Introduction to programming languages
Course team
Developer: Dr Man Leung Wong, Lingnan University
Designer: Mr Chris Baker, OUHK
Coordinator: Dr Tak Sing Li, OUHK
Members: Dr Reggie Kwan, OUHK
Mr Tai Loon Fong, OUHK

External Course Assessor
Prof. To Yat Cheung, City University of Hong Kong

Production
ETPU Publishing Team

Acknowledgement
The OUHK gratefully acknowledges a grant of $100,000,000 from the Hong Kong Government designated for course development.

MT311 Programming Languages and Java Programming was developed in part using resources provided by this generous grant.

Copyright © The Open University of Hong Kong, 2000.
Revised 2003.

All rights reserved.
No part of this material may be reproduced in any form by any means without permission in writing from the President, The Open University of Hong Kong.

The Open University of Hong Kong
30 Good Shepherd Street
Ho Man Tin, Kowloon
Hong Kong
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>1</td>
</tr>
<tr>
<td>Objectives</td>
<td>2</td>
</tr>
<tr>
<td>Motivations for studying concepts of programming languages</td>
<td>3</td>
</tr>
<tr>
<td>Criteria for language evaluation</td>
<td>4</td>
</tr>
<tr>
<td>Readability</td>
<td>4</td>
</tr>
<tr>
<td>Writability</td>
<td>7</td>
</tr>
<tr>
<td>Reliability</td>
<td>8</td>
</tr>
<tr>
<td>Influences on language design</td>
<td>11</td>
</tr>
<tr>
<td>Computer architecture</td>
<td>11</td>
</tr>
<tr>
<td>Programming methodologies</td>
<td>12</td>
</tr>
<tr>
<td>Language categories</td>
<td>13</td>
</tr>
<tr>
<td>Language design tradeoffs</td>
<td>14</td>
</tr>
<tr>
<td>Some major programming languages</td>
<td>16</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>16</td>
</tr>
<tr>
<td>Pascal</td>
<td>17</td>
</tr>
<tr>
<td>C</td>
<td>17</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>18</td>
</tr>
<tr>
<td>C++</td>
<td>18</td>
</tr>
<tr>
<td>LISP</td>
<td>19</td>
</tr>
<tr>
<td>Prolog</td>
<td>20</td>
</tr>
<tr>
<td>Summary</td>
<td>22</td>
</tr>
<tr>
<td>Feedback to activities</td>
<td>23</td>
</tr>
<tr>
<td>Solutions to self-tests</td>
<td>25</td>
</tr>
</tbody>
</table>
Overview

This is the second block of the course. The main aim of this block is to evaluate programming languages in terms of how well they can support software development and to develop the criteria needed for such evaluation. You will also be presented with some design issues and implementation considerations for different constructs of various languages. Four language paradigms, namely the imperative, functional, logic and object-oriented paradigms, will be discussed and we will look at how each paradigm has contributed to software development.

Although this block is not about programming, you should be able to write better programs after studying the internal mechanism of different languages. For example, if you understand how the storage for a variable is allocated, you can avoid declaring variables inside a `for` loop in C.

We begin Unit 6 by looking at why we study programming languages concepts. Then we introduce criteria for evaluating programming languages, namely, readability, writability, cost and reliability. These criteria are not independent of each other, and sometimes they are even in conflict with each other.

After this, we discuss how computer architecture and software development methodologies affect the design of programming languages. Then we present the four categories of programming languages: imperative, functional, logic and object-oriented.

We then move on to discuss some tradeoffs in designing a language. We do not aim to teach you how to design a language. Instead, we hope that you are able to appreciate why a language is designed in ways that initially may seem puzzling or illogical. For example, we will explore why most languages allow an un-initialized variable to be used on the RHS of an assignment statement.

We conclude this unit by giving you an overview of some popular programming languages. These languages will be used from time to time as examples in subsequent units.

The course materials for MT311 Programming Languages and Java Programming consist of the course study units, your two set textbooks and extra readings. Please refer to the text by Sebesta as you work through Unit 6.
Objectives

By the end of Unit 6, you should be able to:

1. Specify some reasons for studying principles and concepts of programming languages.

2. State the criteria for evaluating various programming language constructs and designs.

3. Outline the four categories of programming languages: imperative, functional, object-oriented and logical.

4. Describe some major programming languages.
Motivations for studying concepts of programming languages

After completing this section, you should be able to specify the motivations for studying principles and concepts of programming languages (unit objective 1).

If you have followed our recommendation to take MT210 and MT258 before taking this course, you have already studied two languages. Perhaps you may argue that, having studies those courses, you already know enough about programming languages. Why, then, do we think that you need to study the concepts of programming languages? This is a good question. The following reading provides an answer to you.

