CSC 272 - Software II: Principles of Programming Languages

Lecture 1 - An Introduction

What is a Programming Language?

- A ***programming language*** is a notational system for describing computation in machine-readable and human-readable form.
- Most of these forms are ***high-level languages***, which is the subject of the course.
- Assembly languages and other languages that are designed to more closely resemble the computer's instruction set than anything that is human-readable are ***low-level languages***.

Why Study Programming Languages?

- In 1969, Sammet listed 120 programming languages in common use – now there are many more!
- Most programmers never use more than a few.
  - Some limit their career's to just one or two.
- The gain is in learning about their underlying design concepts and how this affects their implementation.

The Six Primary Reasons

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Better use of languages that are already known
- Overall advancement of computing

Reason #1 - Increased ability to express ideas

- The depth at which people can think is heavily influenced by the expressive power of their language.
- It is difficult for people to conceptualize structures that they cannot describe, verbally or in writing.

Expressing Ideas as Algorithms

- This includes a programmer's to develop effective algorithms
- Many languages provide features that can waste computer time or lead programmers to logic errors if used improperly
  - E. g., recursion in Pascal, C, etc.
  - E. g., GoTos in FORTRAN, etc.

Reason #2 - Improved background for choosing appropriate languages

- Many professional programmers have a limited formal education in computer science, limited to a small number of programming languages.
- They are more likely to use languages with which they are most comfortable than the most suitable one for a particular job.

Reason #3 - Increased ability to learn new languages

- Computer science is a relatively young discipline and most software technologies (design methodology, software development, and programming languages) are not yet mature. Therefore, they are still evolving.
- A thorough understanding of programming language design and implementation makes it easier to learn new languages.

Learning a New Language

- It is easier to learn a new language if you understand the underlying structures of language.

  Examples:
  - It is easier for a BASIC program to FORTRAN than C.
  - It is easier for a C++ programmer to learn Java.
  - It is easier for a Scheme programmer to learn LISP.

Tiobe Index

Position Jul 2012	Position Jul 2011	Delta in Position	Programming Language	Ratings Jul 2012	Delta Jul 2011
1	2	↑	C	18.331%	+1.05%
2	1	↓	Java	16.087%	-3.16%
3	6	↑↑↑	Objective-C	9.335%	+4.15%
4	3	↓	C++	9.118%	+0.10%
5	4	↓	C#	6.668%	+0.45%
6	7	↑	(Visual) Basic	5.695%	+0.59%
7	5	↓↓	PHP	5.012%	-1.17%
8	8	=	Python	4.000%	+0.42%
9	9	=	Perl	2.053%	-0.28%
10	12	↑↑	Ruby	1.768%	+0.44%
11	10	↓	JavaScript	1.454%	-0.79%
12	14	↑↑	Delphi/Object Pascal	1.157%	+0.27%
13	13	=	Lisp	0.997%	+0.09%
14	15	↑	Transact-SQL	0.954%	+0.15%
15	25	↑↑↑↑↑↑↑↑↑↑	Visual Basic .NET	0.917%	+0.43%
16	16	=	Pascal	0.837%	+0.17%
17	19	↑↑	Ada	0.689%	+0.14%
18	11	↓↓↓↓↓↓↓	Lua	0.684%	-0.89%
19	21	↑↑	PL/SQL	0.645%	+0.10%
20	26	↑↑↑↑↑↑	MATLAB	0.639%	+0.19%

Reason #4 - Better understanding of significance of implementation

- It is often necessary to learn about language implementation; it can lead to a better understanding of why the language was designed the way that it was.
- Fixing some bugs requires an understanding of implementation issues.

Reason #5 - Better use of languages that are already known

- To allow a better choice of programming language
- Some languages are better for some jobs than others.
  - Example – FORTRAN and APL for calculations, COBOL and RPG for report generation, LISP and PROLOG for AI, etc.

Better Use of a Language

- To improve your use of existing programming language
- By understanding how features are implemented, you can make more efficient use of them.
- Examples:
- Creating arrays, strings, lists, records.
- Using recursions, object classes, etc.

