An Introduction to Computer Science

Jingde Cheng

Southern University of Science and Technology

An Introduction to Computer Science

- Computer Science: What Is It and Why Study It?
- Computation: What Is It and Why Study It?
- Computability
- Computational Complexity (CS101A class only)
- Algorithms
- Data, Information, and Knowledge, and Their Representations
- Data Storage
- Computer Architecture
- Data Manipulation in Computer Systems
- Programming Languages and Compilers
- Operating Systems
- System Software and Application Software
- Software Engineering (CS101A class only)
- Knowledge Engineering and Artificial Intelligence (CS101A class only)
- Information Security Engineering (CS101A class only)

Programming: What Is It and Why Study It ?

Programming: What Is It and Why Study It ?

Programming: What Is It and Why Study It?

♣ Program [A Dictionary of Computer Science (7th Edition), OUP, 2016]

- "A set of statements that (after translation from programming-language form into executable form) can be executed by a computer in order to produce a desired behavior from the computer."

♣ Program [IEEE Standard Computer Dictionary]

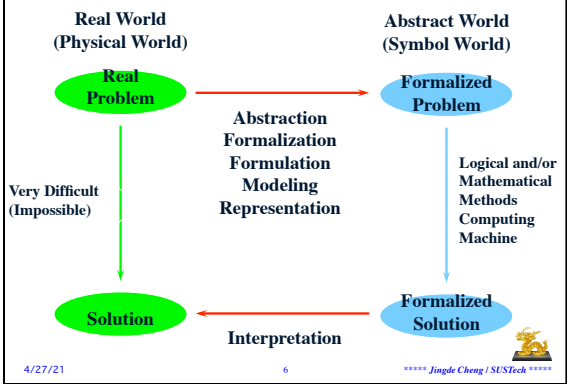
- "A combination of computer instructions and data definitions that enable computer hardware to perform computational or control functions."

Programming: What Is It and Why Study It?

♣ Programming [A Dictionary of Computer Science (7th Edition), OUP, 2016]

- "In the broadest sense, all technical activities involved in the production of a program, including analysis of requirements and all stages of design and implementation. In a much narrower sense it is the coding and testing of a program from some given design."

Real (Physical) World and Abstract (Symbol) World



Programming: What Is It and Why Study It ?

Programming =

Solving Problems by Computers at the Abstract Level of High-level Programming Languages

Programming Language: What Is It and Why Study It ?

Programming Language: What Is It and Why Study It ?

Language: What Is It?

- Language [The Oxford English Dictionary, 2nd Edition]
  - "The whole body of words and of methods of combination of words used by a nation, people, or race."
  - "Words and the methods of combining them for the expression of thought."
- Language [Longman Dictionary of the English Language]
  - "Those words, their pronunciation, and the methods of combining them which are used and understood by a particular people, nation, etc."

Language: What Is It?

- Language [The American Heritage Dictionary of the English Language, 3rd Edition]
  - "The use by human beings of voice sounds, and often written symbols representing these sounds, in organized combinations and patterns in order to express and communicate thoughts and feelings."
  - "A system of words formed from such combinations and patterns, used by the people of a particular country or by a group of people with a shared history or set of traditions."

Programming Language: What Is It and Why Study It?

- Programming language [A Dictionary of Computer Science (7th Edition), OUP, 2016]
  - "A notation for the precise description of computer programs or algorithms. Programming languages are artificial languages, in which the syntax and semantics are strictly defined. Thus while they serve their purpose they do not permit the freedom of expression that is characteristic of a natural language."
- Programming language [IEEE Standard Computer Dictionary]
  - "A language used to express computer programs."

Programming Language: What Is It and Why Study It?

- Program [M. Ben-Ari, 1996]
  - "A program is a sequence of symbols that specifies a computation."
- Programming language [M. Ben-Ari, 1996]
  - "A programming language is a set of rules that specify which sequences of symbols constitute a program, and what computation the program describes."
  - "A programming language is an abstraction mechanism. It enables a programmer to specify a computation abstractly, and to let a program (usually called an assembler, compiler or interpreter) implement the specification in the detailed form needed for execution on a computer."

Programming Language: What Is It and Why Study It?

- **Programming language [M. Ben-Ari, 1996]**
  - "The programming language is one of the most important, not one of the least important, factors that influence the ultimate quality of a software system."
  - "Programming languages exist only for the purpose of bridging the gap in the level of abstraction between the hardware and the real world. There is an inevitable tension between higher levels of abstraction that are easier to understand and safer to use, and lower levels of abstraction that are more flexible and can often be implemented more efficiently."
- **Abstraction [M. Ben-Ari, 1996]**
  - "There is a general truth that arises from the concept of abstraction: The higher the abstraction, the more detail is lost."

Programming Language: What Is It and Why Study It?

- "A good programming language is a conceptual universe for thinking about programming."
  – A. Perlis, NATO Conference on Software Engineering Techniques, Rome, 1969.
- "Programming languages are at the heart of computer science. They are the tools we use to communicate not only with computers but also with people." [Ghezzi and Jazayeri, 1998]
- "Programming languages are notations. They are used for specifying, organizing, and reasoning about computations." [Sethi, 1996]

Programming Language: What Is It and Why Study It?

- "Programming languages are specified by rules for forming correct statements, organizing them into modules, submitting them to a compiler, which translates the code into a language understandable by a particular machine, and finally running the program, i.e., submitting input to the computer, which transforms it into output according to the instructions in the program." [Appleby and VandeKopple, 1997]
- "Any programming language can be considered equivalent to any other, in that each changes values of the store, They can, however, be quite different at both the conceptual and the implementation level. A language is organized around a particular conceptual model." [Appleby and VandeKopple, 1997]

Programming Language: What Is It and Why Study It?

- "Programming languages, like our 'natural' languages, are designed to facilitate the expression and communication of ideas between people. …… However, programming languages differ from natural languages in two ways. First, they have a narrower expressive domain, in that they facilitate only the communication of algorithmic ideas between people. Second, programming languages also enable the communication of algorithmic ideas between people and computing machines." [Tucker and Noonan, 2002]

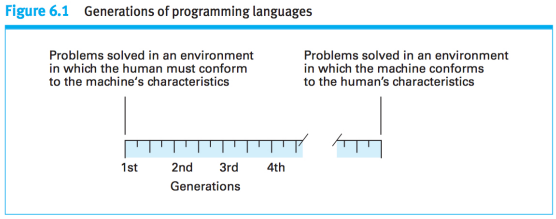
Programming Language: What Is It and Why Study It?

