Theory of Programming Languages

Week 1

Introduction

Course Outline

Instructor

Sarim Baig (sarim.baig@nu.edu.pk)

Office Hours: Mon-Wed (1200 – 1330)

Course Group

Web

http://groups.google.com/group/tplsp12

Local

\\xeon\Spring 2012\Sarim Baig\tpl

Course Outline

Detailed Course Outline

Week 1 Introduction
Reading Chapters 1 Sebesta, and Scott

- ***Why should we study programming languages?***
- ***What is the spectrum of programming languages?***
- How do we evaluate programming languages?
- What factors influences Language design?

(I) Why should we study programming languages?

Following maybe some of the reasons

- *Increased capacity to express ideas*

  PHP: explode(), implode(), split()

  C++?
- *Improved background for choosing appropriate languages*

  What language will you use to parse complicated text file generated by a DNA processing machine?

Perl code for file-reading and output

```
#!/usr/bin/perl -w
require 5.004;
## Open each command line file and print its contents to standard out
foreach $fname (@ARGV) {
   open(FILE, $fname) || die("Could not open $fname\n");
   while($line = <FILE>) {
   print $line;
  }
 close(FILE);
}
```

Python string functions

Perl string functions

C++ string functions

- *Increased ability to learn new languages*
- *Increased ability of design new languages*
- *Advancement of computing*

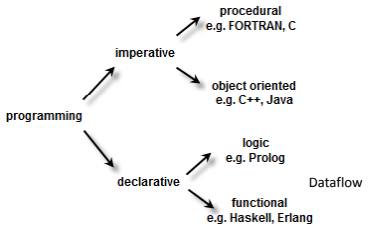
(II) The programming language spectrum

Languages may be classified based on the model of computation

Top Level Division of languages

Declarative Languages: *What the computer has to do…*

Imperative Languages: *How the computer should do what it is supposed to do…*



Functional languages employ a computational model based on the recursive definition of functions. They take their inspiration from the lambda calculus, a formal computational model developed by Alonzo Church in the 1930s. In essence, a program is considered a function from inputs to outputs, defined in terms of simpler functions through a process of refinement. Languages in this category include Lisp, ML, and Haskell.

Fibonacci-Number Programs in C++ and Haskell

Machine's point of view

```
// Fibonacci numbers, imperative style
int fibonacci(int iterations)
{
    int first = 0, second = 1; // seed values

    for (int i = 0; i < iterations; ++i) {
        int sum = first + second;
        first = second;
        second = sum;
    }

    return first;
}

std::cout << fibonacci(10) << "\n";
```

A functional version (in Haskell) has a different feel to it

Programmers point of view

```
-- Fibonacci numbers, functional style

-- describe an infinite list based on the recurrence relation for Fibonacci numbers
fibRecurrence first second = first : fibRecurrence second (first + second)

-- describe fibonacci list as fibRecurrence with initial values 0 and 1
fibonacci = fibRecurrence 0 1

-- describe action to print the 10th element of the fibonacci list
main = print (fibonacci !! 10)
```

Logic or constraint-based languages take their inspiration from predicate logic. They model computation as an attempt to find values that satisfy certain specified relationships, using a goal-directed a search through a list of logical rules. Prolog is the best-known logic language. The term can also be applied to the programmable aspects of spreadsheet systems such as Excel, VisiCalc, or Lotus 1-2-3.

Typical Problem for a logic program

1) There are 5 colored houses in a row, each having an owner, which has an animal, a favorite cigarette, a favorite drink.
2) The English lives in the red house.
3) The Spanish has a dog.
4) They drink coffee in the green house.
5) The Ukrainian drinks tea.
6) The green house is next to the white house.
7) The Winston smoker has a serpent.
8) In the yellow house they smoke Kool.
9) In the middle house they drink milk.
10) The Norwegian lives in the first house from the left.
11) The Chesterfield smoker lives near the man with the fox.
12) In the house near the house with the horse they smoke Kool.
13) The Lucky Strike smoker drinks juice.
14) The Japanese smokes Kent.
15) The Norwegian lives near the blue house.

