CS 4300: Compiler Theory

Chapter 1
Introduction

Dr. Xuejun Liang

Outlines

1. Language Processors
2. The Structure of a Compiler
3. The Evolution of Programming Languages
4. The Science of Building a Compiler
5. Applications of Compiler Technology
6. Programming Language Basics

1. Compilers and Interpreters

- Compilation
  - Translation of a program written in a source language into a semantically equivalent program written in a target language

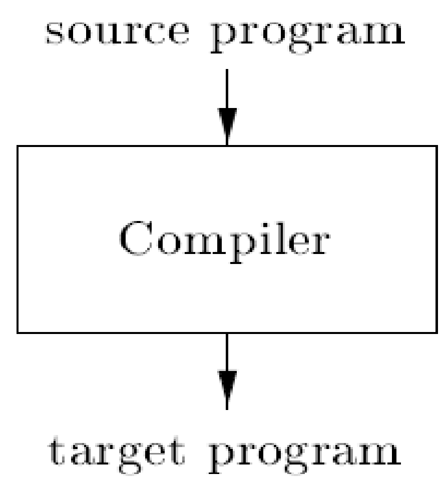


Figure 1.1: A compiler

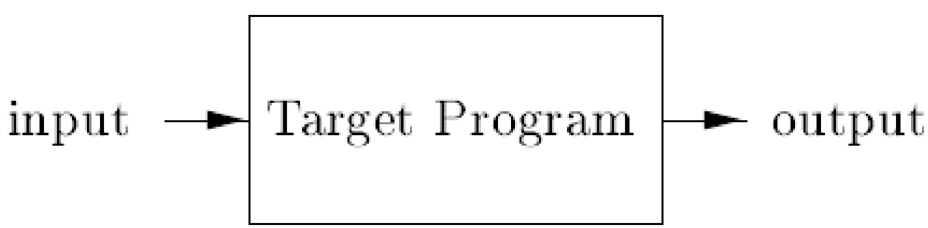


Figure 1.2: Running the target program

Compilers and Interpreters (cont)

- Interpretation
  - Performing the operations implied by the source program

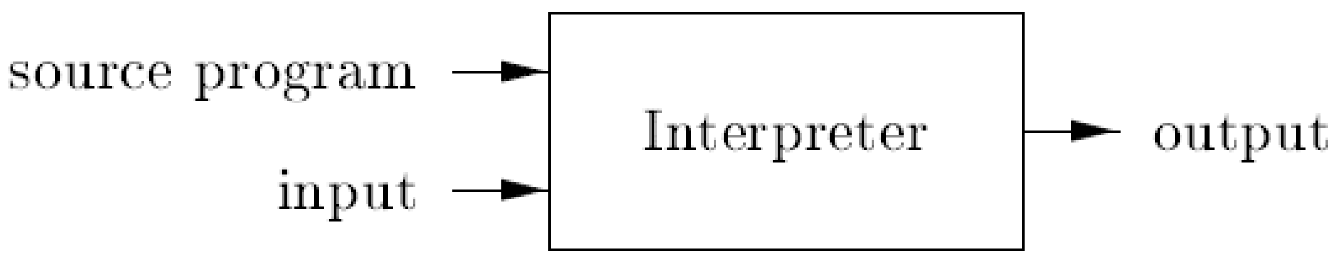


Figure 1.3: An interpreter

Compilers and Interpreters (cont)