Reading

Sebesta, 5th edn, section 1.1, pages 2–5.

In summary, studying the concepts of programming languages helps you to:

• **Simulate a useful facility, which is not supported in the language you are forced to use.** For example, after studying how virtual methods are implemented in C++, you may try to simulate this facility in C.

• **Choose a suitable language for a project.** After you know the criteria for critically evaluating a programming language, you are much more capable of choosing an appropriate language.

• **Learn or design a new language more easily.** Most languages share common fundamental principles. For example, after studying the object-oriented language paradigm, you should find it easier to learn C++ or Java.

• **Write codes that are more efficient or fix a bug more easily.** By studying the implementation and mechanism of different constructs, you should find that you are able to understand a language more deeply. For example, after studying the mechanism of parameter passing, you would avoid passing a large array as value parameter to a function in Pascal.

**Activity 1.1**

Do you need to select programming languages for solving problems in your daily work? If yes, which languages have you selected and what are the reasons for choosing these languages?

In the next section of this unit we will look at some criteria that you can use to evaluate and choose programming languages.
Criteria for language evaluation

After completing this section, you should be able to state the criteria for evaluating various programming language constructs and designs (unit objective 2).

In the 1960s, when FORTRAN was first developed, people were more concerned with what they could do with the language rather than how easily a task could be done. As the size of projects became larger and larger, people found that languages like FORTRAN had problems and that programs were difficult to write and read. This had serious effects on the reliability of the programs and made maintenance very expensive. Consequently programmers realized that when a language is designed, it is important to consider the factors that affect a software’s development. These factors include:

- readability
- writability
- reliability and
- cost.

Let’s discuss each of the four criteria in turn.

Readability

Readability of a program measures how easily it can be understood. If a program is difficult to read, then it is difficult to implement and maintain. Since the maintenance stage is the longest stage in the software life cycle, it is important for a language to have high readability. The following reading discusses the factors that affect the readability of a language.

Reading

Sebesta, 4th edn, section 1.3.1, pages 8–14.

Sebesta, 5th edn, section 1.3.1, pages 8–15.

Five characteristics of programming languages can affect their readability:

1. Overall simplicity of a programming language. There are three factors that determine overall simplicity. They are:

   - Number of basic components. For example, there are 31 different binary operators in C:
Even a competent programmer would find it difficult to remember all of their functions.

- Alternative means for accomplishing the same operation. The method that is used by the author may not be one with which the reader is also familiar. For example, in C, the 3rd element of an array \( A \) can be addressed in two ways:

\[
A[2] \quad \text{or} \quad *(A+2) \quad (\text{Note that the 1st element is } A[0])
\]

If the programmer uses the latter form, then it is much less readable.

- The meanings of operators are re-definable. Programs are difficult to read if such definitions are not of common sense. For example, if \( A \) and \( B \) are records and \( A+B \) is defined to be the sum of one of the fields of the two records (like salary). Unless you trace back to the definition of the operator, it is highly unlikely that you will understand this statement.

2 Number of exceptional rules (orthogonality). If a programming language has only a small number of rules and the exceptions to these rules are few, then the language is easier to learn and therefore easier to read. For example, all data in Smalltalk is object and all respond to the message \texttt{class}, which returns the class type of the object. Therefore, you should have no problem to understand the meaning when you encounter this statement.

3 Control statements. It is known that program readability is severely reduced by indistinguishing usage of \texttt{goto} statements. If a programming language has sufficient control constructs, the need for \texttt{goto} statements can be nearly eliminated. This increases the readability.

4 Data types and data structures. The availability of appropriate facilities for defining user-defined data types and data structures in a language can also improve program readability significantly. For example, assume that an inventory record has two fields: item number and price. In Pascal, you would have defined it as:

```pascal
record inventory_record
  item_number: integer;
  price: integer;
end;
```
If you want 100 such records, you would have to declare the array as:

```plaintext
inventory_records:array[1..100] of inventory_record;
```

Therefore, the item number and price of the 8th record would be written as:

```plaintext
inventory_records[8].item_number
inventory_records[8].price
```

Their meanings are self-explanatory.

However, since FORTRAN does not allow you to declare a record, the array of the records has to be separated into two arrays:

```plaintext
integer itemno(100)
n integer price(100)
```

The item number and price of the 8th record would be written as:

```plaintext
itemno(8) and price(8) respectively.
```

Unless you are the author of this program, it is unlikely that you will notice that `price(8)` is the price of the item with item number `itemno(8)`.

5 Syntax of a programming language. Syntax affects readability in three ways:

- **Identifier forms.** If a language allows longer names, the program would be easier to understand. For example, `inventory_level` is much more self-explanatory than `invlev`.

