

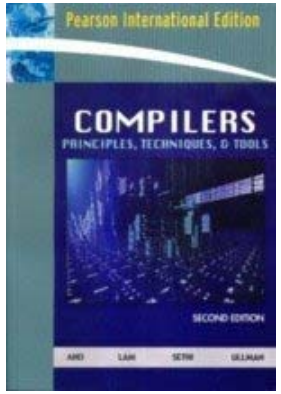
Compiler
編譯器原理

Yuan-Hao Chang (張原豪)
johnsonchang@ntut.edu.tw
Department of Electronic Engineering
National Taipei University of Technology



Syllabus

- 授課教師: 張原豪 (207-2 室、分機 2288)
- 上課時間:星期三 下午5:10~6:00, 星期五 上午10:10~12:00
- 教室: 三教 303, 403
- 教科書：
  - Compilers: Principles, Techniques, and Tools, second edition, 2007
    Authors: Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
    Publisher: Person International Edition
    ISBN: 0-321-49169-6



- 課程網頁:
  - http://www.ntut.edu.tw/~johnsonchang/ 點 Lecturing 或
  - http://www.ntut.edu.tw/~johnsonchang/courses/Compiler201102/
- 成績評量：(subject to changes)

  作業: (10%), 期中考(40%), 期末考(40%), 平時表現(10%)
- 需求：具有 C 或 Java (或其他高階) 程式語言之概念



Overview

- Part 1
  - Introduce motivational material and background issues.
- Part 2
  - Develop a miniature compiler and introduce important concept.
- Part 3
  - Cover lexical analysis, regular expression, finite-state machines, and scanner-generator tools (Lex).
- Part 4
  - Introduce the major parsing methods (top-down LL and bottom-up LR), including the parser generator (Yacc).

Overview (Cont.)

- Part 5
  - Introduce the principal ideas in syntax-directed definition and translation.
- Part 6
  - Use the theory in Part 5 to generate intermediate code.
- Part 7
  - Introduce run-time environment, especially the run-time stack and garbage collection.
- Part 8
  - Object-code generation



Chapter 1
Introduction

Outline

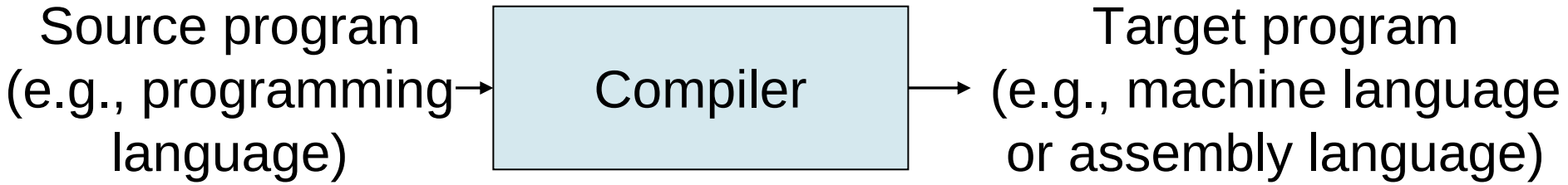
- Language processors
- The structure of a compiler
- The evolution of programming languages
- The science of building a compiler
- Applications of compiler technology
- Programming language basics



Language Processors

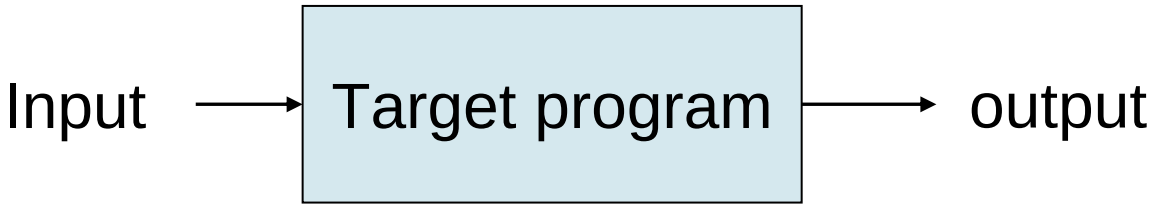
Introduction

- Compilers are fundamental to modern computing.
  - The computing world depends on programming languages.
  - Programming languages depend on compilers to translate the source program to target program.
  - The principles and techniques for compiler design are applicable to many other domains.

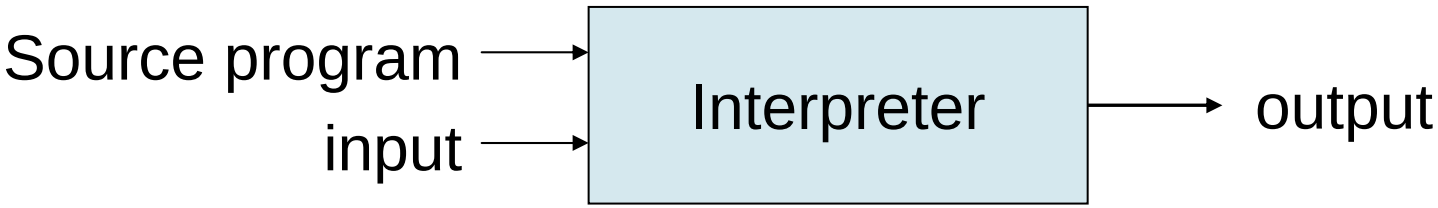


Language Processors

- Machine-language program as the target program produced by a compiler (faster)

