Chapter 1

Basic Principles of Programming Languages

KH ALL RIGHTS RESERVED

Although there exist many programming languages, the differences among them are insignificant compared to the differences among natural languages. In this chapter, we discuss the common aspects shared among different programming languages. These aspects include:

- programming paradigms that define how computation is expressed;
- the main features of programming languages and their impact on the performance of programs written in the languages;
- a brief review of the history and development of programming languages;
- the lexical, syntactic, and semantic structures of programming languages, data and data types, program processing and preprocessing, and the life cycles of program development.

At the end of the chapter, you should have learned:

- what programming paradigms are;
- an overview of different programming languages and the background knowledge of these languages;
- the structures of programming languages and how programming languages are defined at the syntactic level;
- data types, strong versus weak checking;
- the relationship between language features and their performances;
- the processing and preprocessing of programming languages, compilation versus interpretation, and different execution models of macros, procedures, and inline procedures;
- the steps used for program development: requirement, specification, design, implementation, testing, and the correctness proof of programs.

The chapter is organized as follows. Section 1.1 introduces the programming paradigms, performance, features, and the development of programming languages. Section 1.2 outlines the structures and design issues of programming languages. Section 1.3 discusses the typing systems, including types of variables, type equivalence, type conversion, and type checking during the compilation. Section 1.4 presents the preprocessing and processing of programming languages, including macro processing, interpretation, and compilation. Finally, Section 1.5 discusses the program development steps, including specification, testing, and correctness proof.

1.1 Introduction

1.1.1 Programming concepts and paradigms

Millions of programming languages have been invented, and several thousands of them are actually in use. Compared to natural languages that developed and evolved independently, programming languages are far more similar to each other. This is because

- different programming languages share the same mathematical foundation (e.g., Boolean algebra, logic);
- they provide similar functionality (e.g., arithmetic, logic operations, and text processing);
- they are based on the same kind of hardware and instruction sets;
- they have common design goals: find languages that make it simple for humans to use and efficient for hardware to execute;
- designers of programming languages share their design experiences.

Some programming languages, however, are more similar to each other, while other programming languages are more different from each other. Based on their similarities or the paradigms, programming languages can be divided into different classes. In programming language's definition, **paradigm** is a set of basic principles, concepts, and methods for how a computation or algorithm is expressed. The major paradigms we will study in this text are imperative, object-oriented, functional, and logic paradigms.

The **imperative**, also called the **procedural**, programming paradigm expresses computation by fully-specified and fully-controlled manipulation of named data in a stepwise fashion. In other words, data or values are initially stored in variables (memory locations), taken out of (read from) memory, manipulated in ALU (arithmetic logic unit), and then stored back in the same or different variables (memory locations). Finally, the values of variables are sent to the I/O devices as output. The foundation of imperative languages is the **stored program concept**–based computer hardware organization and architecture (von Neumann machine). The stored program concept will be further explained in the next chapter. Typical imperative programming languages include all assembly languages and earlier high-level languages like Fortran, Algol, Ada, Pascal, and C.

The **object-oriented** programming paradigm is basically the same as the imperative paradigm, except that related variables and operations on variables are organized into classes of **objects**. The access privileges of variables and methods (operations) in objects can be defined to reduce (simplify) the interaction among objects. Objects are considered the main building blocks of programs, which support the language features like inheritance, class hierarchy, and polymorphism. Typical object-oriented programming languages include: Smalltalk, C++, Java, and C#.

The **functional**, also called the **applicative**, programming paradigm expresses computation in terms of mathematical functions. Since we express computation in mathematical functions in many of the mathematics courses, functional programming is supposed to be easy to understand and simple to use. However, since functional programming is very different from imperative or object-oriented programming, and most programmers first get used to writing programs in imperative or object-oriented paradigms, it becomes difficult to switch to functional programming. The main difference is that there is no concept of memory locations in functional programming languages. Each function will take a number of values as input (parameters) and produce a single return value (output of the function). The return value cannot be stored for later use. It has to be used either as the final output or immediately as the parameter value of another function. Functional programming is about defining functions and organizing the return values of one or more functions as the parameters of another function. Functional programming languages are mainly based on the **lambda calculus** that will be discussed in Chapter 4. Typical functional programming languages include ML, SML, and Lisp/Scheme.

The **logic**, also called the **declarative**, programming paradigm expresses computation in terms of logic predicates. A logic program is a set of facts, rules, and questions. The execution process of a logic program is to compare a question to each fact and rule in the given fact and rulebase. If the question finds a match, we receive a*yes* answer to the question. Otherwise, we receive a *no* answer to the question. Logic programming is about finding facts, defining rules based on the facts, and writing questions to express the problems we wish to solve. Prolog is the only significant logic programming language.

KH ALL RIGHTS RESERVED

It is worthwhile to note that many languages belong to multiple paradigms. For example, we can say that C++ is an object-oriented programming language. However, C++ includes almost every feature of C and thus is an imperative programming language too. We can use C++ to write C programs. Java is more object-oriented, but still includes many imperative features. For example, Java's primitive type variables do not obtain memory from the language heap like other objects. Lisp contains many nonfunctional features. Scheme can be considered a subset of Lisp with fewer nonfunctional features. Prolog's arithmetic operations are based on the imperative paradigm.

Nonetheless, we will focus on the paradigm-related features of the languages when we study the sample languages in the next four chapters. We will study the imperative features of C in Chapter 2, the object-oriented features of C++ in Chapter 3, and the functional features of Scheme and logic features of Prolog in Chapters 4 and 5, respectively.

1.1.2 Program performance and features of programming languages

A programming language's features include orthogonality or simplicity, available control structures, data types and data structures, syntax design, support for abstraction, expressiveness, type equivalence, and strong versus weak type checking, exception handling, and restricted aliasing. These features will be further explained in the rest of the book. The performance of a program, including reliability, readability, writeability, reusability, and efficiency, is largely determined by the way the programmer writes the algorithm and selects the data structures, as well as other implementation details. However, the features of the programming language are vital in supporting and enforcing programmers in using proper language mechanisms in implementing the algorithms and data structures. Table 1.1 shows the influence of a language's features on the performance of a program written in that language. The table indicates that simplicity, control structures, data types, and data structures have significant impact on all aspects of performance. Syntax design and the support for abstraction are important for readability, reusability, writeability, and reliability. However, they do not have a significant impact on the efficiency of the program. Expressiveness supports writeability, but it may have a negative impact on the reliability of the program. Strong type checking and restricted aliasing reduce the expressiveness of writing programs, but are generally considered to produce more reliable programs. Exception handling prevents the program from crashing due to unexpected circumstances and semantic errors in the program. All language features will be discussed in this book.

Performance / Language features	Efficiency	Readability / Reusability	Writeability	Reliability
Simplicity/Orthogonality	✓	✓	✓	✓
Control structures	✓	✓	✓	✓
Typing and data structures	✓	✓	✓	✓
Syntax design		✓	✓	✓
Support for abstraction		✓	✓	✓
Expressiveness			✓	✓
Strong checking				✓
Restricted aliasing				✓
Exception handling				✓

Table 1.1. Impact of language features on the performance of the programs.

KH ALL RIGHTS RESERVED

1.1.3 Development of programming languages

The development of programming languages has been influenced by the development of hardware, the development of compiler technology, and the user's need for writing high-performance programs in terms of reliability, readability, writeability, reusability, and efficiency. The hardware and compiler limitations have forced early programming languages to be close to the machine language. Machine languages are the native languages of computers and the first generation of programming languages used by humans to communicate with the computer.

Machine languages consist of instructions of pure binary numbers that are difficult for humans to remember. The next step in programming language development is the use of mnemonics that allows certain symbols to be used to represent frequently used bit patterns. The machine language with sophisticated use of mnemonics is called **assembly language**. An assembly language normally allows simple variables, branch to a label address, different addressing modes, and macros that represent a number of instructions. An **assembler** is used to translate an assembly language program into the machine language program. The typical work that an assembler does is to translate mnemonic symbols into corresponding binary numbers, substitute register numbers or memory locations for the variables, and calculate the destination address of branch instructions according to the position of the labels in the program.

KH ALL RIGHTS RESERVED

This text will focus on introducing high-level programming languages in imperative, object-oriented, functional, and logic paradigms.

The first high-level programming language can be traced to Konrad Zuse's Plankalkül programming system in Germany in 1946. Zuse developed his Z-machines Z1, Z2, Z3, and Z4 in late 1930s and early 1940s, and the Plankalkül system was developed on the Z4 machine at ETH (Eidgenössisch Technische Hochschule) in Zürich, with which Zuse designed a chess-playing program.

The first high-level programming language that was actually used in an electronic computing device was developed in 1949. The language was named Short Code. There was no compiler designed for the language, and programs written in the language had to be hand-compiled into the machine code.

The invention of the compiler was credited to **Grace Hopper**, who designed the first widely known compiler, called **A0**, in 1951.

The first primitive compiler, called Autocoder, was written by Alick E. Glennie in 1952. It translated Autocode programs in symbolic statements into machine language for the Manchester Mark I computer. Autocode could handle single letter identifiers and simple formulas.

The first widely used language, Fortran (FORmula TRANslating), was developed by the team headed by John Backus at IBM between 1954 and 1957. Backus was also the system co-designer of the IBM 704 that ran the first Fortran compiler. Backus was later involved in the development of the language Algol and the Backus-Naur Form (BNF). BNF was a formal notation used to define the syntax of programming languages. Fortran II came in 1958. Fortran III came at the end of 1958, but it was never released to the public. Further versions of Fortran include ASA Fortran 66 (Fortran IV) in 1966, ANSI Fortran 77 (Fortran V) in 1978, ISO Fortran 90 in 1991, and ISO Fortran 95 in 1997. Unlike assembly languages, the early versions of Fortran allowed different types of variables (real, integer, array), supported procedure call, and included simple control structures.

Programs written in programming languages before the emergence of structured programming concepts were characterized as spaghetti programming or monolithic programming. **Structured programming** is a technique for organizing programs in a hierarchy of modules. Each module had a single entry and a single exit point. Control was passed downward through the structure without unconditional branches (e.g., *goto*

statements) to higher levels of the structure. Only three types of control structures were used: sequential, conditional branch, and iteration.

Based on the experience of Fortran I, Algol 58 was announced in 1958. Two years later, Algol 60, the first block-structured language, was introduced. The language was revised in 1963 and 1968. Edsger Dijkstra is credited with the design of the first Algol 60 compiler. He is famous as the leader in introducing structured programming and in abolishing the *goto* statement from programming.

Rooted in Algol, **Pascal** was developed by **Niklaus Wirth** between 1968 and 1970. He further developed Modula as the successor of Pascal in 1977, then Modula-2 in 1980, and Oberon in 1988. Oberon language had Pascal-like syntax, but it was strongly typed. It also offered type extension (inheritance) that supported object-oriented programming. In Oberon-2, type-bound procedures (like *methods* in object-oriented programming languages) were introduced.

