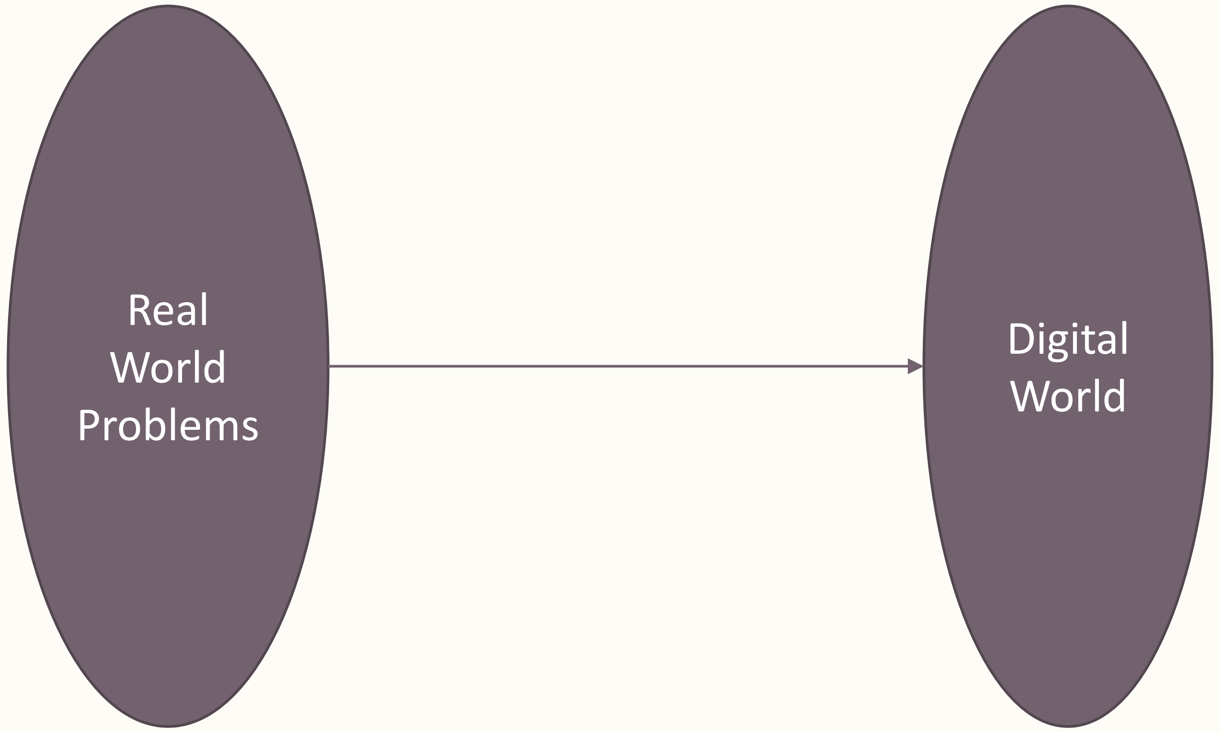
Programming Languages Principles

Introduction

Programming Language

A notation for describing computation in machine-readable and human-readable form



What is a programming language?

Machine-readability

1. Language must have simple enough structure for efficient translation
2. There must be an algorithm to translate the language. The algorithm must be unambiguous and finite
   The algorithm cannot be too complex. Otherwise it would take more time to translate the program than to execute it.
3. The programming language is restricted to something called a Context-Free language (i.e. no interdependencies in it)

What is a programming language?

Human-readability

1. The language must provide abstractions of the actions the computer must take.
2. These abstractions must be fairly easy to understand by humans.
3. The language must keep humans as far as possible from the internal workings of the computer.
4. Readability also involves the ease with which large programs can be written. If we need to fix a bug, how easy is it to find, and how easy is it to fix? (localization of data and functions)
5. Also a language must make it easy for teams of programmers to program and communicate with each other (Software Development and Software Engineering)

Abstractions in programming languages

Basically two types

- Data abstractions
    - Simplify the behavior and the attributes of the data for the programmers
- Control abstractions
    - Simplified view of the modification of execution path
- Abstractions also have levels
    - Basic –collects the most localized machine information
    - Structured – collects intermediate information about the structure of a program
    - Unit – collects large scale information in a program

