Concepts of Programming Languages: Introduction

Dr. Sherif G. Aly

Textbook and Partial Credit: Louden

Introduction

- Objective:

  To introduce and study the major principles and concepts underlying all programming languages.

What is a Programming Language?

- A notation for communicating to a computer what we want it to do.
- How accurate is the definition above?

What is a Programming Language?

- A notation for communicating to a computer what we want it to do.
  - Before the 1940s, computers were programmed using hardwiring: switches were set by the programmer to connect the internal wiring of a computer to perform requested tasks!

What is a Programming Language?

- John von Neumann envisioned a computer should not be hard wired to do specific tasks.
- A series of codes stored as data should determine the actions taken by a central processing unit.

What is a Programming Language?

- Soon, the assembly language was born.
  - Example:

LDA #2	(Load 2 into the accumulator)
STA X	(Store the present value in the accumulator in register X).

What is a Programming Language?

- The assembly language is
  - Very machine dependent
  - Very low in abstraction
  - More difficult to program
  - However, very fast and provides tremendous programming control.

What is a Programming Language?

- Higher levels of programming languages are needed to:
  - Provide a higher level of abstraction.
  - Improve the ability to write concise, more understandable instructions.
  - Could be used with little change from machine to machine.
  - Capture generalizations such as loops, assignments, conditions, etc.

What is a Programming Language?

- The same assembly code before:
  - LDA #2 (Load 2 into the accumulator)
  - STA X (Store the present value in the accumulator in register X).
- Could be written in C as:
  - X=2

What is a Programming Language?

- ~~A notation for communicating to a computer what we want it to do.~~

- A programming language is a notational system for describing computation in machine-readable and human-readable form.

Computation

- Computation is any process that can be carried out by a computing machine such as:
  - Mathematical calculations.
  - Data manipulation.
  - Text processing.
  - Information storage and retrieval

Computation

- Computation is formally described using a Turing Machine.
    - A Turing Machine is:
        - A very simple computer that is capable of carrying out all known computations (however, not very efficiently).
    - Church's Thesis:
        - It is not possible to build a machine that is inherently more powerful than a Turing Machine.

Computation

- A programming language is Turing Complete if:
  - It can be used to describe any computation performed by a Turing Machine.
- A programming language is Turing Complete if it has (trivially)
  - Integer variables and arithmetic.
  - Sequential execution of statements.
  - Assignment, selection (if), and looping (while) !!!

Machine Readability

- For a language to be machine readable it must:
  - Have a simple structure to allow for efficient translation.
    - There must be an algorithm for translation: unambiguous and finite.
    - The algorithm must not have a very high complexity.
  - Most programming languages can be translated in time that is proportional to the size of the program.

Machine Readability

- Usually machine readability is ensured by restricting the structure of a programming language to that of:

Context Free Languages

- Translation should be based on the same structure also.

Human Readability

- A programming language should provide abstractions of complex computational tasks, yet that have to be easy to understand.
- Programming languages tend to resemble natural languages (English, German, etc.).
- The goal is to reduce the effort required to read and understand a complex program.

Human Readability

- Large programs also require many programmers to simultaneously write different parts of the program.
- A programming language is not only a means for describing a computation, but:
  - Is also now part of a software development environment.
  - Should promote and enforce a software design methodology.
  - (Software Engineering!)

Human Readability

- Software development environments do not only contain facilities to write and translate programs but also:
  - Manipulates files.
  - Keeps records of changes.
  - Performs debugging.
  - Testing.
  - Analysis.

Human Writability

- What about writability?
- Should not a programming language be easy to write also?
- To some extent, yes, readability tends to be more important as many people will be reading and maintaining code after you write it.

Abstractions in Programming Languages

- Data abstractions
  - Strings
  - Numbers
  - Trees
- Control abstractions
  - Loops
  - Conditional statements
  - Procedure calls.

Data Abstractions – Simple or Basic

- Abstract the internal representation of common data values.
- Integer data values are usually stored using two's complement for example.
- Floating point data values are stored using a mantissa representation (mantissa sign, value, exponent sign, value).

