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
Generations of programming languages

Problems solved in an environment in which the human must conform to the machine’s characteristics

Problems solved in an environment in which the machine conforms to the human’s characteristics
Second-generation: Assembly language

A mnemonic system for representing machine instructions

- Mnemonic names for op-codes
- Identifiers: Descriptive names for memory locations, chosen by the programmer
Assembly Language Characteristics

• One-to-one correspondence between machine instructions and assembly instructions
  – Programmer must think like the machine
  – the names of all registers, controller registers, instructions, masks, memory partitions, ....

• Inherently machine-dependent

• Converted to machine language by a program called an assembler
Assembly Language Characteristics

\( a = x + y + z; \)
- mov r0 x
- mov r1 y
- add r0 r1
- mov temp r0
- mov r0 z
- mov r1 temp
- add r0 r1
- mov a r0

3\textsuperscript{rd} generation

\( a = x + y + z; \)
- mov r0 x
- mov r1 y
- add r0 r1
- mov r1 z
- add r0 r1
- mov a r0

2\textsuperscript{nd} generation

Copyright © 2008 Pearson Education, Inc. Publishing as Pearson Addison-Wesley
Program Example

<table>
<thead>
<tr>
<th>Machine language</th>
<th>Assembly language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt; generation</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; generation</td>
</tr>
<tr>
<td>156C</td>
<td>LD R5, Price</td>
</tr>
<tr>
<td>166D</td>
<td>LD R6, ShippingCharge</td>
</tr>
<tr>
<td>5056</td>
<td>ADDI R0, R5 R6</td>
</tr>
<tr>
<td>30CE</td>
<td>ST R0, TotalCost</td>
</tr>
<tr>
<td>C000</td>
<td>HLT</td>
</tr>
</tbody>
</table>
Third Generation Language

- Uses high-level primitives
  - Similar to our pseudocode in Chapter 5
- Machine independent (mostly)
- Examples: FORTRAN, COBOL
- Each primitive corresponds to a sequence of machine language instructions
- Converted to machine language by a program called a compiler
Third Generation Languages

Some languages are supported by Interpreter.

- Java, BASIC
- JVM (Java virtual machine)
- One statement at a time
Third generation language

- *Drawback*

Step-by-step statements

• Refraining programmers from high-level abstraction of problems
• Burdening the productivity of programmers
## Software complexities (1/2)

<table>
<thead>
<tr>
<th>Year</th>
<th>Operating System</th>
<th>SLOC (Million)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1993</td>
<td>Windows NT 3.1</td>
<td>4-5</td>
</tr>
<tr>
<td>1994</td>
<td>Windows NT 3.5</td>
<td>7-8</td>
</tr>
<tr>
<td>1996</td>
<td>Windows NT 4.0</td>
<td>11-12</td>
</tr>
<tr>
<td>2000</td>
<td>Windows 2000</td>
<td>&gt;29</td>
</tr>
<tr>
<td>2001</td>
<td>Windows XP</td>
<td>40</td>
</tr>
<tr>
<td>2003</td>
<td>Windows Server 2003</td>
<td>50</td>
</tr>
</tbody>
</table>

**Vincent Maraia**

*Build Master, The: Microsoft's Software Configuration Management Best Practices*

*Addison-Wesley*

*October 2005*

## Software complexities (2/2)

<table>
<thead>
<tr>
<th>Operating System</th>
<th>SLOC (Million)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debian 2.2</td>
<td>55-59</td>
</tr>
<tr>
<td>Debian 3.0</td>
<td>104</td>
</tr>
<tr>
<td>Debian 3.1</td>
<td>215</td>
</tr>
<tr>
<td>Debian 4.0</td>
<td>283</td>
</tr>
<tr>
<td>OpenSolaris</td>
<td>9.7</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>8.8</td>
</tr>
<tr>
<td>Mac OS X 10.4</td>
<td>86</td>
</tr>
<tr>
<td>Linux kernel 2.6.0</td>
<td>5.2</td>
</tr>
<tr>
<td>Linux kernel 2.6.29</td>
<td>11.0</td>
</tr>
<tr>
<td>Linux kernel 2.6.32</td>
<td>12.6</td>
</tr>
</tbody>
</table>

[source lines of code](http://en.wikipedia.org/wiki/Source_lines_of_code)
Software cost estimates

<table>
<thead>
<tr>
<th>Software</th>
<th>Estimates (LOC/P-month)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time embedded systems</td>
<td>40-160</td>
</tr>
<tr>
<td>Systems programs</td>
<td>150-400 LOC/P-month</td>
</tr>
<tr>
<td>Commercial applications</td>
<td>200-800 LOC/P-month</td>
</tr>
</tbody>
</table>

*including all necessary activities in software development.