The **C** programming language was invented and first implemented by Dennis Ritchie at DEC between 1969 and 1973, as a system implementation language for the nascent Unix operating system. It soon became one of the dominant languages at the time and even today. The predecessors of C were the typeless language BCPL (Basic Combined Programming Language) by Martin Richards in 1967 and then the B written by Ken Thompson in 1969. C had a weak type checking structure to allow a higher level of programming flexibility.

Object-oriented programming concepts were first introduced and implemented in the **Simula** language, which was designed by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center (NCC) between 1962 and 1967. The original version Simula I was designed as a language for discrete event simulation. However, its revised version, Simula 67, was a full-scale general-purpose programming language. Although Simula never became widely used, the language was highly influential on the modern programming paradigms. It introduced important object-oriented concepts like classes and objects, inheritance, and late binding.

One of the object-oriented successors of Simula was **Smalltalk**, designed at Xerox PARC, led by Alan Kay. The versions developed included Smalltalk-72, Smalltalk-74, Smalltalk-76, and Smalltalk-80. Smalltalk also inherited functional programming features from Lisp.

Based on Simula 67 and C, a language called "**C with classes**" was developed by Bjarne Stroustrup in 1980 at Bell Labs, and then revised and renamed as **C++** in 1983. C++ was considered a better C (e.g., with strong type checking), plus it supported data abstraction and object-oriented programming inherited from Simula 67.

Java was written by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems. It was called Oak at first and then renamed Java when it was publicly announced in 1995. The predecessors of Java were C++ and Smalltalk. Java removed most non-object-oriented features of C++ and was a simpler and better object-oriented programming language. Its two-level program processing concept (i.e., compilation into an intermediate bytecode and then interpretation of the bytecode using a small virtual machine) made it the dominant language for programming Internet applications. Java was still not a pure object-oriented programming language. Its primitive types, integer, floating-point number, Boolean, etc., were not classes, and their memory allocations were from the language stack rather than from the language heap.

Microsoft's **C#** language was first announced in June 2000. The language was derived from C++ and Java. It was implemented as a full object-oriented language without "primitive" types. C# also emphasizes component-oriented programming, which is a refined version of object-oriented programming. The idea is to be able to assemble software systems from prefabricated components.

KH ALL RIGHTS RESERVED

Functional programming languages are relatively independent of the development process of imperative and object-oriented programming languages. The first and the most important functional programming language, **Lisp**, short for LISt Processing, was developed by John McCarthy at MIT. Lisp was first released in 1958. Then Lisp 1 appeared in 1959, Lisp 1.5 in 1962, and Lisp 2 in 1966. Lisp was developed specifically for artificial intelligence applications and was based on the lambda calculus. It inherited its algebraic syntax from Fortran and its symbol manipulation from the Information Processing Language, or **IPL**. Several dialects of Lisp were designed later, for example, Scheme, InterLisp, FranzLisp, MacLisp, and ZetaLisp.

As a Lisp dialect, **Scheme** was first developed by G. L. Steele and G. J. Sussman in 1975 at MIT. Several important improvements were made in its later versions, including better scope rule, procedures (functions) as the first-class objects, removal of loops, and sole reliance on recursive procedure calls to express loops. Scheme was standardized by the IEEE in 1989.

Efforts began on developing a common dialect of Lisp, referred to as Common Lisp, in 1981. Common Lisp was intended to be compatible with all existing versions of Lisp dialects and to create a huge commercial product. However, the attempt to merge Scheme into Lisp failed, and Scheme remains an independent Lisp dialect today. Common Lisp was standardized by the IEEE in 1992.

Other than Lisp, John Backus's **FP** language also belongs to the first functional programming languages. FP was not based on the lambda calculus, but based on a few rules for combining function forms. Backus felt that lambda calculus's expressiveness on computable functions was much broader than necessary. A simplified rule set could do a better job.

At the same time that FP was developed in the United States, **ML** (Meta Language) appeared in the United Kingdom. Like Lisp, ML was based on lambda calculus. However, Lisp was not typed (no variable needs to be declared), while ML was strongly typed, although users did not have to declare variables that could be inferentially determined by the compiler.

Miranda is a pure functional programming language developed by David Turner at the University of Kent in 1985–86. Miranda achieves referential transparency (side effect-free) by forbidding modification to global variables. It combines the main features of **SASL** (St. Andrews Static Language) and **KRC** (Kent Recursive Calculator) with strong typing similar to that of ML. SASL and KRC are two earlier functional programming languages designed by Turner at the University of St Andrews in 1976 and at University of Kent in 1981, respectively.

There are many logic-based programming languages in existence. For example, **ALF** (Algebraic Logic Functional language) is an integrated functional and logic language based on Horn clauses for logic programming, and functions and equations for functional programming. Gödel is a strongly typed logic programming language. The type system is based on a many-sorted logic with parametric polymorphism. **RELFUN** extends Horn logic by using higher-order syntax, first-class finite domains, and expressions of nondeterministic, nonground functions, explicitly distinguished from structures.

The most significant member in the family of logic programming languages is the **Horn logic**-based **Prolog**. Prolog was invented by Alain Colmerauer and Philippe Roussel at the University of Aix-Marseille in 1971. The first version was implemented in 1972 using Algol. Prolog was designed originally for natural-language processing, but it has become one of the most widely used languages for artificial intelligence. Many implementations appeared after the original work. Early implementations included **C-Prolog**, **ESLPDPRO**, **Frolic**, LM-Prolog, Open Prolog, SB-Prolog, and UPMAIL Tricia Prolog. Today, the common Prologs in use are AMZI Prolog, **GNU Prolog**, LPA Prolog, **Quintus Prolog**, SICSTUS Prolog, SNI Prolog, and **SWI-Prolog**.

KH ALL RIGHTS RESERVED

1.2 Structures of programming languages

This section studies the structures of programming languages in terms of four structural layers: lexical, syntactic, contextual, and semantic.

1.2.1 Lexical structure

Lexical structure defines the vocabulary of a language. Lexical units are considered the building blocks of programming languages. The lexical structures of all programming languages are similar and normally include the following kinds of units:

- **Identifiers**: Names that can be chosen by programmers to represent objects like variables, labels, procedures, and functions. Most programming languages require that an identifier start with an alphabetical letter and can be optionally followed by letters, digits, and some special characters.
- **Keywords**: Names reserved by the language designer and used to form the syntactic structure of the language.
- **Operators**: Symbols used to represent the operations. All general-purpose programming languages should provide certain minimum operators such as mathematical operators like +, −, *, /, relational operators like <, ≤, ==, >, ≥, and logic operators like AND, OR, NOT, etc.
- **Separators**: Symbols used to separate lexical or syntactic units of the language. Space, comma, colon, semicolon, and parentheses are used as separators.
- **Literals**: Values that can be assigned to variables of different types. For example, integer-type literals are integer numbers, character-type literals are any character from the character set of the language, and string-type literals are any string of characters.
- **Comments**: Any explanatory text embedded in the program. Comments start with a specific keyword or separator. When the compiler translates a program into machine code, all comments will be ignored.

1.2.2 Syntactic structure

Syntactic structure defines the grammar of forming sentences or statements using the lexical units. An imperative programming language normally offers the following kinds of statements:

- **Assignments**: An assignment statement assigns a literal value or an expression to a variable.
- **Conditional statements**: A conditional statement tests a condition and branches to a certain statement based on the test result (*true* or *false*). Typical conditional statements are *if-then*, *if-then-else*, and *switch* (*case*).
- **Loop statements**: A loop statement tests a condition and enters the body of the loop or exits the loop based on the test result (*true* or *false*). Typical loop statements are *for-loop* and *while-loop*.

The formal definition of lexical and syntactic structures will be discussed in Section 1.2.5.

1.2.3 Contextual structure

Contextual structure (also called **static semantics**) defines the program semantics before dynamic execution. It includes variable declaration, initialization, and type checking.

Some imperative languages require that all variables be initialized when they are declared at the contextual layer, while other languages do not require variables to be initialized when they are declared, as long as the variables are initialized before their values are used. This means that initialization can be done either at the contextual layer or at the semantic layer.

Contextual structure starts to deal with the meaning of the program. A statement that is lexically correct may not be contextually correct. For example:

KH ALL RIGHTS RESERVED

```
stringstr = "hello";
int i = 0;
int j = i + str;
```

All declaration statements are lexically correct, but the last statement is contextually incorrect because it does not make sense to add an integer variable to a string variable.

More about data type, type checking, and type equivalence will be discussed in Section 1.3.

1.2.4 Semantic structure

Semantic structure describes the meaning of a program, or what the program does during the execution. The semantics of a language are often very complex. In most imperative languages, there is no formal definition of semantic structure. Informal descriptions are normally used to explain what each statement does. The semantic structures of functional and logic programming languages are normally defined based on the mathematical and logical foundation on which the languages are based. For example, the meanings of Scheme procedures are the same as the meanings of the lambda expressions in lambda calculus on which Scheme is based, and the meanings of Prolog clauses are the same as the meanings of the clauses in Horn logic on which Prolog is based.

1.2.5 BNF notation

BNF (Backus-Naur Form) is a Meta language that can be used to define the lexical and syntactic structures of another language. Instead of learning BNF language first and then using BNF to define a new language, we will first use BNF to define a simplified English language that we are familiar with, and then we will learn BNF from the definition itself.

A simple English sentence consists of a subject, a verb, and an object. The subject, in turn, consists of possibly one or more adjectives followed by a noun. The object has the same grammatical structure. The verbs and adjectives must come from the vocabulary. Formally, we can define a simple English sentence as follows:

```
<sentence>   ::= <subject><verb><object>
<subject>    ::= <noun> | <article><noun> | <adjective><noun> |
                 <article><adjective><noun>
<adjective>  ::= <adjective> | <adjective><adjective>
<object>     ::= <subject>
<noun>       ::= table | horse | computer
<article>    ::= the | a
<adjective>  ::= big | fast | good | high
<verb>       ::= is | makes
```

In the definitions, the symbol “::=” means that the name on the left-hand side is defined by the expression on the right-hand side. The name in a pair of angle brackets “<>” is **nonterminal**, which means that the name needs to be further defined. The vertical bar “|” represents an “or” relation. The boldfaced names are **terminal**, which means that the names need not be further defined. They form the vocabulary of the language.

We can use the sentence definition to check whether the following sentences are syntactically correct.

```
fast high big computer is good table                    1
the high table is a good table                          2
a fast table makes the high horse                       3
the fast big high computer is good                      4
```

KH ALL RIGHTS RESERVED

```
good table is high                                                5
a table is not a horse                                            6
is fast computer good                                             7
```

The first sentence is syntactically correct, although it does not make much sense. Three adjectives in the sentence are correct because the definition of an adjective recursively allows any number of adjectives to be used in the subject and the object of a sentence. The second and third sentences are also syntactically correct according to the definition.

