Chapter 6: Programming Languages

6.1 Historical Perspective

6.2 Traditional Programming Concepts

6.3 Procedural Units

6.4 Language Implementation

6.5 Object Oriented Programming

6.6 Programming Concurrent Activities

6.7 Declarative Programming

1st Generation: Machine Language

\*Machine language

\*Operations in op-codes

 Operands

 Numerical values

 Register number

 Memory location address

2nd Generation: Assembly Language

\*A mnemonic system for representing programs

 \*Mnemonic: easy to remember

\*More descriptive

 \*Enabling programming without tables such as the one in Appendix C

\*Things are mnemonic

 \*Op-codes in mnemonic names

 \*Registers in mnemonic names

 \*Memory locations in mnemonic names of the programmer’s choice (Identifiers/variables)

Just a Little Step Further

\*One-to-one correspondence between machine instructions and assembly instructions

\*Inherently machine-dependent

\*Converted to machine language by a program called an assembler

\*Thing are easier to remember, yes.

\*But programmer still needs to think like the machine!

Third Generation Language

\*Uses high-level primitives

 \*Similar to our pseudocode in Chapter 5

\*Machine independent (mostly)

\*Each primitive corresponds to a short sequence of machine language instructions

 Converted to machine language by a program called compiler

\*Examples: FORTRAN, COBOL, BASIC

Compilers vs. Interpreters

\*Compilers

 \*Compile several machine instructions into short sequences to simulate the activity requested by a single high-level primitive

 \*Produce a machine-language copy of a program that would be executed later

\*Interpreters

 \*Execute the instructions as they were translated

Imperative Paradigm

\*Procedural paradigm

\*Develops a sequence of commands that when followed, manipulate data to produce the desired result

\*Approaches a problem by trying to find an algorithm for solving it

Object-Oriented Paradigm

\*Grouping/classifying entities in the program

 \*Entities are the objects

 \*Groups are the classes

 \*Objects of a class share certain properties

 \*Properties are the variables or methods

\*Encapsulation of data and procedures

 \*Lists come with sorting functions

\*Natural modular structure and program reuse

 \*Inheriting from mother class definitions

\*Many large-scale software systems are developed in the object oriented fashion

Object vs. Class

\*Some objects can be categorized into the same class

 \*Desk , chair -> furniture

\*Objects in the same class might share the same property

 \*Desk, chair -> four legs

Declarative Paradigm

\*Emphasizes

 \*“What is the problem?”

 \*Rather than“What algorithm is required to solve the problem?”

\*Implemented a general problem-solving algorithm

\*Develops a statement of the problem compatible with the algorithm and then applies the algorithm to solve it

Functional Paradigm

\*Views the process of program development as connecting predefined “black boxes,” each of which accepts inputs and produces outputs

\*Mathematicians refer to such“boxes”as functions

\*Constructs functions as nested complexes of simpler functions

LISP Expressions

(Divide (Sum Numbers)

(Count Numbers))

(First (Sort List))

Advantages of FP

\*Constructing complex software from predefined primitive functions leads to well-organized systems

\*Provides an environment in which hierarchies of abstraction are easily implemented, enabling new software to be constructed from large predefined components rather than from scratch

Types of Statements

\*Declarative statements

 \*Define customized terminology that is used later in the program

\*Imperative statements

 \*Describe steps in the underlying algorithms

\*Comments

 \*Enhance the readability of a program

Declaration Statements

\*Data terms

 \*Variables變量

 \*Literals文字

 \*Constants常數/固定的( const int )

\*Data types

 \*Common types

 \*Integer, real, character, Boolean

 \*Decides

 \*Interpretation of data

 \*Operations that can be performed on the data

\*Declaring data terms with proper types

\*Variable Declarations

 \*Pascal

Length, width: real;

Price, Tax, Total: integer;

 \*C, C++, Java

float Length, width;

int Price, Tax, Total;

 \*FORTRAN

REAL Length, Width

INTEGER Price, Tax, Total

\*Data structure

 \*Conceptual shape of data

 \*Common data structure

 \*Homogeneous array

 \*Heterogeneous array

Declaration of a 2D Array

\*C

int Scores[2][9];

\*Java

int Scores[][]=new int [2][9];

\*Pascal

Scores: array[3..4, 12..20] of integer;

Assignment Statements

\*C, C++, Java

Total = Price + Tax;

\*Ada, Pascal

Total := Price + Tax;

\*APL

Total <- Price + Tax;

Operators

\*Operator precedence

 \*Operator priority

 \*Plus and minus

 \*Multiply and divide

 \*Add and subtract

\*Operator overloading

 \*Exact function depends on the operand data types

 \*12 + 43

 \*‘abc’+ ’def’

Control Statements

\*Alter the execution sequence of the program

\*goto is the simplest control statement

Types of Controls

\*for / if…else/switch/while

Comments

\*For inserting explanatory statements (internal documentation)

\*C++ and Java

/\* This is a comment \*/

// This is a comment

Explain the program, not to repeat it

Procedures

\*A procedure

 \*A set of instructions for performing a task

 \*Used as an abstract tool by other program units

\*Control

 \*Transferred to the procedure at the time its services are required

 \*Returned to the original program unit (calling unit) after the procedure is finished