- **Special words.** Consider the following C program fragment:

```c
for (i=1; i<10; i++) {
    res=a[i]-b[i];
    if (res>0) {
        temp=a[i];
        sum[i]=0;
        j=i;
        while (sum<100 && j<10) {
            sum[i]+=a[j];
            j++;
        }      
    } else {
        temp=b[i];
        sum[i]=0;
        j=i;
    }
```

C uses ‘{’ and ‘}’ to enclose a block of statements. Therefore it is sometimes very difficult to decide which ‘{’ and ‘}’ form a pair. In a small fragment like the one above, we can use indentation to identify the statements inside a block. This will not work if there are many statements inside a block because the beginning and end of the block are too far away to be displayed on one screen. If the same program fragment was written in Ada, it would look like this:

```ada
for i in 1..9 loop
    res:=a[i]-b[i];
    if res>0 then
        temp:=a[i];
        sum[i]:=0;
        j:=i;
        while sum<100 and j<10 loop
            sum[i]:=sum[i]+a[j];
            j:=j+1;
        end loop;
    else
        temp:=b[i];
        sum[i]:=0;
        j:=i;
        while sum<100 and j<10 loop
            sum[i]:=sum[i]+b[j];
            j:=j+1;
        end loop;
    end if;
end loop;
```

This fragment of program is easier to understand because different `end` statements are used for ending different types of blocks.

**Writability**

Writability is a measure of how easily a language can be used to write programs. The following reading discusses the factors that affect the writability of a programming language.

**Reading**

Sebesta, 4th edn, section 1.3.2, pages 15–17.

Sebesta, 5th edn, section 1.3.2, pages 15–17.
After studying the previous reading, you should have noted that program writability is affected by the following factors:

1. **Simplicity** and fewer exceptional rules. If a language consists of a small number of primitive constructs and there are only a few exceptional rules, then it is easier to write because there is less to remember.

2. **Support for abstraction.** Abstraction allows programmers to hide the implementation details of complicated data structures and operations, and thus encourages and simplifies the use of these details. For example, a stack can be implemented using pointer or array. The user of the stack would not be affected if this underlying structure were changed.

3. **Expressivity.** This means that a language has convenient ways of specifying computation. For example, there is a multiple assignment statement in C like:

   \[
   a=b=c=1; 
   \]

   If this is to be done in Pascal, it has to be written as three separate assignment statements:

   \[
   a:=1; \\
   b:=1; \\
   c:=1; 
   \]

**Reliability**

A program is reliable if it performs its specified operations under all conditions. The next reading describes the language features that affect program reliability.

**Reading**

Sebesta, 4th edn, section 1.3.3, pages 17–18.

Sebesta, 5th edn, section 1.3.3, pages 17–18.

After reading the above notes, you should now see that the following features can affect program reliability:

- **Type checking.** If the parameters passed to a function are not those expected, then errors will occur. Therefore, it is important to check that the types are correct. Usually, this is done in most languages by restricting the programmer to declare a function first before it can be used somewhere. Languages like FORTRAN, and the original C language do not do any type checking for parameters of function calls. Therefore, they are very unreliable.

- **Exception handling.** This is a mechanism that intercepts run-time errors, takes corrective procedures and then continues the program
execution. Without this, a program that encountered errors might stop or continue to do something that would cause damages to the system.

• **Alias-ing.** This refers to the situation of having two or more different referencing methods for the same memory location. Alias-ing significantly reduces program reliability. In Pascal, you can define a variant record like this:

```pascal
type Shape=(Square, Rectangle);
Dimensions=record
    case WhatShape: Shape of
        Square: (Side1: real);
        Rectangle: (Length, Width: real)
    end;
```

In this example, `Dimensions` can have two forms: one for a `Square` and the other for a `Rectangle`. This means that the memory occupied by this record may be accessed by either the `Side1` field or the `Length` field. However, Pascal will not complain when a `Dimensions` record of a `Square` is used as `Rectangle`. This would, of course, be an error because the `Length` and `Width` fields would be rubbish.

• **Readability and Writability.** Usually the easier the program to write the more likely it is to be correct. However, we will later show that sometimes writability can decrease reliability. The easier the program is to read, the easier it is to spot errors.

Before you continue on to the next reading, please attempt the following activity. This activity asks you to think about how cost may influence the choice of a language program.

**Activity 1.2**

There are many factors that affect the cost of a programming language. Can you list some of them?

Were you able to list many factors? The next reading discusses these factors. After completing this reading, you can compare the list of factors you created in Activity 1.2 with the factors mentioned in the readings.

**Reading**

Sebesta, 4th edn, section 1.3.4, pages 18–20.