The fourth and fifth sentences are syntactically incorrect because a noun is missing in the object of the sentences. The sixth sentence is incorrect because “not” is not a terminal. The last sentence is incorrect because the definition does not allow a sentence to start with a verb.

After we have a basic understanding of BNF, we can use it to define a small programming language. The first five lines define the lexical structure, and the rest defines the syntactic structure of the language.

```
<letter>      ::=  a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit>       ::=  0|1|2|3|4|5|6|7|8|9
<symbol>      ::=  _|@|.|~|?|#|$
<char>        ::=  <letter>|<digit>|<symbol>
<operator>    ::=  +|-|*|/|%|<|>|==|<=|>=|and|or|not
<identifier>  ::=  <letter>|<identifier><char>
<number>      ::=  <digit>|<number><digit>
<item>        ::=  <identifier>|<number>
<expression>  ::=  <item>|(<expression>)|
                   <expression><operator><expression>
<branch>      ::=  if <expr>then {<block>} |
                   if <expr>then {<block>}else {<block>}
<switch>      ::=  switch<expr>{<sbody>}
<sbody>       ::=  <cases> | <cases>; default :<block>
<cases>       ::=  case<value>:<block> |
                   <cases> ; case<value>:<block>
<loop>        ::=  while <expr>do {<block>}
<assignment>  ::=  <identifier>=<expression>;
<statement>   ::=  <assignment>|<branch>|<loop>
<block>       ::=  <statement>|<block>;<statement>
```

Now we use the definition to check which of the following statements are syntactically correct.

```
sum1 = 0;                                                         1
while sum1 <= 100 do {                                            2
sum1 = sum1 + (a1 + a2) * (3b % 4*b); }                           3
if sum1 == 120 then 2sum - sum1 else sum2 + sum1;                 4
p4#rd_2 = ((1a + a2) * (b3 % b4)) / (c7 - c8);                    5
_foo.bar = (a1 + a2 - b3 - b4);                                   6
(a1 / a2) = (c3 - c4);                                            7
```

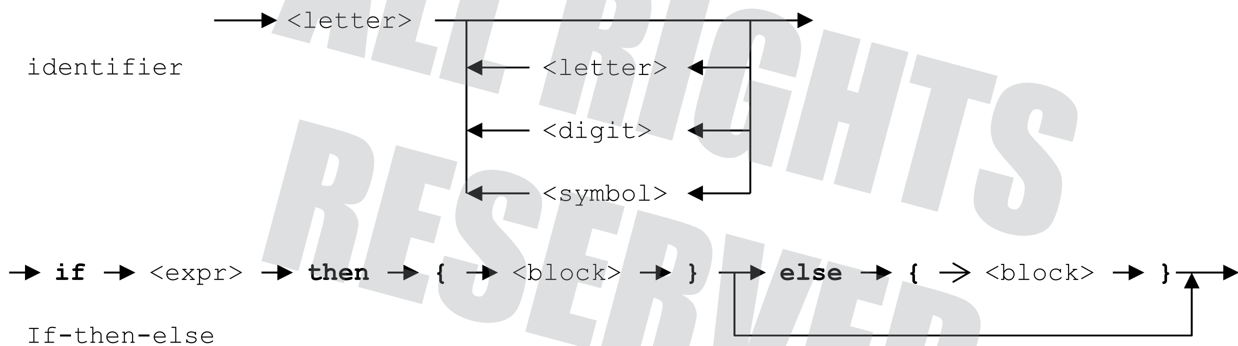
According to the BNF definition of the language, statements 1 and 2 are correct. Statements 3 and 4 are incorrect because 3*b* and 2*sum* are neither acceptable identifiers nor acceptable expressions. Statement 5 is incorrect. Statement 6 is incorrect because an identifier must start with a letter. Statement 7 is incorrect because the left-hand side of an assignment statement must be an identifier.

KH ALL RIGHTS RESERVED

1.2.6 Syntax graph

BNF notation provides a concise way to define the lexical and syntactic structures of programming languages. However, BNF notations, especially the recursive definitions, are not always easy to understand. A graphic form, called a **syntax graph**, also known as **railroad tracks**, is often used to supplement the readability of BNF notation. For example, the identifier and the if-then-else statement corresponding to the BNF definitions can be defined using the syntax graphs in Figures 1.1. The syntax graph for the identifier requires that an identifier start with a letter, may exit with only one letter, or follow the loops to include any number of letters, digits, or symbols. In other words, to check the legitimacy of an identifier, we need to travel through the syntax graph following the arrows and see whether we can find a path that matches the given identifier. For instance, we can verify that `len_23` is a legitimate identifier as follows. We travel through the first `<letter>` once, travel through the second `<letter>` on the back track twice, travel through the `<symbol>` once, and finally travel through the `<digit>` twice, and then we exit the definition. On the other hand, if you try to verify that `23_len` is a legitimate identifier, you will not be able to find a path to travel through the syntax graph.

Using the if-then-else syntax graph in Figure 1.1, we can precisely verify whether a given statement is a legitimate if-then-else statement. The alternative route that bypasses the else branch signifies that the else branch is optional. Please note that the definition of the if-then-else statement here is not the same as the if-then-else statement in C language. The syntax graph definitions of various C statements can be found in Chapter 2.



KH ALL RIGHTS RESERVED

Figure 1.1. Definition of identifier and if-then-else statement.

As another example, Figure 1.2 shows the definitions of a set data structures, including the definitions of value, string, array, bool, and number.

In syntax graphs, we use the same convention that terminals are in boldfaced text and nonterminals are enclosed in a pair of angle brackets.

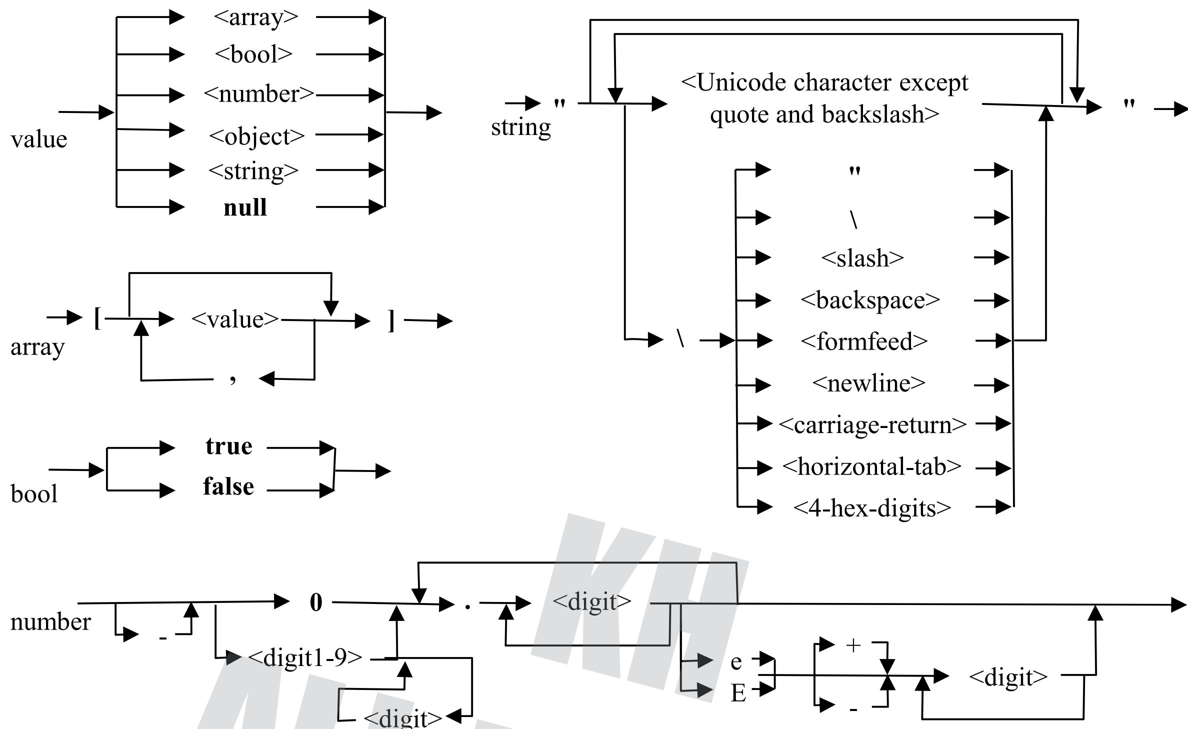


Figure 1.2. Definitions of different data structures.

KH ALL RIGHTS RESERVED

1.3 Data types and type checking

In this section, we examine data types, type equivalence, and type checking that is part of the contextual structure of a programming language.

1.3.1 Data types and type equivalence

A **data type** is defined by the set of primary values allowed and the operations defined on these values. A data type is used to declare variables (e.g., integer, real, array of integer, string, etc.).

Type checking is the activity of ensuring that the data types of operands of an operator are legal or equivalent to the legal type.

Now the question is, what types are equivalent? For example, are *int* and *short* types, and *int* and *float* types in C language equivalent? Are the following operations legal in C?

```
int i = 3; short j = 5;float n, k = 3.0;
n = i + j + k;
```

The answers to these questions are related to the type equivalence policy of programming languages. There are two major type equivalence policies: structural equivalence and name equivalence. If the **structural equivalence** policy is used, the two types are equivalent if they have the same set of values (data range) and the same operations. This policy follows the stored program concept and gives programmers the maximum flexibility to manipulate data. The **stored program concept** suggests that instruction and data are stored in computer memory as binary bit patterns and it is the programmer's responsibility to interpret the meanings of the bit patterns. Algol 68, Pascal, and C are examples of

languages that use structural equivalence policy. For example, in the following C code, a type *salary* and a type age are defined:

```
typedef integer salary;
typedef integer age;
int i = 0; salary s = 60000; age a = 40;
i = s + a;
```

If structural equivalence policy is used, the statement `"i = s + a;"` is perfectly legal because all three variables belong to types that are structurally equivalent. Obviously, adding an *age* value to a *salary* value and putting it into an integer type variable normally does not make sense and most probably is a programming error. Structural equivalence policy assumes programmers know what they are doing.

On the other hand, if **name equivalence** policy is used, two types are equivalent only if they have the same name. Since no programming language will normally allow two different data types to have the same name, this policy does not allow any variable of one type to be used in the place where a variable of another type is expected without explicit type conversion. If name equivalence policy is used, the statement `"i = s + a;"` is then illegal.

Ada, C++, and Java are examples where the name equivalence policy is used. Name equivalence enforces a much stronger discipline and is generally considered a good way to ensure the correctness of a program.