Reason #6 - Overall advancement of computing

- Frequently, the most popular language may not be the best language available.
- E.g., ALGOL 60 did NOT displace Fortran.
  - They had difficulty understanding its description and they didn't see the significance of its block structure and well-structured control statements until many years later.

Programming Domains

- Scientific Applications
- Business Applications
- Artificial Intelligence
- Web Software

Numerically-Based Languages

- Many of the earliest computers were used almost exclusively for scientific calculations and consequently many of the earliest attempts at languages were for scientific purposes.
- Grace Murray Hopper's ***A-0*** and John Backus's ***Speedcoding*** ere designed to compile simple arithmetic expressions.

FORTRAN

- John Backus's team at IBM developed FORTRAN (for ***FOR***mula ***TRAN***slator) in 1955-1957.
- While FORTRAN was designed for numerical computation, it included control structures, conditions and input/output.
- FORTRAN's popularity led to FORTRAN II in 1958, FORTRAN IV in 1962, leading to its standardization in 1966, with revised standards coming out in 1977 and 1990.

Business Languages

- Commercial data processing was one of the earliest commercial applications of computers.
- Grace Murray Hopper et. al. at Univac developed FLOWMATIC, an English-like language for business applications.
- The U.S. Defense Dept. sponsored the effort to develop COBOL (***Co***mmon ***B***usiness-***O***riented ***L***anguage), which was standardized in 1960, revised in 1961 & 1962, re-standarized in 1968, 1974, and 1984.

Artificial Intelligence

- Artificial Intelligence deals with emulating human-style reasoning on a computer.
- These applications usually involve symbolic computation, where most of the symbols are names and not numbers.
- The most common data structure is the list, not the matrix or array as in scientific computing and not the record as in business computing
- Artificial intelligence requires more flexibility than other programming domains.

Artificial Intelligence Languages

- The first AI language was IPL (International Processing Language, developed by the Rand Corporation. Its low-level design led to its limited use.
- John McCarthy of MIT developed LIST for the IBM 704 (which eventually led to Scheme and Common LISP). LISP is a recursion-oriented, list-processing language that facilitated game-playing programs.
- Yngve of MIT developed COMIT, a string-processing language, which was followed by AT&T's SNOBOL.
- Prolog was developed by Colmerauer, Roussel and Kowalski based on predicate calculus and mathematical logic.

Systems Languages

- Assembly languages were used for a very long time operating systems programming because of its power and efficiency.
- CPL, BCPL, C and C++ were later developed for this purpose.
- Other languages for systems programming included PL/I, BLISS, and extended ALGOL.

Web Software

- Eclectic collection of languages:
  - **Markup** (e.g., HTML) – used for annotating a document in a manner that can be distinguished from the text.
  - **Scripting** (e.g., PHP) - the language that enable the script to run these commands and typically include control structures such as if-then-else and while-do.
  - **General-purpose** (e.g., Java) – can be used for a wide range of programming jobs.

Language Evaluation Criteria

- **<u>Readability</u>** – the ease with which programs can be read and understood.
- **<u>Writability</u>** – the ease with which programs can be developed for a given program domain.
- **<u>Reliability</u>** – the extent to which a program will perform according to its specifications.

What Do We Mean By Machine Readability?

- A language is considered machine-readable if it can be translated efficiently into a form that the computer can execute.
- This requires that:
  - A translation algorithm exists.
  - The algorithm is not too complex.
- We can ensure machine readability by requiring that programming languages be ***<u>context-free languages</u>***.

What Do We Mean By Human Readability?

- It is harder to define human readability in precise terms.
- Generally this requires a programming language to provide enough abstractions to to make the algorithms clear to someone who is not familiar with the program's details.
- As programs gets larger, making a language readable requires that the amount of detail is reduced, so that changes in one part of a program have a limited effect on other parts of the program.

What Contributes to Readability?

There are five characteristics of programming languages that contribute to readability:

- Simplicity
- Orthogonality
- Control Statements
- Data types and Structures
- Syntax

Simplicity

- Programming languages with a large number of basic components are harder to learn; most programmers using these languages tend to learn and use subsets of the whole language.
- Complex languages have multiplicity (more than one way to accomplish an operation).
- Overloading operators can reduce the clarity of the program's meaning

An Example of Multiplicity

- All of the following add one to the variable count in C:

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

Do they mean the same thing?