Sebesta, 5th edn, section 1.3.4, pages 18–20.
In summary, the seven factors that influence the cost are:

- **The training cost.** If the language is simpler, with fewer exceptional rules and the programmer is more experienced, then the cost is less.

- **The cost of writing programs.** Higher writability reduces the cost of writing programs.

- **The cost of compiling programs.** This includes the time and resources that are required in compiling a program. For example, some compilers have to read in a program source twice (known as 2 passes compilers). They are therefore more time consuming and require more resources than those that have to read in a program only once.

- **The cost of running programs.** Interpreted languages like Basic have higher running costs than compiled languages like Fortran. But, of course, Basic programs need not be compiled in the first place and therefore have no cost in compilation. There are always tradeoffs between compilation cost and running cost.

- **The cost of the language implementation system.** Some languages like OCCAM require a special hardware platform known as transputer to run. Therefore they are more expensive.

- **The cost of reliability.** For critical systems, the cost is very high.

- **The cost of maintaining programs in the language.** Higher readability would reduce this cost as the program could be easily understood and therefore would make enhancement and bug fixing easier.

---

### Self-test 1.1

You should now have a good general idea of the criteria for evaluating various programming language designs. To help consolidate your grasp of these criteria, please complete this self-test. Feedback to the questions below is provided at the end of this unit.

The purpose of this exercise is to get you thinking about how readability is influenced by the availability of language constructs. In this exercise, I assume that you have knowledge of a high-level programming language such as C or Pascal.

In C, the closing reserved word `}` is used for a number of control statements including if statement, if/else statement, for statement etc. Similarly, the closing reserved word `end` is used for a number of control statements in Pascal.

- What readability problem is caused by using the same closing reserved word for a number of different statements? Explain your answer by using one or more programs.

- What readability problem is caused by using short identifiers? Explain your answer by using one or more programs.
Influences on language design

There are two more factors that can affect the design of a language:

- computer architecture
- software development methodology.

Computer architecture

All programs have to be executed on a hardware platform. Therefore, the architecture of the hardware platform would affect the design of a language. One extreme example is the FORTRAN IV restrictions on array subscripts because of the addressing hardware of the IBM 709.

Most current computers have the so-called von Neumann architecture. They are therefore called von Neumann computers. There are two major components in the architecture:

- Memory, which stores program instructions and data;
- Central Processing Unit (CPU); which executes program instructions.

The two components are separate. Therefore, instructions and data must be transmitted first from memory to the CPU before the CPU can execute the instructions. After the execution of the instructions, results produced by the CPU must be moved back to memory. A number of constructs and features of imperative programming languages model the computing process of von Neumann architecture.

The next reading describes imperative languages and the relationship between von Neumann architecture and these languages.

Reading

Sebesta, 4th edn, section 1.4.1, pages 20–22.
Sebesta, 5th edn, section 1.4.1, pages 20–22.

The previous reading also presents functional programming languages. In these languages, computations are performed by applying functions to given parameters. These languages have the characteristic that computation can be done without:

- the kind of variables used in imperative languages
- assignment statements
- iteration statements.

However, because these languages do not model the computation process of von Neumann architecture, they are not as efficient as imperative languages on von Neumann computers. As a result, functional programming languages are not as popular as imperative languages.
Programming methodologies

Programming languages are major vehicles for developing software systems. As software systems become more complicated and the development cost is increasing, different system development methodologies have emerged to formalize the system development process, reduce development cost, and improve system reliability. Some examples of these methodologies are:

- **Top-down design and stepwise refinement.** The main task is subdivided into smaller tasks. These smaller tasks are further subdivided until they are small enough to be implemented as a module or a subprogram. For example, the program for controlling an automatic teller machine can be subdivided into the following sub-tasks: password validation, balance checking, money delivery, updating account. Then these tasks may be further divided.

- **Object-oriented design.** The objects exist in the real world are modeled by encapsulating their attributes and operations. Then the whole system is modeled by the interactions between these objects. For example, one of the objects in the automatic teller machine system is the customer. The attributes of the customer are customer number, name, etc. An operation of a customer is money withdrawal.

- **Process-oriented design.** Different processes are modeled separately and the whole system is modeled by the interactions between these processes. This is mainly used in real time concurrent systems. For example, in a nuclear power plant system, there should be a process that regulates the water temperature and a process waiting to receive commands from a control panel. When the latter process receives a command to regulate the water temperature, it passes a signal to the former process to do the job.

The next reading explains how these system design methodologies affect the design of programming languages.

**Reading**

Sebesta, 4th edn, section 1.4.2, pages 22–23.

Sebesta, 5th edn, section 1.4.2, pages 22–23.

The stepwise refinement methodology promotes the use of subprograms and modules. For example, standard Pascal does not support modules and therefore is not very useful when this methodology is used.