- "Programming languages provide the abstractions, organizing principles, and control structures that programmers use to write good programs." [Mitchell, 2003]
- "Programming languages are the medium of expression in the art of computer programming. An ideal programming language will make it easy for programmers to write programs succinctly and clearly. Because programs are meant to be understood, modified, and maintained over their lifetime, a good programming language will help others read programs and understand how they work." [Mitchell, 2003]
- "A good language for large-scale programming will help programmers manage the interaction among software components effectively." [Mitchell, 2003]

Programming Languages: Generations

- The first-generation languages: Machine languages
- The second-generation languages: Assembly languages
- The third-generation languages: High-level languages
- The fourth-generation languages: ……

Figure 6.1 Generations of programming languages



Assembly Language: What Is It ?

Assembly Language: What Is It ?

Machine Code Program Example [F-CS-18]

Table 9.1 Code in machine language to add two integers

Hexadecimal	*Code in machine language*			
$(1FEF)_{16}$	0001	1111	1110	1111
$(240F)_{16}$	0010	0100	0000	1111
$(1FEF)_{16}$	0001	1111	1110	1111
$(241F)_{16}$	0010	0100	0001	1111
$(1040)_{16}$	0001	0000	0100	0000
$(1141)_{16}$	0001	0001	0100	0001
$(3201)_{16}$	0011	0010	0000	0001
$(2422)_{16}$	0010	0100	0010	0010
$(1F42)_{16}$	0001	1111	0100	0010
$(2FFF)_{16}$	0010	1111	1111	1111
$(0000)_{16}$	0000	0000	0000	0000

Machine language is the only language understood by the computer hardware, which is made of electronic switches with two states: off (representing 0) and on (representing 1).

The only language understood by a computer is machine language.

Assembly Program Example (Fig 2.2, Fig 2.7)

As a more extensive example, the machine language routine

```
156C
166D
5056
306E
C000
```

which adds the contents of memory cells 6C and 6D and stores the result at location 6E (Figure 2.7 of Chapter 2) might be expressed as

```
LD R5,Price
LD R6,ShippingCharge
ADDI R0,R5 R6
ST R0,TotalCost
HLT
```

using mnemonics. (Here we have used LD, ADDI, ST, and HLT to represent *load*, *add*, *store*, and *halt*. Moreover, we have used the descriptive names `Price`, `ShippingCharge`, and `TotalCost` to refer to the memory cells at locations 6C, 6D, and 6E, respectively. Such descriptive names are often called **identifiers.**) Note that the mnemonic form, although still lacking, does a better job of representing the meaning of the routine than does the numeric form.

Assembly Program Example [F-CS-18]

Table 9.2 Code in assembly language to add two integers

Code in assembly language				*Description*
LOAD	RF	Keyboard		Load from keyboard controller to register F
STORE	Number1	RF		Store register F into Number1
LOAD	RF	Keyboard		Load from keyboard controller to register F
STORE	Number2	RF		Store register F into Number2
LOAD	R0	Number1		Load Number1 into register 0
LOAD	R1	Number2		Load Number2 into register 1
ADDI	R2	R0	R1	Add registers 0 and 1 with result in register 2
STORE	Result	R2		Store register 2 into Result
LOAD	RF	Result		Load Result into register F
STORE	Monitor	RF		Store register F into monitor controller
HALT				Stop

A special program called an **assembler** is used to translate code in assembly language into machine language.

Machine Languages and Assembly Languages

- **Problems in using machine languages**
  - **Fallible: Programming in any machine language is fallible.**
  - **Inefficient: Programming in any machine language is inefficient.**
  - **Unreadable and hard-rememberable: Any machine program is unreadable and hard-rememberable.**
- **Assembly languages and assemblers**
  - ***Assembly languages*** **assign mnemonic letter codes to each machine-language instruction. Programmers use these letter codes in place of binary digits.**
  - **Because every program that is executed on a computer eventually must be in the form of the computer's machine language, a program called an *assembler* reads each of the instructions in mnemonic form and translates it into the machine-language equivalent.**

Assembly Language and Assembler [DL-CS-16]

Assembly language A low-level programming language in which a mnemonic represents each of the machine-language instructions for a particular computer

Assembler A program that translates an assembly-language program in machine code

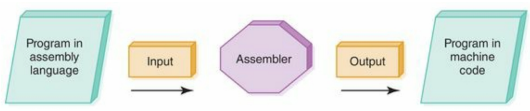


FIGURE 6.5 Assembly process

Assembly Languages: Advantages and Disadvantages

- Advantages of assembly languages
  - Assembly languages are better than machine languages on fallibility, efficiency, readability, and rememberability.
- Disadvantages of assembly languages
  - Programming in assembly languages are still fallible, and inefficient, and assembly programs are still unreadable and hard-rememberable.
  - Each assembly (machine) language is specific to a particular computer architecture but is not generally portable across multiple computer architectures. Any assembly (machine) program must be inherently machine dependent (unportable).
  - A programmer is still forced to think in terms of the small, incremental steps of the machine's language.

High-Level Programming Language: What Is It ?

High-Level Programming Language: What Is It ?

A Philosophy on Design

- Primitives in construction and primitives in design
  - The elementary primitives in which a product must ultimately be constructed are not necessarily the primitives that should be used during the product's design.
- Primitives in design process
  - The design process is better suited to the use of high-level primitives, each representing a concept associated with a major feature of the product.
- Primitives in implementation
  - Once the design is complete, these primitives can be (sometime, automatically) translated to lower-level concepts relating to the details of implementation.