Orthogonality

- For a programming language to be orthogonal, language constructs should not behave differently in different contexts.
- The fact that Modula-2's constant expressions may not include function calls can be viewed as a nonorthogonality.

Examples of Nonorthogonalities

- Other examples of nonorthogonalities include:
  - In Pascal functions can only return scalar values or pointers.
  - In C/C++, arrays types cannot be returned from a function
  - In C, local variables must be at the beginning of a block.
  - C passes ALL parameters by value except arrays (passed by reference).

Example – IBM vs. VAX Assembler

- IBM Assembler

```
A  Reg1, memory_cell ; Reg1 = Reg1 + memocell
AR Reg1, Reg2 ; Reg1 = Reg1 + Reg2
```

- VAX Assembler

```
ADDL     operand1, operand2
```

Control Statements

- In the 1950s and 1960s, the **`goto`** was the most common control mechanism in a program; however, it could make programs less readable.
- The introduction of **`while`**, **`for`** and **`if-then-else`** eliminate the need for **`gotos`** and led to more readable programs.

Data Types and Structures

- A more diverse set of data types and the ability of programmers to create their own increased program readability:
  - Booleans make programs more readable:
    `TimeOut = 1` vs. `TimeOut = True`
  - The use of records to store complex data objects makes programs more readable:
    ```
    CHARACTER*30 NAME(100)
    INTEGER AGE(100), EMPLOYEE_NUM(100)
    REAL SALARY(100)
    ```
    Wouldn't it better if these were an array of records instead of 4 parallel arrays?

Syntax

- Most syntactic features in a programming language can enhance readability:
  - **Identifier forms** – older languages (like FORTRAN) restrict the length of identifiers, which become less meaningful
  - **Special words** – in addition to `while`, `do` and `for`, some languages use special words to close structures such as endif and endwhile.
  - **Form and meaning** – In C a `static` variable within a function and outside a function mean two different things – this is undesirable.

Writability

- Historically, writability was less important than efficiency than efficiency. As computers have gotten faster, the reverse has become true to a certain extent.
- Writability must be considered within the context of the language's target problem domain.
  - E.g., COBOL handles report generating very well but matrices poorly. The reverse is true for APL.
- A large and diverse set of construct is easier to misuse than a smaller set of constructs that can be combined under a consistent et of rules. (This is simple and orthogonal)

Writability and Abstraction

- A programming language should be able to support data abstractions that a programmer is likely to use in a given problem domain.
- Example – implementing binary trees in FORTRAN, C++ and Java.

Reliability

- Reliability is the assurance that a program will not behave in unexpected or disastrous ways during execution.
- This sometimes requires the use of rules that are extremely difficult to check at translation or execution time.
  - ALGOL68's rule prohibiting dangling reference assignments (referring to objects that have been de-allocated).
- Reliability and efficiency of translation are frequently diametrically opposed.

Contributing Factors To Reliability

- **Type Checking** – a large factor in program reliability. Compile-time type checking is more desireable. C's lack of parameter type checking leads to many reliability problems.
- **Exception Handling** – the ability to catch run-time errors and make corrections can prevent reliability problems.
- **Aliasing** – having two or more ways of referencing the same data object can cause unnecessary errors.

Cost of Use

- Cost of program execution
  - A slower program is more expensive to run on a slower computer.
  - In an era of faster, cheaper computer, this is less of a concern.
- Cost of program translation
  - Optimizing compilers are slower than some other compilers designed for student programs, which will not run as many times..
- Cost of program creation, testing and use
  - How quickly can you get the program executing ***correctly***.
- Cost of program maintenance
  - How expensive will it be to modify the program when changes are needed in subsequent years?

Influences on Language Design

Other factors have had a strong influence on programming language design:

- Computer Architecture
- Programming Methodologies

Computer Architecture

- Most computers are still based on the von Neumann architecture, which view memory as holding both instructions and data interchangably.
- This has influenced the development of imperative languages and has stifled the adaption of functional languages.
- As parallel processing computers are developed, there have been several attempts made to develop languages that exploit their features.