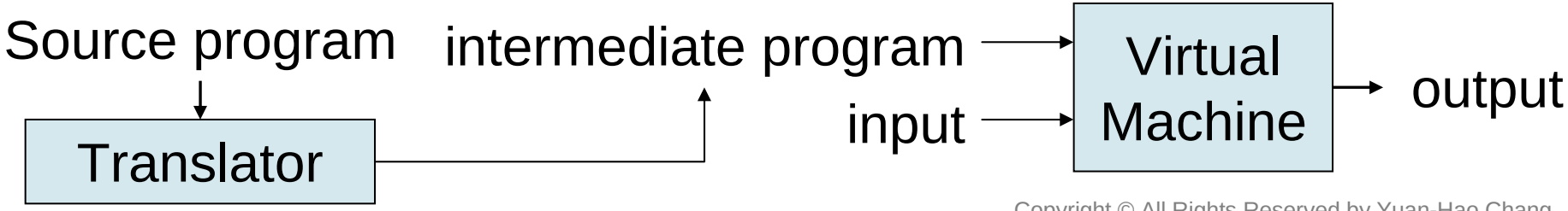


- Another language processor: Interpreter
  - It executes the source program statement by statement (better error diagnostics)

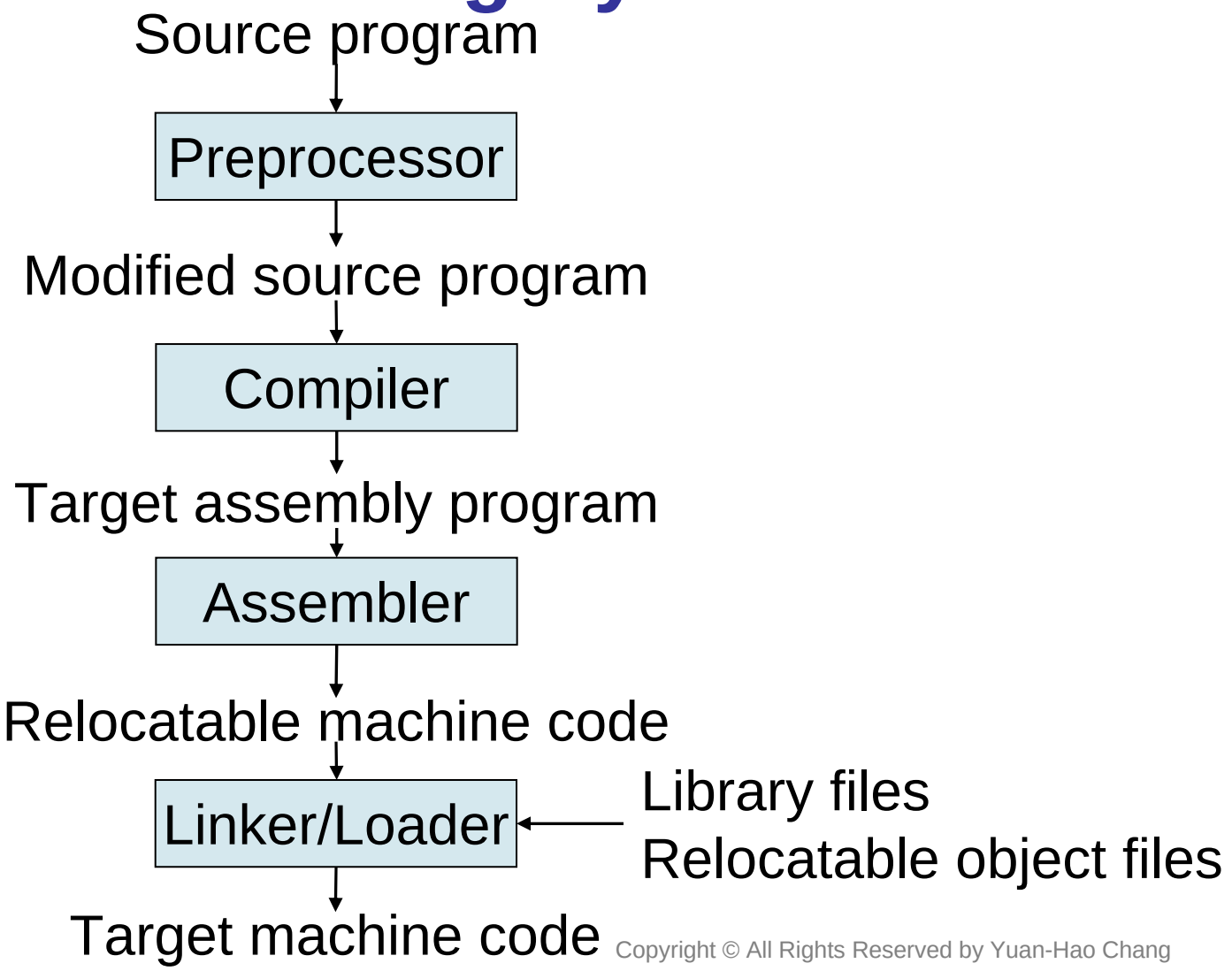


Language Processors (Cont.)

- Hybrid compiler
  - Combine compilation and interpretation
  - E.g., Java language processor
    - Java source program is first compiled into an intermediate form called *bytecodes*.
    - The bytecodes are then interpreted by a virtual machine.
  - High portability (cross platform)
    - Bytecodes compiled on one machine can be interpreted on another machine.



Language Processing System



Language Processing System (Cont.)

- Preprocessor
  - Collect source programs
  - Expand macros
- Compiler
  - It usually produces assembly-language program because assembly language is easier to produce and to debug.
- Assembler
  - Produce relocatable machine code
- Linker
  - Link pieces of a large program together
  - Resolve external memory addresses
- Loader
  - Put all of the executable object files into memory for execution



The Structure of a Compiler



The Structure of a Compiler

Lexeme: 語彙
Syntax: 語法
Semantic: 語意

- A compiler consists of two major parts
  - Analysis part (front end)
    - Break the source program into pieces with the grammatical structure.
    - Provide informative messages when syntactically ill formed or semantically unsound.
    - Collect and organize information from source program to the symbol table, which is passed to the synthesis part.
    - Generate the intermediate representation.
  - Synthesis part (back end)
    - Construct the target program from the intermediate representation and the symbol table.