*Ian Sommerville*

*Software cost estimation, chapter 29*

*Software Engineering, 5th edition, Addison-Wesley*

*modified by Spiros Mancoridis 1998*
The evolution of programming paradigms
The composition of a typical imperative program or program unit

Program

The first part consists of declaration statements describing the data that is manipulated by the program.

The second part consists of imperative statements describing the action to be performed.
Data Types
- Basic

- Integer: Whole numbers
- Real (float): Numbers with fractions
- Character: Symbols
- Boolean: True/false
Basic Variable Declarations
- *examples in C*

```c
float    Length, Width;
int      Price, Total, Tax;
char     Symbol;
```
Structured (inductive) data types

- A two-dimensional array with two rows and nine columns

<table>
<thead>
<tr>
<th>Scores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Scores \((2, 4)\) in FORTRAN where indices start at one.

Scores \([1][3]\) in C and its derivatives where indices start at zero.
Array implementations

Fortran: c(2,9) static allocation.

- **compile-time:**
  - allocate a memory buffer c of size 2*9*4 bytes.
- **run-time:** $c(i,j) \text{ is } *(c +(i-1)*9+j-1)$

C: c[2][9] dynamic allocation.

- **compile-time:**
  - may allocate a memory buffer of size 2*4 bytes.
- **run-time:** $c[i][j] \text{ is } *(*(c+i)+j)$
The conceptual structure of the heterogeneous array Employee

```
#define ofs_name 0
#define ofs_age 4
#define ofs_rating 8

*(Employee+ofs_name)
*(Employee+ofs_age)
*(Employee+ofs_rating)
```
The for loop structure

*representation in C++, C#, and Java*

```
for (int Count = 1; Count < 4; Count++)
    body;
```
Procedural Units

- Local versus Global Variables
- Formal versus Actual Parameters
- Passing parameters by value versus reference
- Procedures versus Functions
The flow of control involving a procedure

Calling program unit requests procedure.

Calling program unit continues.

Control is transferred to procedure.

Procedure is executed.

Control is returned to calling environment when procedure is completed.
The procedure `ProjectPopulation` written in the programming language C

Starting the head with the term "void" is the way that a C programmer specifies that the program unit is a procedure rather than a function. We will learn about functions shortly.

```c
void ProjectPopulation (float GrowthRate)
{
    int Year;
    Population[0] = 100.0;
    for (Year = 0; Year <= 10; Year++)
        Population[Year+1] = Population[Year] + (Population[Year] * GrowthRate);
}
```

The formal parameter list. Note that C, as with many programming languages, requires that the data type of each parameter be specified.

This declares a local variable named `Year`.

These statements describe how the populations are to be computed and stored in the global array named `Population`. 
Executing the procedure Demo and passing parameters by value

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

b. and the procedure manipulates its copy.

c. Thus, when the procedure has terminated, the calling environment has not been changed.
Executing the procedure Demo and passing parameters by reference

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

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

c. and are, therefore, preserved after the procedure has terminated.
Unified view of passing by reference and value in C

```c
#define FAILURE 0
#define SUCCESS 1
int deposit(int credit, int *balance_ptr) {
    if (credit <= 0)
        return(FAILURE);
    *balance_ptr = *balance_ptr + credit;
    return(SUCCESS);
}

static int balance = 0;
void atm (int credit) {
    if (deposit(credit, &balance) == FAILURE)
        exit(0);
}
```
The function **CylinderVolume** written in the programming language C

The function header begins with the type of the data that will be returned.

```c
float CylinderVolume (float Radius, float Height)
{
    float Volume;
    Volume = 3.14 * Radius * Radius * Height;
    return Volume;
}
```

- Declare a local variable named Volume.
- Compute the volume of the cylinder.
- Terminate the function and return the value of the variable Volume.
The translation process
Lexical analysis & parsing

• Lexical analysis
  – Token analysis

• Parsing
  – Syntax definition
  – Automatic syntax analysis
Parsing
- syntax specifications

Syntax language: Backus-Naur Form (BNF)
A BNF rule in syntax diagram of

*if-then-else pseudocode statement*
Parsing
- syntax specifications

Syntax diagrams for simple algebraic expression
Parsing
- parse tree for the string \(x + y x z\)
BNF example:
- for if-statement

if $\text{Boolean-expression}$ then $\text{Statement}$ [else $\text{Statement}$ ]
Two distinct parse trees for the statement - *Ambiguity*

\[
\text{if } B1 \text{ then }\]
\[
\text{if } B2 \text{ then } S1 \text{ else } S2
\]
Stack implementation for local variables and parameters (1/3)