Data Abstractions – Simple or Basic

- Memory locations containing data values are abstracted by giving them names using variables.
- The kind of data value is also abstracted using a data type (int, double, float, etc.)
- Example:
  - int x; (C, C++, and Java)
  - var x: integer;   (Pascal)

Data Abstractions – Structured

- Data structures are the principal method for abstracting collections of related data.
  - Arrays
    - int a[10]   (C, C++)
    - int a[] = new int[10]   (Java)
    - typedef int Intarray[10]; (A new non-internal data type called Intarray. It is an array of ten ints).
  - Class
  - Struct

Data Abstractions – Unit Abstractions

- In large programs, it is necessary to collect related code into specific locations within a program.
- Examples:
  - Packages in Ada and Java
  - Modules in ML and Haskell
- A class may also be viewed as a unit abstraction that provides data encapsulation and information hiding by providing access conventions and restrictions.

Data Abstractions – Unit Abstractions

- Unit abstractions facilitate reusability: the ability to reuse data abstractions in different programs.

- Such data abstractions represent:
  - Components: Operationally complete pieces of a program or user interface.
  - Containers: Data structures containing other user-defined data.

Data Abstractions – Unit Abstractions

- Unit abstractions form the basis for language library mechanisms.

Control Abstractions - Basic

- Typical basic control abstractions are those that combine a few machine instructions into a more understandable abstract statement.
- Example: Assignment
  - x = x + 10
- It abstracts the fetching of the value of x, performing a computation and storage of a value into a location denoted by a variable name.

Control Abstractions - Basic

- Example: GOTO

```
…
GOTO 10
…
10 CONTINUE
```

- Abstracts a jump operation to transfer control elsewhere in the program.

Control Abstractions - Structured

- Divide a program into groups of instructions.
- Examples:
  - case (Pascal)
  - switch (C, C++, Java)
  - while

Control Abstractions - Structured

- Example (C):

```
if (x > 0)
{
   …
} else
     {
          …
     }
```

Control Abstractions - Structured

- Example (Ada):

```
if x>0.0 then
        …
        …
        …
else
        …
end if;
```

Opening a group of nested statements is automatic!

Control Abstractions - Structured

- Structured control structures can be nested.
- Procedures, sometimes called subprograms or routines are also powerful structured control abstractions.
- Procedures must be declared, then invoked.

Control Abstractions - Structured

Formal Parameters

- Example: Ada gcd declaration

```
procedure gcd (u, v: in integer; x: out integer) is
      y, t, z: integer;
  begin
      z := u;
      y := v;
      loop
                  exit when y = 0;
                  t := y;
                  y := z mod y;
                  z := t;
      end loop;
      x := z;
  end gcd;
```

Control Abstractions - Structured

- Example: Ada gcd call

```
gcd(8, 18, d)
```

Call and Actual Parameters

Control Abstractions - Structured

- Example: Fortran subroutine declaration

```
SUBROUTINE gcd (u, v, x)
…
END
```

- Fortran call:

```
CALL gcd( a, b, d)
```

Control Abstractions - Structured

- Procedures are more complex mechanisms than selection or looping.
- They require storing information about the program at the point of the call.

Control Abstractions - Structured

- Functions are simply procedures that return a value or result.
- Some languages such as C and C++ have void functions (return no value).

Control Abstractions - Unit

- Control abstractions can also be grouped into files, packages, and units exactly like data abstractions.

Computational Paradigms

- Imperative (also called procedural).
- Object Oriented
- Functional
- Logic
- Parallel (Paradigm on its own?)
- Declarative

The Imperative Paradigm

Computational Paradigms - Imperative

- Sequential execution of instructions.
- The use of variables representing memory locations.
- The use of assignment to change the value of variables.
- Containing loops.

Computational Paradigms - Imperative

- It is not necessary for a programming language to describe computation exactly as such.
- The requirement that a computation be described as a sequence of instructions, each operating on a single piece of data is called the von Neumann bottleneck.
- It restricts the ability of the language to indicate parallel , or non-deterministic computation upon multiple pieces of data.