Phases of a Compiler

Character stream

Lexical Analyzer (scanner)

Token stream

Syntax Analyzer (parser)

Syntax tree

Semantic Analyzer

Syntax tree

Intermediate Code Generator

Intermediate representation

Analysis part

Symbol Table

Used by all phases of the compiler

Machine-Independent Code Optimizer (optional)

Intermediate representation

Synthesis part

Code Generator

Target-machine code

Machine-Dependent Code Optimizer (optional)

Target-machine code

Lexical Analysis - Scanning (語彙分析)

- The lexical analyzer
  - Read the source program
  - Group the characters into meaningful sequences called lexemes
  - Output each lexeme as a token as the following form: *<token-name, attribute-value>*
    - ***token-name***: an abstract symbol used in syntax analysis
    - ***attribute-value***: point to the entry in the symbol table for this token
  - Tokens in the symbol table are needed for semantic analysis and code generation.

Lexical Analysis (Cont.)

- E.g., (for example)

  position = initial + rate * 60 (1.1)

  - position: a lexeme mapped to the token <id, 1>, where
    - id: an abstract symbol standing for *identifier*
    - 1: the entry for "position" in the symbol table to hold information about the identifier, such as its name and type.
  - Assignment symbol =: a lexeme mapped into the token <=>
    - No attribute-value so that it can be omitted
  - initial: a lexeme mapped to the token <id, 2>
  - +: a lexeme mapped to the token <+>
  - rate: a lexeme mapped to the token <id, 3>
  - *: a lexeme mapped to the token <*>
  - 60: a lexeme mapped to the token <60>

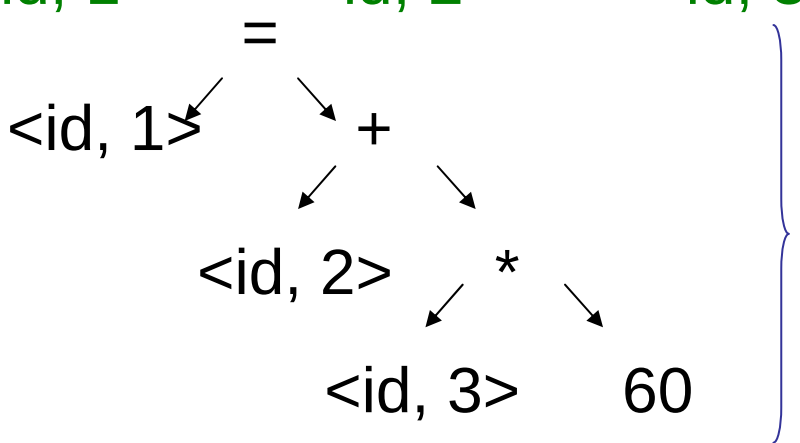
1	position	…
2	initial	…
3	rate	…

Symbol Table

<id, 1> <=> <id, 2> <+> <id, 3> <*> <60> (1.2)

Syntax Analysis – Parsing (語法分析)

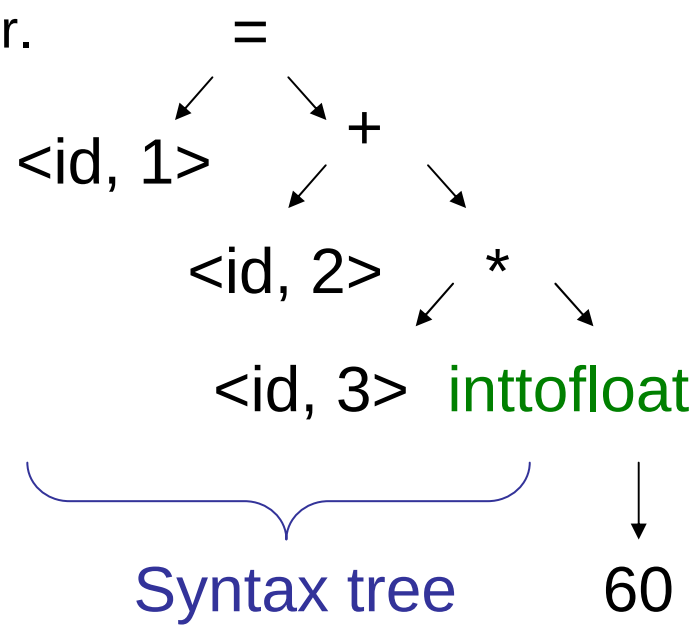
- Syntax analyzer uses tokens produced by lexical analyzer to create syntax trees that are usually used in syntax and semantic analysis
  - Interior node
    - It represents an operation.
    - Its children represent the arguments of the operation.
  - E.g., <id, 1> <=> <id, 2> <+> <id, 3> <*> <60>



Syntax tree

Semantic Analysis (語義分析)

- Semantic analyzer uses the syntax tree and the symbol table to check the semantic consistency, and then update the syntax tree and symbol table.
  - Type checking
    - E.g., an array index should be an integer.
  - Type conversion (coercion)
    - E.g., 3.5 + 5 (The compiler converts the integer into a floating-point number)





Intermediate Code Generation

- Important properties of intermediate representation:
  - Easy to produce
  - Easy to translate into the target machine.
- Three-address code (a commonly used intermediate form)
  - Consist of a sequence of assembly-like instructions with three operands, each of which acts like a register.
    E.g., <id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

```
t1 = inttoflota(60)
t2 = id3 * t1                (1.3)
t3 = id2 + t2
id1 = t3
```

Intermediate Code Generation (Cont.)