The object-oriented methodology promotes the use of object-oriented programming languages. Therefore, when this methodology becomes more popular, object-oriented programming languages would also gain in popularity.

The process-oriented programming methodology promotes the use of process-oriented programming languages like CSP or Ada.
Language categories

After completing this section, you should be able to outline the four categories of programming languages (unit objective 3).

There are many programming languages available today. In order to systematically analyse and compare the features and constructs of these languages, they have been classified into four categories:

1. Imperative languages. The procedures on how to perform the computation are stated explicitly, e.g., C, Pascal;

2. Object-oriented languages. The behavior of different objects are stated explicitly, e.g., Smalltalk, Java;

3. Functional programming languages. The result, but not the procedures, of applying a function to some parameters are stated explicitly, e.g., LISP;

4. Logic programming languages. The rules that have to be observed are stated explicitly, e.g., Prolog.

The next reading briefly introduces these categories

**Reading**

Sebesta, 4th edn, section 1.5, pages 23–24.

Sebesta, 5th edn, section 1.5, pages 23–24.

The previous reading is only a brief introduction of the four categories of programming languages. These categories will be discussed in detail in the following units of this course. Specifically, the features of imperative languages will be presented in Units 8 and 9. Object-oriented languages will be described in Unit 10. Functional and logic programming languages are discussed in Unit 11.
Language design tradeoffs

We have so far presented the factors that are used to evaluate a programming language. These factors are sometimes in conflict with each other. For example, FORTRAN does not require programmers to declare a variable first before use. Writability is increased because the programmer can use a new variable wherever it is needed. If you were writing a Pascal program and you wanted to use a new variable, then you had to go back to the beginning of the function or procedure and declare the variable there. However FORTRAN is less reliable because the compiler would not be able to report an error if the programmer has made a typo in an identifier name.

Activity 1.3

Do you think that, in general, the readability and reliability of a programming language are in conflict with each other? Think about your answer before you look at my comments below, and try to give reasons to support your answer.

Indeed, the four criteria for evaluating programming languages — readability, writability, reliability, and cost — are sometimes conflicting. The next reading illustrates the problem of design tradeoffs. Please look at my additional feedback about readability and reliability at the end of the unit.

Reading

Sebesta, 4th edn, section 1.6, pages 24–25.

Sebesta, 5th edn, section 1.6, pages 24–25.

After completing the reading above, you should understand that there are conflicts between reliability and cost, and between readability and writability. Consequently, if it is necessary for you to design a new programming language for your work, you must consider these conflicts carefully and make compromises and tradeoffs.

The following self-test will help you consolidate this idea of tradeoffs among different criteria.
Self-test 1.2

The purpose of this exercise is to test your understanding of the tradeoffs between readability and writability. In this exercise, I assumed that you have knowledge about some high-level programming languages.

Write a program segment in a high-level programming language to demonstrate the conflict between readability and writability.

Don’t forget to look at my suggested answers after you have completed this self-test.
Some major programming languages

After completing this section, you should be able to describe the evolution of the major programming languages (unit objective 4).

In this section, you will look at the historical background and design of a number of major programming languages. I would like to use the four categories described in the previous section of the unit to classify several of these languages into the following categories:

- imperative languages
  - FORTRAN
  - Pascal
  - C
- object-oriented languages
  - Smalltalk
  - C++
- functional programming language
  - LISP
- logic programming language
  - Prolog

**FORTRAN**

FORTRAN is an imperative programming language designed for scientific computations. The next reading introduces its historical background and different versions of FORTRAN. With your background in the course MT258 Programming and Problem Solving, you should be able to understand the FORTRAN program that is provided at the end of the reading.

**Reading**

Sebesta, 4th edn, section 2.3, pages 44–49.
Sebesta, 5th edn, section 2.3, pages 44–49.

Having worked through this reading, you should know that FORTRAN has the following language constructs:
• input and output formatting statements;
• user-defined subroutines that can be compiled separately;
• an arithmetic IF statement;
• a logical IF statement;
• an IF/ELSE statement;
• the DO loop statement;
• a logical loop control statement;
• explicit type declarations for variables;
• character string processing statements;
• array processing functions;
• a case statement;
• dynamic array.

Moreover, only FORTRAN 90 supports recursive subroutines, optional parameters and keyword parameters for subroutines.

**Pascal**

Pascal is an imperative language designed for teaching programming. Its simplicity and expressivity make it particularly suitable for this purpose. The next reading introduces you to Pascal.

**Reading**

Sebesta, 4th edn, section 2.12.1, pages 77–79.

Sebesta, 5th edn, section 2.12.1, pages 77–79.