Who has a zebra and who drinks water?

Dataflow languages model computation as the flow of information (tokens) among primitive functional nodes. They provide an inherently parallel model: nodes are triggered by the arrival of input tokens, and can operate concurrently. Id and Val are examples of dataflow languages. Sisal, a descendant of Val, is more often described as a functional language.

- The spectrum of Programming Languages is scattered around **Programming Domains**
  - Business Applications
    e.g. COBOL (**CO**mmon **B**usiness-**O**riented **L**anguage)

```
IF SALARY > 8000 OR SUPERVISOR-SALARY OR = PROV-SALARY
    ADD YEARS TO AGE
```

    JAVA, C#, Objective C etc. etc.

  - Scientific Applications
    e.g. MATLAB, R

- Systems Programming
  PL/S (IBM computers, 60's and 70's)
  BLISS (Digital computers, just above assembly)
  ALGOL (Burroughs Computers, 60's and 70's)
  C (UNIX)

- Scripting Languages
  - Evolved over the last 30 years
  - A scripting language is used by putting a list of commands, called a script, in a file to be executed. The first of these languages, named sh (for shell), began as a small collection of commands that were interpreted as calls to system subprograms that performed utility functions, such as file management and simple file filtering. To this basis were added variables, control flow statements, functions, and various other capabilities, and the result is a complete programming language.
  - Web Programming has given great rise to the popularity of scripting languages: PHP, JavaScript, being examples.

Week 1 Introduction
Reading Chapters 1 Sebesta, and Scott

- Why should we study programming languages?
- What is the spectrum of programming languages?
- ***How do we evaluate programming languages?***
- What factors influences Language design?

(III) Language Evaluation Criteria

- There are no fixed criteria to evaluate programming languages – but there are some essential features that impact software design.
- **Characteristics** of a programming language influence the **criteria** that impact software design.

Characteristics impact Criteria impact Software Design

	Criteria		
Characteristic	*Readability*	*Writability*	*Reliability*
Simplicity/orthogonality	•	•	•
Control structures	•	•	•
Data types & structures	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Readability – What does it mean, and what affects it?

Readability is the ease with which programs can be read and understood in a particular programming language.

Readability is important because it facilitates **maintenance** – which is crucial as software grows and evolves.

How does one measure readability and what characteristics influence it?

Which of the following loops is more readable?

```
PERFORM VARYING WS-BOTTLE-NUM FROM 98 BY -1
              UNTIL WS-BOTTLE-NUM < 2
END-PERFORM
```

COBOL

Pretty close to an English sentence. Compare that to this sample for a C-style language (C#)

```
for (WSBOTTLENUM = 98; WSBOTTLENUM >= 2; WSBOTTLENUM--)
{
}
```

Which one is more obvious? Pretend you are not an experienced programmer.

BASIC (Visual Basic in this case) is somewhere in the middle.

```
For WSBOTTLENUM = 98 To 1 Step -1
```

- Succinctness and readability are (usually) inversely proportional?
  - Simple division in C++ and COBOL
    C++: k = i / 10
    COBOL: divide i by 10 giving k
  - Simple addition in C++ and COBOL
    C++: k = i + 10
    COBOL: add i to 10 giving k

- **Overall simplicity** of a language enhances readability.
  - A language with *large number of basic components* results in less-readable code as various programmers might use different subsets of the components. A diversity in the basic components, such as, data structures, memory structures and control structures make the codes more complicated and less uniform – e.g. **if or switch? while or do-while?**
  - *Feature multiplicity*: having more than one way to accomplish an operation, reduces multiplicity. For example, there are four ways to increment a simple variable in C:

```
count=count+1
count+=1
count++
++count
```

- *Operator overloading* makes code less readable as various objects use the same operator for different tasks. (But does it not make the code more readable too?)
  - » Readability becomes an issue, particularly, when the overloaded operators are not mathematically standard.