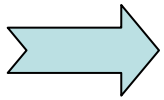
- Three-address code
  - There is *at most one operator* on the right side.
  - The compiler must generate a *temporary name* to hold the value computed by a three-address instruction.
  - An instruction might have fewer than three operands.

```
t1 = inttoflota(60)
t2 = id3 * t1                (1.3)
t3 = id2 + t2
id1 = t3
```

Code Optimization

- Machine-independent code-optimization is to improve the intermediate code.
  - E.g., faster, shorter code, target code consuming less power
- Code optimization might slow down the compilation.

```
t1 = inttoflota(60)
t2 = id3 * t1          (1.3)
t3 = id2 + t2
id1 = t3
```



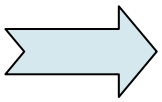
```
t1 = id3 * 60.0        (1.4)
id1 = id2 + t1
```

- Convert 60 to 60.0 automatically.
- t3 is used only once.

Code Generation

- Map the intermediate representation to the target language.
  - E.g., if the target is machine code,
    - Registers or memory locations are selected for each variable used in the program.
    - Then the intermediate representation is translated into machine instructions.

```
t1 = id3 * 60.0      (1.4)
id1 = id2 + t1
```



dest.

Immediate constant

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2          (1.5)
ADDF  R1, R1, R2
STF   id1, R1
```

- F stands for for floating point.
- LD(load)/ST(store) are used to access memory.
- The issue of storage allocation is ignored here.

Translation of An Assignment Statement

Character stream

position = initial + rate * 60

Lexical Analyzer (scanner)

Token stream

Syntax Analyzer (parser)

Syntax tree

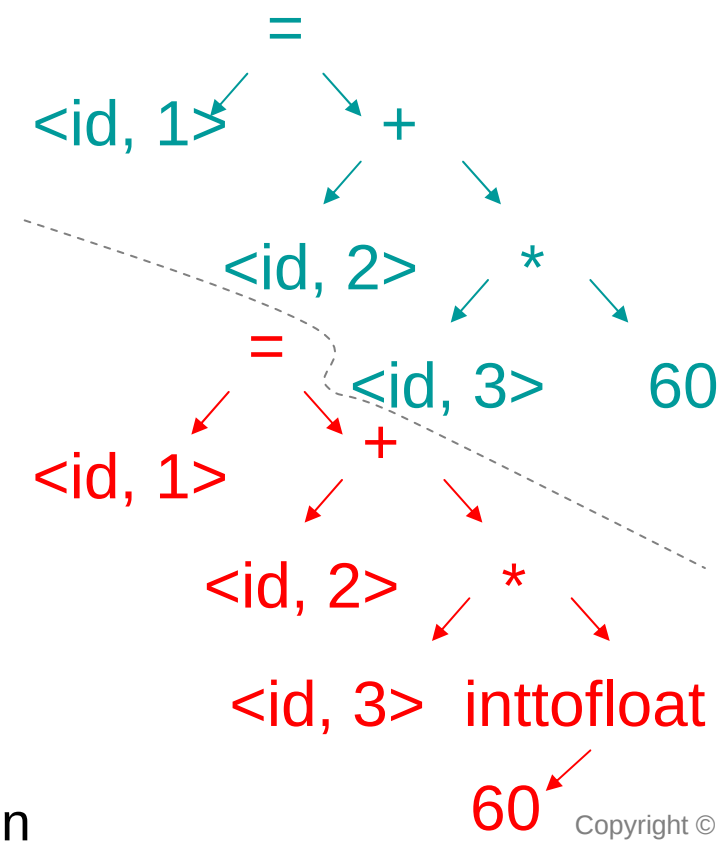
Semantic Analyzer

Syntax tree

Intermediate Code Generator

Intermediate representation

<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>



1	position	…
2	initial	…
3	rate	…

Symbol Table

```
t1 = inttoflota(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Translation of An Assignment Statement (Cont.)

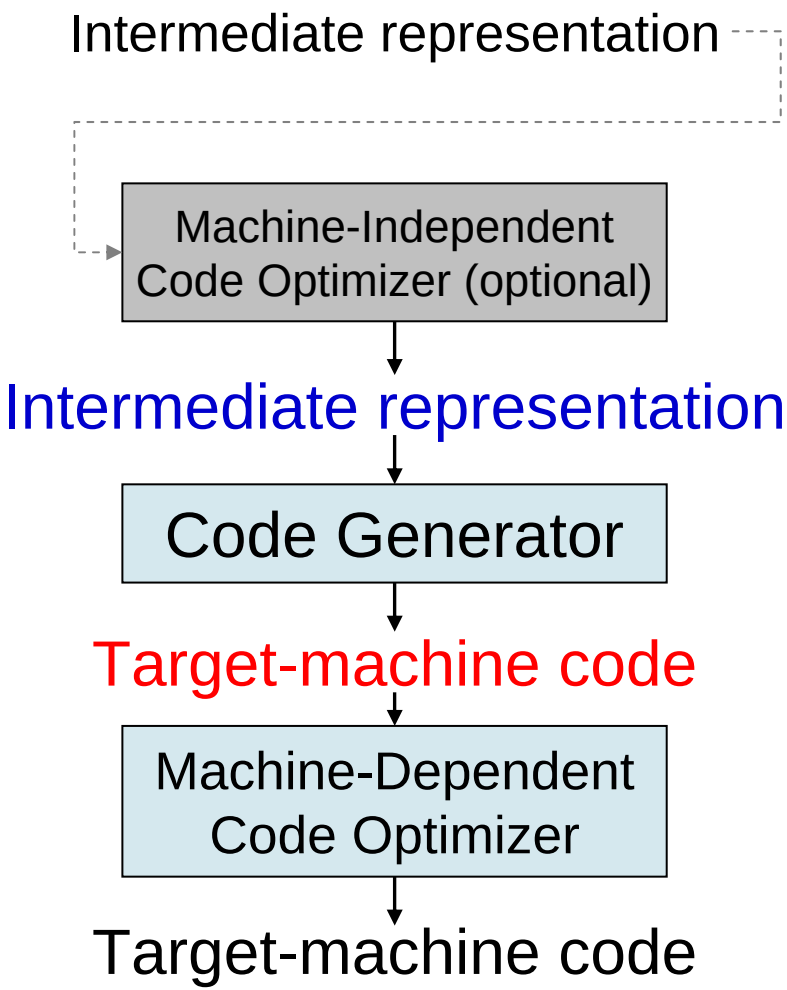
```
t1 = inttoflota(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF   R2,  id3
MULF  R2,  R2, #60.0
LDF   R1,  id2
ADDF  R1,  R1, R2
STF   id1, R1
```