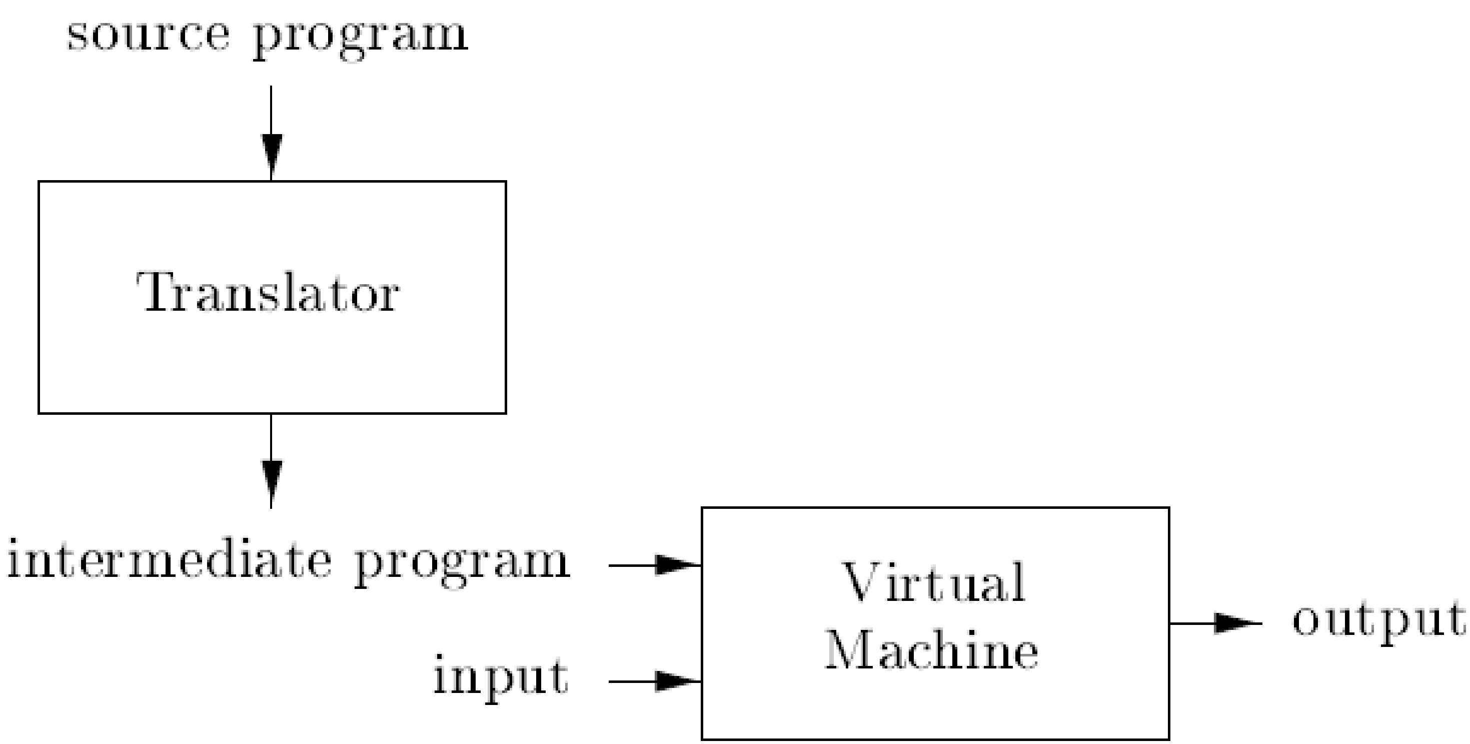