Computational Paradigms - Imperative

- Imperative programming languages become only one paradigm, or pattern, for programming languages to follow.
- Examples: C, Pascal, core Ada, FORTRAN

The Object Oriented Paradigm

Computational Paradigms – Object Oriented

- Of enormous importance in the last decade.
- Very successful in allowing programmers to write reusable, extensible code that mimics the real world.
- It is merely an extension of the imperative paradigm (sequential execution, changing set of memory locations).
- However, programs now consist of a large number of very small pieces whose interactions are carefully controlled, yet easily changed.

Computational Paradigms – Object Oriented

- Based on the notion of an object.
  - Loosely coupled
  - Collection of data and operations.
- Many programming languages group objects together into classes.
- Objects thus become instances of classes.

Computational Paradigms – Object Oriented

- Example:
  - Java
  - Smalltalk
  - C++

The Functional Paradigm

Computational Paradigms - Functional

- Bases the description of computation on the evaluation of functions, or the application of functions to known values.
- Sometimes called applicative languages.
- The basic mechanism is the evaluation of a function.
- This involves the passing of values as parameters to functions, and obtaining returned values.

Computational Paradigms - Functional

- Passive data, no sequential control.
- All actions performed by function evaluation (call), particularly recursion.
- No variables exist !
- Repetitive operations are not expressed by loops (which require control variables to terminate), rather by recursive functions!

Computational Paradigms - Functional

- Is very much considered the opposite of object oriented programming.
- Examples: Lisp (Scheme), ML, Haskell

Computational Paradigms - Functional

- But why functional programming??
- It does away with variables and loops.
- Becomes more independent of the machine.
- Because they resemble mathematics, it is easier to draw precise conclusions about their behavior!

Computational Paradigms - Functional

- The recursive function theory in mathematics established the following property:

A programming language is Turing complete if it has integer values, arithmetic functions on those values, and if it has a mechanism for defining new functions using existing functions, selection, and recursion.

Computational Paradigms - Functional

- Example Use Ada to create a functional version of GCD:

```
procedure gcd_prog is

  function gcd (u, v: in integer) return integer is
  begin
    if v = 0 then
      return u;
    else
      return gcd(v, u mod v);
    end if;
  end gcd;
```

Recursion, yet no loops or variables!

Computational Paradigms - Functional

- LISP is a more functional oriented language than Ada.
- LISP programs are list expressions
  - Sequences of entities separated by spaces and surrounded by parentheses.
  - (+ 2 3) simply means 2 + 3
  - (gcd 8 18) simply means call gcd and pass it 8 and 18.

Computational Paradigms - Functional

- Example Use LISP (Scheme Dialect) to create the GCD:

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

If v=0 return u, notice no return statement!

Otherwise call gcd recursively with v and the modulus

Computational Paradigms - Functional

- Example Use Haskell to write GCD:

```
gcd u v = if v ==0 then u else gcd v (u ‘mod’ v)
```

The Logic Paradigm

Computational Paradigms - Logic

- Based on symbolic logic.
- A program consists of a set of statements that describe what is true about a desired result.
- As opposed to giving a particular sequence of statements that must be executed in a fixed order to produce the result.

Computational Paradigms - Logic

- A pure logic programming language has no need for control abstractions such as loops or selections.
- Control is supplied by the underlying system.
- Logic programming is sometimes called declarative programming, since properties are declared, but no execution sequence is specified.
- Very high level languages.

Computational Paradigms - Logic

- Example: Prolog
- In Prolog, the form of a program is a sequence of statements, called clauses which are of the form:
  - a :- b, c, d
  - Means a is true
    - if b, c, d are true

Computational Paradigms - Logic

- Unlike functional programs, Prolog needs variables.
- Variables do not represent memory locations as in imperative programming, but behave more as names for the results of partial computations.
- Variables in Prolog must be in upper case.

Computational Paradigms - Logic

- Example: Compute the GCD again using logic programming.

- The properties are as follows:
  - The GCD of u and v is u if v=0
  - The GCD of u and v is the same as the GCD of v and u mod v if v is not equal to zero.

Computational Paradigms - Logic

- Example: Compute the GCD again using logic programming.

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

The GCD of U and V is U if:
V is equal to zero

The GCD of U and V is X provided that V is not equal to zero. But what is X??