1	position	…
2	initial	…
3	rate	…

Symbol Table





Symbol-Table Management

- The symbol table
  - Is a data structure containing a record for each variable name (e.g., *identifiers* and *keywords*) with fields for the attributes of the name.
  - Should be designed to allow the compiler
    - To find the record for each name quickly and
    - To store or retrieve data from that record quickly.



Grouping of Phases into Passes

- A pass
  - Groups several phases together and
  - Reads an input file and writes an output file.
- E.g.,
  - Pass 1 consists of the front-end (analysis) phases:
    - Lexical analysis, syntax analysis, semantic analysis, and intermediate code generation
  - Pass 2 consists of the back-end (synthesis) phase:
    - Code generation for a particular target machine

Note: code optimization might be an optional pass.

Grouping of Phases into Passes (Cont.)

- Benefits
  - Combine different front ends with the back end for a target machine.
  - Combine a front end with back ends for different target machines. (e.g., gcc for different processors)



Compiler-Construction Tools

- Scanner generators (e.g., Lex) :
  - Produce lexical analyzers from a regular-expression description of the tokens of a language.
- Parser generators (e.g., Yacc):
  - Automatically produce syntax analyzers from a grammatical description of a programming language.
- Syntax-directed translation engines:
  - Produce collections of routines for walking a parse tree and generating intermediate code.
- Code-generator generators:
  - Produce a code generator from a collection of rules for translating the intermediate language.

Compiler-Construction Tools (Cont.)

- Data-flow analysis engines:
  - Help gather information on value transmission from one part of a program to the other part.
  - Data-flow analysis is a key part of code optimization.
- Compiler-construction toolkits:
  - Provide an integrated set of routines for constructing various phases of a compiler



The Evolution of Programming Languages & The Science of Compilers

The Move to High-Level Languages

- In the 1940's:
  - The first electronic computer appeared and programmed in machine language by sequences of 0's and 1's.
- In the early 1950's:
  - Development of mnemonic(助記符號) assembly languages
    - Instructions for mnemonic representation of machine instructions
    - Macro instruction for frequently used sequences of machine instructions with parameters.
- In the late 1950's:
  - Development of higher-level languages
    - Fortran (for scientific computing), Cobol (for business data processing), and Lisp (for symbolic computation)
- In the following decades:
  - Thousands of high-level programming languages are developed.

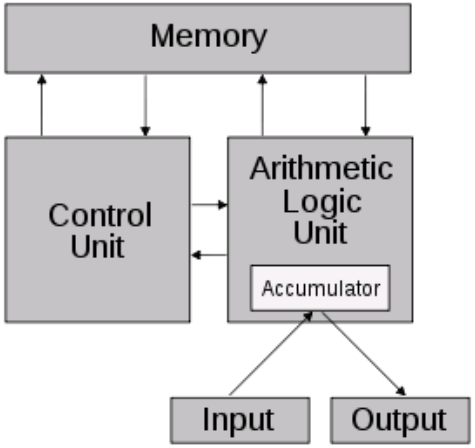
Classification of Programming Languages

- By Generation:
  - First generation: machine languages
  - Second generation: assembly languages
  - Third generation: higher-level languages
    - E.g., Fortran, Cobol, Lisp, C, C++, C#, and Java
  - Fourth generation: languages for specific applications
    - E.g., SQL for database queries, and Postscript for text formatting
  - Fifth generation: logic- and constraint-based languages
    - E.g., Prolog and OPS5



Classification of Programming Languages (Cont.)

- Imperative (命令) / Declarative (陳述)
  - Imperative language: specify how a computation is to be done.
    - E.g., C, C++, C#, and Java
  - Declarative language: specify what computation is to be done.
    - E.g., Functional languages such as ML and Haskell, and constraint logic languages such as Prolog
- *von Neumann* language
  - Based on the von Neumann architecture
    - E.g., Fortran, and C
- Object-oriented language
  - A program consists of a collection of objects.
  - E.g., Smalltalk, C++, C#, and Java
- Scripting language
  - Interpreted languages called scripts, such as JavaScript, Perl, and PHP, Python, and Tcl