1.3.2 Type checking and type conversion

Besides type equivalence, type-checking policy is another important characteristic of a programming language. We can vaguely differentiate strongly typed and weakly typed languages. In a (truly) **strongly typed language:**

- every name in a program must be associated with a single type that is known at the compilation time;
- name equivalence is used; and
- every type inconsistency must be reported.

C++ and Java are strongly typed languages. Functional programming languages are normally weakly typed because mathematical functions are not typed. However, ML is strongly typed, although users did not have to declare variables that could be inferentially determined by the compiler.

On the other hand, in a **weakly typed language:**

- not every name has to be associated with a single type at the compilation time;
- structural equivalence is used; and
- a variable of a subtype is acceptable, and implicit type conversion, called **coercion**, is allowed.

Type T1 is considered the **subtype** of T2, if the set of data of T1 is a subset of data of T2 and the set of operations of T1 is a subset of operations of T2. For example, an *integer* type can be considered a subtype of *floating-point* type. C, Scheme, and Prolog are weakly typed programming languages. Typeless languages like BCPL are weakly typed.

If a data type is a subtype of another type, it does not mean that the two types have the equivalent or similar structure. For example, in a 32-bit computer, an integer number 3 is represented in a simple binary form, as shown in the upper part of Figure 1.3. On the other hand, a floating-point number is normally represented in a three-segment format (IEEE 754 Standard), with the leftmost bit representing the sign (positive or negative), the next 8 bits representing the exponent, and the remaining 23 bits representing the fraction of the number, as shown in the lower part of Figure 1.3.

KH ALL RIGHTS RESERVED

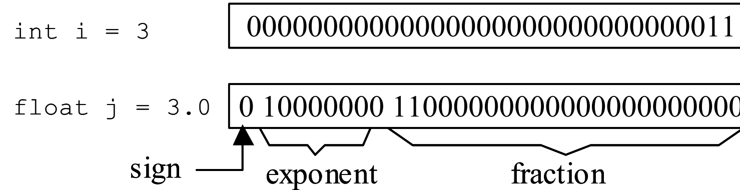


Figure 1.3. Integer and floating-point numbers have different internal structures.

In a strongly typed programming language, you can still use different types of variables in a single expression. In this case, one has to do explicit type conversion. In C/C++, explicit type conversion is called **typecasting**: the destination type in parentheses is placed before the variable that has an incompatible type. In the following statement, explicit conversion is used to convert variable `s` of *salary* type and variable `a` of *age* type into `int` type:

```
i = (int)s + (int)a;
```

Strong type checking trades flexibility for reliability. It is generally considered a good policy and is used in most recent programming languages.

1.3.3 Orthogonality

The word **orthogonality** refers to the property of straight lines meeting at right angles or independent random variables. In programming languages, orthogonality refers to the property of being able to combine various language features systematically. According to the way the features are combined, we can distinguish three kinds of orthogonality: compositional, sort, and number orthogonality.

Compositional orthogonality: If one member of the set of features S_1 can be combined with one member of the set of features S_2, then all members of S_1 can be combined with all members of S_2, as shown in Figure 1.4.

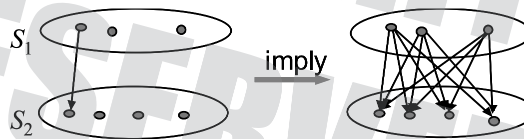


Figure 1.4. Compositional orthogonality.

For example, assume that S_1 is the set of different kinds of declarations: (1) plain, (2) initializing, and (3) constant. For example

```
(1) typex i; (2) typex i = 5; (3) const typex i = 5;
```

Set S_2 is data types: `boolean, int, float, array`.

If the language is compositionally orthogonal, then we can freely combine these two sets of features in all possible ways:

- Plain boolean, plain int, plain float, plain array

```
(1) boolean b; (2) int i; (3) float f; (4) array a[];
```

- Initializing boolean, initializing int, initializing float, and initializing array

```
(1) boolean b = true;
(2) int i = 5;
(3) float f = 4.5;
(4) array a[3] = {4, 6, 3};
```

KH ALL RIGHTS RESERVED

- Constant boolean, constant int, constant float, constant array

```
(1) const boolean b = true;
(2) const int i = 5;
(3) const float f = 7.5;
(4) const array a[3] = {1, 2, 8};
```

Sort orthogonality: If one member of the set of features S_1 can be combined with one member of the set of features S_2, then *this* member of S_1 can be combined with all members of S_2, as shown in Figure 1.5.

For example, if we know that `int i` is allowed (the combination plain-int), then, according to sort orthogonality, plain boolean, plain int, plain float, and plain array will be allowed, that is:

```
(1) boolean b;
(2) int i;
(3) float f;
(4) array a[];
```

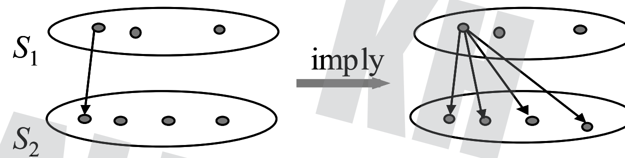


Figure 1.5. Sort orthogonality 1.

However, since sort orthogonality does not allow other members of S_1 to combine with members of S_2, we cannot conclude that initializing and constant declarations can be applied to the data types in S_2.

The sort orthogonality can be viewed from the other side. If one member of the set S_1 can be combined with one member of the set S_2, then all members in S_1 can be combined with the members of S_2, as shown in **Figure 1.6.**

For example, if the plain declaration can be combined with the `int` type, we conclude that the plain, initializing, and constant declarations can be applied to the `int` type.

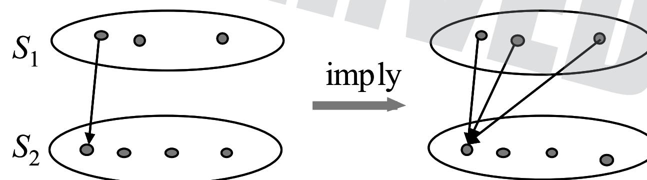


Figure 1.6. Sort orthogonality 2.

For example, if we know that `const array a` is allowed (the combination constant and array), according to the sort orthogonality, then plain array, initializing array, and constant array will be allowed:

```
(1) array a[];
(2) array a[3] = {1, 2, 8};
(3) const array a[3] = {1, 2, 8};
```

However, since the sort orthogonality does not allow other members of S_2 to combine with members of S_1, we cannot conclude that `boolean`, `int`, and `float` type can be declared in three different ways.

KH ALL RIGHTS RESERVED

Number orthogonality: If one member of the set of features *S* is allowed, then zero or multiple features of S are allowed. For example, if you can declare a variable in a particular place (in many languages, e.g., C++ and Java, you can put declarations anywhere in the program), you should be able to put zero (no declaration) or multiple declarations in that place. In a class definition, if you can define one member, you should be allowed to define zero or multiple members.

1.4 Program processing and preprocessing

This section discusses what preparations need to be done before the computer hardware can actually execute the programs written in a high-level programming language. Typical techniques used to do the preparation work are preprocessing, interpretation, and compilation.

1.4.1 Interpretation and compilation

Interpretation of a program is the direct execution of one statement at a time sequentially by the interpreter. **Compilation**, on the other hand, first translates all the statements of a program into assembly language code or machine code before any statement is executed. The compiler does not execute the program, and we need a separate loader to load the program to the execution environment (hardware or a runtime system that simulates hardware). A **compiler** allows program modules to be compiled separately. A **linker** can be used to combine the machine codes that were separately compiled into one machine code program before we load the program to the execution environment. Figure 1.7 shows typical processing phases of programs using compilation.

KH ALL RIGHTS RESERVED

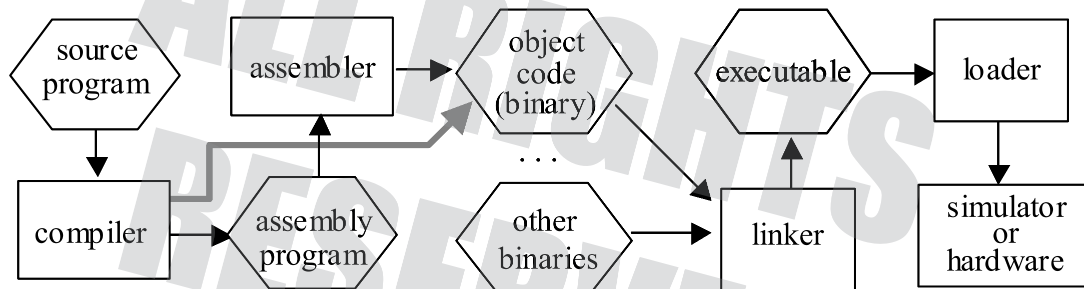


Figure 1.7. Compilation-based program processing.

The advantage of interpretation is that a separate program-processing phase (compilation) is saved in the program development cycle. This feature allows the program to be updated without stopping the system. The interpreter can immediately and accurately locate any errors. However, the execution speed with interpretation is slower than the execution of machine code after the compilation. It is also more difficult to interpret programs written in very high-level languages.

To make use of the advantages of both compilation and interpretation, Java offers a combined solution to program processing. As shown in **Figure 1.8**, Java source program is first translated by a compiler into an assembly language-like intermediate code, called **bytecode**. Then the bytecode is interpreted by an interpreter called **Java Virtual Machine** (JVM). The advantage of using the intermediate code is that the compiler will be independent of the machine on which the program is executed. Thus, only a single compiler needs to be written for all Java programs running on any machine. Another advantage is that the bytecode is a low-level language that can be interpreted easily. It thus makes JVM small enough to be integrated into an Internet browser. In other words, Java bytecode programs can be transferred over the Internet and executed in the client's browser. This ability makes Java the dominant language for Internet application development and implementation.

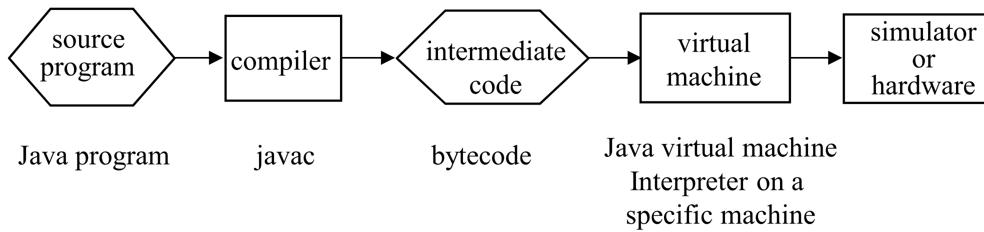


Figure 1.8. Java processing environment.