    Set s(3,4,5);

    s++

- **Orthogonality** in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language. Furthermore, every possible combination of primitives is legal and meaningful.
  - For example: the following statement has different meanings:

    a=a+1 ; //a is an integer variable

    a=a+1; //a is an integer pointer (character pointer, double pointer?)

- Following is an example of orthogonality in VAX as compared to IBM Mainframe:

  In IBM main frame has two different instructions for adding the contents of a register to a memory cell or another register.

  A Reg1, memory_cell

  AR Reg1, Reg2

  In contrast, VAX has only one statement for addition (hence greater orthogonality):

  ADDL operand1, operand2

- **How does Orthogonality affect readability?**

Here are some examples of non-orthogonality in C:

- C has two kinds of built-in data structures, arrays and records (structs). Records can be returned from functions, but arrays cannot.
- A member of a struct can have any type except void or a structure of the same type.
- An array element can be any data type except void or a function.
- Parameters are passed by value, unless they are arrays, in which case they are passed by reference.

Reading

Chapter on Orthogonality from "The Art of Unix Programming"

- Orthogonality and Readability
  - Orthogonality is closely related to simplicity: The more orthogonal the design of a language, the fewer exceptions the language rules require, Fewer exceptions mean a higher degree of regularity in the design, which makes the language easier to learn, read, and understand. (Example on non-orthogonality from English: DO is doo, Go is not goo)
  - Too much Orthogonality can be problematic. Following is an example from one of the most orthogonal language, ALGOL, in which assignment and arithmetic primitives can also be used with control and logical structures, as everything returns a value:

    if((if x<y then a else b fi):=

    (b+ if a>b then c:=c+1 else c:=c-1 fi)

) fi then …

- Control Statements affect readability

The structured programming revolution of the 1970s was in part a reaction to the poor readability caused by the limited control statements of some of the languages of the 1950s and 1960s. In particular, it became widely recognized that indiscriminate use of **goto** statements severely reduces program readability.

Following constraints were suggested on the use of goto statements

- They must precede their targets, except when used to form loops.
- Their targets must never be too distant
- Their numbers must be limited.

Similarly: features like **break, continue** reduce readability

- Data Types, Structures and readability examples
  - **true** and **false** for Boolean types

    sum_is_too_big = 1

    sum_is_too_big = true
  - features likes **enum** and macros in C++
  - structs for collecting data to one place, instead of using several arrays. For example, to store employee data.

- Readability and Syntax Considerations

- **Identifier forms: flexible composition.**
  - The ***length*** of the name of a variable vs. the ***meaning*** of a variable
  - Identifier length cannot exceed 6 in FORTRAN 77

- **Special words and methods of forming compound statements**
  Readability is influenced by **the forms of a language's special words**.
  { }, begin/end, end if/end loop
  Block style: begin end in Pascal
  { } in C++
  end if, end loop in Ada

- **Form and meaning: self-descriptive constructs, meaningful keywords**
  Designing statements so that their appearance at least partially **indicates their purpose** is an obvious aid to readability.
  - Example: the **static and const** keywords in C++

Some uses of **static** in C++

```
class user
{
  private:
  int id;
  static int next_id;

  public:
  static int next_user_id()
  {
    next_id++;
    return next_id;
  }
  /* More stuff for the class user */
  user()
  {
    id = user::next_id++; //or, id = user.next_user_id();
  }
};
int user::next_id = 0;
```

```
for(int x=0; x<10; x++)
{
  for(int y=0; y<10; y++)
  {
    static int number_of_times = 0;
    number_of_times++;
  }
}
```

```
class Clock{

public:
    static void showTime(){};
};
```

```
static class Clock{

public:
    void showTime(){};
} clk;
```

- Some uses of **const** in C++

  – Variable pointer to constant data
  const int* ptr;
  int const * ptr;

  – Constant pointer to variable data
  int * const ptr;

  – Constant pointer to constant data
  const int * const ptr;
  int const * const ptr;

Writability – What does it mean, and what affects it?