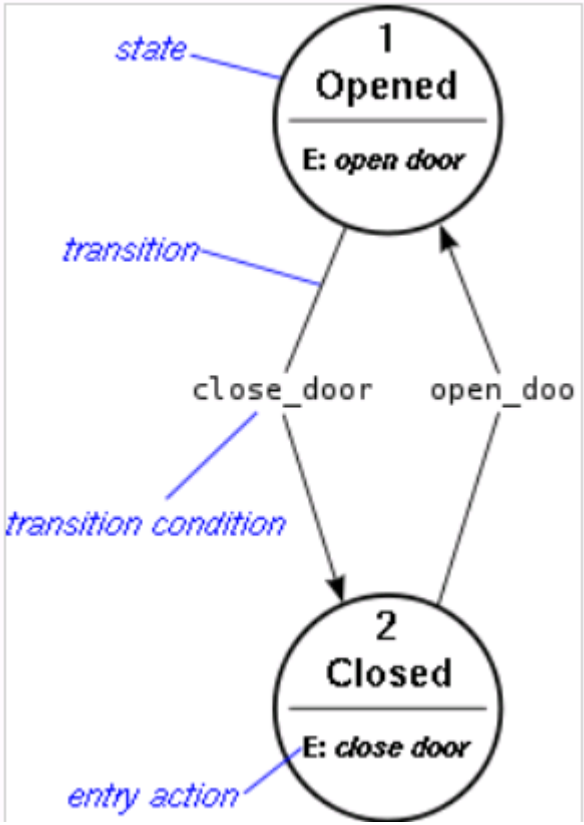


Impacts on Compilers

- Compilers and programming languages are closely related.
  - Compiler can help reduce overhead of programs.
  - Compilers are critical in making high-performance computer architecture effective on user applications.
- Compiler writing is challenging.
  - A compiler itself is a large program. (參考書:人月神話)
- A compiler must translate correctly.
- The problem of generating the optimal target code is undecidable. (NP-hard problem)

Modeling in Compiler Design

- Fundamental models:
  - Finite-state machines and regular expressions are useful for describing lexical units of programs.
    - Finite-state machine is a model of behavior composed of a finite number of states, transitions between those states, and actions.
    - Regular expressions provide a concise and flexible means for matching strings of text.
  - Context-free grammars are used to describe the syntactic structure such as the nesting of parentheses or control constructs.



An Example of Finite-State Machine

Objectives of Compiler Code Optimization

- The optimization must be correct.
- The optimization must improve the performance of many programs.
  - E.g., Speed, code size in embedded system, and power in mobile devices
- The compilation time must be kept reasonable.
- The engineering effort required must be manageable.



Applications of Compiler Technology

Compiler Technology for High-Level Programming Languages

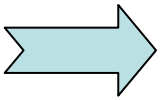
- Register keyword (in C language):
  - Let programmers to control registers.
- Data-flow optimization
  - E.g., reduce redundant load-store operation to variables.
- Procedure inlining
  - Replacement a procedure call by the body of the procedure
- Development trend: Increase levels of abstraction to handle things for programmers.
  - We use ***Java*** as an example:
    - Type-safe: an object can only be used in related types.
    - Boundary checks for arrays
    - No pointers
    - Built-in garbage collection

Optimization for Computer Architectures

- The evolution of computer architectures has led to new demand for new compiler technology.
- E.g., high performance systems usually adopt:
  - Parallelism:
    - Instruction level: Multiple instructions are executed simultaneously.
      - E.g., VLIW (Very Long Instruction Word) such as Intel IA64 to process a vector of data at the same time. (Compilers could adopt the instruction set.)
    - Processor level: Different threads of the same application are run on different processors.
      - Compilers could translate sequential program into multiprocessor code.
  - Memory hierarchies:
    - Registers (B), caches (KB), physical memory (MB~GB), secondary storage (GB~TB) ~ two to three orders of magnitude
    - Compilers can change
      - The order of instructions and layout of data to improve the effectiveness of memory hierarchy. (especially data caches)
      - The layout of code to improve the effectiveness of instruction caches.

Optimization for Computer Architectures (Cont.)

```
for (i=0;i<2;i++) {
   for (j=0;j<2;j++) {
      a[ j ][ i ] = 0;
   }
}
```



```
for (i=0;i<2;i++) {
   for (j=0;j<2;j++) {
      a[ i ][ j ] = 0;
   }
}
```

a[0][0]	a[1][0]	a[2][0]	a[0][1]	a[1][1]	a[2][1]	a[0][2]	a[1][2]	a[2][2]

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]	a[2][0]	a[2][1]	a[2][2]

Layout of data

Design of New Computer Architectures

- Compilers are now developed in the processor-design stage to evaluate the HW architecture.
- Some HW architectures:
  - RISC (Reduced Instruction-Set Computer)
    - Compilers can use small number of simple instructions to reduce the redundancies in complex instructions. (Optimization issue) (Note: CISC: Complex Instruction-Set Computer)
    - E.g., PowerPC, MIPS (Note: x86 is with CISC but adopts many design ideas of RISC)
  - Specialized architecture
    - E.g., Data-flow machine, VLIW, SIMD (Single instruction multiple data) array of processors, multiprocessors with shared/distributed memory.

Program Translation Assisted by Compilers

- Binary Translation
  - Translate the binary code for one machine to the other.
- Hardware Synthesis
  - Hardware designs are now described in high-level languages like Verilog, VHDL (Very high-speed integrated circuit Hardware Description Language)
  - Hardware designs are typically designed at the register transfer level (RTL):
    - Variables represent registers.
    - Expressions represent combinational logic.
  - Hardware-synthesis tools translate RTL into gates automatically. (circuit optimization takes hours.)
- Database Query Interpreters
  - SQL (Structured Query Language) that can be interpreted or compiled
- Compiled Simulation (Emulation)
  - Instead of writing a simulator that interprets the design, it is faster to compile the design to produce machine code that simulates the particular design.