High-Level Programming Languages

- Why high-level programming languages are necessary?
  - In order to provide high-level abstraction (high-level primitives).
  - In order to provide program portability (machine-independency).
- The third-generation languages
  - FORTRAN (FORmula TRANslator): Developed by IBM for scientific and engineering applications (1957).
  - COBOL (COmmon Business-Oriented Language): Developed by the U.S. Navy for business applications (1959).
  - ALGOL, APL, PL/I, BASIC, C, Pascal, Ada (83, 95, 2005, 2012), Java, ……

High-Level Programming Languages

- The third-generation (imperative) languages
  - The third generation of programming languages differed from previous generations in that their primitives were both higher level (in that they expressed instructions in larger increments) and machine independent (in that they did not rely on the characteristics of a particular machine).
  - In general, the approach to third-generation languages was to identify a collection of high-level primitives in which software could be developed.
- Other high-level programming languages
  - Functional programming languages: LISP, ML, Scheme, ...
  - Logic programming languages: Prolog, ...

High-Level Programming Languages and Compilers [DL-CS-16]

Compiler A program that translates a high-level language program into machine code

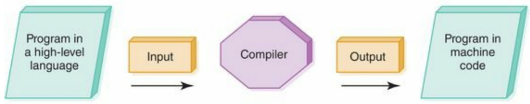


FIGURE 9.2 Compilation process

High-Level Programming Languages and Interpreters [DL-CS-16]

Interpreter A program that inputs a program in a high-level language and directs the computer to perform the actions specified in each statement

♣ **Interpreter**

- ◆ An interpreter is a program that translates and executes the statements in sequence.
- ◆ Interpreters can be viewed as simulators or virtual machines that "understand" the language in which a program is written.

High-Level Programming Languages and Interpreters [DL-CS-16]

♣ **The difference between interpreter and translator (assembler or compiler)**

- ◆ Unlike an assembler or compiler that produces machine code as output, which is then executed in a separate step, an interpreter translates a statement and then immediately executes the statement.
- ◆ The translator (assembler or compiler) simply produces an equivalent program in the appropriate machine language, which must then be run. The interpreter executes the input program directly.

High-Level Programming Languages and Interpreters [F-CS-18]

♣ **Interpretation**

- ◆ Interpretation refers to the process of translating each line of the source program into the corresponding line of the object program and executing the line.
- ◆ There are two trends in interpretation: that used by some languages before Java (1996) and the interpretation used by Java.

♣ **First approach to interpretation**

- ◆ Each line of the source program is translated into the machine language of the computer being used and executed immediately.
- ◆ If there are any errors in translation and execution, the process displays an error message and the rest of the process is aborted.

High-Level Programming Languages and Interpreters [F-CS-18]

♣ **Second approach to interpretation**

- ◆ To achieve portability, the translation of the source program to object program is done in two steps: compilation and interpretation.
- ◆ A Java source program is first compiled to create *Java bytecode*, which looks like code in a machine language, but is not the object code for any specific computer: it is the object code for a virtual machine, called the *Java Virtual Machine* or *JVM*.
- ◆ The bytecode then can be compiled or interpreted by any computer that runs a JVM emulator, i.e., the computer that runs the bytecode needs only a JVM emulator, not the Java compiler.

High-Level Programming Languages and Interpreters [DL-CS-16]

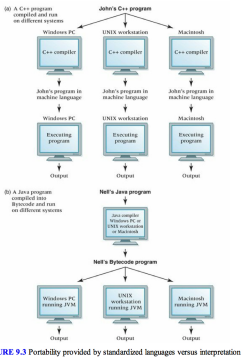


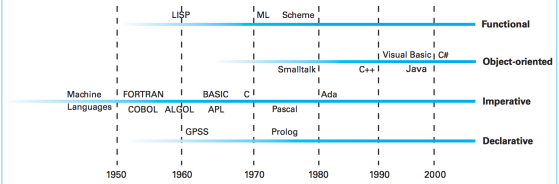
FIGURE 9.3 Portability provided by standardized languages versus interpretation

Various High-Level Programming Languages

Various High-Level Programming Languages

Programming Languages: Evolution

Figure 6.2 The evolution of programming paradigms



Programming Languages: Evolution [Sebesta-16]

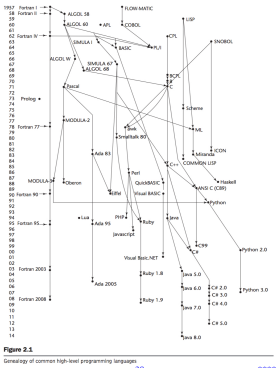
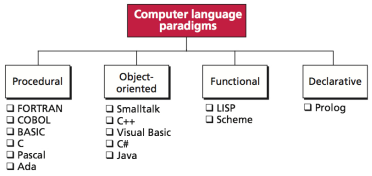


Figure 2.1
Genealogy of common high-level programming languages

Programming Languages: Categories [F-CS-18]

Figure 9.2 *Categories of programming languages*



Programming (Software Development) Paradigms

♣ **The imperative (procedural) paradigm**

- The imperative paradigm defines the programming process to be the development of a sequence of commands that, when followed, manipulate data to produce the desired result.
- Following the imperative paradigm, the programming process includes finding an algorithm to solve the problem at hand and then expressing that algorithm as a sequence of commands.

♣ **The declarative paradigm**

- A declarative programming system applies a pre-established general-purpose problem-solving algorithm to solve problems presented to it.
- Following the declarative paradigm, the task of a programmer is developing a precise statement of the problem rather than of describing an algorithm for solving the problem.

The Declarative Paradigm: Functional Programming

♣ **Programs as functions**

- Under functional programming paradigm, a functional program is viewed as a function that accepts inputs and produces outputs.

♣ **Functional programming as building functions**

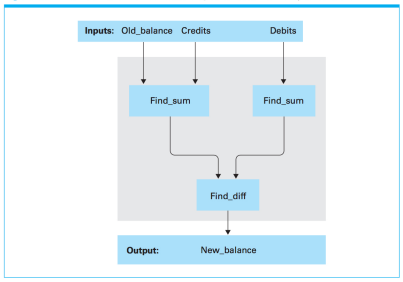
- A functional program is constructed by connecting smaller predefined program units (predefined functions) so that each unit's outputs are used as another unit's inputs in such a way that the desired overall input-to-output relationship is obtained.
- In short, the programming process under the functional paradigm is that of building functions as nested complexes of simpler functions.