Microsoft's Visual Studio extends the Java environment's compilation and interpretation processing into a two-step compilation process. As shown in Figure 1.9, in the first compilation step, a high-level language program is compiled to a low-level language called **intermediate language (IL).** The IL is similar in appearance to an assembly language. Programs in IL are managed and executed by the **common language runtime** (CLR). Similar to Java's bytecode, the purpose of IL is to make the CLR independent of the high-level programming languages. Compared to JVM, CLR has a richer type system, which makes it possible to support many different programming languages rather than one language only, as on JVM. On the basis of the type system, nearly any programming language, say X, can be easily integrated into the system. All we need to do is to write a compiler that translates the programs of the X language into the IL. Before an IL program can be executed, it must be translated to the machine code (instructions) of the processor on which the programs are executing. The translation job is done by a **Just-In-Time** (JIT) compiler embedded in CLR. The JIT compiler uses a strategy of compile-when-used, and it dynamically allocates blocks of memory for internal data structures when each method is first called. In other words, JIT compilation lies between the complete compilation and statement-by-statement interpretation.

Unlike the Java environment, Visual Studio is language agnostic. Although C# is considered its flagship, Visual Studio is not designed for a specific language. Developers are open to use the common libraries and functionality of the environment while coding their high-level application in the language of their choice.

KH ALL RIGHTS RESERVED

1.4.2 Preprocessing: macro and inlining

Many programming languages allow programmers to write macros or **inline** procedures that preserve the structure and readability of programs while retaining the efficiency. The purpose of **program preprocessing** is to support macros and inline procedures. The preprocessing phase is prior to the code translation to the assembly or machine code. The preprocessor is normally a part of the compiler.

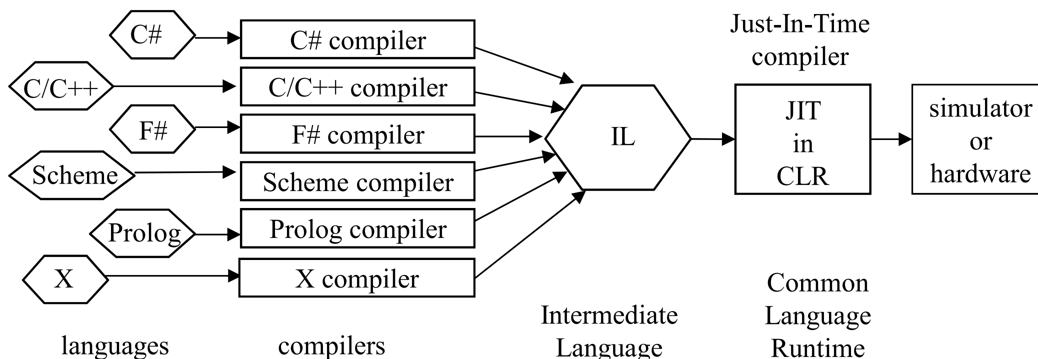


Figure 1.9. Microsoft's Visual Studio. Net programming environment.

In different programming languages, macros can be defined in different ways. In Scheme, we can introduce a **macro** by simply adding a keyword macro in the head of a procedure definition. In C/C++, a macro is introduced by a construct, which is different from a procedure definition:

```
#define name body
```

The `#define` construct associates the code in the *body* part to the identifier *name* part. The body can be a simple expression or rather complex procedure-like code, where parameter passing is allowed. For example:

```
#define MAXVAL 100
#define QUADFN(a,b) a*a + b*b - 2*a*b
...
x = MAXVAL + QUADFN(5,16);
y = MAXVAL - QUADFN(1,3);
```

The last two statements will be replaced by the macro preprocessor as:

```
x = 100 + 5*5 + 16*16 - 2*5*16;
y = 100 - 1*1 + 3*3 - 2*1*3;
```

where `MAXVAL` is replaced by `100` and `QUADFN(5,16)` is replaced by `a*a + b*b - 2*a*b`, with parameter passing: `a` is given the value `5` and `b` is given the value `16`, and `a` is given the value `1` and `b` is given the value `3`, in the two statements, respectively.

Macros are more efficient than procedure calls because the body part of the macro is copied into the statement where the macro is called. No control flow change is necessary. This process is also called **inlining**. On the other hand, if the macro is implemented as a procedure/function:

```
#define MAXVAL 100
int QUADFN(a,b) {return a*a + b*b - 2*a*b;}
...
x = MAXVAL + QUADFN(5,16);
y = MAXVAL - QUADFN(1,3);
```

a procedure will cause a control flow change that usually needs to save the processor environment, including registers and the program counter (return address), onto the stack. This process sometimes is called **out-lining**. Obviously, inlining is much more efficient than out-lining.

Macro preprocessing is a very simple and straightforward substitution of the macro *body* for the macro *name*. It is the programmer's responsibility to make sure that such a simple substitution will produce correct code. A slight overlook could lead to a programming error. For example, if we want to define a macro to obtain the absolute value of a variable, we write the following macro in C:

```
#define abs(a) ((a<0) ? -a : a)
```

where the C statement `((a<0) ? -a : a)` returns `-a` if `a<0`; otherwise, it returns `a`.

This macro definition of the absolute-value function looks correct, but it is not. For example, if we call the macro in the following statement:

```
j = abs(2+5);    // we expect 7 to be assigned to j.
```

The statement does produce a correct result. However, if we call the macro in the following statement:

```
j = abs(2-5);    // we expect +3 to be assigned to j.
```

KH ALL RIGHTS RESERVED

The statement will produce an incorrect result. The reason is that the macro-processor will replace "`abs(2-5)`" by "`((a<0) ? -a : a)`" and then replace the parameter "`a`" by "`2-5`", resulting in the statement:

```
j = ((2-5 < 0) ? -2-5 : 2-5);
```

Since `(2-5 < 0)` is true, this statement will produce the result of `-2-5 = -7`, which is assigned to the variable `j`. Obviously, this result is incorrect, because we expect `+3` to be assigned to `j`.

Examine a further example. If we write a statement:

```
j = abs(-2-5);
```

The macro-processor will replace "`abs(-2-5)`" by "`((a<0) ? -a : a)`" and then replace the parameter "`a`" by "`-2-5`," resulting in the statement:

```
j = ((-2-5 < 0) ? --2-5 : -2-5);
```

The "`--2`" in the preprocessed statement may result in a compiler error.

The problem in this `abs(a)` macro is that we expect the expression that replaces `a` to be a unit. All operations within the unit should be done before `−a` is performed. This is the case when we write a function or procedure. However, the macro replacement does not guarantee this order of operation, and the programmer must understand the difference. A correct version of the `abs(a)` macro is:

```
#define abs(a) ((a<0) ? -(a) : a)    // correct version of abs(a) macro
```

Putting the "`a`" in a pair of parentheses guarantees that the operations within `a` are completed before `-a` is performed.

Owing to the nature of simple textual replacement, a macro may cause side effects. A **side effect** is an unexpected or unwanted modification of a state. When we use a global variable to pass values between the caller procedure and the called procedure, a modification of the global variable in the called procedure is a side effect.

Next, examine the side effect in the correctly defined `abs(a)` macro discussed earlier. If we call the macro in the following code:

```
i = 3;
j = abs(++i);    // we expect 4 to be assigned to j.
```

According to the way a macro is preprocessed, the following code will be produced by the macro-processor:

```
i = 3;
j = ((++i < 0) ? -(++i) : ++i);
```

When the second statement is executed, variable `i` will be incremented twice. According to the definition of the expression `++i`, variable `i` will be incremented every time before `i` is accessed. There is another similar expression in C/C++: `i++`, which increments variable `i` every time after `i` is accessed. Similarly, C/C++ have expressions `--i` and `i--`, etc.

In the earlier statement, variable `i` will be accessed twice: first when we assess `(++i < 0)`, and then the second `++i` will be accessed after the condition is assessed as false. As a result, number 5, instead of 4, will be assigned to the variable `j`.

Macros can be used to bring (inline) a piece of assembly language code into a high-level language program. For example:

```
#define PORTIO __asm          \
```

KH ALL RIGHTS RESERVED

```
{                               \
   __asm mov al, 2              \
   __asm mov dx, 0xD007         \
   __asm out al, dx             \
}
```

The back slash \ means that there is no line break when performing macro replacement. When we make a macro call to `PORTIO` in the program, this macro call will be replaced by:

```
__asm { __asm mov al, 2 __asm mov dx, 0xD007 __asm out al, dx }
```

where `__asm` is the C/C++ keyword for assembly instructions. If the compiler is translating the program into assembly code, nothing needs to be done with a line that starts with `__asm`. If the compiler is translating the program into machine code, the compiler will call the assembler to translate this line into machine code.

For the execution of macros, different runtimes (execution engines) may process the translated code indifferent order. As an example, we consider the following code, which has two pairs of functions and macros.

```
/* Side effect, Macro versus Function */
#include <stdio.h>
#pragma warning(disable : 4996)
#define mac1(a,b) a*a + b*b - 2*a*b
#define mac2(a,b) a*a*a + b*b*b - 2*a*b
int func1(int a, int b) { return (a*a + b*b - 2 * a*b); }
int func2(int a, int b) { return (a*a*a + b*b*b - 2 * a*b); }
main() {
   int a, b, i, j, fncout, macout;
   printf("Please enter two integers\n");
   scanf("%d%d", &a, &b);
   i = a;
   j = b;
   fncout = func1(++i, ++j);
   printf("i = %d\tj = %d\n", i, j);
   i = a;
   j = b;
   macout = mac1(++i, ++j);
   printf("i = %d\tj = %d\n", i, j);
   printf("fncout1 = %d\tmacout1 = %d\n\n", fncout, macout);
   i = a;
   j = b;
   fncout = func2(++i, ++j);
   printf("i = %d\tj = %d\n", i, j);
   i = a;
   j = b;
   macout = mac2(++i, ++j);
   printf("i = %d\tj = %d\n", i, j);
   printf("fncout2 = %d\tmacout2 = %d\n", fncout, macout);
```

KH ALL RIGHTS RESERVED

```
}
```

Each of the pairs, (mac1, func1) and (mac2, func2), is supposed to implement the same functionality and give the same output. If we run the code on Visual Studio 2013, the outputs are as follows:

```
Please enter two integers
5
6
i = 6   j = 7
i = 8   j = 9
fncout1 = 1     macout1 = 1

i = 6   j = 7
i = 9   j = 10
fncout2 = 475   macout2 = 1549
```

How are these outputs generated? We will manually trace the execution as follows. After the macro replacement, the two macro calls will be replaced by the following statements, respectively:

```
macout = mac1(++i, ++j);→      macout = ++i*++i + ++j*++j - 2*++i*++j;

macout = mac2(++i, ++j);→      macout = ++i*++i*++i + ++j*++j*++j - 2*++i*++j;
```

In Visual Studio, the order of execution is to apply to all the unary operations (++) first, in the order of their appearances, before doing any arithmetic calculations at all.