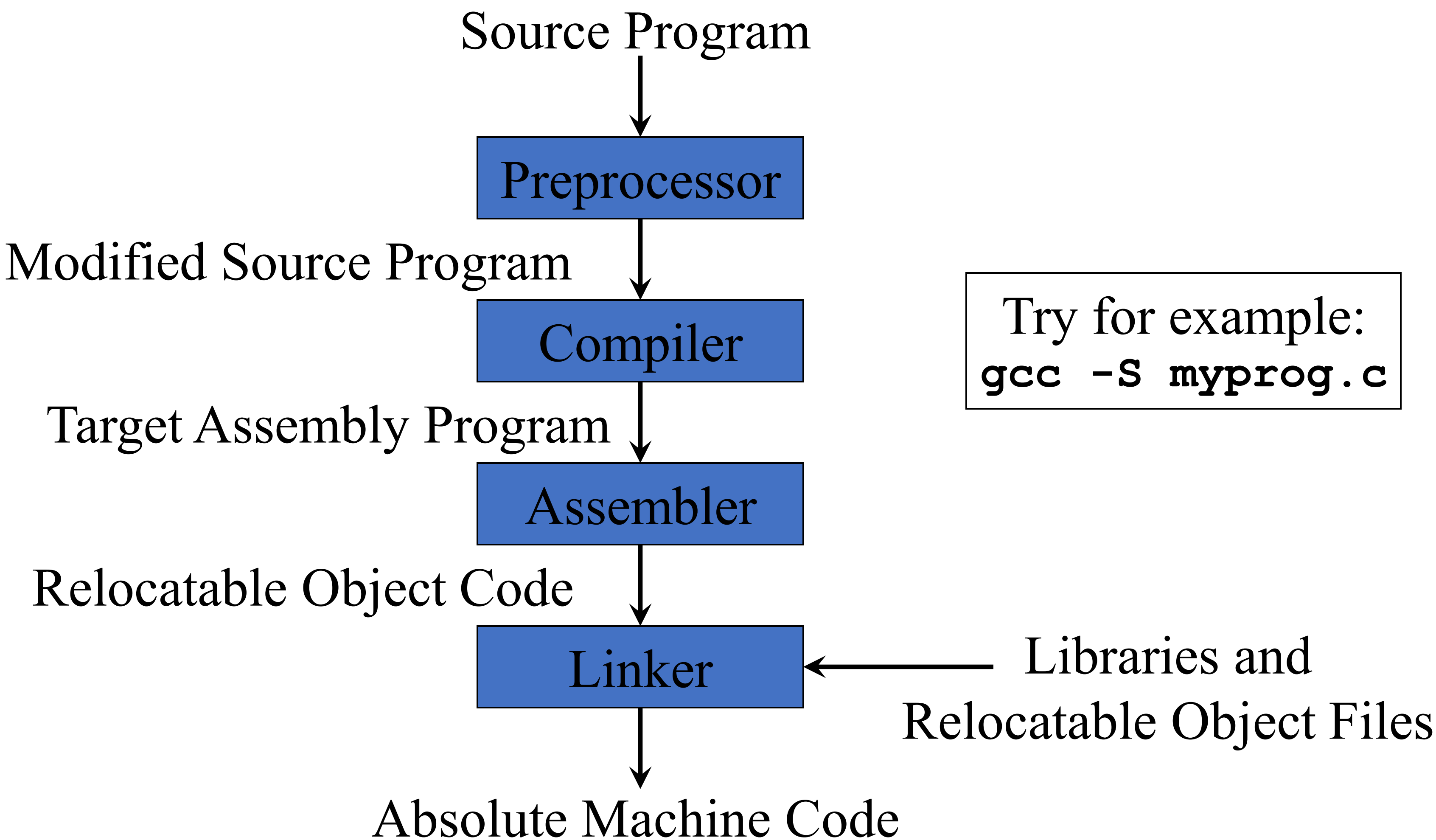