**C**

The next reading gives you a brief description of the background and design of C.

**Reading**


Sebesta, 5th edn, section 2.12.2, pages 79–82.
**Activity 1.4**

Briefly write down some reasons why C is popular. Compare your answers to my feedback provided the end of this unit.

---

**Smalltalk**

Smalltalk is an object-oriented programming language that will be described in detail in *Unit 10*. It has the four basic components of all object-oriented programming languages, namely:

1. classes
2. objects
3. inheritance and
4. polymorphism.

In the next reading, you should focus mainly on the historical background of Smalltalk and only briefly look at its appearance and format. At this stage of the course, you are not required to understand this program.

**Reading**

Sebesta, 4th edn, section 2.15, pages 91–94.

Sebesta, 5th edn, section 2.15, pages 91–94.

---

**C++**

C++ is an enhancement of C that combines features of object-oriented and imperative programming languages. In the next reading, the background and design of C++ is briefly described.

**Reading**

Sebesta, 4th edn, section 2.16, pages 94–96 (excluding 2.16.4).

Sebesta, 5th edn, section 2.16, pages 94–96 (excluding 2.16.4).

After completing the previous reading, you should know that C++ has the following characteristics:

- C++ supports function parameter type checking and conversion.
• It supports the definitions of classes, subclasses, and friendly classes.
• There are public, private, and protected access control modifiers.
• Classes in C++ can have constructor and destructor functions.
• It provides single and multiple inheritance.
• Operators and functions can be overloaded.
• Programmers can define virtual functions to achieve the goal of polymorphism.
• It has exception handling facilities.

Activity 1.5

Imagine that a software company has developed all of its software products in C. Now, the director is considering developing new products in C++, and thus needs to employ you as a consultant to study the feasibility of realizing this goal. What are your suggestions to him?

When you have thought about your suggestions, refer to the feedback at the end of this unit.

LISP

LISP is a functional programming language used mostly in Artificial Intelligence applications. Pure LISP has two kinds of data structures:

1. **Atoms**, which are either symbols or numeric literals.
2. **Lists**, which are specified by delimiting their elements with parentheses. For example, (1 2 3) and ((1 2) (3 4) (5) 6) are lists. The former is a simple list whose elements are atoms. The latter is a nested list whose elements can be atoms or lists.

The next reading gives a brief overview of LISP. You are not required to understand this program, so please just skim through the reading and familiarize yourself with its format.

Reading

Sebesta, 4th edn, section 2.4, pages 49–55.
Sebesta, 5th edn, section 2.4, pages 49–55.

The readings in this section should have given you a basic idea of what functional programming languages are. I will discuss functional programming languages in more detail in Unit 10.
Prolog

Prolog is a logic programming language that uses a formal logic notation to communicate computational processes to computers. Like LISP, it is used mainly in developing artificial intelligence applications. Prolog is a non-procedural because programs written in it specify what kind of results are desirable, rather than how to compute the results.

The next reading gives you some background information on Prolog. As with the functional programming languages, more details about logic programming languages will be discussed in Unit 10.

Reading

Sebesta, 4th edn, section 2.13, pages 84–85.
Sebesta, 5th edn, section 2.13, pages 84–85.

Now, there is a self-test for you to complete. This self-test helps you to assess your understanding of some of the reading that you’ve done in this section of the unit. When you finish the self-test, be sure to check your answers against the sample answers provided at the end of this unit.

Self-test 1.3

The purpose of this exercise is to test your ability of reading and understanding simple FORTRAN programs. Explain the action of the program given below line by line. In this FORTRAN program, the line number at the beginning of each line is not part of the program.

```
1 INTEGER INTLIST(99)
2 INTEGER LIST_LEN, COUNTER, SUM, AVERAGE, RESULT
3 RESULT = 0
4 SUM = 0
5 READ *, LIST_LEN
6 IF ((LIST_LEN .GT. 0) .AND. (LIST_LEN .LT. 100)) THEN
7     DO 100 COUNTER = 1, LIST_LEN
8         READ *, INTLIST(COUNTER)
9         SUM = SUM + INTLIST(COUNTER)
10     100 CONTINUE
11     AVERAGE = SUM / LIST_LEN
12     DO 200 COUNTER = 1, LIST_LEN
13         IF (INTLIST(COUNTER) .LT. AVERAGE) THEN
14             RESULT = RESULT + 1
15         END IF
16     200 CONTINUE
17 PRINT *, RESULT
18 ELSE
```
20 PRINT *, 'LENGTH VALUE IS NOT LEGAL'
21 END IF
22 STOP
23 END
Summary

In this unit you have thought about some reasons for studying concepts of programming languages. Unit 1 has introduced you briefly to some key considerations in language programming.

