# COMPILER CONSTRUCTION

Principles and Practice

Kenneth C. Louden
San Jose State University

# 1. INTRODUCTION

# 1.1 What is a compiler?

- A computer program translates one language to another


- A compiler is a complex program
  - From 10,000 to 1,000,000 lines of codes
- Compilers are used in many forms of computing
  - Command interpreters, interface programs

# Brief History of Compiler

- The first compiler was developed between 1954 and 1957
  - The FORTRAN language and its compiler by a team at IBM led by John Backus
  - The structure of natural language was studied at about the same time by Noam Chomsky

# 1.2 Programs related to Compiler

# 1) Interpreters

- Execute the source program immediately rather than generating object code
- Examples: BASIC, LISP, used often in educational or development situations
- Speed of execution is slower than compiled code by a factor of 10 or more
- Share many of their operations with compilers

# Difference between Compiler and Interpreter

## Compiler

- It converts whole code at a time.
- It is faster.
- Requires more memory.
- Errors are displayed after entire program is checked.
- Example: C, C++, JAVA.

## Interpreter

- It converts the code line by line.
- It is slower.
- Requires less memory.
- Errors are displayed for every instruction interpreted (if any).
- Example: GW BASIC, Ruby, Python

# Differences between compilers & interpreters

| | Compilers | Interpreters |
|---|---|---|
| *Translation of source program* | the whole program before execution | one line at a time when it is run |
| *Frequency of translation* | each line is translated once | has to be translated every time it is executed - slower |
| *Object program* | can be saved for future execution without the source program | no object program is generated, so, source program and interpreter must be present for execution |

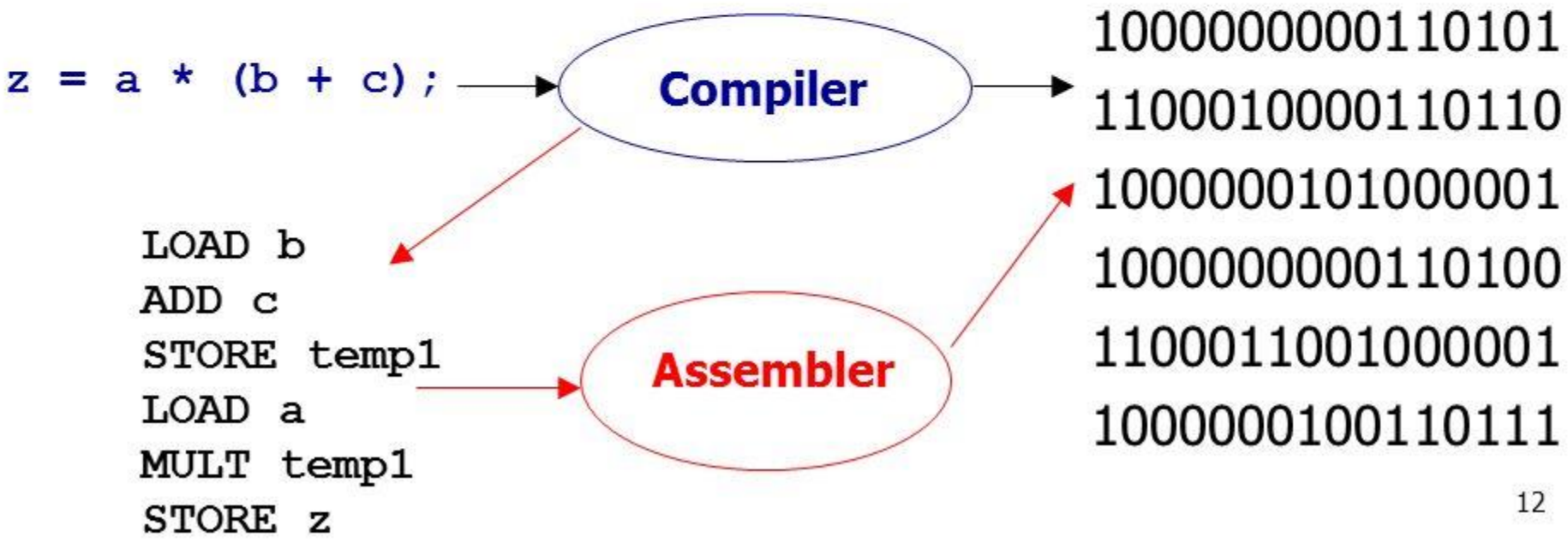
# 2) Assemblers

- A translator for the assembly language of a particular computer
- Assembly language is a symbolic form of one machine language
- A compiler may generate assembly language as its target language and an assembler finished the translation into object code