Figure 1.4: A hybrid compiler

Language Preprocessing System



Try for example:
`gcc -S myprog.c`

2. The Structure of a Compiler

- Lexical Analysis
- Parsing (Syntax Analysis)
- Semantic Analysis
- Optimization
- Code Generation

The first 3, at least, can be understood by analogy to how humans comprehend English.

Analysis and Synthesis

- There are two parts to compilation:
  - **Analysis** breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program.
    - The analysis part also collects information about the source program and stores it in a data structure called a symbol table
  - **Synthesis** constructs the desired target program from the intermediate representation and the information in the symbol table

The Phases of a Compiler

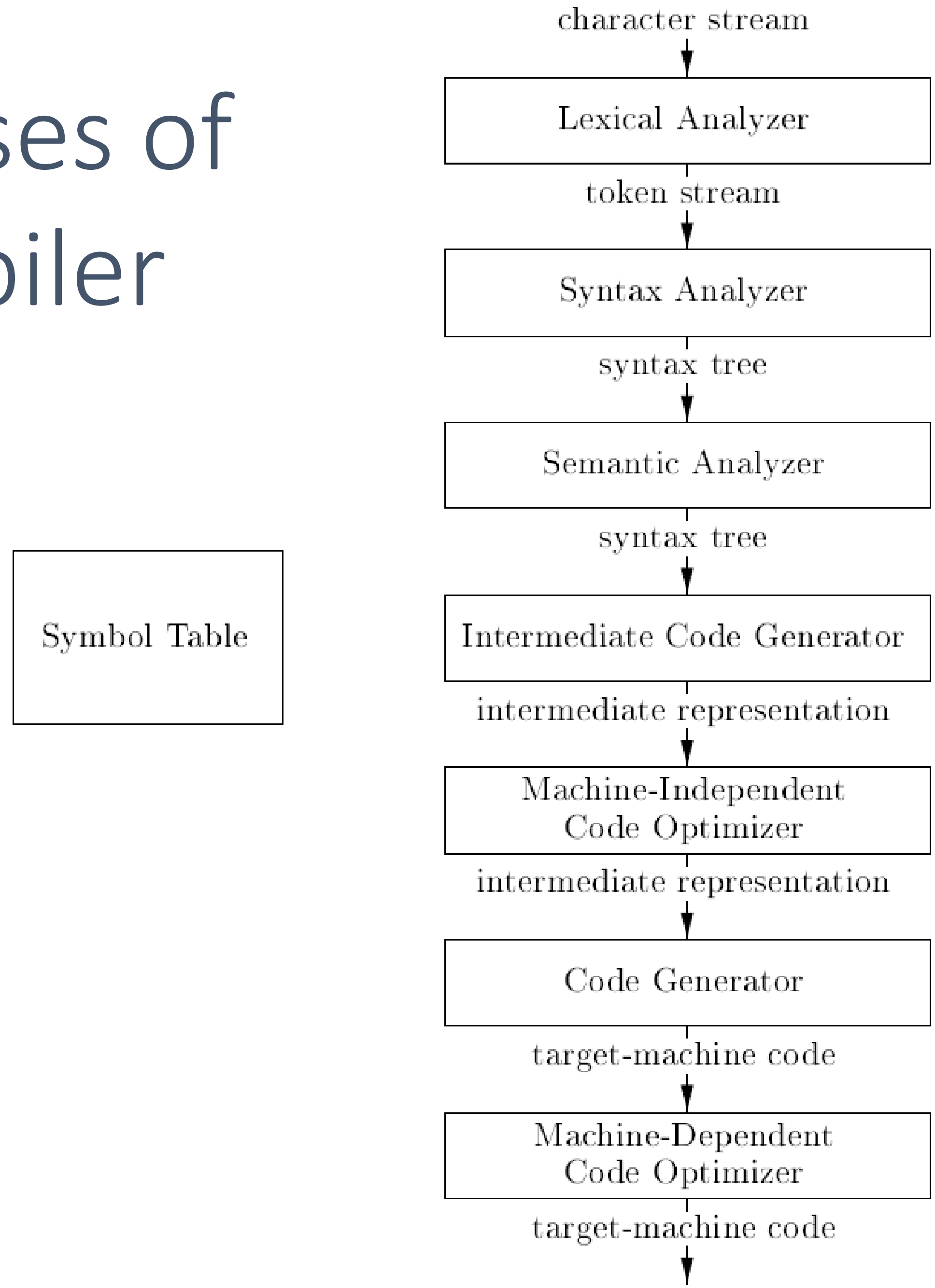


Figure 1.6: Phases of a compiler

Example

position = initial + rate * 60

Optimization Phase:

Automatically modify programs so that they

– Run faster
– Use less memory
– In general, conserve some resource
– Preserve correctness