The Declarative Paradigm: Functional Programming

♣ **LISP program example**

- (Find_diff (Find_sum Old_balance Credits) (Find_sum Debits))

Figure 6.3 A function for checkbook balancing constructed from simpler functions



The Declarative Paradigm: Logic Programming

♣ **Programs as logic formulas**

- Under logic programming paradigm, a logic program is written as a sequence of logic formulas that express facts and inference rules about some problem domain. Such a program is used by asking it questions, which it attempts to answer by consulting the facts and rules.
- Logic programs do not state exactly how a result is to be computed but rather describe the form of the result. Therefore, Logic programs are declarative rather than procedural in the sense that only the specifications of the desired results are stated rather than detailed procedures for producing them.

♣ **Logic programming as knowledge representation and deduction**

- Logic programming provides both the relevant information and inference method for computing desired results.

The Declarative Paradigm: Logic Programming

♣ **Prolog program example**

- The fact that a turtle is faster than a snail could be represented by the Prolog statement
  faster (turtle, snail),
  and the fact that a rabbit is faster than a turtle could be represented by
  faster (rabbit, turtle).
- The inference rule (faster (X, Y) AND faster (Y, Z)) → faster (X, Z) would be expressed in Prolog as
  faster (X, Z) :- faster (X, Y), faster (Y, Z).
- For the program, ask question "faster (rabbit, snail)?" will get result "YES"; ask question "faster (W, snail)?" will get results "faster (turtle, snail)" and "faster (rabbit, snail)".

The Object-Oriented Programming (Software Development) Paradigm

♣ **The object-oriented paradigm**

- Following this paradigm, a software system is viewed as a collection of units, called *objects*, each of which is capable of performing the actions that are immediately related to itself as well as requesting actions of other objects.
- These objects interact together to solve the problem.

♣ **Objects and methods**

- Each of these objects would encompass a collection of procedures (called *methods*) describing how that object is to respond to the occurrence of various events.
- Thus the entire system would be constructed as a collection of objects, each of which knows how to respond to the events related to it.

The Object-Oriented Programming (Software Development) Paradigm

♣ **Classes and instances**

- An object can consist of data together with a collection of methods for performing activities on the data. These features must be described by statements in the written program. This description of the object's properties is called a *class*.
- Once a class has been constructed, it can be applied anytime an object with those characteristics is needed. Thus, several objects can be based on the same class.
- An object that is based on a particular class is said to be an *instance* of that class.

The Object-Oriented Programming (Software Development) Paradigm

♣ **Object-oriented paradigm and imperative paradigm**

- The methods within an object are essentially small imperative program units.
- This means that most programming languages based on the object-oriented paradigm contain many of the features found in imperative languages.

Programming Language Concepts

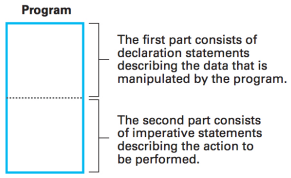
Programming Language Concepts

Programming Language Concepts: Statements

- **Declarative statements**
  - ***Declarative statements*** **define customized terminology that is used later in the program, such as the names used to reference data items.**
- **Imperative statements**
  - ***Imperative statements*** **(assignment statements and control statements) describe computational execution steps in the underlying algorithms.**
- **Comments**
  - ***Comments*** **enhance the readability of a program by explaining its esoteric features in a more human-compatible form.**

Programming Language Concepts: Statements

Figure 6.4 The composition of a typical imperative program or program unit



Declaration A statement that associates an identifier with a variable, an action, or some other entity within the language that can be given a name so that the programmer can refer to that item by name

Programming Language Concepts: Variables

- **Variables**
  - **High-level programming languages allow locations in main memory to be referenced by descriptive names rather than by numeric addresses.**
  - **Such a name is known as a *variable (name)*, in recognition of the fact that by changing the value stored at the location, the value associated with the name changes as the program executes (by assignment statements).**
- **Notes**
  - **Variables in programs are intrinsically different from variables in mathematics.**
  - **In programs, variables are names of memory "boxes".**

Programming Language Concepts: Data Types

- **Data types**
  - **Variables be identified via a declarative statement prior to being used elsewhere in the program.**
  - **These declarative statements also require that the programmer describe the type of data that will be stored at the memory location associated with the variable.**
  - **Such a type is known as a *data type* and encompasses both the manner in which the data item is encoded and the operations that can be performed on that data.**
- **Note**
  - **To define data type accurately is an important means to improve the quality of programs.**

Programming Language Concepts: Constants and Literals

- **Constants**
  - **High-level programming languages allow descriptive names to be assigned to specific, non-changeable values. Such a name is called a *constant*.**
- **Literals**
  - **An explicit appearance (representation) of a value is called a *literal*.**
- **Notes**
  - **To use constants rather than literals is a good programming practice.**
  - **The use of literals is not good programming practice because literals can mask the meaning of the statements in which they appear.**

Programming Language Concepts: Expressions

- **Arithmetic expressions**
  - **An arithmetic expression is a sequence of identifiers, separated by arithmetic operators, that evaluates to a numerical value.**
- **Logic (Boolean) expressions**
  - **A logic (Boolean) expression is a sequence of identifiers, separated by compatible operators, that evaluates to either true or false.**
  - **A Boolean expression can be any of the following:**
    A Boolean variable,
    An arithmetic expression followed by a relational operator followed by an arithmetic expression, and
    A Boolean expression followed by a logic (Boolean) operator followed by a Boolean expression.

Programming Language Concepts: Relational Operators

♣ **Relational operators**

- ◆ A relational operator is one that compares two numerical values.
- ◆ A relational operator between two arithmetic expressions is asking if the relationship exists between the two expressions.

Programming Language Concepts: Relational Operators

Symbol	Meaning	Example	Evaluation
<	Less than	Number1 < Number2	True if Number1 is less than Number2; false otherwise
<=	Less than or equal	Number1 <= Number2	True if Number1 is less than or equal to Number2; false otherwise
>	Greater than	Number1 > Number2	True if Number1 is greater than Number2; false otherwise
>=	Greater than or equal	Number1 >= Number2	True if Number1 is greater than or equal to Number2; false otherwise
!= or <> or /=	Not equal	Number1 != Number2	True if Number1 is not equal to Number2; false otherwise
= or ==	Equal	Number1 == Number2	True if Number1 is equal to Number2; false otherwise

Programming Language Concepts: Assignment Statements

♣ **Assignment statement**

- ◆ The most basic imperative statement is the ***assignment statement***, which requests that a value be assigned to a variable (or more precisely, stored in the memory area identified by the variable).

