less than or equal to:

E ` e1 : v1
E ` e2 : v2
v1 _ v2
E ` e1 _ e2 : True
E ` e1 : v1
E ` e2 : v2
v1 6_ v2
E ` e1 _ e2 : False
I Or (slightly imprecise) shorthand
E ` e1 : v1
E ` e2 : v2
E ` e1 _ e2 : v1 _ v2
I What about the other predicates?

Semantics of Statements

I Now, all we have left are the statements
I However, there is one big problem: Statements do not evaluate to anything!
I Instead, statements update the values of variables
I In other words, they change E!
I Therefore, the rules for statements will produce a new environment
I Specifically, they are of the form E ` S : E0
I Changing the environment is the technical way of having side effects in the language

I Let's start with the sequencing statement S1; S2:
E ` S1 : E1
E1 ` S2 : E2
E ` S1; S2 : E2
I Observe here that S1 produces a new environment E1
I We then use this new environment to evaluate S2 and return E2

Basic Statements

I Here is the assignment statement
E ` e : v
E0 = E[id v]
E ` id = e : E0
I Observe that it is possible that id already had a value in E
I In this case, this rule overrides the value of id with the current Value

Semantics of the Conditional

I Here are operational semantics of the conditional
E ` C : true

E ` S1 : E0
E ` if(C) then S1 else S2 fi : E0
E ` C : false
E ` S2 : E0
E ` if(C) then S1 else S2 fi : E0
I Observe that there are two different proof rules used.
I Expressions and conditionals return values, while statements return environments

Semantics of the While loop

I Let's finish with semantics for the last statement: While loop
I This is tricky because the loop may execute any number of times
I Let's start with the base case where the predicate is false:
E ` C : false
E ` while(C) do S od : E

I Now, what about the case where the condition is true?
I In this case, we want to:
I Execute one iteration of the loop, producing a new environment E0
I Repeat the evaluation of the loop with E'
I Here is the rule to do just that:
E ` C : true
E ` S : E0
E0 ` while(C) do S od : E00
E ` while(C) do S od : E00

E ` C : true
E ` S : E0
E0 ` while(C) do S od : E00
E ` while(C) do S od : E00
I Question: How does this rule make progress?
I Answer: It uses the new environment E0 when reevaluating the loop body
I Is it possible that this rule does not terminate? Yes, if the loop is non-terminating

Putting it all together

I We saw how to give operational semantics for a simple imperative language
I Key difference: Side effects
I Side effects are encoded in operational semantics by producing a new environment
I Also observe that for imperative languages, all expressions always evaluate to concrete values

Pointers

I In the language IMP1, perhaps the biggest missing feature is pointers
I A pointer is a reference to a memory location
I Pointers are naturally supported by hardware through load and store instructions
I In fact, pretty much all code turns into pointer manipulation at the assembly level

Why Pointers?

I What are pointers good for?
I Call-by-reference in a call-by-value language
I Clever and efficient data structures
I Avoid copying of data if it can be shared
I It is not uncommon for pointers to to be 100x faster than copying data!
I For this reason, pointers are essential for most performance-critical task.

A Simple Pointer Language

I Let us consider the following simple language with pointers we will call IMP2:
P ! " | P1;P1 | S
S ! if(C) then S1 else s2 fi | id = e | _ id = e
| while(C) do S od | id = alloc
e ! id | e1 + e2 | e1 − e2 | int | _ id
C ! e1 _ e2 | e1 = e2 | notC | C1 and C2
I This is the same as IMP1, just with a load and store operation
I Here, I am using C syntax for loading and storing
I Addition: Alloc allocates fresh memory

Operational Semantics with Pointers

I We want to give operational semantics to this language
I But how can we handle pointers?
I Recall: So far, we only had a environment.
I The environment mapped variables to values
I But how can we look up the value of a pointer?

Operational Semantics with Pointers Cont.