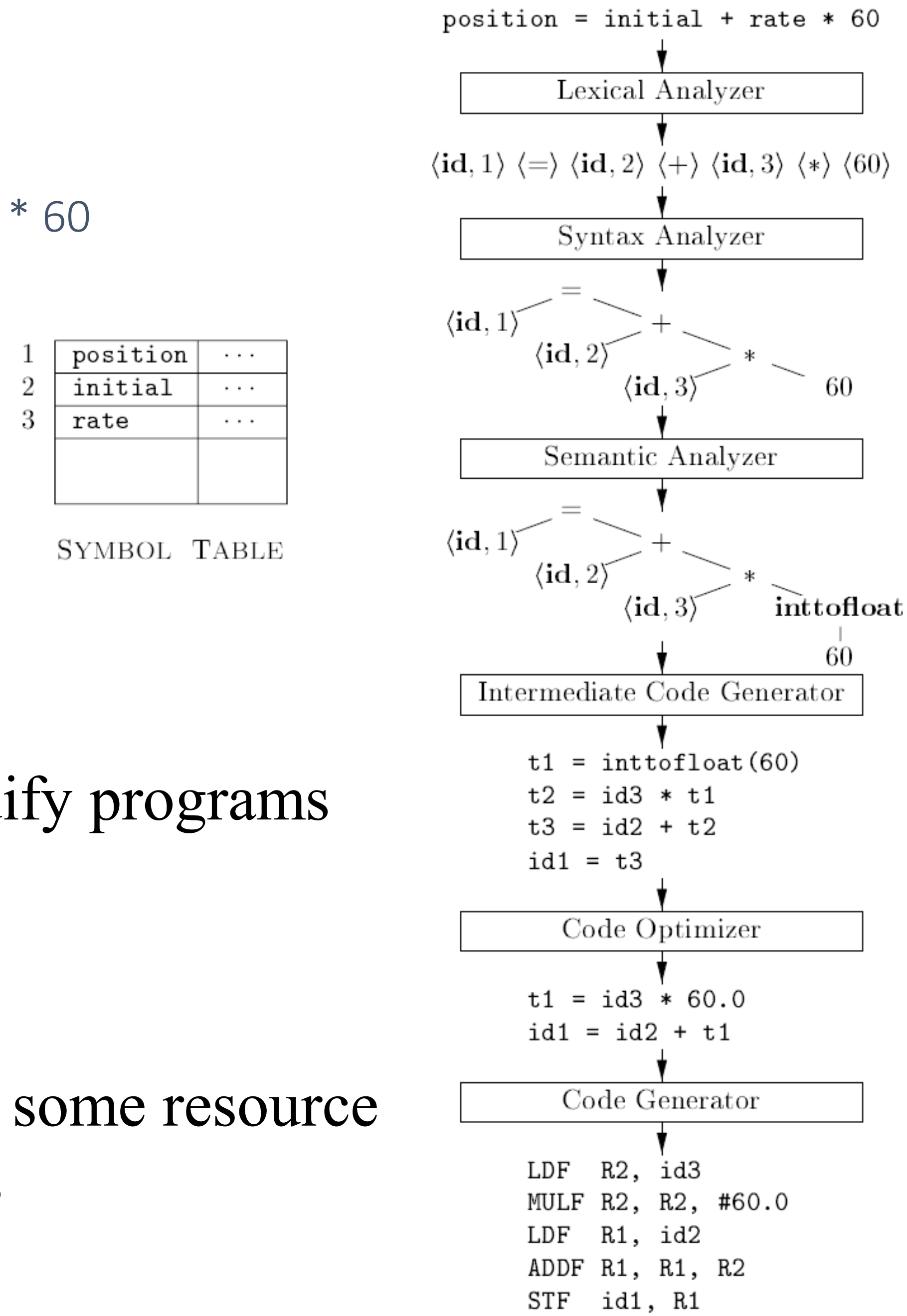


Figure 1.7: Translation of an assignment statement

The Grouping of Phases

- Compiler *front* and *back ends*:
  - Front end: *analysis* (*machine independent*)
  - Back end: *synthesis* (*machine dependent*)
- Compiler *passes:*
  - A collection of phases is done only once (*single pass*) or multiple times (*multi pass*)
    - Single pass: usually requires everything to be defined before being used in source program
    - Multi pass: compiler may have to keep entire program representation in memory

Compiler-Construction Tools

- Software development tools are available to implement one or more compiler phases
  - Scanner generators
  - Parser generators
  - Syntax-directed translation engines
  - Code-generator generators
  - Data-flow analysis engines
  - Compiler- construction toolkits

5. Applications of Compiler Technology

- Implementation of High-Level Programming Languages
- Optimizations for Computer Architectures
- Design of New Computer Architectures
- Program Translations
- Software Productivity Tools

Other Tools that Use the Analysis-Synthesis Model

- *Editors* (syntax highlighting)
- *Pretty printers* (e.g. Doxygen)
- *Static checkers* (e.g. Lint and Splint)
- *Interpreters*
- *Text formatters* (e.g. TeX and LaTeX)
- *Silicon compilers* (e.g. VHDL)
- *Query interpreters/compilers* (Databases)

Why Study Compilers

- Increase capacity of expression
- Improve understanding of program behavior
- Increase ability to learn new languages
- Learn to build a large and reliable system
- **See many basic CS concepts at work**

6. Programming Language Basics

- The Static/Dynamic Distinction
- Environments and States
- Static Scope and Block Structure
- Explicit Access Control
- Dynamic Scope
- Parameter Passing Mechanisms
- Aliasing

Static Verse Dynamic Scope

- The **scope** of a declaration of x is the region of the program in which uses of x refer to this declaration
- A language uses **static scope** or **lexical scope** if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses **dynamic scope**
- With dynamic scope, as the program runs, the same use of x could refer to any of several different declarations of x

Environments and States

- The environment is a mapping from names to locations in the store
- The state is a mapping from locations in store to their values