- stack in array: S[0..n]
- stack top pointer: sp = -1 initially
- push(S, a) { S[++sp] = a; }
- pop (S) { if (sp >= 0) return S[sp--]; }
- top(S) { return S[sp]; }
Stack implementation for local variables and parameters (2/3)

Example:
F(n) {
    if (n <= 1) return n;
    return F(n-1) + F(n-2);
}

F(3) → F(2), F(1); F(2) → F(1), F(0); F(1); F(0);
### Stack implementation for local variables and parameters (3/3)

\[ F(3) \rightarrow F(2), F(1); \quad F(2) \rightarrow F(2), F(1); \quad F(1); \quad F(0); \]

**procedure-call stack**

<table>
<thead>
<tr>
<th>Push</th>
<th>Push</th>
<th>Push</th>
<th>Pop</th>
<th>Pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>*(top-0)</td>
<td>top</td>
<td>c₀</td>
<td>c₁</td>
<td>1</td>
</tr>
<tr>
<td>F(3)</td>
<td>F(2)</td>
<td>F(2)</td>
<td>F(2)</td>
<td>F(0)</td>
</tr>
<tr>
<td>*(top-4)</td>
<td>top</td>
<td>c₀</td>
<td>c₁</td>
<td>1</td>
</tr>
<tr>
<td>F(3)</td>
<td>F(2)</td>
<td>F(2)</td>
<td>F(2)</td>
<td>F(2)</td>
</tr>
<tr>
<td>*(top-0)</td>
<td>top</td>
<td>c₀</td>
<td>c₁</td>
<td>0</td>
</tr>
<tr>
<td>F(3)</td>
<td>F(3)</td>
<td>F(3)</td>
<td>F(3)</td>
<td>c₀</td>
</tr>
<tr>
<td>*(top-4)</td>
<td>top</td>
<td>c₀</td>
<td>c₁</td>
<td>1</td>
</tr>
<tr>
<td>F(3)</td>
<td>F(3)</td>
<td>F(3)</td>
<td>F(3)</td>
<td>F(3)</td>
</tr>
</tbody>
</table>

*Note: The diagram illustrates the stack operations for the given procedure calls.*
Some issues in code optimization

Rearrange the code, to cut down

• the execution time
• the memory usage
• the power consumption for embedded computing.
Some issues in code optimization

Rearrange the code, to cut down

• the execution time

```plaintext
x=x+1;
x=x+y;

mov a, x
mov b, 1
add a, b
mov x, a
mov a, x
mov b, y
add a, b
move x, a

mov a, x
mov b, 1
add a, b

mov b, y
add a, b
move x, a
```
Some issues in code optimization

Rearrange the code, to cut down
- the execution time

\[
\begin{align*}
x &= x + 1; \\
y &= x + y; \\
z &= x \times z;
\end{align*}
\]

out-of-order (ooo) execution optimization
Some issues in code optimization

Rearrange the code, to cut down
• the execution time

\[
\begin{align*}
  x &= x + 1; \\
  *y &= x + *y; \\
  z &= x \times z;
\end{align*}
\]

out-of-order (ooo) execution optimization

\[
\begin{align*}
  x &= x + 1; \\
  z &= x \times z; \\
  *y &= x + *y;
\end{align*}
\]

Is this correct?

How do we know if \( y \) is \&\( z \) ?

Pointer analysis or shape analysis
OO (Object-Oriented) programming

Motivation:

3rd generation programming languages

• do not provide abstraction to data
• do not provide adequate protection to data.
• do not support management of large-scale projects
OO (Object-Oriented) programming

Motivation:

- 3\textsuperscript{rd} generation programming languages do not provide abstraction to data

```c
struct account_type *my_acc;
```

- How to withdraw from my\_acc ?
- How to deposit to my\_acc ?
- How to close my\_acc ?
- How to multiply my acc as float ?
Motivation:
• 3\textsuperscript{rd} generation programming languages do not provide adequate protection to data.

```c
struct account_type *my_acc;
if (my_acc == "my_account") {
    my_acc = "account_closed";
}
```
OO (Object-Oriented) programming

Objects = data + methods

- data: operand with a (real-world) meaning.
- methods: restricted operations on the data.