♣ **The syntax and semantics of assignment statement**

- ◆ The assignment statement normally takes the syntactic form of a variable, followed by a symbol representing the assignment operation, and then by an expression indicating the value to be assigned.
- ◆ X := arithmetic/logic expression
- ◆ The semantics of such a statement is that the expression is to be evaluated and the result stored as the value of the variable.
- ◆ For examples, Z := X + Y; Z := X and Y.

Programming Language Concepts: Control Statements

♣ **If statement**

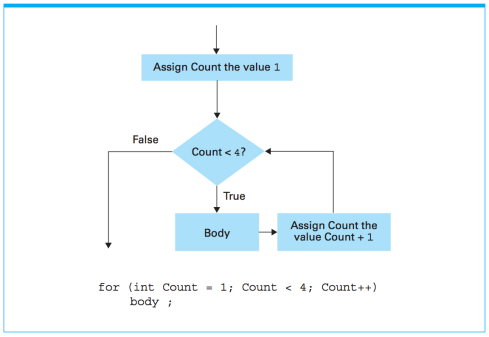
- ◆ if (condition) statementA
  else statementB; (in C, C++, C#, and Java)
- ◆ IF condition THEN
  statementA;
  ELSE statementB;
  END IF; (in Ada)

♣ **While statement**

- ◆ while (condition)
  {loop body} (in C, C++, C#, and Java)
- ◆ WHILE condition LOOP
  loop body
  END LOOP; (in Ada)

Programming Language Concepts: For-loop Statement

Figure 6.7 The for loop structure and its representation in C++, C#, and Java

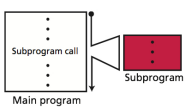


Programming Language Concepts: Subprograms

♣ **Subprogram**

- ◆ We can give a section of code, called a ***subprogram***, a name and then use that name as a statement in another part of the ***main program*** (also called the ***calling unit***).
- ◆ When the name is encountered, the processing in the calling unit is suspended while the named code section executes.
- ◆ When the named code section finishes executing, processing resumes with the statement just below where the name occurred.

Figure 9.11 *The concept of a subprogram*



Programming Language Concepts: Subprograms [DL-CS-16]

- **Two basic forms of subprogram**
  - ***Void subprograms (procedures)*: named code section that does a particular task; used as a statement in the calling unit.**
  - ***Value-returning subprograms (functions)*: named code section that also does a task but returns a single value to the calling unit; used in an expression in the calling unit where the returned value is then used in the evaluation of the expression.**
- **Subprograms as powerful tools for abstraction**
  - **The listing of a named subprogram allows the reader of the program to see that a task is being done without having to be bothered with the details of the task's implementation.**

Programming Language Concepts: Subprograms [DL-CS-16]

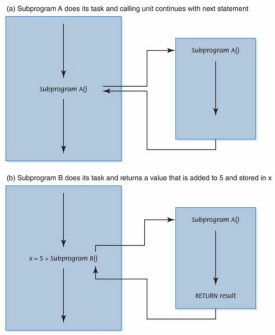


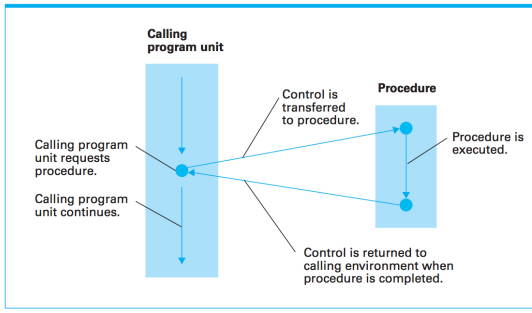
FIGURE 7.14 Subprogram flow of control

Programming Language Concepts: Procedural Units

- **Procedure**
  - **A *procedure* is a set of instructions for performing a task that can be used as an abstract tool by other program units.**
  - **Control is transferred to the procedure at the time its services are required and then returned to the original program unit after the procedure has finished.**
  - **The process of transferring control to a procedure is often referred to as *calling* or *invoking* the procedure. A program unit that requests the execution of a procedure is called the *calling unit*.**
- **Function**
  - **A *function* is a program unit similar to a procedure except that a value is transferred back to the calling program unit as "the value of the function."**

Programming Languages Concepts: Calling a Procedure

Figure 6.8 The flow of control involving a procedure



Programming Language Concepts: Parameters

- **Parameters**
  - **If a subprogram needs data to execute its task, we place the name of the data value (as generic terms) in parentheses by the subprogram heading.**
  - **Such generic terms within procedures are called *parameters*.**
- **Formal and actual parameters**
  - **The generic terms used within the subprogram are called *formal parameters* and the precise meanings assigned to these formal parameters when the subprogram is applied are called *actual parameters*.**
  - **Then, the values of the actual parameters are effectively transferred to their corresponding formal parameters, and the subprogram is executed .**

Programming Language Concepts: Parameters

- **Transferring data between actual and formal parameters**
  - **The task of transferring data between actual and formal parameters is handled in a variety of ways by different programming languages.**
- **Calling (Passing) by value**
  - **A duplicate of the data represented by the actual parameters is produced and given to the subprogram.**
  - **Using this approach, any alterations to the data made by the subprogram are reflected only in the duplicate, the data in the calling program unit are never changed.**
  - **Passing parameters by value protects the data in the calling unit from being mistakenly altered by a poorly designed subprogram.**

Programming Language Concepts: Parameters

♣ Calling (Passing) by reference

- To give the subprogram direct access to the actual parameters by telling it the addresses of the actual parameters in the calling program unit.
- Passing parameters by reference allows the subprogram to modify the data residing in the calling environment.

Programming Language Concepts: Parameters

♣ Example

- procedure Demo (Formal)
  Formal := Formal + 1;
- Suppose that the variable Actual was assigned the value 5 and we called Demo with the statement Demo (Actual).