So Create Y as U Mod V

X is the result of GCD applied on V and Y

The Parallel Paradigm

- Schools vary in labeling this as a paradigm on its own.
- No sequential execution involved.
- Examples: Java (Threads), Ada (Tasks)

The Declarative Paradigm

- State what needs computing, but not how (sequence).
- Logic and functional paradigms share this property.

Computational Paradigms

- Even though a programming language may exhibit most or all of the properties of one of the four paradigms, few languages adhere purely to one paradigm!
- We were able to write a functional version of GCD using Ada.
- Scheme LISP which is generally considered to be functional, does permit variables to be declared and assigned to, definitely an imperative feature!

Computational Paradigms

- Scheme programs can also be written in an object oriented style.
- We can refer to a “Style” as following one or more of the paradigms.
- It is up to you which paradigm to use, based on which is more appropriate.

Examples of Mostly “Pure” Languages

- Imperative: (old) FORTRAN
- Functional: Haskell
- Object Oriented: Smalltalk

Language Definition

- There has been increasing acceptance of the need for programming languages to have definitions that are formally precise.

- Precise definitions allow:
  - Definitions of the effect of language constructs.
  - Mathematical reasoning about programs.
  - Standardization of languages.

Language Definition

- Standardization organizations:
  - ANSI (American National Standards Institute)
  - ISO (International Organization for Standardization).

- Such organizations have published definitions for many languages including:
  - Pascal
  - FORTRAN
  - C
  - C++
  - Ada
  - Prolog

Language Definition

- Language definition is loosely divided into two parts:
  - Syntax
  - Semantics

Language Definition

- Syntax:
  - Like the grammar of natural languages.
  - Defines the structure of a program, and how parts can be combined together.
  - Usually formally defined using a context-free language.

Language Definition

- Syntax Example:
  - An if statement consists of
    - The word "if" followed by
    - An expression inside parenthesis followed by
    - A statement followed by
    - An optional else consisting of the word "else" and another statement.

Language Definition

- Syntax Example in Context Free Grammar:

<if-statement> ::= if (<expression>) <statement> [else <statement>]

Optional

OR

If-statement → if (expression) statement [else statement]

Language Definition

- Syntax:
  - An issue closely related to the syntax of a programming language is its lexical structure.
  - Similar to spelling in a natural language.
  - The words in the programming language are usually called tokens.
  - Example: if, else, +, <=

Language Definition

- Semantics:
  - Denotes the meaning of a language and the actual result of execution.
  - Is more complex and difficult to describe precisely.
  - Usually described in English, but can be done mathematically also.

Language Definition

- Semantics Example (If in C):
  - *An if-statement is executed by first evaluating its expression, which must have arithmetic pointer or 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 is 0, the statement following the else is executed.*

Language Definition

- Semantics Example (If in C):
  - An if-statement is executed by first evaluating its expression, which must have arithmetic pointer or 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 is 0, the statement following the else is executed.

What if the expression evaluates to false and there is no else?? The statement above does not describe this!

Language Definition

- Semantics Example (If in C):
  - An if-statement is executed by first evaluating its expression, which must have arithmetic pointer or 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 is 0, the statement following the else is executed.

Is the If statement described above <u>safe</u>?? There should not be another statement in the language capable of executing the body of the if, without evaluating the expression!

If (x!=0) y = 1/x;

If the If statement is save, the division here is adequately protected from division by zero situations!

Language Definition

- Semantics:
  - The alternative to this informal description is to use a formal method.
  - However, no formal method (yet) is analogous to the use of context free grammars for describing syntax.
  - Formal semantic notational systems include:
    - Operational semantics
    - Denotational semantics.
    - Axiomatic semantics.

Language Translation

- Semantics:
  - The alternative to this informal description is to use a formal method.
  - However, no formal method (yet) is analogous to the use of context free grammars for describing syntax.
  - Formal semantic notational systems include:
    - Operational semantics
    - Denotational semantics.
    - Axiomatic semantics.

Language Translation

- For a programming language to be useful, it must have a translator:
  - A program (in hardware or software) that accepts other programs written in the language in question and either:
    - Executes them directly.
    - Transforms them into a form suitable for execution.