Example:

- a car that we can maintain, drive, park, …
- a pistol that we can trigger, load, hold, …
An object-oriented approach to the translation process

- Source program
- Lexical analyzer
- Code generator
- Parser
- Object program
Objects and Classes

• **Object**: Active program unit containing both data and procedures
• **Class**: A template from which objects are constructed

An object is called an **instance** of the class.
The structure of a class describing a laser weapon in a computer game

class LaserClass
{
    int RemainingPower = 100;

    void turnRight ( )
    {
        ...
    }

    void turnLeft ( )
    {
        ...
    }

    void fire ( )
    {
        ...
    }
}
Components of an Object

• **Instance Variable**: Variable within an object  
  – Holds information within the object

• **Method**: Procedure within an object  
  – Describes the actions that the object can perform

• **Constructor**: Special method used to initialize a new object when it is first constructed
A class with a constructor

class LaserClass
{
  int RemainingPower;

  LaserClass (InitialPower)  
  { RemainingPower = InitialPower;  

  void turnRight (  )  
  { ... }  

  void turnLeft (  )  
  { ... }  

  void fire (  )  
  { ... }  

}
Object Integrity

- **Encapsulation**: A way of restricting access to the internal components of an object
  - Private versus public
Our LaserClass definition using encapsulation as it would appear in a Java or C# program

class LaserClass
{
    private int RemainingPower;

    public LaserClass (InitialPower)
    {
        RemainingPower = InitialPower;
    }

    public void turnRight ( )
    {
        ...
    }

    public void turnLeft ( )
    {
        ...
    }

    public void fire ( )
    {
        ...
    }
}
Additional Object-oriented 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
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
Spawning threads

- Calling program unit requests procedure.
- Procedure is activated.
- Both units execute simultaneously.
Controlling Access to Data

- **Mutual Exclusion**: A method for ensuring that data can be accessed by only one process at a time
- **Monitor**: A data item augmented with the ability to control access to itself
Declarative Programming (4th generation)

rapid prototyping
programming by users.
for a specific application area.
Examples:
  • Table-driven programming, eDeveloper
  • IBM RPG (Report Program Generator)
  • CASE (Computer-Aided Software Engineering) tools
  • data management: SAS, SPSS, Excel
5GL (Fifth Generation Languages)

- **Resolution**: Combining two or more statements to produce a new statement (that is a logical consequence of the originals).
  - Example: \((\text{rain} \implies \text{happy}) \land (\neg \text{rain} \implies \text{smoke})\)
    resolves to \((\text{happy} \lor \text{smoke})\)
  - **Resolvent**: A new statement deduced by resolution
  - **Clause form**: A statement whose elementary components are connected by the Boolean operation OR
- **Unification**: Assigning a value to a variable so that two statements become “compatible.”
Resolution principle (1/3)

facts: \( \neg \text{rain} \rightarrow \text{happy}, \text{rain} \rightarrow \text{smoke} \)

queries ? (\( \text{happy} \lor \neg \text{smoke} \))

\[
\begin{align*}
\text{rain} \lor \text{happy} & \quad \neg \text{rain} \lor \text{smoke} \\
\text{happy} \lor \text{smoke}
\end{align*}
\]
Resolution principle (2/3)

facts: \( \neg rain \rightarrow happy, \ rain \rightarrow smoke, \ \neg smoke \)

queries ? happy

clauses: \( (\neg rain \lor happy), \ (rain \lor smoke), \ \neg smoke, \ \neg happy \)

proof by refutation.
Resolution principle (3/3)
- different proof with AI

facts: \( \neg \text{rain} \rightarrow \text{happy}, \text{rain} \rightarrow \text{smoke}, \neg \text{smoke} \)

queries ? \text{happy}

clauses: \( (\neg \text{rain} \lor \text{happy}), (\text{rain} \lor \text{smoke}), \neg \text{smoke}, \neg \text{happy} \)
Prolog

- **Fact**: A Prolog statement establishing a fact
  - Consists of a single predicate
  - Form: `predicateName(arguments)`.  
    - Example: `parent(bill, mary)`.  

- **Rule**: A Prolog statement establishing a general rule
  - Form: `conclusion :- premise`.  
    - :- means “if”  
    - Example: `wise(X) :- old(X)`.  
    - Example: `faster(X,Z) :- faster(X,Y), faster(Y,Z)`.  

Functional programming language

Checkbook balancing constructed from simpler functions

**Comparison**

**Imperative:**
- Total_credits ← sum of all credits.
- Temp_balance ← Old_balance + Total_balance;
- Total_debits ← sum of all Debits;
- Balance ← Temp_balance – Total_debits;

**Functional:**

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

\( \text{Output: } \text{New_balance} \)