♣ Calling (Passing) by value

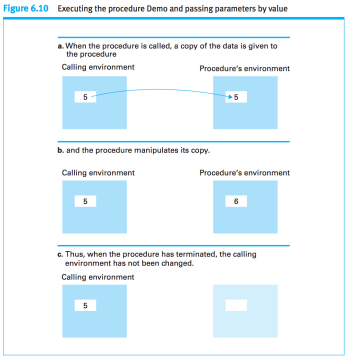
- If parameters were passed by value, the change to Formal in the procedure would not be reflected in the variable Actual (Figure 6.10).

♣ Calling (Passing) by reference

- If parameters were passed by reference, the value of Actual would be incremented by one (Figure 6.11).

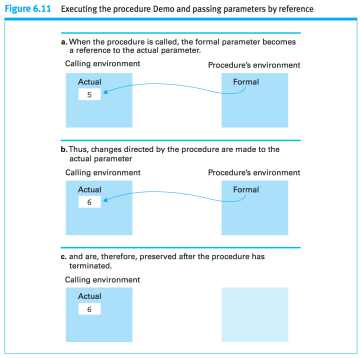
Programming Languages Concepts: Calling by Value

Figure 6.10 Executing the procedure Demo and passing parameters by value



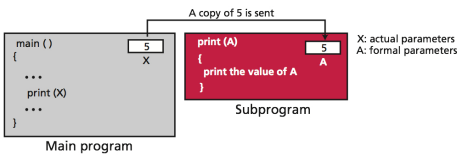
Programming Languages Concepts: Calling by Reference

Figure 6.11 Executing the procedure Demo and passing parameters by reference



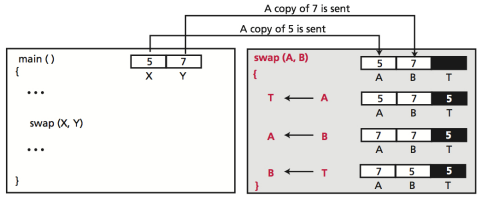
Programming Language Concepts: Parameters [F-CS-18]

Figure 9.12 *An example of pass by value*



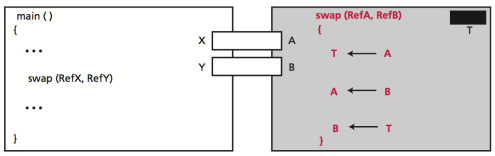
Programming Language Concepts: Parameters [F-CS-18]

Figure 9.13 *An example in which pass by value does not work*



Programming Language Concepts: Parameters [F-CS-18]

Figure 9.14 *An example of pass by reference*



X and Y: actual parameters
A and B: formal parameters
T: local (temporary) parameters

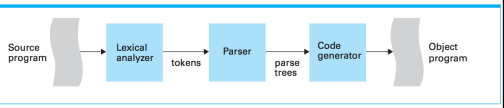
Programming Language Implementation: Compilation

Programming Language Implementation: Compilation

Programming Language Implementation: Compilation

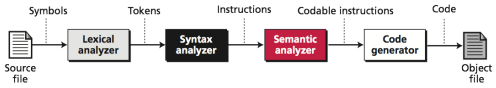
- **Program compilation (translation)**
  - The process of converting a program from one description (representation) language to another is called *compilation* (*translation*).
  - The program in its original form is called the *source program*, and the translated version is called the *object program*.
  - The translation process consists of three activities: *lexical analysis*, *parsing*, and *code generation*, that are performed by three units in the compiler (translator) known as the *lexical analyzer*, *parser*, and *code generator*.

Figure 6.13 The translation process



Source Code Translation Process [F-CS-18]

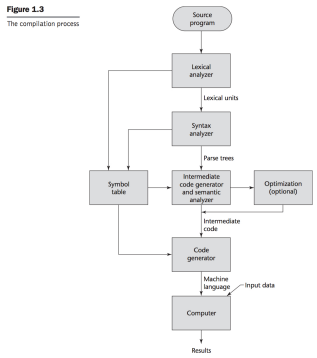
Figure 9.1 *Source code translation process*



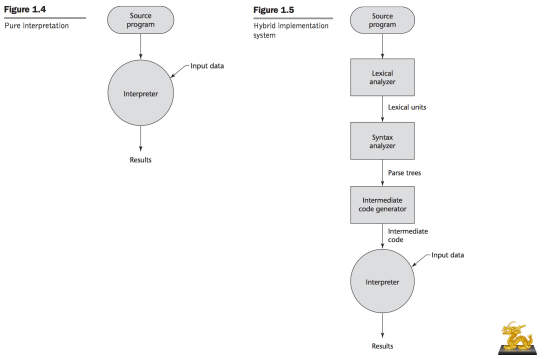
Source Code Translation Process [F-CS-18]

- **The lexical analyzer**
  - A lexical analyzer reads the source code, symbol by symbol, and creates a list of *tokens* in the source language.
- **The syntax analyzer**
  - The syntax analyzer parses a set of tokens to find (well formed) instructions.
- **The semantic analyzer**
  - The semantic analyzer checks the sentences created by the syntax analyzer to be sure that they contain no ambiguity.
- **The code generator**
  - After unambiguous instructions are created by the semantic analyzer, each instruction is converted to a set of machine instructions for the computer on which the program will run. This is done by the code generator.

The Compilation Process [Sebesta-16]



The Interpretation Process [Sebesta-16]



Programming Language Implementation: Lexical Analysis

♣ **Lexical analysis**

- Lexical analysis is the process of recognizing which strings of symbols from the source program represent a single entity called a *token*.
- The lexical analyzer reads the source program symbol by symbol, identifying which groups of symbols represent tokens, and classifying those tokens according to whether they are numeric values, words, arithmetic operators, and so on.
- The lexical analyzer encodes each token with its classification and hands them to the parser.

Programming Language Implementation: Parsing

♣ **Parsing**

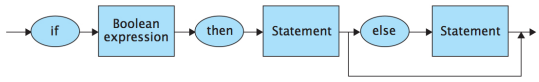
- The parser views the program in terms of lexical units (tokens) rather than individual symbols.
- It is the parser's job to group these units into statements.
- Indeed, parsing is the process of identifying the grammatical structure of the program and recognizing the role of each component.

♣ **Grammar and syntax diagram**

- The parsing process is based on a set of rules that define the syntax of the programming language. These rules are called a *grammar*.
- One way of expressing these rules is by means of *syntax diagrams*, which are pictorial representations of a language's grammatical structure.