Language Translation

- A translator is primarily one of three types:
  - Interpreter
  - Compiler
  - Pseudo-Interpreter

Language Translation

- Interpreter:
  - Executes a program directly.
  - Is a one step process:
    - Both the program and input are provided to the interpreter.The output is then obtained.
    - Can be viewed as a simulator for a machine whose “machine language” is the language being translated.

Language Translation

- Interpreter:

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

Language Translation

- Compiler:
  - Produces an equivalent program in a form suitable for execution.
  - Is at least a two step process:
    - The original program (source) is input to the compiler.
    - The new program (target) is output from the compiler.
  - The target program may then be executed, if it is in a form suitable for execution (i.e. machine language).

Language Translation

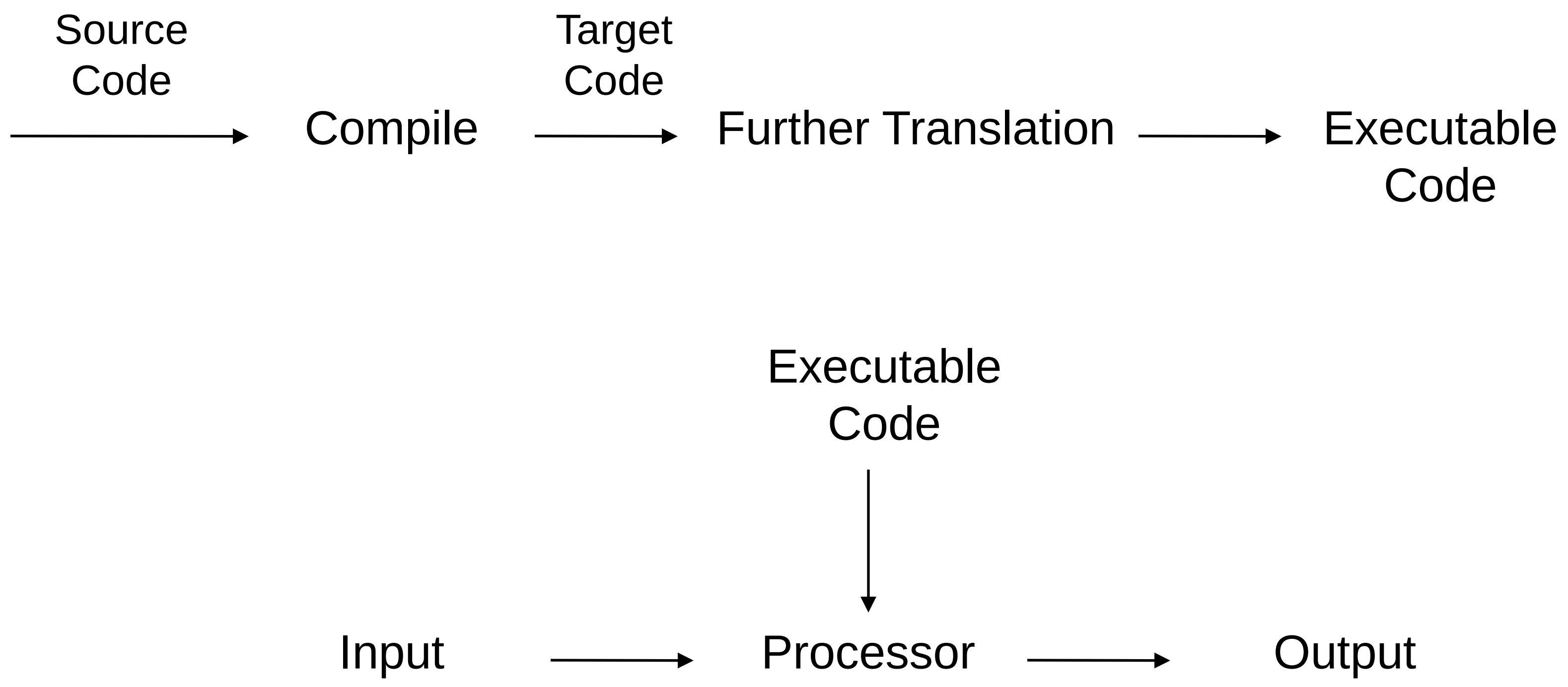
- Compiler:
  - Commonly the target language is assembly language.
  - The target program must then be translated by an assembler into an object program
  - The object program must then be linked with other object programs.
  - Then loaded into appropriate memory locations before it can be executed.