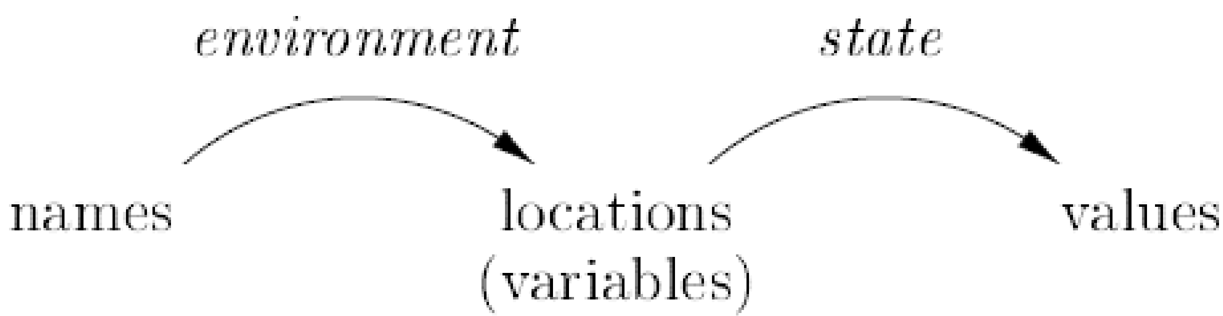


Figure 1.8: Two-stage mapping from names to values

Block Structure Example

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

$$B_3 \subset B_2$$

$$B_4 \subset B_2$$

$$B_2 \subset B_1$$

Figure 1.10: Blocks in a C++ program

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

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

Static Scope and Block Structure

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

Figure 1.11: Scopes of declarations in Example 1.6

The static-scope rule for variable declarations in a block structured languages is as follows. If declaration D of name x belongs to block B, then the scope of D is all of B, except for any blocks B' nested to any depth within B, in which x is redeclared

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

DECLARATION	SCOPE
`int a = 1;`	B_1
`int b = 1;`	$B_1 - B_2$
`int b = 2;`	B_2

Dynamic Scope

- Dynamic Scope
  - A use of a name x refers to the declaration of x in the most recently called, not yet terminated, procedure with such a declaration
- Analogy Between Static and Dynamic Scoping
  - The dynamic rule is to time as the static rule is to space.
  - While the static rule asks us to find the declaration whose unit (block) most closely surrounds the physical location of the use, the dynamic rule asks us to find the declaration whose unit (procedure invocation) most closely surrounds the time of the use

Dynamic Scope: two cases

- Macro expansion in the C preprocessor

```
#define a (x+1)
int x = 2;
void b() { int x = 1; printf("%d\n", a); }
void c() { printf("%d\n", a); }
void main() { b(); c(); }
```

Polymorphism

Overriding

Virtual method

- Method resolution in object-oriented programming
  - There is a class A with a method named m() .
  - B is a subclass of A, and B has its own method named m().
  - There is a use of m of the form x.m(), where x is an object of class A.

C++ Method (Function) Overriding

```
#include <iostream>
using namespace std;

class A {
  public: void m() {
    cout << "This is A!" << endl;
  }
};

class B : public A {
  public: void m() {
    cout << "This is B!" << endl;
  }
};
```

```
int main(){
  A* s1 = new A;
  s1 -> m();

  A* s2 = new B;
  s2 -> m();

  return 0;
}
```

C++ Virtual Method (Function)

```cpp
#include <iostream>
using namespace std;

class A {
  public: virtual void m() {
    cout << "This is A!" << endl;
  }
};

class B : public A {
  public: virtual void m() {
    cout << "This is B!" << endl;
  }
};
```

```cpp
int main(){
  A* s1 = new A;
  s1 -> m();

  A* s2 = new B;
  s2 -> m();

  return 0;
}
```

Java Method (Function) Overriding

```
class A {
  public void m() { System.out.println("This is A!"); }
}

class B extends A {
  public void m() { System.out.println("This is B!"); }
}

class Override {
  public static void main(String[] args) {
    A s1 = new A();
    s1.m();

    A s2 = new B();
    s2.m();
  }
}
```

Parameter Passing Mechanisms

- Call-by-value
  - the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure
- Call-by-reference
  - the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter
- Java uses call-by-value exclusively. But, anything other than a basic type such as an integer or real is a pointer to the actual object. Thus, the called procedure is able to affect the value of the object itself (except the basic type).