# Compilers vs. Assemblers

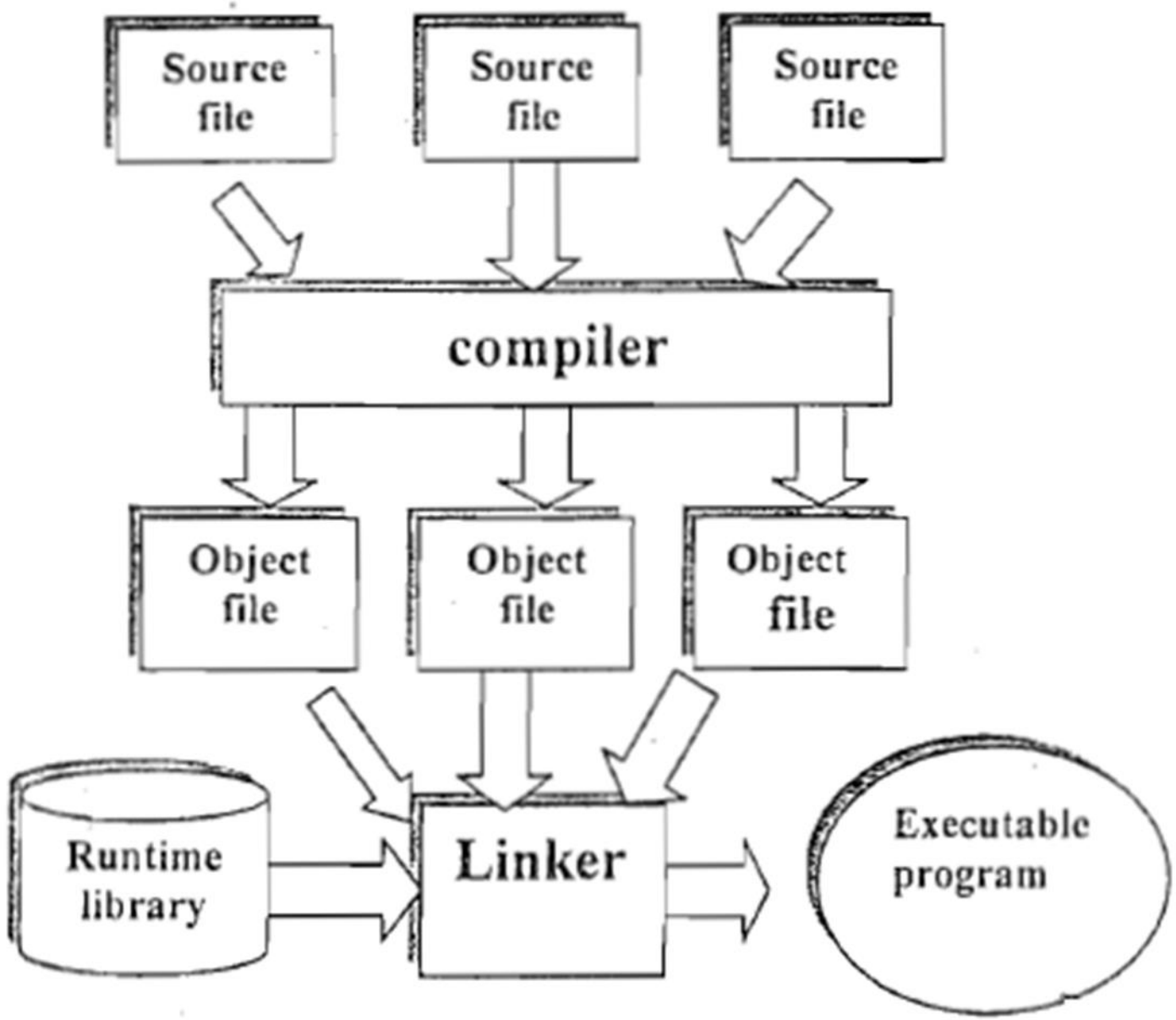
An assembler translates one assembly-language statement into one machine-language statement.