Software Productivity Tools

- Testing is the primary technique for locating errors so as to improve software productivity.
- Finding all program errors is undecidable. (NP-hard)
  - Practical error detectors are often neither sound nor complete. (Reported errors are not all real errors, and only partial errors are found.)
- Some detection techniques
  - Type checking
  - Bounds checking
    - E.g., data flow analysis to detect buffer overflow.
  - Memory-management tools
    - E.g., garbage collection to prevent memory leaks



Programming Language Basics



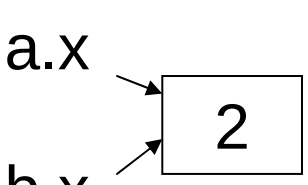
The Static / Dynamic Distinction

- Policies
  - Static policy:
    - A decision allowed to be made at compile time
  - Dynamic policy:
    - A decision allowed to be made at run time
- Scope:
  - The scope of a declaration of *x* is the region of the program in which uses of *x* refer to this declaration(宣告).
    - Static scope (or lexical scope): based on space
      - Determine the scope of a declaration by looking only at the program.
      - E.g., C and Java, and most languages
    - Dynamic scope: based on time
      - The same use of *x* could refer to any of several different declarations of *x*.



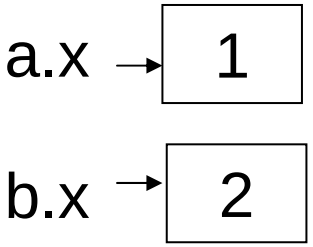
Static Variable in Java

- A variable is a *name* for a *location* in memory to hold a data *value*.
  - In Java, a "static" (class) variable means that all the instances of the class share the same copy of the variable.
  - E.g.,



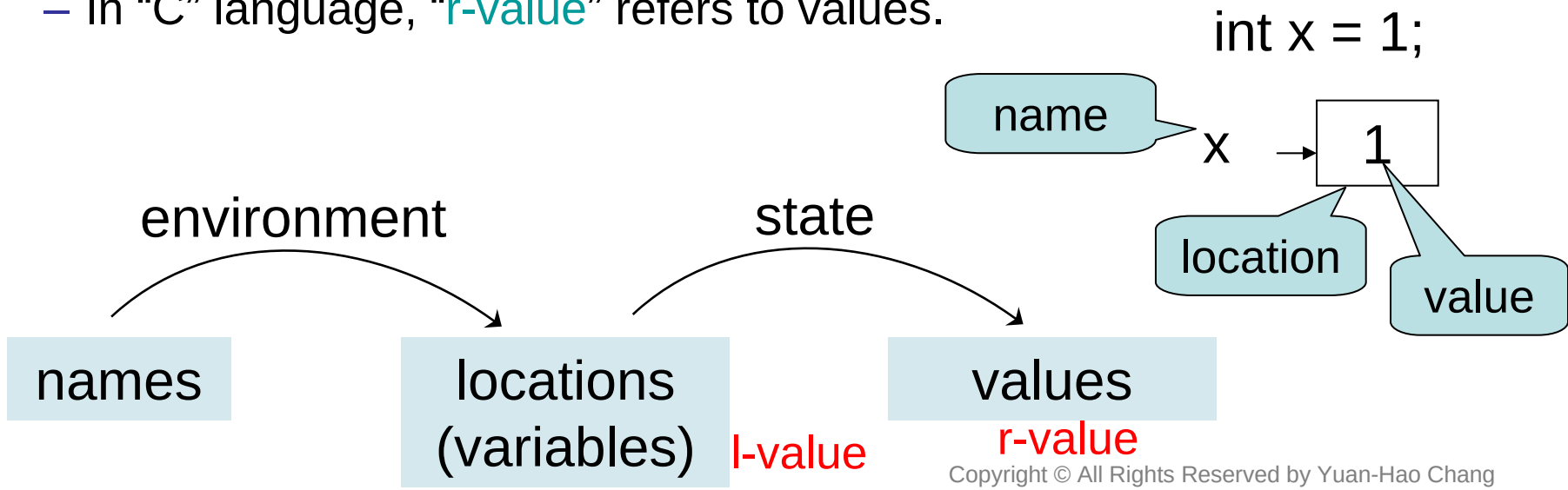

```
class stcla {
    public static int x;
    …
}
…
a = new stcla();
b = new stcla();
a.x = 1;
b.x = 2;
System.out.write(a.x); // 2
```

```
class stcla {
    public int x;
    …
}
…
a = new stcla();
b = new stcla();
a.x = 1;
b.x = 2;
System.out.write(a.x); // 1
```



Two-Stage Mapping from Names to Values

- Environment:
  - A mapping from names to locations in the store or memory.
  - In “C” language, “l-values” refers to the locations.
- State:
  - A mapping from locations in the store to their values.
  - In “C” language, “r-value” refers to values.



Two Declarations of the Name i

- The *local i* is local to function *f*.
- The *local i* is given a place on the run-time stack.
- Declarations in C must precede their use.
  - The function *g* can access neither *local i* nor *global i*.
  - The function *h* can't access local i, but can access global i.

```
g() {…}
…
int i;          /* global i */
…
void f(…) {
   int i;       /* local i */
   …
   i = 3;       /* use of local i */
   …
}
…
x = i + 1;      /* use of global i */
…
h() {…}
```

Binding

- Static/dynamic binding of names to locations:
  - Most binding of names to locations is dynamic.
  - E.g., in the previous example, *local i* is dynamic binding, but the *global i* is static binding.
    (Here the relocation issue is ignored since the loader and the operating system will handle it.)
- Static/dynamic binding of locations to values:
  - Most bindings of locations to values are dynamic.
  - The value in a location could be changed from time to time.
  - Exception: constants
    - E.g., #define ARRAYSIZE 1000 /* C language */

Names, Identifiers, and Variables

- ***Identifier***: a string of characters to identify(識別) an entity(實體)
  - All identifiers are names, but not all names are identifiers.
  - E.g., x.y → x, y, and x.y are names, but only x and y are identifiers.
- ***Variable***: a particular location of the store
  - An identifier could be declared more than once, and each declaration introduces a new variable.
  - E.g., an identifier (or a name) local to a recursive procedure will refer to different locations of the store at different times. (e.g., recursion to solve **Fibonacci series: 1, 1, 2, 3, 5, …**.)