For func1 and mac1, the calculations are done as follows, respectively:

```
func1: 6*6 + 7*7 - 2*6*7 = 36 + 49 - 84 = 1
mac1:  8*8 + 9*9 - 2*8*9 = 64 + 81 - 144 = 1
```

In this example, func1 and mac1 happen to have generated the same result. This is a pure coincident. For the func2 and mac2, the calculations are done as follows, respectively, which generated different results:

```
func1: 6*6*6 + 7*7*7 - 2*6*7 = 216 + 343 - 84 = 475
mac1:  9*9*9 + 10*10*10 - 2*9*10 = 729 + 1000 - 180 = 1549
```

Now, we run the same code on GNU GCC. The following results are generated:

```
5
6
i = 6   j = 7
i = 8   j = 9
fncout1 = 1     macout1 = -31

i = 6   j = 7
i = 9   j = 10
fncout2 = 475   macout2 = 788
```

As can be observed that, the function implementations generate the same results as that generated on Visual Studio. However, the outputs of the macros on GCC are completely from that on Visual Studio. The reason is that GCC uses a different order of evaluations. Now we explain how macout1 = -31 and macout2 = 788 are generated.

KH ALL RIGHTS RESERVED

GCC calculates the unary operations for the operands of each operator in pair and makes the same variable in the operation to have the same value.

For mac1: `macout = ++i*++i + ++j*++j - 2*++i*++j;` the macro is evaluated as follows:

```
mac1:  7*7 + 8*8 – (2*8)*9 = 49 + 64 – 144 = -31
```

where, the value of the first pair of variables ++i is 7, the value of the second pair of variables ++i is 8. Then, 2 and ++i will form a pair, resulting (2*8), and it result will form a pair with the last ++i, which obtain a value 9.

For mac2: `macout = ++i*++i*++i + ++j*++j*++j - 2*++i*++j;` the macro is evaluated as follows:

```
mac1:  (7*7)*8 + (8*8)*9 – (2*9)*10 = 392 + 576 – 180 = 788
```

Notice that the macros generate different values when there exist side effects, for example, when ++i is used as the input. If no side effects are involved, the macros should generate the same results as their function implementations, and macros should generate the same results running different execution environments.

This discussion shows that macros are similar to and yet different from procedures and functions and that both writing macros (ensuring correctness) and using macros (understanding the possible side effects) can be difficult and challenging. Can we write and use macros (obtain better efficiency) exactly in the same way as we write and use procedures and functions (obtain the same simplicity)? Efforts have been made to do that, and we are making good progress. As mentioned earlier, in Scheme, we can write macros in the same way in which we write procedures. However, we still cannot use macros in exactly the same way we use procedures. This issue will be discussed in the chapter on Scheme programming (Chapter 4). In C++, "inline" procedures/functions are allowed. All that is needed is to add a keyword –inline (in C) and **inline** (in C++) in front of the definition of a procedure/function. For example:

```
_inline int sum(int i, int j) {
  return i + j;
}
```

However, the inline procedure/function is slightly different from a macro. A macro call will always be replaced by the body of the macro definition in the preprocessing stage. The macro-processor will not check the syntax and semantics of the definition. On the other hand, for an inline procedure/function call, the compiler (not the preprocessor) will try to replace the call by its body. There are two possibilities: If the procedure/function is simple enough, the compiler can do the replacement and can guarantee the correctness of the replacement; that is, the inlined code must work exactly in the same way as an ordinary procedure/function call. If the body of the procedure is too complicated (e.g., uses many variables and complex expressions), the compiler will not perform inlining. The inline procedure in this case will work like an ordinary procedure.

Java has a similar mechanism called **final method**. If a method is declared **final**, it cannot be overridden in a subclass. This is because a call to a final method may have been replaced by its body during compilation, and thus late binding cannot associate the call to a method redefined in a subclass.

*1.5 Program development

This section briefly introduces the main steps of the program development process, including requirement, specification, design, implementation, testing, proof, and related techniques. Understanding these steps and the related techniques involved is extremely important. However, a more detailed discussion of these topics is beyond the scope of this text.

KH ALL RIGHTS RESERVED

1.5.1 Program development process

Development of a program normally goes through the following process:

Requirement is an informal description, from the user's point of view, of the functionality of the program to be developed. Normally, requirements are informal. For example, "I want the program to sort all numbers in an increasing order" is an informal requirement.

Specification is a formal or semiformal description of the requirement from the developer's point of view. The specification describes what needs to be done at the functionality level. A formal specification is defined by a set of preconditions on the inputs and a set of post conditions on the outputs. For example, a formal specification of "sorting numbers" can be defined as follows:

Input: $(x_1, x_2, ..., x_n)$
Preconditions on inputs:

$(\forall x_i)(x_i \in I)$, where I is the set of all integer numbers.

Output: $(x_{i1}, x_{i2}, ..., x_{in})$
Postconditions on outputs:

$(\forall x_{ij})(\forall x_{ik})\ ((x_{ij} \in I \wedge x_{ij} \in I \wedge j < k) \rightarrow x_{ij} \leq x_{ik})$.

The **design** step translates what needs to be done (functional specification) into how to do it (procedural steps or algorithm). For example, devising a sorting algorithm to meet the specification belongs to the design step. An algorithm is usually written in a pseudo language that does not have the mechanical details of a programming language. A **pseudo language** focuses on clear and accurate communication with humans, instead of humans and machines.

The **implementation** step actualizes or instantiates the design step using a real programming language. Writing programs in real programming languages is the main topic of this text and will be discussed in much more detail in the following chapters.

The **testing and correctness proof** step tries to show that the implementation does the work defined in the design step or in the specification step. The development process has to return to the implementation or design steps if the implementation does not meet the requirements of the design or the specification.

The **verification and validation** step tries to show that the implementation meets the specification or the user's requirements. The development has to return to design or specification steps if necessary.

In fact, numerous refined phases and iterations within and between these steps can occur during the entire development cycle.

1.5.2 Program testing

In this and the next subsections, we present more detail of the testing and correctness proof step, and related techniques in the program development process.

A **test case** is a set of inputs to a program and the expected outputs that the program will produce if the program is correct and the set of inputs is applied to the program. We also use the **input case** to refer to the input part of a test case. **Program testing** is the process of executing a program by applying predesigned test cases with the intention of finding programming errors in a given environment. Testing is related to the environment. For example, assume that your program runs correctly on a GNU GCC C/C++ environment. If you move your program to a Visual Studio C/C++ environment, you need to retest your program because the environment has been changed. **Debugging** is the process of finding the locations and the causes of errors and fixing them.

KH ALL RIGHTS RESERVED

If a program has a limited number of possible inputs, we could choose all possible inputs as the test cases to test the program. This approach is called **exhaustive testing**. If the outputs are correct for all test cases, we have proved the program's correctness.

However, in many cases, a program can take an unlimited number of inputs, or the number of test cases is too big to conduct exhaustive testing. We have two ways to deal with the problem: use incomplete testing or use a formal method to prove the program's correctness.

If incomplete testing is used, the question is how we should choose (generate) the limited subset of test cases. Functional testing and structural testing are two major approaches used to generate incomplete test cases. In **functional testing**, we try to generate a subset of test cases that can cover (test) all functions or subfunctions of the program under test. Functional testing is also called **black-box testing** because we generate test cases without looking into the structure or source code of the program under test. In **structural testing**, we try to generate a subset of test cases that can cover (test) particular structures of the program under test. For example, we can try to cover all:

- statements in the program,
- branches of the control flow, or
- paths from the program's entry point to the program's exit point.

Structural testing is also called **glass-box testing** or **white-box testing** because it requires detailed knowledge of the control structure and the source code of the program under test.

Both functional testing and structural testing can be considered so-called partition testing. **Partition testing** tries to divide the input domain of the program under test into different groups so that the inputs in the same group are equivalent in terms of their testing capacity. For example, if we are conducting functional testing, we can consider all inputs that will cause the same subfunction to be executed as a group. On the other hand, if we are conducting structural testing, we can consider all inputs that will cause the same program path to be executed as a group. Then, we choose:

- one or several test cases from each group of inputs, and
- one or several inputs on the boundaries between the groups

to test the program. For example, if an integer input is partitioned into two groups—negative and nonnegative—then zero is on the boundary and must be chosen as an input case. Obviously, if the partition is done properly, partition testing will have a fair coverage of all the parts of the program under test.

Program testing is a topic that can take an entire semester to study. We do not attempt to teach the complete program testing theory and practice in this section. In the rest of the section, we will use a simple example to illustrate some of the important concepts related to the topic.

Example: The `gcd` function in the following C program is supposed to find the greatest common divisor of two nonnegative integers.

```
#include <stdio.h>
int gcd (int n0, int m0) { // n0 >= 0 and m0 >= 0 and (n0 ≠ 0 or m0 ≠ 0)
   int n, m;
   n = n0;
   m = m0;
   while (n != 0 && n != m) { // (n ≠ 0) AND (n ≠ m)
        if (n < m)
              m = m – n;
```

KH
ALL RIGHTS
RESERVED

```
        else
              n = n - m;
   }
   return m;
}
void main() {
   int i, j, k;
   scanf("%d\n%d", &i, &j);   // input integers i and j from the keyboard
   k = gcd(i, j);             // call function gcd
   printf("%d\n", k);         // output the greatest common divisor found
}
```

First, we “randomly” pick out the following test cases to test the program.

Input (i, j)	Expected Output k	Actual Output k
(6, 9)	3	3
(10, 5)	5	5
(0, 4)	4	4
(5, 7)	1	1
(8, 29)	1	1

We find that the actual output equals the expected output in all these test cases. Now the question is, is this program correct?

As we know, testing can never prove the correctness of a program unless all possible test cases have been used to test the program. However, a set of test cases that can cover different parts of the program can certainly increase the confidence of the correctness of the program.

Now we apply structural testing to generate systematically a better set of test cases. We assume that we aim at covering all the branches in the `gcd` function. To make sure we cover all branches, we first draw the function’s flowchart, as shown in Figure 1.10. A **flowchart** is an abstraction of a program that outlines the control flows of the program in a graphic form. In the flowchart in Figure 1.10, each branch is marked with a circled number.

KH ALL RIGHTS RESERVED

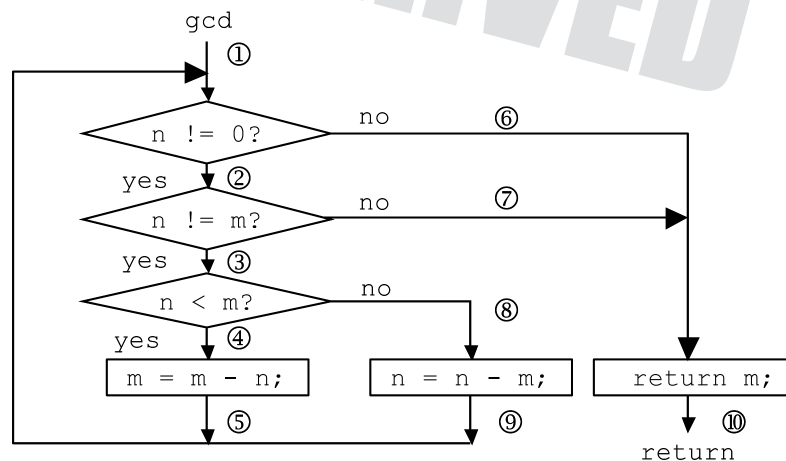


Figure 1.10. Flowchart of gcd function.