Writability is a measure of how easily a language can be used to create programs for a chosen problem domain.

While assessing writability it is important to keep in mind the domain for which a language is intended.

- Simplicity and Orthogonality

  If a language has a large number of different constructs, some programmers may not be familiar with all of them. This can lead to a misuse of some features and a disuse of others that may be either more elegant or more efficient, or both, than those that are used.

Example – Which would you use as a C++ programmer? See: Obfuscated Code

```
if(x==y){
    cout<<"Hello";
}else
{
    cout<<"Hello";
}
```

```
x==y?cout<<"Hello": cout<<"World!";
```

- Support for Abstraction
  – Programming languages can support two kinds of abstraction
    - Process abstraction
      – For example: A **sort(…)** function abstracts **findMind(…) and swap(…)** functions.
    - Data abstraction
      – In the following example: an actual animal is abstracted by the class Animal, which is abstracted by the class LivingThing. Advantages?

```
public class Animal extends LivingThing
{
    private Location loc;
    private double energyReserves;

    public boolean isHungry() {
        return energyReserves < 2.5;
    }
    public void eat(Food f) {
        // Consume food
        energyReserves += f.getCalories();
    }
    public void moveTo(Location l) {
        // Move to new location
        loc = l;
    }
}
```

```
thePig = new Animal();
theCow = new Animal();
if (thePig.isHungry()) {
    thePig.eat(tableScraps);
}
if (theCow.isHungry()) {
    theCow.eat(grass);
}
theCow.moveTo(theBarn);
```

- How abstraction facilitates writability?
  - Example: How will you implement the Factory design pattern in C++?

```
class Computer
{
public:
    virtual void Run() = 0;
    virtual void Stop() = 0;
};
class Laptop: public Computer
{
public:
    virtual void Run(){mHibernating = false;};
    virtual void Stop(){mHibernating = true;};
private:
    bool mHibernating; // Whether or not the machine is hibernating
};
class Desktop: public Computer
{
public:
    virtual void Run(){mOn = true;};
    virtual void Stop(){mOn = false;};
private:
    bool mOn; // Whether or not the machine has been turned on
};

class ComputerFactory
{
public:
    static Computer *NewComputer(const std::string &description)
    {
        if(description == "laptop")
            return new Laptop;
        if(description == "desktop")
            return new Desktop;
        return NULL;
    }
};
```

- **Expressivity** in a programming language can refer to several different characteristics
  - Commonly it means that a language has a relatively convenient, rather than a cumbersome way of specifying a particular computation.

For example

- Do we really need a **for** loop?
- Do we really need a **do-while** loop?
- Do we really need a **switch** statement?
- Do we really need a **++** operator?
- Do we really need a **[]** operator in the C++ STL vector?

Reliability – What does it mean, and what affects it?

- A program is said to be **reliable** if it performs to its specifications under all conditions.
- Language features directly affect reliability – and though we would like to increase reliability at all costs, there are tradeoffs.

- **Type Checking**
  - Maybe compile-time or run-time (expensive): See example here
  - Type checking is desirable for reliability
  - The earlier errors in programs are detected, the less expensive it is to make the required repairs.

- Exception Handling

The ability of a program to intercept run-time errors (as well as other unusual conditions detected by the program), take corrective measures, and then continue is a great aid to reliability.

- Readability and Writability
  - Both readability and writability influence reliability.
  - A program written in a language that does not support natural ways to express the required algorithms will necessarily use unnatural approaches.
    - **Example? Find a case in C++ and JAVA – where an algorithm has a more natural expression in one and less so in the other. Argue for you choice.**

Week 2 Introduction (Continued)
Reading Chapters 1 Sebesta, and Scott

- Why should we study programming languages?
- What is the spectrum of programming languages?
- How do we evaluate programming languages?
- ***What factors influences Language design?***