Data Abstractions

Basic Abstractions

- Abstraction of the internal data representation of common data
- Simple example is the data variable and the data type
    - int my_age;
- Integer data values are often stored in 2’s complement representation
- Floating point data values-IEEE single precision and double precision representation
- Basic abstraction is atomic

Structured Abstractions

```
struct {
        char name[20];
        int age;
        float salary;
    } employee;
```

```
int a[10]; in Java
INTEGER A(10) (in FORTRAN)
typedef int IntArray (in C)
```

- Abstraction of a collection of data (a “data structure”)
- Data structure is an abstraction that hides the component parts allowing the programmer to view them as ONE thing
- Group operations, such as, sorting and searching are possible
- Group of items- a simple array of integers, or a record of data
- Text files
- This abstraction is not atomic
- Allows the programmers to build such an abstraction
- Allows to access and modify the elements

Unit Abstractions

- Abstraction of a collection of data PLUS related code
- A good example of this would be ADT- a “class” in Java
- Uses “data encapsulation” and “information hiding”
- Unit abstractions should be developed so that they are “reusable” in other programs-package in java
- Unit abstractions can be used to form a “library” that others can use when programming.
- Must also have standard interface definitions to improve the “interoperability” between programming modules.
- Application programming interface, docs generated by the programs about its structure

Control Abstractions

Basic Abstractions

- Combining machine instructions into a more understandable form.
- The “assignment statement” e.g. x = x + 3;
- Syntactic sugar
  - A mechanism that allows the programmer to replace a complex notation with a simpler shorthand notation
  - X+=3

```
List<String>
WithNos=Stream.of("a","1ab","1A", "2A")
                             .filter(d2-
>Character.isDigit(d2.charAt(0)))
      .collect(toList());
```

```
 if (x>0)
{       r1=sqrt(x);
        r2= - r1;
}else
{       numSolns = 0;
        r1 = r2 = 0;
}
```

Structured Abstractions

```
Iterator<String> iter=
exampleList.iterator();
while(iter.hasNext())
     System.out.println(iter.next());
```

- Divide the program into sets of instructions:
- Branch instructions-if-else, switch-case statements
- Iterator and loops
- Procedure or subroutine, example-Ada
  - Declaration, invocation or activation
- Procedure calls require a runtime environment
- Function is a procedure that returns a value to its caller
- Can be recursive
- Unlike procedures, functions can be understood independently of the Von Neumann concept
- Higher order functions

Control Abstractions

Unit Abstractions

1. A collection of functions, methods or subroutines that are related in some way.
2. Can be translated separately, so that programmers do not need to know the details of how things are done.
3. Examples: java.applet.Applet, math.h

Other abstractions

1. For parallel processing

1.3 Computational Paradigms

- Features of a language based on the Von Neumann model
  - Variable represent memory locations
  - Assignment allows to operate on those memory locations
- An "imperative language"
    - *Sequential execution of instructions*
    - *Variables representing memory locations*
    - *Assignment to change values of variables*
  - The requirement that computation can be described as a sequence of instructions operating on a single piece of data is referred to as Von Neumann bottleneck.
  - Problems
    - Non deterministic computation
    - parallelism

1.3 Computational Paradigms

Object-oriented Programming

1. Based on “object” = memory locations + operations on them.
2. Each object is sort of like its own computer.
3. Objects are grouped into classes – having same properties
4. An object is an “instance” of a class.
5. Next page gives example of a “GCD” operation, but defined in a class.

Computational Paradigms

```
public class IntWithGcd
{ public IntWithGcd( int val) { value = val; }
  public int intValue( ) { return value; }
  public int gcd( int v )
  {   int z = value;
      int y  = v;
      while( y != 0)
      { int t = y;
        y = z % y;
        z = t;         }
       return z;    }
  private int value;     }
```

Functional Programming

- ❑ A program is a description of a specific computation
- ❑ “WHAT” of the computation + “HOW” of the computation

Virtual black box

- A program becomes equivalent to a mathematical function
- Programs, procedures and functions as functions
- It only distinguishes between input and output

Functional Programming

- The functional paradigm has evaluation of functions as its description of computation.
- Also called “applicative” language.
- Does not use much variable definition or assignment, except what a function may need to do its job.
- Based on “passed” parameters and “returned” values”
- Repetitive operations are based on function recursion, rather than on looping mechanisms.
- Doing away with variables and loops makes the language
  - More independent of the machine model
  - More like mathematics and can make decisions on behavior better