I Idea: Add one level of indirection in the environment.
I We used to have one environment that maps variables to values
I Now, we will have:
I An environment E mapping variables to addresses

I A store S mapping addresses to values stored at this address
I The store is emulating memory when executing a program!

The Store

I This means that our operational semantics will now be of the form
. . .
E, S ` . . .
I Specifically, expression rules will be of the form:
. . .
E, S ` e : v
I Conditional rules are of the form:
. . .
E, S ` e : bool
I And statement rules are of the form:
. . .
E, S ` e : E0, S0
I Statements now both change the environment and the store!

The Store in Action

I Let start with expressions and take a look at the rule for id
I Recall, in IMP1 the operational semantics for id just returned E(id)
I Now, let's write the same rule for IMP2:
l1 = E(id)
v = S(l1)
E, S ` id : v

The alloc Statement

I Intended semantics of alloc: Return a fresh address in S that is not used by anyone else
I Here are the operational semantics of alloc:
lf fresh
S0 = S[lf 0]
S00 = S0[E(v) lf]
E, S ` id = alloc : E, S00

Load in IMP2

I Next: The load expression
I What do we have to do to load a value?
I Look up the address l1 of the variable in E
I Look up the value of l1 in S as v1
I Look up the value of v1 in S
I Here is the rule for load:
l1 = E(id)
v1 = S(l1)

v2 = S(v1)
E, S ` _id : v2

Store in IMP2

I Next: The store statement _id = e
I What do we have to do to store a value?
I Look up the address l1 of the variable v in E
I Look up the value of l1 in S as l2
I Change the value of l2 in S to e's value
I Here is the rule for store:
E, S ` e : v
l1 = E(id)
l2 = S(l1)
S0 = S[l2 v]
E, S ` _id = e : E, S0

Storage for Variables

I So far, we have been sloppy about the storage associated with variables
I Specifically, we have assumed that every variable can be looked up in E
I But this is clearly not the case unless some rule adds them to E!
I Question: How can we solve this problem?
I Solution 1: Two cases for each rule where we use a variables
I One case if the variable is already in E
I One case if variable is not yet in E
I Solution 2: Add variable declarations to our language
I Specifically, add a declare id statement
I Semantics of declare id:
lf fresh
E0 = E[id lf]
E, S ` declare id : S,E0
I This is the solution preferred by most imperative languages

Aliasing

I As soon as we allow pointers, we also allow aliasing
I Two pointers alias if they point to the same memory location
I Here is a simple example program:

```
declare x, y;
x = alloc;
y = x;
*x = 3;
*y = 4;
```

I What is the value of *x?
I In one sense, aliasing is great.

I In fact, many the cases where pointers are really useful involve some kind of aliasing
I However, in another sense, aliasing is awful
I Because of aliasing, storing a value into any location can potentially change every other location's value!
I This is very bad news for any kind of expressive type system

Run-time errors

I Question: What kind of new run-time errors can happen in IMP2?
I Run-time errors everywhere!
I This is another typical "side effect" of adding pointers to a Language

Even More Features

I Another popular feature of imperative languages: arrays
I Array is nothing but a list of values
I Indexed by position
I Corresponds to a contiguous region of memory
I Popular because fast
I Accessing an element only requires adding to the base pointer
I Can perform in-place updates of values

Arrays

I Important: What is called an array in Python is not what we are talking about here!
I Python arrays are lists of values
I These lists can even contain elements of different type
I We are talking about the C/Java style array here

Array Language

I Consider the following modified language we will call IMP3:
P ! " | P1;P1 | S
S ! if(C) then S1 else s2 fi | id = e | id[e1] = e2
| while(C) do S od | id = alloc | declare id
e ! id | e1 + e2 | e1 − e2 | int | id[e]
C ! e1 _ e2 | e1 = e2 | not C | C1 and C2
I Observe that load and store are replaces with array load and store) pointer arrays are a generalization of pointers
I Also, assume that alloc allocates arrays of infinite size