\*The process of transferring control to a procedure is often referred to as calling or invoking the procedure

Pass by Value

a. When the procedure is called, a copy of the datais given to the procedure

5 → 5

b. and the procedue manipulates its copy

5 6

c. Thus, when the procedure has terminated, the calling environment has not been changed calling environment

5 ○

Pass by Reference

a. When the procedure is called, the formal parameter becomes a reference to the actual parameter

5 → 5

b. Thus changes directed by the procedure are made to the actual parameter

5 6

6 ← 6

c. and are, therefore, preserved after the procedure has terminated

6 ○

Functions

\*The 6th type of control

\*A program unit similar to procedure unit except that a value is transferred back to the calling unit

Input/Output Statements

\*I/O statements are often not primitives of programming languages

\*Not really a control

\*Most programming languages implement I/O operations as procedures or functions

The Translation Process

Source program → Lexical(詞語的) analyzer → Parser(語法分析程式) → Code generator → Object program

Lexical Analyzer

\*Reads the source program symbol by symbol, identifying which groups of symbols represent single units, and classifying those units

\*As each unit is classified, the lexical analyzer generates a bit pattern known as a token to represent the unit and hands the token to the parser

\*Like mapping words according to a dictionary, except the dictionary here is much smaller and non-ambiguous

Parsing

\*Group lexical units (tokens) into statements

\*Identify the grammatical structure of the program

\*Recognize the role of each component

Syntax(語法) Diagram

\*Pictorial representations of a program’s grammatical structure

\*Nonterminals (rectangles)矩形 statement / expression

\*Requires further description

\*Terminals (ovals)橢圓 --- if / else / ……

\*圓形 --- + / - / x / y / z / ……

Parse Tree

\*Pictorial form which represents a particular string conforming to a set of syntax diagrams

\*The process of parsing a program is essentially that of constructing a parse tree for the source program

\*A parse tree represents the parser’s understanding of the programmer’s grammatical composition

Syntax Tree Ambiguity

\*There could be multiple syntax trees for one statement

\*When the results are the same, it is OK

\*When the results are not the same, we call the statement an ambiguous statement

Code Generation

\*Given the parse tree, create machine code

 \*Z  X + Y;

 \*Load X

 \*Load Y

 \*ADDI X Y

\*Complication

 \*When X is an integer and Y is a floating point number

 \*Convert X from integer to floating point number

 \*Use ADDF instead

Code Optimization

Line 1. X  Y + Z;

Line 2. W  X + Z;

\*Values of Y, Z, and X already in registers after Line 1

\*No need to store the values back to memory and then load again for Line 2.

Intertwined Process (相互交織的過程)

\*Lexical analyzer

 \*Recognize a token

 \*Pass to parser

\*Parser

 \*Analyze grammatical structure

 \*Might need another token

 \*Back to lexical analyzer

 \*Recognize a statement

 \*Pass to code generator

\*Code generator

 \*Generate machine code

 \*Might need another statement

 \*Back to code generator

Extended Process



Linker

\*Most programming environments allow the modules of a program to be developed and translated as individual units at different times

\*Linker links several

 \*Object programs

 \*Operating system routines and utility software

 #include <xxxx.h>

\*To produce a complete, executable program (load module) that is in turn stored as a file in the mass storage system

Loader

\*Often part of the operating system’s scheduler

\*Places the load module in memory

\*Important in multitasking systems

 \*Exact memory area available to the programs is not known until it is time to execute it

 \*Loader also makes any final adjustments that might be needed once the exact memory location of the program is known (e.g. dealing with the JUMP instruction)

Software Development Package

\*Editor

 \*Often customized

 \*Example

 Color for reserved words

 Aligned indentation

\*Translator

 \*The compiler/interpreter

 \*The most important part

\*Debugger

 \*To allow easy tracking of program states

Objects and Classes

\*Object

 \*Active program unit containing both data and procedures

\*Class

 \*A template for all objects of the same type

An Object is often called an instance of the class.

Components of an object

\*Instance variable

 \*Variable within an object

\*Method

 \*Function or procedure within an object

 \*Can manipulate the object’s instance variables

\*Constructor

 \*Special method to initialize a new object instance

Encapsulation

\*A way of restricting access to the internal components of an object

\*Private vs. Public

Additional Concepts

\*Inheritance

 \*Allows new classes to be defined in terms of previously defined classes

\*Polymorphism

 \*Allows method calls to be interpreted by the object that receives the call

 \*For example

 \*draw()

 \*Different for circle vs. square object

Program Concurrent Activities

\*Parallel or concurrent processing

\*Simultaneous execution of multiple processes

\*True concurrent processing requires multiple CPUs

\*Can be simulated using time-sharing with a single CPU

\*Examples: Ada task and Java thread

Basic Idea

\*Creating new process

\*Handling communication between processes

\*Problem accessing shared data

 \*Mutually exclusive access over critical regions

 \*Mechanism on the program

 \*Data accessed by only one process at a time

 \*Monitor

 \*Mechanism on the data

 \*A data item augmented with the ability to control access to itself

Prolog

\*PROgramming in LOGic

\*A Prolog program consists of a collection of initial statements upon which the underlying algorithm bases its deductive reasoning