Programming Language Implementation: Syntax Diagrams

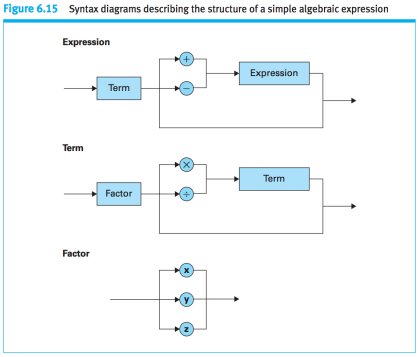
Figure 6.14 A syntax diagram of our if-then-else pseudocode statement



- Terms that appear in ovals are called *terminals*.
- Terms that require further description (those in rectangles) are called *non-terminals*; in a complete description of a language's syntax the non-terminals are described by additional diagrams.

Programming Language Implementation: Syntax Diagrams

Figure 6.15 Syntax diagrams describing the structure of a simple algebraic expression



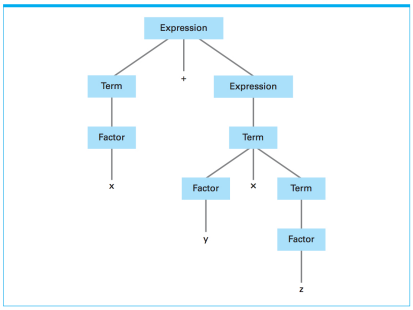
Programming Language Implementation: Parse Trees

♣ **Parse tree**

- A particular string conforms to a set of syntax diagrams can be represented in a pictorial form by a *parse tree*.
- The process of parsing a program is essentially that of constructing a parse tree for the source program.
- Indeed, a parse tree represents the parser's interpretation of the program's grammatical composition.
- For this reason the syntax rules describing a program's grammatical structure must not allow two distinct parse trees for one string, since this would lead to ambiguities within the parser.

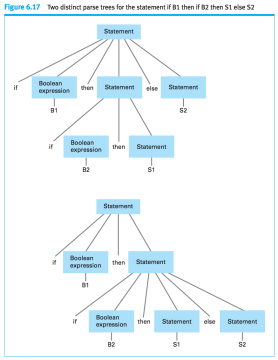
Programming Language Implementation: Parse Trees

Figure 6.16 The parse tree for the string x + y + z based on the syntax diagrams in Figure 6.15



Programming Language Implementation: Parse Trees

Figure 6.17 Two distinct parse trees for the statement if B1 then if B2 then S1 else S2



Programming Language Implementation: Symbol Table

♣Symbol table

- As a parser analyzes the grammatical structure of a program, it is able to identify individual statements and to distinguish between the declarative statements and imperative statements.
- As it recognizes the declarative statements, it records the information being declared in a table called the ***symbol table***.
- Thus the symbol table contains such information as the names of the variables appearing in the program as well as what data types and data structures are associated with those variables.

Programming Language Implementation: Code Generation

♣Code generation

- The final activity in the compilation (translation) process is code generation, which is the process of constructing the machine-language instructions to implement the statements recognized by the parser such that the machine instruction program (object program) can be really executed on the target computer.
- This process involves numerous issues; the most important one, of course, is to generate correct codes; the other important one is to produce efficient machine-language versions of programs.

The Process of Compiling C Programs [TB-OS-15]

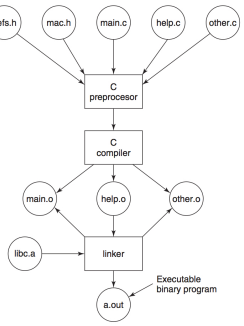


Figure 1-30. The process of compiling C and header files to make an executable.

Important Programming Languages

Important Programming Languages

Programming Language C (and Unix)

- The C programming language and UNIX operating system
  - CPL --> BCPL(M. Richards, Cambridge U, 1967) --> B(K.L. Thompson, AT&T Bell Lab, 1970) --> C(D.M. Richie, AT&T Bell Lab, 1972).
  - UNIX in B(K.L. Thompson, AT&T Bell Lab, 1960s-1970) --> UNIX in C(K. Thompson & D.M. Richie, AT&T Bell Lab, 1973)
  - K.L. Thompson & D.M. Richie, 1983 Turing award
- C
  - The first system software programming language.
  - The most efficient programming language.
  - The most widely used programming language.
  - The most easy programming language.
  - The "most unreliable" programming language.

Programming Language Ada

- The Ada programming language
  - Ada 83 is the first programming language such that a completely separate group has defined the requirements for the programming language to be designed.
  - Ada 83 is the first internationally standardized programming language to include concurrency as an intrinsic feature in the language.
  - Ada 95 is the first object-oriented programming language which has been internationally standardized.
  - Ada 2012 is the world's premier programming language for engineering safe, secure and reliable software.
  - Ada 2012 is the most powerful programming language.

Facts about Ada: Commercial Aviation

- Boeing 777 Project: "Working Together with Ada"
  - 99% programs of Boeing 777 are written in Ada.
  - "Working Together" is the project name Boeing chose in 1990 when it first entertained the idea of producing its 777.
- Commercial Aviation
  - Airbus 320, 330, 340; Airbus 380; Beechjet 400A (US business jet); Beech Starship I (US business turboprop); Beriev BE-200 (Russian forest fire patrol plane); Boeing 737-200, -400, -500, -600, -700, -800; Boeing 747-400; Boeing 757, 767; Boeing 777; Boeing 787; Boeing 787 Air Conditioning Control Unit; Canadair Regional Jet; Embraer CBA-123 and CBA-145 (Brazilian-made regional airliners); Fokker F-100 (Dutch DC-9-size airliner - American Airlines flies these); Ilyushin 96M (Russian jetliner); Lockheed-Martin Hercules "hurricane chaser"; Saab 2000; Tupolev TU-204 (Russian jetliner)

Facts about Ada: Air Traffic Management Systems

- Australia,
- Belgium, Brazil,
- Canada, China, Czech Republic,
- Denmark,
- Finland, France,
- Germany, Greece,
- Hong Kong, Hungary,
- India, Ireland,
- Kenya,
- Netherlands, New Zealand,
- Pakistan,