To obtain a good coverage of the branches, we need to analyze the features of the input domain of the program. For the given program, the input domain has the following features:

- The program takes two nonnegative integers as input. The boundary value is 0.
- The branches of the program are controlled by the relative values of a pair of integers. We should consider choosing equal and unequal pairs of numbers.
- The great common divisors are related to the divisibility of integers, or the prime and nonprime numbers. We should choose some prime numbers and some nonprime numbers.

On the basis of the analysis, we can choose, for example, these values for both `i` and `j`: boundary value 0, two prime numbers 2 and 3, and two nonprime numbers 9 and 10.

The combination of the two inputs generates the following input cases:

```
(0, 0)  // This case is not allowed according to the precondition.
(0, 2), (0, 3), (0, 9), (0, 10)
(2, 0), (2 2), (2, 3), (2, 9), (2, 10)
(3, 0), (3, 2), (3, 3), (3, 9), (3, 10)
(9, 0), (9, 2), (9, 3), (9, 9), (9, 10)
(10, 0), (10, 2), (10, 3), (10, 9), (10, 10)
```

We can apply all these test cases to test the program. We can also reduce the number of test cases by partitioning the input cases into groups: two input cases belong to the same group if they cover the same branches in the same order. Table 1.2 lists the groups, a representative from each group, the expected output of the representative input case, and the branches covered by the groups.

Groups partitioned	Representative	Expected gcd output	Branches covered
(0, 2),(0, 3),(0, 9),(0, 10)	(0, 2)	2	①⑥⑩
(2 2),(3, 3), (9, 9),(10, 10)	(2 2)	2	①②⑦⑩
(2, 0),(3, 0), (9, 0),(10, 0)	(2, 0)	2	①②③⑧⑨
(2, 3),(2, 9), (3, 10), (9, 10)	(2, 3)	1	①②③④⑤⑧⑨⑩
(2, 10), (3, 9)	(2, 10)	2	①②③④⑤⑦⑩
(3, 2),(9, 2), (10, 3),(10, 9)	(3, 2)	1	①②③⑧⑨④⑤⑩
(9, 3), (10, 2)	(9, 3)	3	①②③⑦⑧⑨⑩

Table 1.2. Input case partitions and branch coverage.

If we choose the representative input from each group and the expected output as the test cases, we obtain a test case set that can cover all the branches in the flowchart. Applying this set of test cases, we will find that the input case (2, 0) will not be able to produce the expected output. In fact, a dead-looping situation will occur. Thus, we successfully find that the program is incorrect.

KH ALL RIGHTS RESERVED

1.5.3 Correctness proof

To prove the **correctness** of a program, we need to prove that, for all predefined inputs (inputs that meet the preconditions), the program produces correct outputs (outputs that meet the postconditions).

Program proof consists of two steps: **partial correctness** and **termination**. A program is partially correct if an input that satisfies the preconditions is applied to the program, and if the program terminates, the output satisfies the postconditions. A program terminates if, for all inputs that meet the preconditions, the program stops in finite execution steps. A program is totally correct (**total correctness**) if the program is partially correct and the program terminates.

The idea of partial correctness proof is to prove that any input that meets the preconditions at the program entry point will be processed, step by step through the statements between the entry point and the exit point, and the postconditions will be satisfied at the exit point. Obviously, if there is no loop in the program, it is not too hard to do the step-by-step proof. However, if there is a loop in the program, the step-by-step approach will not work. In this case, we need to use the loop invariant technique to derive the condition to be proved through the loop. A **loop invariant** is a condition that is true in each iteration of the loop. To prove a condition is a loop invariant, we can use mathematical induction: We prove that the condition is true when the control enters the loop for the first time (iteration 1). We assume that the condition is true at iteration `k`, and prove that the condition will be true at the iteration `k+1`.

Finding the loop invariant that can lead to the postconditions is the most challenging task of the correctness proof. You must have a deep understanding of the problem in order to define a proper loop invariant.

Proving the termination is easy if we design the loops following these guidelines. The loop variable:

- is an enumerable variable (e.g., an integer);
- has a lower bound (e.g., will be greater than or equal to zero);
- will strictly decrease. **Strictly decrease** means that the value of the loop variable must be strictly less than the value of the variable in the previous iteration. The "<" relation is strict, while "≤" is not.

If you do not follow the guidelines, you may have trouble proving the termination of even a simple program. In an exercise given at the end of the chapter, a very simple example is given where many inputs have been tried, and the program always stops. However, so far, nobody can prove that the program terminates for all inputs.

Now we will study a similar example that we used in the last section to illustrate the proof concepts that we discussed here. The program is given in a pseudo language. Since we do not actually have to execute the program, we do not have to give the program in a real programming language.

```
gcd (n0, m0)
// precondition: (n0≥ 0 ∧ m0≥ 0) ∧ (n0≠ 0 ∨ m0≠ 0) ∧ (n0, m0 are integer)
   n = n0;
   m = m0;
   while n ≠ 0 do
// loop invariant:   (n ≥∧ m ≥ 0) ∧ (n ≠ 0 ∨ m ≠ 0) ∧
//                   max{u: u|n and u|m} = max{u: u|n0 and u|m0}
         if n ≤ m
         then m = m − n
         else swap(n, m)
```

KH
ALL RIGHTS
RESERVED

```
   output(m)
// postconditions:   m = max{u: u|n0 and u|m0}
```

To prove the partial correctness, we need to prove, for any integer pair (n0, m0) that meets the preconditions, the loop invariant is true in every iteration of the loop. When the control completes all the iterations of the loop and reaches the exit point of the program, the postconditions will be true. As we discussed, finding the loop invariant is the most difficult part. Now we are given the condition that should be a loop invariant, and we only need to prove that the condition is indeed a loop invariant. The given condition is:

```
(n ≥∧ m ≥ 0) ∧ (n ≠ 0 ∨ m ≠ 0) ∧
max{u: u|n and u|m} = max{u: u|n0 and u|m0}
```

We need to prove it is a loop invariant. We can use mathematical induction to prove it in the following steps:

(1) Prove the condition is true at the iteration 1: it is obvious.

(2) Assume the condition is true at iteration k.

(3) Prove the condition is true at iteration k+1.

Since the conversion made in each iteration is:

```
gcd(n, m) ⇒ gcd(n, m − n), or
gcd(n, m) ⇒ gcd(m, n)
```

According to mathematical facts:

```
        gcd(n, m) = gcd(n, m − n), and
        gcd(n, m) = gcd(m, n)
```

Thus, from iteration k to iteration k+1, the condition will remain to be true. Therefore, we have proved the condition is a loop invariant.

Next, we need to prove that the loop invariant leads to the postconditions. It can be easily seen from the program that if the loop invariant is true when the control leaves the loop, the postconditions will indeed be true. Thus, we have proved the partial correctness of the program.

To prove that the program terminates, we can use the following facts:

(1) The loop variable is (n, m). The loop variable is enumerable.

(2) If we consider the dictionary order, that is:

```
(n, m) > (n, m − n), if n ≥ m // e.g. (3, 6) > (3, 3) in dictionary order
(n, m) > (m, n),     if n > m. // (6, 3) > (3, 6) in dictionary order
```

we can see that there is strictly decreasing order on the loop variable (n, m) based on the dictionary order.

(3) There is a lower bound on the value that (n, m) can take, that is (0, 0).

Since the loop variable (n, m) is enumerable, it decreases strictly, and there is a lower bound (0, 0); the loop must terminate.

In an exercise given at the end of the chapter, a variation of the gcd program is suggested. Try to prove its partial correctness and its termination.

KH ALL RIGHTS RESERVED

1.6 Summary

In this chapter, we introduced in Section 1.1 the concepts of the four major programming paradigms: imperative, object-oriented, functional, and logic. We looked at the impact of language features on the performance of the programs written in the language. We briefly discussed the development of languages and the relationships among different languages. We then discussed in Section 1.2 the structures of programs at four levels: lexical, syntactic, contextual, and semantic. We illustrated our discussion on the lexical and syntactic levels by introducing the BNF notation and syntax graph. We used BNF notation to define the lexical and syntactic structures of a simple programming language. In Section 1.3, we studied the important concepts in programming languages, including data types, type checking, type equivalence, and type conversion. Orthogonality was used to examine the regularity and simplicity of programming languages. Finally, in Section 1.4, we briefly discussed program processing via compilation, interpretation, and a combination of the two techniques. The emphasis was on the macro and inline procedures/functions in C/C++. We studied how to define and use macros, and what their strengths and weaknesses are when compared to ordinary procedures/functions. Section 1.5 outlined the program development process and discussed programming testing and proof techniques through examples.

KH
ALL RIGHTS
RESERVED

1.7 Homework and programming exercises

1. Multiple Choice. Choose only one answer for each question. Choose the best answer if more than one answer are acceptable.

1.1 Stored Program Concept (von Neumann machine) is one of the most fundamental concepts in computer science. What programming paradigm most closely follows this concept?

❑ imperative ❑ object-oriented ❑ functional ❑ logic

1.2 If you teach someone to make a pizza, you are in fact teaching

❑ imperative programming ❑ functional programming

If you teach someone to order a pizza from a restaurant, you are in fact teaching

❑ imperative programming ❑ functional programming

1.3 Because of hardware constraints, early programming languages emphasized

❑ efficiency ❑ orthogonality ❑ reliability ❑ readability

1.4 What factor is generally considered more important in modern programming language design?

❑ readability ❑ writeability ❑ efficiency ❑ None

1.5 The main idea of structured programming is to

❑ reduce the types of control structures. ❑ increase the types of control structures.

❑ make programs execute faster. ❑ use BNF to define the syntactic structure.

1.6 In the following pseudo code, which programming language allows the mixed use of data types?

```
int i = 1; char c = 'a';   // declaration and initialization
c = c + i;                 // execution of an assignment statement
```

❑ Ada ❑ C ❑ Java ❑ All of them

1.7 In the layers of programming language structure, which layer performs type checking?

❑ lexical ❑ syntactic ❑ contextual ❑ semantic

1.8 The contextual structure of a programming language defines

❑ how to form lexical units from characters.

❑ how to put lexical units together to form statements.

❑ the static semantics that will be checked by the compiler.

❑ the dynamic semantics of programs under execution.