Static Scope and Block Structure

- Most languages adopt static scope.
  - The scope of a declaration is determined *implicitly* by where the declaration appears.
  - C++, Java, and C# provide *explicit* control over scopes through keywords, such as ***public***, ***private***, and ***protected***.
- Block structure
  - Block is a grouping of declarations and statements, and can be used to define a static scope.
    - E.g., “C” uses { }, Pascal uses “begin” and “end”.
  - Block structure allows blocks to be nested inside each other.
    - A declaration D belongs to a block B if B is the most closely nested block containing D. → That is, D is located within B, not within any block within B.



A Block Example with C++

```
Main () {
    int a = 1;                                   B1
    int b = 1;
    {
        int b = 2;                               B2
        {
            int a = 3;                           B3
            cout << a << b; // 3 2
        }
        {
            int b = 4;                           B4
            cout << a << b;  // 1 4
        }
        cout << a << b;  // 1 2
    }
    cout << a << b;  // 1 1
}
```

Declaration	Scope
int a = 1;	B_1 - B_3
int b = 1;	B_1 - B_2
Int b = 2;	B_2 - B_4
int a = 3;	B_3
int b = 4;	B_4

Note: B_1 - B_3 stands for B_1 exclusive B_3

Explicit Access Control

- Java and C++ provide explicit control with the keywords "**private**", "**protected**", and "**public**" so as to support *encapsulation(封裝)*.
  - Private names give a scope within that class and its friend classes (in C++ term).
  - Protected names are accessible to subclasses (or the same package in Java).
  - Public names are accessible from outside the class.

Explicit Access Control (Cont.)

- An example with Java

```
package P1;          // cla_a.java
public class cla_a {
    public int x;
    protected int y;
    private int z;
    …
}
package P2;          // cla_b.java
public class cla_b extends cla_a {
    public test() {
        x = 1; // accessible
        y = 1; // accessible
        z = 1; // not accessible
    }
    …
}
```

Program entry

```
import P1.*;          // Main.java
import P2.*;
class Main {
    static void main(String[] args) {
        cla_a a = new cla_a();
        cla_b b = new cla_b();
        a.x = 1; // accessible
        a.y = 1; // not accessible
        a.z = 1; // not accessible
        b.test();
    }
}
```



Dynamic Scope

- Dynamic scope
  - It is based on factors that can be known only when the program executes.
  - A use of a name *x* refers to the declaration of *x* in the most recently called procedure with such a declaration.
    - E.g.:
      - Method resolution (in object-oriented programming)

Notes:
- In C, procedures are implemented as functions that returns value, where procedures don't return any value.
- In object-oriented languages, procedures of a class are also implemented as functions called methods, and variables of a class are called attributes.



Dynamic Scope Resolution

- Essentials for polymorphic(多型) procedures.
  - A procedure that has two or more definitions depends only on the types of the arguments.

E.g.:

1. There is a class C with a method named m().
2. D is a subclass of C, and D has its own method named m().
3. There is a use of m of the form x.m(), where x is an object of class C.

→ Normally, it is ***impossible*** to tell at compile time where x will be of class C or D.

Dynamic Scope Resolution (Cont.)

```
public class cla_c {  // cla_c.java
    public void m() {
        System.out.println("c");
    }
}
// cla_d.java
public class cla_d extends cla_c {
 public void m() {
        System.out.println("d");
    }
}
```

```
public class cla_a {  // cla_a.java
    public void test(cla_c x) {
        x.m();
    }
}

class Main {        // Main.java
    static void main(String[] args) {
    cla_c c = new cla_c();
    cla_d d = new cla_d();
    cla_a a = new cla_a();
    a.test(c); // "c"
    a.test(d); // "d"
    }
}
```

Declarations (宣告) and Definitions (定義)

- Declarations / Definitions
  - Declarations tell us about the types of things.
    - E.g., int i;
  - Definitions tell us about their values or contents.
    - E.g., i = 1;
      int i = 1;
- For example:
  - In C++, a method is declared in a class definition.
  - It is common to define a C function in one file and declare it in other files where the function is used.

```
// definition
int Callee() {      // Callee.c
    …
}
```

```
int Callee(); // declaration
int CallerA() {     // CallerA.c
    Callee();
    …
}
```

```
int Callee();   // declaration
int CallerB() {     // CallerB.c
    Callee();
    …
}
```



Parameter Passing

- Parameters
  - Actual parameters: used in the call of a procedure
  - Formal parameters: used in the procedure definition
- Mechanisms
  - Call-by-value
    - The value of the actual parameter is passed to the callee.
  - Call-by-reference
    - The address of the actual parameter is passed to the callee, and the corresponding formal parameters can't point to any other address.
  - Call-by-address(/pointer)
    - The address of the actual parameter is passed to the callee, and the corresponding formal parameters can point to other addresses.
  - Call-by-name
    - The actual parameter literally substitutes the formal parameter in the callee (similar to a macro).

Call-by-address

```
int main() {
    int x = 5;
    foo(&x);
}
void foo(int *x) {
    (*x)++;
}
```

Call-by-reference

```
int main() {
    int x = 5;
    foo(x);
}
void foo(int &x) {
    x++;
}
```

Aliasing

- Aliasing is that two formal parameters refer to the same location.

```
void q(char *x, char *y) {
    x[10] = 2;
    printf("%d\n", y[10]); // 2
}

void p () {
    char a[20];
    q(a, a);
}
```

The formal variables *x* and *y* are aliasing to each other. (C language)