Semantics of IMP3

I The only new statements are array load and array store
I They replace load and store from IMP2
I Question: How can we emulate pointer load and store in IMP3?

l Answer: Pointer load id = *e is the same as id = e[0]
l Pointer store *id = e is the same as id[0] = e

On to the Operational Semantics

l Fortunately, the only change from IMP2 are the array load and store
l Therefore, we only need to write two new rules
l First order of business: Array load

Load in IMP3

l What do we have to do to load a value in an array?
l Specifically, how do we process id[e]?
l Evaluate e to vi
l Look up the address l1 of the variable id in E
l Look up the value of l1 in S as v1
l Add the index vi to v1 as v2
l Look up the value of l2 in S

Performance Tasks

CMSC308 Programming Languages

MODULE 2

Functional and Imperative Programming

Activity 2: User define function

Write a program in C# Sharp to create a user define function.

Pictorial Presentation:

Sample Output

```
See, how to create an user define function :
------------------------------------------------
Welcome Friends!
Have a nice day!
```

Instruction:

File name format should be:

CMSC308m2_lastname_firstname_year&section

You can submit it anytime before the end of the semester but take note, i can see the date of your submission and any of your activities in our group on my end. Please be proactive. Keep safe and good luck.

Rubric:
Hands-On Activity rubric

Understanding Directed Assess

Basic Adobe Photoshop Hands-on Activity

The activity is about proficiency with the functions and application in Programming Languages.

Score	COMPLIANCE *Project Guidelines and Compliance*	KNOWLEDGE *Ability to use required tools*	PRESENTATION *Demonstration of objective or competency in presentation*	SKILL *Uses a number of elements and principles*	
Excellent **90-100**	Project guidelines followed completely and all the required elements are present	Shows ability to use a variety of tools expertly and effectively -- at an excellent level.	The objective or competency was executed with thorough and precise judgments concluding in a carefully crafted product, presentation, or behavior.	Student used a skillful number of elements and principles to present their information.	
Proficient **80-89**	Project guidelines are mostly complete and all required elements are present.	Shows ability to use essential tools capably and effectively -- at a proficient level.	The objective or competency was executed with discernable and Correct judgments concluding in a proficiently crafted product, presentation, or behavior.	Student used an effective number of elements and principles to present their information.	
Adequate **70-79**	There is a missing important project requirement, or a guideline not followed	Shows ability to use basic tools competently and effectively -- at a satisfactory level.	The objective or competency was executed with apparent and acceptable judgments concluding in a satisfactorily crafted product, presentation, or behavior.	Student used a minimal number of elements and principles to present their information.	
Limited **60-69**	There are missing 2 or more important project requirements, or project guidelines not followed	Did not demonstrate ability to use basic tools competently and effectively at a satisfactory level.	The objective or competency was executed with superficial and vague Judgments concluding in an indistinctly crafted product, presentation, or behavior.	Student used an Inadequate number of elements and principles to present their information .	
Final Score					

This rubric came from ***http://barkerwchs.weebly.com/rubrics.html*** and was modified to suit the needs of the course.

Learning Resources

- Khan, Aslam. Grokking Functional Programming . Manning Publications(2015). p. 475. ISBN 9781617291838(https://web.archive.org/web/20160331201242/http://www.manning.com/khan)
- https://wiki.haskell.org/index.php?title=Functional_programming&oldid=59163
- https://www.tutorialspoint.com/functional_programming/index.htm
  Paul Chiusano and Runar Bjarnason
  Foreword by Martin Odersky/September 2014/ ISBN 9781617290657
- . **Programming in Haskell** /Graham Hutton/Cambridge University Press /
  aperback: ISBN 978-1316626221; Kindle: /SIN B01JGMEA3U
- **Functional Programming for the Object-Oriented Programmer" by Brian Marick**