Examples of Lambda Expressions

- The identity function:

$$I =_{def} \lambda x.\ x$$

- A function that given an argument y discards it and computes the identity function:

$$\lambda y.\ (\lambda x.\ x)$$

- A function that given a function f invokes it on the identity function

$$\lambda f.\ f\ (\lambda\ x.\ x)$$

Lambda Expressions in Programming Languages

- ❑ it allows functions to be treated as data values
- ❑ Features
  - ❑ do not have a specific name
  - ❑ not associated with any class unlike a java method
  - ❑ can be passed as an argument to a method or stored as a variable (passed around)
  - ❑ concise syntax – not verbose like inner classes
- ❑ `(parameters)→ expressions;`
- ❑ `(parameters)→ {statements;}`

- ❑ `()→ {return "CR";}`
- ❑ `()→ "CR"`

1.3 Computational Paradigms

Another even weirder example from LISP

```
(define (gcd u v) (if (= v 0) u (gcd v (modulo u v))))
```

1.3 Computational Paradigms

Logic programming

1. Based on symbolic logic.
2. Statements are given about what is true of an object
3. No need for control abstraction such as loops
4. Control is supplied by the underlying system
5. Often called “declarative programming”
6. “Very High Level” languages – e.g. Prolog
    1. gcd of u and v is u if v = 0;
    2. gcd of u and v is same as gcd of v and u mod v, if v is not 0

Computational Paradigms

In Prolog

```
gcd(U, V, U) :- V= 0.
gcd(U, V, X) :- not (V = 0),
                    Y is U mod V,
                    gcd(U. Y, X).
```

a :- b, c, d means a is true if b, c, and d are true.

How would you “read” the above code?

A final word – languages generally do not follow only one paradigm exclusively. There is a mix-match to make things as easy as possible for the programmers.

Language Definition

1. A programming language needs a “precise” definition
2. We must know what each construction in a language will do so that we know what the computer will do.
3. A precise definition also helps make the language machine independent. (Follow standards set by other organizations such as ANSI or ISO.)
4. Programmers will need to know how programs interact; this also implies that a good definition is required.
5. With good definition, the program design phase of software development goes much easier.

Language Syntax

- Syntax of a language is like the grammar of a regular language.
- Describes how parts of the language can be put together to form other parts.
- Usually uses some sort of rules.
- <if-statement> ::= if(<expression>) <statement>
- [else <statement>]
- Also important is the “lexical structure”. This is the structure of the words of the language. Each word is a “token”. For example “if” and “else” are tokens.
- Syntax also defines punctuation, when to use “;”, etc.

Language Definition

Language Semantics

1. This is much more complex and difficult to define.
2. Basically semantics deals with the “meaning” of statements in the language.
3. “IF YOU DO THIS” then “THE COMPUTER WILL DO THAT”
4. Example: An if-statement is executed by first evaluating its expression, which must have arithmetic or pointer type, including all side effects, and if it compares unequal to 0, the statement following the expression is executed. If there is an else part, and the expression evaluates to 0, that statement following the “else” is executed.
5. But even the above semantic definition has problems. (What?)

Operational Semantics

- Definitional interpreters or compilers
- Specifies how an arbitrary program would be executed on a hypothetical machine
- It defines the behavior of programs in terms of an abstract machine that is simple enough to be completely understood and simulated by any user
  - Answers questions about program behavior
- Reduction machine is a collection of steps in reducing programs by applying their operations to values

Denotational Semantics

- Uses functions to describe the semantics of a programming language
- Programs are converted to mathematical functions
- A function describes semantics by associating values to syntactically correct constructs
- Syntactic Domain
  - E: Expression
  - N: Number
  - D: Digit
  - $E \rightarrow E_1$ ‘+’ E_2
- Semantic domains
  - Domain v: Integer={….,-1,0,1,2,…}
  - +: Integer x Integer $\rightarrow$ Integer
- Semantic functions
  - E:Expression$\rightarrow$Integer

Axiomatic Semantics

- A program or statement or language construct
  - Describes the effect its execution has on assertions about the data manipulated by the program
  - Elements of mathematical logic are used here to specify the semantics

Assertions

```
– class TestingAxioms {
–
–         public static void main(String args[]) {
–                 Scanner sc=new Scanner(System.in);
–                 System.out.println("Enter age:");
–                 int value=sc.nextInt();
–                 assertValue>=18:"valid";
–                 System.out.println("Value is " + value);
–         }
– }

– Java –ea <>
```