Facts about Ada: Air Traffic Management Systems

- Scotland, Singapore, South Africa, Spain, Sweden,
- Taiwan,
- United Kingdom, UK New En-Route Center (NERC),
- United States FAA En Route Automation Modernization (ERAM), United States FAA Conflict Detection Tool (URET), United States FAA Air Route Traffic Control Centers Display System Replacement (DSR),
- Vietnam,
- European Air Traffic Flow Management

Augusta Ada Byron: The First Programmer in the World

Ada Augusta, the First Programmer[2]

On December 10, 1815 (the same year that George Boole was born), a daughter—Augusta Ada Byron—was born to Anna Isabella (Annabella) Byron and George Gordon, Lord Byron. At that time in England Byron's fame derived not only from his poetry but also from his wild and scandalous behavior. The marriage was strained from the beginning, and Annabella left Byron shortly after Ada's birth. By April of 1816, the two had signed separation papers. Byron left England, never to return. Throughout the rest of his life he regretted that he was unable to see his daughter. At one point he wrote of her,

I see thee not. I hear thee not.
But none can be so wrapt in thee.

Before he died in Greece, at age 36, he exclaimed,

Oh my poor dear child! My dear Ada!
My God, could I but have seen her!

Meanwhile, Annabella, who eventually was to become a baroness in her own right, and who was educated as both a mathematician and a poet, carried on with Ada's upbringing and education. Annabella gave Ada her first instruction in mathematics, but it soon became clear that Ada was gifted in the subject and should receive more extensive tutoring. Ada received further training from Augustus DeMorgan, today famous for one of the basic theorems of Boolean algebra. By age eight, Ada had demonstrated an interest in mechanical devices and was building detailed model boats.

When she was 18, Ada visited the Mechanics Institute to hear Dr. Dionysius Lardner's lectures on the "Difference Engine," a mechanical calculating machine being built by Charles Babbage. She became so interested in the device that she arranged to be introduced to Babbage. It was said that, upon seeing Babbage's machine, Ada was the only person in the room to understand immediately how it worked and to recognize its significance. Ada and Charles Babbage became lifelong friends. She worked with him, helping to document his designs, translating writings about his work, and developing programs for his machines. In fact, Ada today is recognized as the first computer programmer in history.

When Babbage designed his Analytical Engine, Ada foresaw that it could go beyond arithmetic computations and become a general manipulator of symbols, and thus would have far-reaching capabilities. She even suggested that such a device eventually could be programmed with rules of harmony and composition so that it could be produce "scientific" music. In effect, Ada foresaw the field of artificial intelligence more than 150 years ago.

In 1842, Babbage gave a series of lectures in Turin, Italy, on his Analytical Engine. One of the attendees was Luigi Menabrea, who was so impressed that he wrote an account of Babbage's lectures. At age 27, Ada decided to translate the account into English, with the intent to add a few of her own notes about the machine. In the end, her notes were twice as long as the original material, and the document, "The Sketch of the Analytical Engine," became the definitive work on the subject.

It is obvious from Ada's letters that her "notes" were entirely her own and that Babbage was acting as a sometimes unappreciated editor. At one point, Ada wrote to him,

I am much annoyed at your having altered my Note. You know I am always willing to make any required alterations myself, but that I cannot endure another person to meddle with my sentences.

Ada gained the title Countess of Lovelace when she married Lord William Lovelace. The couple had three children, whose upbringing was left to Ada's mother while Ada pursued her work in mathematics. Her husband was supportive of her work, but for a woman of that day such behavior was considered almost as scandalous as some of her father's exploits.

Ada died in 1852, just one year before a working Difference Engine was built in Sweden from one of Babbage's designs. Like her father, Ada lived only to age 36, and even though they led very different lives, she undoubtedly admired him and took inspiration from his unconventional and rebellious nature. In the end, Ada asked to be buried beside him at the family's estate.

Some Other Programming Languages [F-CS-18]

♣C++

- The C++ language was developed by Bjarne Stroustrup at Bell Laboratory as an improvement of the C language.
- It uses classes to define the general characteristics of similar objects and the operations that can be applied to them.
- Three principles were used in the design of the C++ language: encapsulation, inheritance, and polymorphism.

♣Java

- Java was developed at Sun Microsystems, Inc.
- It is based on C and C++, but some features of C++ are removed to make the language more robust; it is totally class-oriented.

Some Other Programming Languages [F-CS-18]

♣LISP

- LISP (LISt Programming) was designed by a team of researchers at MIT in the early 1960s.
- It is a functional programming language in which everything is considered a list.

♣Scheme

- The Scheme language (developed at MIT in the early 1970s) defines a set of primitive functions that solves problems.
- The function name and the list of inputs to the function are enclosed in parentheses.
- The result is an output list, which can be used as the input list to another function.

Some Other Programming Languages

♣SPACK

- SPARK is a formally defined programming language based on the Ada language, intended for the development of high integrity software used in systems where predictable and highly reliable operation is essential; it facilitates the development of applications that demand safety, security, or business integrity.
- SPARK 2014, based on Ada 2012, was released on April 30, 2014.
- The SPARK language consists of a well-defined subset of the Ada language that uses contracts to describe the specification of components in a form that is suitable for both static and dynamic verification.

Some Other Programming Languages

♣Python

Python

Python is a programming language that was created by Guido van Rossum in the late 1980s. Today it is popular in developing Web applications, in scientific computation, and as an introductory language for students. Python emphasizes readability, and includes elements of the imperative, object-oriented, and functional programming paradigms. Python is also an example of a modern language that uses a form of fixed formatting. It uses indentation to denote program blocks, rather than punctuation marks or reserved words.

An Introduction to Computer Science

- Computer Science: What Is It and Why Study It?
- Computation: What Is It and Why Study It?
- Computability
- Computational Complexity (CS101A class only)
- Algorithms
- Data, Information, and Knowledge, and Their Representations
- Data Storage
- Computer Architecture
- Data Manipulation in Computer Systems
- Programming Languages and Compilers
- Operating Systems
- System Software and Application Software
- Software Engineering (CS101A class only)
- Knowledge Engineering and Artificial Intelligence (CS101A class only)
- Information Security Engineering (CS101A class only)