By now you should have completed the following:

1. used some criteria for evaluating different programming language designs and constructs;
2. learned the four categories of programming languages;
3. thought about the major tradeoffs during language design;
4. explored the historical background and design of nine major programming languages.

If you met the objectives for this unit, you are ready to move on to Unit 2 and explore the fundamental semantic issues of various programming languages.

Don’t forget to look at the list of optional references on the next page and be sure to follow up any topics that interest you.
Feedback to activities

Activity 1.3

In general, higher readability means that a program is easier to understand. This means that bug fixing is easier and therefore that the program is more reliable. Therefore, readability and reliability are, in general, not conflicting each other.

Activity 1.4

C is popular for a number of reasons. Some of these reasons include:

- **C is simple and small.** The C language does not contain many of the built-in features present in other programming languages such as PL/1 and COBOL. For example, COBOL has about 300 reserved words and extensive capabilities to do input and output to several types of files. These features greatly increase the complexity of the COBOL language. On the other hand, C has only 32 reserved words. Moreover, input and output facilities are not built into the C language. Instead, C provides input and output facilities through a library of functions. Consequently, C is a simple and small language.

- **C is portable.** C is one of the most portable high-level languages. A C program can run on several platforms (PCs, Macintoshes, workstations, mini-computers, and mainframes) with no modifications if it is written properly. Portability eliminates the cost of developing the application several times for different platforms. Consequently, many software developers choose C to write their programs.

- **C is available in all UNIX machines.** Many mainframes, mini-computers, workstations use the UNIX operating system. Because UNIX was developed mainly in C, all versions of UNIX contain a version of C as their native language. Because C is available in all UNIX machines, developers find that it is convenient to develop programs on these machines.

- **C has the advantages of high-level and low-level languages.** A low-level language allows the programmer to control the hardware and operating system of a computer system directly. However, low-level languages are difficult to learn because they are highly symbolic and the programmer must take care of all the details of the program. Low-level language programs are difficult to maintain and debug. Low-level language programs are not portable. A program is specific to the hardware and operating system of the computer system on which it is developed.

On the other hand, high-level languages, such as FORTRAN, COBOL, ALGOL and Pascal, are easier to learn, maintain, and debug. However, these languages do not allow the programmer to control a computer system directly.
Although C is a high-level language, it also contains many low-level capabilities of controlling the hardware and operating system. Consequently, C contains both the advantages of high-level and low-level languages.

This list is not exhaustive and I’m sure that you have also thought of some other reasons why C is so popular.

**Activity 1.5**

Some possible suggestions that you could give include:

- *The availability of experienced C++ programmers.* We should ensure that there are enough experienced C++ programmers to develop new products in C++. We can employ new C++ programmers or encourage some existing C programmers to learn C++.

- *The availability of appropriate development tools.* There are many C++ development tools including C++ compilers, debuggers, integrated development environments (IDE). We must evaluate these tools to determine if they are compatible with each other and with the underlying operating system and hardware. After determining the most appropriate tools, we should purchase them.

- *The availability of powerful hardware.* Since C++ is a more powerful language than C, its hardware requirement is much more demanding than C. We should ensure that there are enough powerful machines for software development in C++.

- *The portability of existing software products.* Usually, new software products are based on the program codes of existing software products, thus, we should determine if the existing program codes can be ported to C++.

- *The migration plan.* Although all new products will be developed in C++, there should be a number of C programmers to maintain and support the existing products for a long period. Thus, we should arrange an appropriate migration plan and schedule. The plan will describe the numbers of C and C++ programmers at each phase, and how to migrate existing C programmers to other job positions.

Don’t worry if your suggestions are slightly different than mine. Other reasonable suggestions are also acceptable.
Solutions to self-tests

Self-test 1.1

We will use a program in C to demonstrate the readability problem caused by using } as the closing reserved word. In the following C program, the numbers at the beginning of each line are not part of the program, they are used as labels for later discussion.

```
void main() {
    int intlist[99], listlen, counter, sum, average, result;
    result = 0;
    sum = 0;
    scanf("%d", &listlen);
    if ((listlen > 0) && (listlen < 100)) {
        for (counter = 0; counter < listlen; counter++) {
            scanf("%d", &intlist[counter]);
            sum = sum + intlist[counter];
        }
        average = sum / listlen;
        for (counter = 0; counter < listlen; counter++)
            if (intlist[counter] < average)
                result = result + 1;
        printf("%d\n", result);
    }
    else
        printf("list length is not incorrect\n");
}
```

The above C program gets an integer as input and stores it into the integer variable listlen. If the value in listlen is greater than zero and less than 100, the program repeatedly gets a number of integer values and stores them into the array intlist. After getting all input values, the program calculates the average of all input values and finds the number of input values that are smaller than the average.