Programming Methodologies

- New methods of program development have led to advances in language design:
- These have included:
  - structured programming languages
  - data abstraction in object-oriented languages

Language Categories

- There are four different programming language paradigms:
  - Imperative
  - Functional
  - Declarative
  - Object-Oriented

Imperative Languages

- Imperative languages are command-driven or statement-oriented languages.
- The basic concept is the machine state (the set of all values for all memory locations).
- A program consists if a sequence of statements and the execution of each statement changes the machine state.
- Programs take the form:

```
statement1;
statement2;
 … …
```

- FORTRAN, COBOL, C, Pascal, PL/I are all imperative languages.

Functional Languages

- An functional programming language looks at the function that the program represents rather than the state changes as each statement is executed.
- The key question is: What function must be applied to our initial machine and our data to produce the final result?
- Statements take the form:

  $\texttt{function}_n\texttt{(function}_1\texttt{, function}_2\texttt{, … (data)) …)}$
- ML, Scheme and LISP are examples of functional languages.

Example GCD in Scheme

```
;; A Scheme version of Greatest
;; Common divisor
(define (gcd u v)
  (if (= v 0) u
      (gcd v (modulo u v))))
```

A Function GCD in C++

```
//gcd() - A version of greatest common
//        divisor written in C++ in
//        function style
int  gcd(int u, int v)
{
  if (v == 0)
     return(u);
  else
     return(v, u % v);
}
```

Rule-Based Languages

- Rule-based or ***declarative*** languages execute checking to see if a particular condition is true and if so, perform the appropriate actions.
- The enabling conditions are usually written in terms of predicate calculus and take the form:

  $\texttt{condition}_1 \rightarrow \texttt{action}_1$

  $\texttt{condition}_2 \rightarrow \texttt{action}_2$

  … … …
- Prolog is the best know example of a declarative language.

GCD in Prolog

means "if"

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

clauses in Prolog

Object-Oriented Languages

- In object-oriented languages, data structures and algorithms support the abstraction of data and endeavor to allow the programmer to use data in a fashion that closely represents its real world use.
- Data abstraction is implemented by use of
  - ***Encapsulation*** – data and procedures belonging to a class can only be accessed by that classes (with noteworthy exceptions).
  - ***Polymorphism*** – the same functions and operators can mean different things depending on the parameters or operands,
  - ***Inheritance*** – New classes may be defined in terms of other, simpler classes.

GCD in Java

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

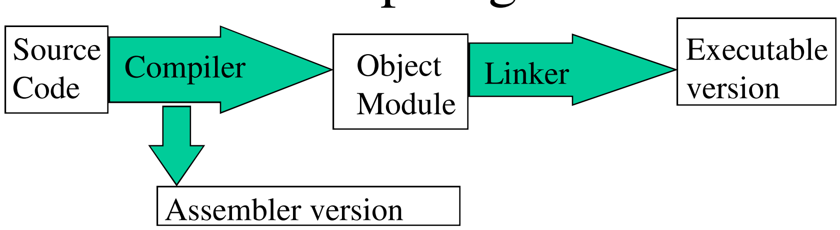
Language Design Trade-offs

- Frequently, design criteria will be contrdictory:
  - Reliability and cost of execution
  - In APL, expressivity and writability conflict with readability
  - Flexbilty and safety (e.g., variant records as a safety loophole in Pascal).

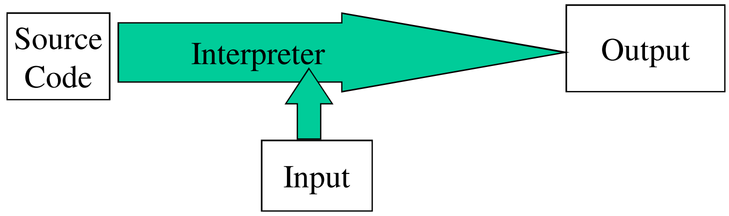
Implementation Methods

- Compilation
- Pure Interpretation
- Hybrid Implementation Systems

The Compiling Process



The Pure Interpretation Process



The Hybrid Interpretation Process