Properties of Formal Semantics Specification

- Complete
  - Every correct terminating program must have an associated semantics given by the rules
- Consistent
  - Same program cannot be given two different conflicting semantics
- Independent
  - No rule should be derivable from other rules

Language Translation

- All programming languages must have a “translator”
- A translator is another program that accepts programs written in the language and that
  - a) executes the program directly or
  - b) transforms them into a form suitable for execution at a later time.
- Interpreter = direct execution
- Compiler = creates file for later execution

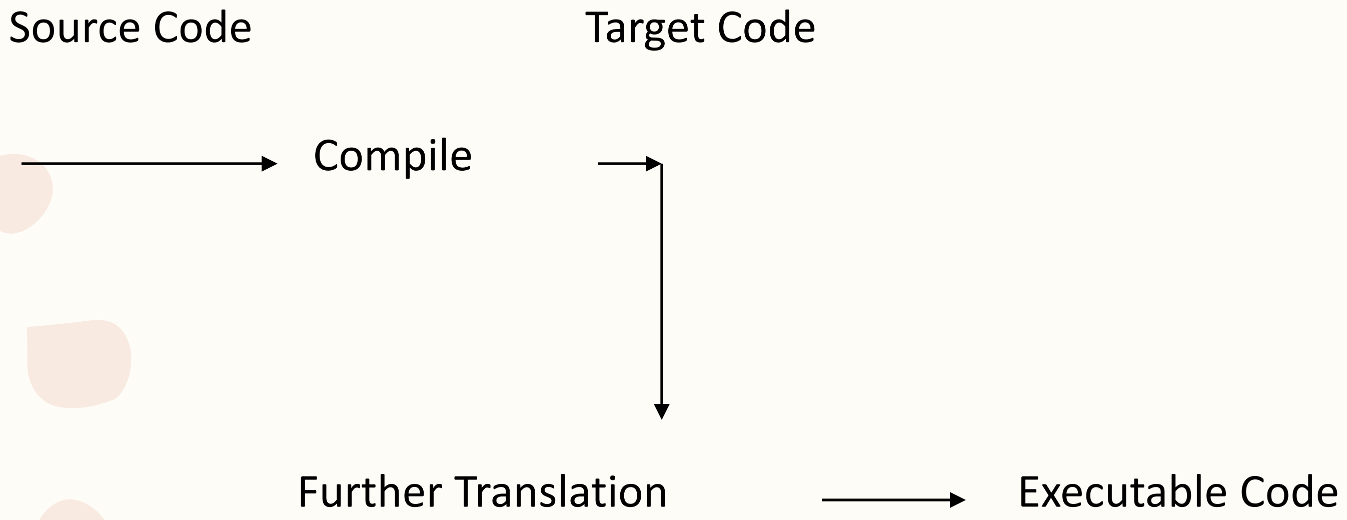
Language Translation

Interpreters

$$\begin{tikzcd}
 & \text{Source Code} \arrow[d] & \\
\text{Input} \arrow[r] & \text{Interpreter} \arrow[r] & \text{Output}
\end{tikzcd}$$

Language Translation

Compiler (part 1)



Language Translation

Compiler (part 2)

$$\begin{tikzcd}
 & \text{Executable Code} \arrow[d] & \\
\text{Input} \arrow[r] & \text{Processor} \arrow[r] & \text{Output}
\end{tikzcd}$$

Language Translation

1. Be careful! Some compilers do not translate source code directly into machine language.
2. Some compilers produce “intermediate code” that is then interpreted by another program
3. Java is like this. JAVAC creates a .class file, which is then interpreted by the JVM (Java Virtual Machine)

Language Translation

All language translators must do similar operations:

1. Lexical Analyzer (scanner) – reads text and separates text into sequences of characters, based on punctuation – creates tokens
2. Syntax Analyzer (parser) – determines structure of the tokens and syntactic correctness of the source code
3. Semantic Analyzer – determines the meaning of the program and how to create the target code
4. These phases are not separate; they tend to go be done back-and-forth until the program is translated.
5. Also must maintain a “runtime environment” that contains space for variables, data structures, code, etc.

Discussion Topics

- Difference between data structure and abstract data types
- An abstraction allows programmers to say more with less in their code. Discuss about its pros and cons with examples