Language Translation

- Compiler:
  - Sometimes the target language is another programming language.
  - Another compiler must then be used to obtain an executable object program.

Language Translation

- Compiler:



Language Translation

- Pseudo-Interpreter:
  - Intermediate between interpreters and compilers.
  - A source program is compiled into an intermediate language.
  - The intermediate language is then interpreted.
  - Example: Java

Language Translation

- Sometimes preprocessors are needed.
- Preprocessors are run prior to translation.
- They convert the program into a form suitable for translation.

Language Translation

- Both compilers and interpreters must perform similar operations (phases):
  - Lexical analysis (scanning).
    - Converts the textual representation of the program as a sequence of characters into a form easier to process (tokens) representing keywords, identifiers, constants, etc.
  - Syntax analysis (parsing).
    - Determines the structure of the sequence of tokens.
  - Semantic analysis.
    - Determines the meaning of a program.

Language Translation

- Such phases do not occur separately, but are usually combined.
- A translator must also maintain a runtime environment to:
  - Allocate memory space to store program data.
  - Keep track of the progress of execution.
- Since a compiler does not execute code directly, a compiler will maintain the runtime environment indirectly by adding suitable operations to the target code.

Language Translation

- The properties of a programming language that can be determined prior to execution are called static properties.
    - Lexical and syntactic structure.
- Properties that can be determined only during execution are called dynamic properties.

Language Translation

- Programming languages can be designed to be more suitable for interpretation or compilation.
- A more dynamic language is more suitable for interpretation.
- A language with strong static structure is more suitable for compilation.

Language Translation

- Usually imperative languages have more static properties and are compiled.
- Usually functional and logic programming languages have more dynamic properties and are interpreted.
- A compiler or interpreter can exist for any language of course, regardless of static or dynamic properties.

Language Translation

- Static allocation:
  - All variables are assumed to occupy a fixed position in memory for the duration of program execution.
  - A fully static environment may be used.
  - To the opposite extreme, fully dynamic environments may be used if all variables do not occupy a fixed position in memory.

Language Translation

- Stack based environments:
  - Midway between fully static and fully dynamic environments.
  - Like C and Ada, both have static and dynamic aspects.

Language Translation

- Efficiency of Compilers Vs. Translators:
  - Interpreters are inherently less efficient than compilers.
  - They must simulate the actions of the source program on the underlying machine.
  - Compilers can boost efficiency of the target code by performing optimizations and performing several passes to analyze the source program in detail.

Language Translation

- Efficiency of Compilers Vs. Translators:
  - A programming language needing efficient execution is more likely to be compiled than interpreted.
  - Interpreters on the other hand usually have an interactive mode and can provide immediate output. Example:
    - > gcd(8 18)   ;;calls the gcd function
    - >2   ;;immediately provides output.

Language Translation

- Error Classification:
  - <u>Lexical</u>: character-level error, such as illegal character (hard to distinguish from syntax).
  - <u>Syntax</u>: error in structure (e.g., missing semicolon or keyword).
  - <u>Static semantic</u>: non-syntax error prior to execution (e.g., undefined vars, type errors).
  - <u>Dynamic semantic</u>: non-syntax error during execution (e.g., division by 0).
  - <u>Logic</u>: programmer error, program not at fault. (e.g. computing the average of two numbers is attempted by dividing by 3!)

Language Translation

- Error Reporting:
  - A compiler will report lexical, syntax, and static semantic errors. It cannot report dynamic semantic errors.
  - An interpreter will often only report lexical and syntax errors when loading the program. Static semantic errors may not be reported until just prior to execution. Indeed, most interpreted languages (e.g. Lisp, Smalltalk) do not define *any* static semantic errors.
  - No translator can report a logic error.