However, the readability of this program is reduced by using the same symbol } as the closing word. For example, the } symbol at line 10 is paired with the { symbol at line 7. If readers do not read the program carefully enough, it is possible to misunderstand that the } symbol is paired with the { symbol at line 6. Similarly, the } symbol at line 16 is paired with the { symbol at line 6, it is possible to misunderstand that the } symbol is paired with the { symbol at line 7.

From the simple program above, you should realize that it is not easy to find statement groups for a short program, and it will be very difficult and tedious to identify statement groups for long programs containing thousands of lines of codes.

We will modify the C program above to demonstrate the readability problem caused by using short identifiers. In the following C program, identifiers with only a single letter are used:
20 void main() {
21     int i[99], l, c, s, a, r;
22     r = 0;
23     s = 0;
24     scanf("%d", &l);
25     if ((l > 0) && (l < 100)) {
26         for (c = 0; c < l; c++) {
27             scanf("%d", &i[c]);
28             s = s + i[c];
29         }
30         a = s / l;
31         for (c = 0; c < l; c++)
32             if (i[c] < a)
33                 r = r + 1;
34             printf("%d
", r);
35     }
36     else
37         printf("list length is not incorrect\n");
38 }

Comparing this program with the program in the previous page, you can realize that this program is much more difficult to understand, because it is difficult to know the meanings and usage of each variable. For example, you may not know easily that \(s\) is the sum of all input values, \(a\) is the average of all input values, and \(l\) is the number of input values.

**Self-test 1.2**

We will use two program segments in C to demonstrate the tradeoff between readability and writability. The first program segment implements a stack using an array. It also includes a `push` function that stores an element into the stack. In this program segment, the numbers at the beginning of each line are not part of the program, they are used as labels for later discussion.

```c
1  #define stacklen 100
2  int stack[stacklen]
3  int top = -1
4
5  void push(int element) {
6      if (top < stacklen - 1) {
7          top = top + 1;
8          stack[top] = element;
9      }
10  } else
11      printf("Stack full\n");
12 }
```

In this program segment, `stack` is an integer array that can contain 100 elements, `top` is an variable that stores the current position of the `top` of the stack. The function `push` is defined from lines 5 to 12. Firstly, the function checks if the stack is full (line 6). If it is not full, the function increases the value of `top` by 1, so that `top` is pointing to the next available slot of the stack. The input element is then stored in the slot
pointed to by \texttt{top}. If the stack is full, the function simply displays a message telling that the stack is full.

You should be able to understand the above program segment easily because its readability is relatively high. On the other hand, the next program has higher writability.

\begin{verbatim}
13    #define stacklen 100
14    int stack[stacklen]
15    int top = -1
16
17    void push(int element) {
18        if (top < stacklen - 1) {
19            stack[++top] = element;
20        }
21    } else
22        printf("Stack full\n");
23 }
\end{verbatim}

By comparing these two program segments, you can find that the second one is shorter because the statements at lines 7 and 8 are replaced by an equivalent statement 19. Thus, the second program segment has higher writability. However, it is more difficult to understand this statement if you are not familiar enough with the syntax of C.

\textit{Self-test 1.3}

We will explain the FORTRAN program line by line. The first line defines an integer array \texttt{INTLIST} containing 99 elements. The second line defines 5 integer variables. They are \texttt{LIST\_LEN}, \texttt{COUNTER}, \texttt{SUM}, \texttt{AVERAGE}, and \texttt{RESULT}. The integer variable \texttt{RESULT} is initialized to 0 at line 3. Similarly, the variable \texttt{SUM} is initialized to 0. The number of input values is inputted into the variable \texttt{LIST\_LEN} at line 5. The lines 6 and 7 determine if \texttt{LIST\_LEN} is greater than 0 and less than 100. If it is, the statements from lines 8 to 18 are executed. Otherwise, the statement at line 20 displays the message ‘LENGTH VALUE IS NOT LEGAL’.

The statements from lines 8 to 11 form a \texttt{DO} loop that repeatedly reads a number of input values and stores them into the array \texttt{INTLIST}. The sum of the input values is also calculated and stored in the variable \texttt{SUM}.

The statement at line 12 calculates the average of the input values and stores it into the variable \texttt{AVERAGE}.

The statements from lines 13 to 17 form a \texttt{DO} loop that repeatedly examine each element stored in the array \texttt{INTLIST}. If an element value is less than the average of all input values (line 14), the value of the variable \texttt{RESULT} is increased by 1 (line 15). The purpose of the \texttt{DO} loop is to determine the number of input values that are smaller than the average of all input values.

The statement at line 18 prints the number of input values that are smaller than the average of all input values.