1.9 Interpretation is not efficient if

❑ the source program is small.

❑ the source program is written in an assembly language.

❑ the difference between source and destination is small.

❑ multi-module programming is used.

KH ALL RIGHTS RESERVED

1.10 What is the difference between an inline function and a macro in C++?

❑ There is no difference between them.

❑ The *inline* functions are for Java only. There are no inline functions in C++.

❑ *Inlining* is a suggestion to the compiler, while a *macro* definition will be enforced.

❑ A *macro* definition is a suggestion to the compiler, while *inlining* will be enforced.

1.11 Macros-Processing takes place during which phase?

❑ Editing ❑ Pre-processing ❑ Compilation ❑ Execution

1.12 Assume a function requires 20 lines of machine code and will be called 10 times in the main program. You can choose to implement it using a function definition or a macro definition. Compared with the function definition, macro definition will lead the compiler to generate, for the entire program,

❑ a longer machine code but with shorter execution time.

❑ shorter machine code but with longer execution time.

❑ the same length of machine code, with shorter execution time.

❑ the same length of machine code, with longer execution time.

2. Compare and contrast the four programming paradigms: imperative, object-oriented, functional, and logic.

3. Use the library and Internet resources to compile a table of programming languages. The table must include all programming languages mentioned in the section 1.1.3 on the development of programming languages. The table should contain the following columns:

- name of the programming language,
- year the language was first announced,
- authors/inventors of the language,
- predecessor languages (e.g., C++ and Smalltalk are predecessors of Java),
- programming paradigms (e.g., Java belongs to imperative and object-oriented programming paradigms).

The table should be sorted by the year.

4. What is strong type checking, and what are its advantages? List a few languages that use nearly strong type checking in their program compilations.

5. What is weak type checking, and what are its advantages? List a few languages that use weak type checking in their program compilations.

6. What is orthogonality? What are the differences between compositional, sort, and number orthogonality?

7. Compare and contrast a macro and an inline function in C++. Which one is more efficient in execution time? Which one is easier for programmers to write?

8. Compare and contrast the C++ inline function and Java's final method.

KH ALL RIGHTS RESERVED

9. Which type equivalence leads to strong type checking, structural equivalence, or name equivalence? Explain your answer.

10. Use BNF notation to define a small programming language that includes the definition of variable, math-operator, math-expression, condition-operator, condition-expression, assignment statement, loop statement, switch statement, and a block of statements. A variable must start and end with a letter (an unusual requirement). A single letter is a valid variable. The definition of the expression must be able to handle nested expressions like `2*(x + y)` and `5*((u + v)*(x - y))`. The language must include the following statements:

    Assignment: It assigns (the value of) a variable or an expression to a variable. For example, `x = 2*(y + z)`.

    Conditional: `if-then` and `if-then-else`. The condition in the statement can be simply defined as an expression.

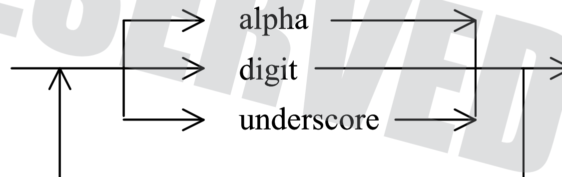
    For-loop: For example, `for (i = 0; i <10; i=i+1){`a block of statements`}`

    Switch: It must have an unlimited number of cases.

    Statement: A statement can be an assignment, conditional, for-loop, or switch statement.

    Block: One statement is a block. Zero or multiple statements in curly braces and separated by ";" is also a block, according to number orthogonality. For example, `i=i+2;` is a block. `{i=i+2; for (k=0; k<i; k=k+1) {i=i+1; s=2*i}}` is also a block.

11. The following syntax graph defines the identifiers of a programming language, where *alpha* is the set of characters "a" through "z" and "A" through "Z", *digit* is the set of characters "0" through "9", and *underscore* is the character "_".



11.1 Which of the following strings can be accepted by the syntax graph (choose all correct answers)?

❑ _FooBar25_ ❑ 2_5fooBar_ ❑ Foo&Bar ❑ 12.5 ❑ Foo2bar

11.2 Give the syntax graph that defines the identifiers always *starting* with a letter from the set *alpha*.

11.3 Give the BNF definition equivalent to the syntax graph of the question above.

12. Given the C program below, answer the following questions.

```
#define min1(x,y) ((x < y) ? x : y)
#define min 10
#include <stdio.h>
int min2(int x, int y) {
    if (x < y) return x;
```

KH ALL RIGHTS RESERVED

```
    else return y;
}
void main() {
    int a, b;
    scanf("%d %d", &a, &b);
    if (b < min)
        printf("input out of range\n");
    else {
        a = min1(a, b++);
        printf("a = %d, b = %d\n", a, b);
        a = min2(a, b++);
        printf("a = %d, b = %d\n", a, b);
    }
}
```

12.1 Give the exact C code of the statement "`a = min1(a, b++);`" after the macro processing.

12.2 Give the exact C code of the statement "`a = min2(a, b++);`" after the macro processing.

12.3 Give the exact C code of the statement "`if (b < min)`" after the macro processing.

12.4 Assume 60 and −30 are entered as inputs. What is the exact output of the program?

12.5 Assume 50 and 30 are entered as inputs. What is the exact output of the program?

KH ALL RIGHTS RESERVED

13. Assume a programming language has two sets of features. S_1 is the set of three different kinds of declarations: (1) plain, (2) initializing, and (3) constant. That is,

(1) `typex i;` (2) `typex i = 5;` (3) `const typex i = 5;` S_2 is the set of data types: (a) `bool`, (b) `int`, (c) `float`, (d) `array`, (e) `char`.

13.1 If the language guarantees sort orthogonality 1 in Figure 1.5, and we know that `int i` is allowed (the combination plain-int), list the allowed combinations of the features of the two sets that can be implied by the sort orthogonality.

13.2 If the language guarantees sort orthogonality in Figure 1.6, and we know that `const array a` is allowed (the combination constant-array), list the allowed combinations of the features of the two sets that can be implied by the sort orthogonality.

13.3 If the language guarantees the compositional orthogonality, list the combinations of the two sets of features allowed. Write a simple C program that exercises all the declarations in this question. Each declared variable must be used at least once in the program. The purpose of the program is to test whether C supports compositional orthogonality. The program thus does not have to be semantically meaningful. Test the program on Visual Studio or GNU GCC, and submit a syntax error-free program. *Note:* a variable can be declared only once. You must use different variable names in the declarations.

14. Programming exercise.

You are given the following simple C program.

```
/* assign1.c is the file name of the program. */
```

```
#include <stdio.h>     // use C style I/O library function
main () {              // main function
     int i = 0, j = 0; // initialization
     printf("Please enter a 5-digit integer \n");
     scanf("%d", &i);  // input an integer
     i = i % 10000;    // modulo operation
     printf("Please repeat the number you entered\n");
     scanf("%d", &j);  // input an integer
     j = j % 10000;    // modulo operation
     if (i == j)       // conditional statement
          printf("The number you entered is %d\n", i);
     else              // else branch
          printf("The numbers you entered are different\n");
}
```

14.1 Enter the program in a development environment (e.g., Visual C++ or GNU GCC). Save the file as assign1.c. If you are not familiar with any programming environment, please read Section B.2 of Appendix B.

14.2 Compile and execute the program.

14.3 Read Chapter 2, Section 2.1. Modify the given program. Change `<stdio.h>` to `<iostream>`. Change `printf` to `cout` and change `scanf` to `cin`, etc. Save the file as assign1.cpp

14.4 Compile and execute the program.

15. Macros are available in most high-level programming languages. The body of a macro is simply used to replace a macro call during the preprocessing stage in compilation. A macro introduces an "inline" function that is normally more efficient than an "out-line" function. However, macros suffer from side effects, unwanted or unexpected modifications of variables. Macros should be used cautiously. The main purpose of the following program is to demonstrate the differences between a function and a macro. Other purposes include learning different ways of writing comments, formatted input and output, variable declaration and initialization, unary operation ++, macro definition/call, function definition/call, if-then-else and loop structures, etc. Study the following program carefully and answer the following questions.

```
/*   The purpose of this program is to compare and contrast a function
     to a macro. It shows the side effects of a macro and an incorrect
     definition of a macro. The macros/functions abs1(x), abs2(x), and
     abs3(x) are supposed to return the absolute value of x.  */
#define abs1(a) ((a<0) ? -(a) : (a))  // macro definition
#define abs2(a) ((a<0) ? -a : a)      // macro definition
#include <stdio.h>
int abs3(int a) {                     // function definition
     return ((a<0) ? -(a) : (a));  // --> if(a < 0) return -a else return
a;
}
void main() {
     int i1 = 0, i2 = 0, i3 = 0, j1 = 0, j2 = 0, j3 = 0;
```

KH ALL RIGHTS RESERVED

```
    printf("Please enter 3 integers\n");
    scanf("%d %d %d", &i1, &i2, &i3);
    while (i1 != 123) {        // 123 is used as sentinel
         j1 = abs1(++i1 - 2);  // call a macro
         j2 = abs2(++i2 - 2);  // call a macro
         j3 = abs3(++i3 - 2);  // call a function
         printf("j1 = %d, j2 = %d, j3 = %d\n", j1, j2, j3);
         printf("Please enter 3 integers\n");
         scanf("%d %d %d", &i1, &i2, &i3);
    }
}
```

15.1 Desk check (manually trace) the program. What would be the outputs of the program when the following sets of inputs are applied to the program?

```
i1, i2, i3 = 9, 9, 9   j1, j2, j3 =
i1, i2, i3 = -5, -5, -5    j1, j2, j3 =
i1, i2, i3 = 0, 0, 0   j1, j2, j3 =
```

15.2 Enter the program into a programming environment and execute the program using the inputs given in the previous question. What are the outputs of the program?

15.3 Explain the side effects that occurred in the program.

15.4 Which macro is incorrectly defined? Explain your answer.

15.5 Change the macros in the program into inline functions.

KH ALL RIGHTS RESERVED

16. Consider the `gcd` program in Section 1.5.3. What would happen if the else-branch `swap(n, m)` in the program were changed to `n = n - m`?

16.1 Can we still prove the partial correctness?

16.2 Can we prove the termination?

16.3 Write a C program to implement the original algorithm and find a set of test cases to test your program. The test case set must cover the branches of the program.

17. Given the following algorithm in pseudo code:

```
termination(n)// precondition: n is any integer
    while n ≠ 1 do
         if even(n)
         then     n := n/2
         else     n := 3n + 1
    output(n) // postcondition: n = 1
```

17.1 Prove the program is partially correct.

17.2 Discuss whether this program terminates or not.

17.3 Write a C program to implement the algorithm, generate a set of test cases that can cover all branches of the program, and use the set of test cases to test the program.