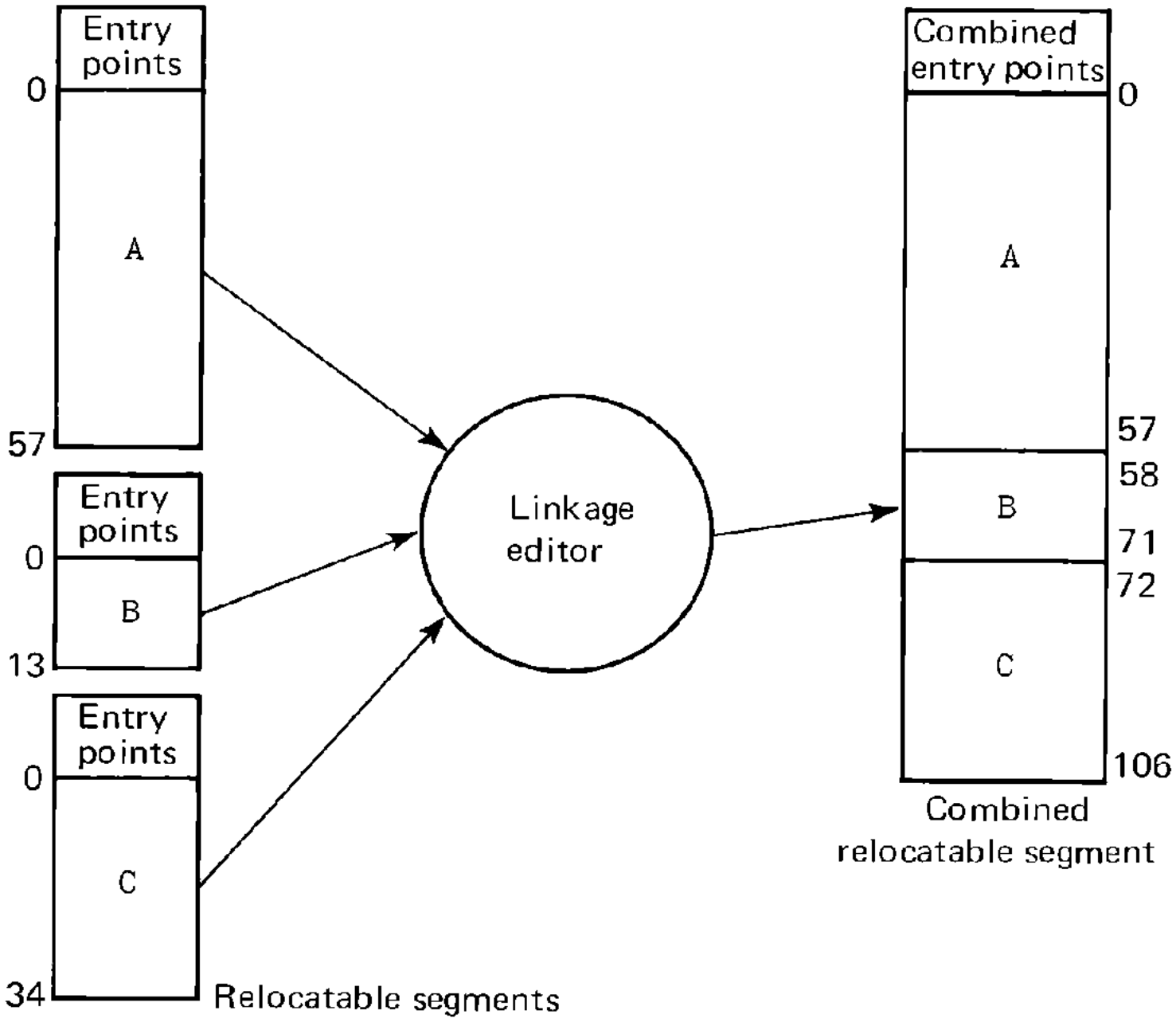
A compiler translates one high-level statement into multiple machine-language statements, *so it is much more difficult to write a correct compiler than an assembler.*



# 3) Linkers

- Collect separate object files into a directly executable file
- Connect an object program to the code for standard library functions and to resource supplied by OS
- Becoming one of the principle activities of a compiler, depends on OS and processor


Source file
Source file
Source file
compiler
Object file
Object file
Object file
Runtime library
Linker
Executable program


Entry points
0
A
57
Entry points
0
B
13
Entry points
0
C
34
Relocatable segments
Linkage editor
Combined entry points
0
A
57
58
B
71
72
C
106
Combined relocatable segment

# 4) Loaders

- Resolve all re-locatable address relative to a given base
- Make executable code more flexible
- Often as part of the operating environment, rarely as an actual separate program

# FUNDAMENTAL PROCESS OF LOADERS

- **Allocation :** the space for program is allocated in the main memory, by calculating the size of the program.
- **Loading –** brings the object program into memory for execution.
- **Relocation –** modifies the object program so that it can be loaded at an address different from the location originally specified.

# Relocatable Code

- **The output of most assemblers is a stream of relocatable binary code.**
  - In relocatable code, operand addresses are relative to where the operating system chooses to load the program.
  - The origin directive of the assembler implies or specifies the load point.
  - Absolute (nonrelocatable) code is most suitable for device and operating system control programming.
- **When relocatable code is loaded for execution, special registers provide the base addressing.**
- **Addresses specified within the program are interpreted as offsets from the base address.**

# 5) Preprocessors

- Delete comments, include other files, and perform macro substitutions
- Required by a language (as in C) or can be later add-ons that provide additional facilities

# Preprocessor

- Preprocessor processes source program before it is passed to compiler.


- Produce a source code file with the preprocessing commands properly sorted out.

# 6) Editors

- Compiler have been bundled together with editor and other programs into an interactive development environment (IDE)
- Oriented toward the format or structure of the programming language, called structure-based
- May include some operations of a compiler, informing some errors

# 7) Debuggers

- Used to determine execution error in a compiled program
- Keep tracks of most or all of the source code information
- Halt execution at pre-specified locations called breakpoints
- Must be supplied with appropriate symbolic information by the compiler

# 8) Profilers

- Collect statistics on the behavior of an object program during execution
  - Called Times for each procedures
  - Percentage of execution time
- Used to improve the execution speed of the program

# 9) Project Managers

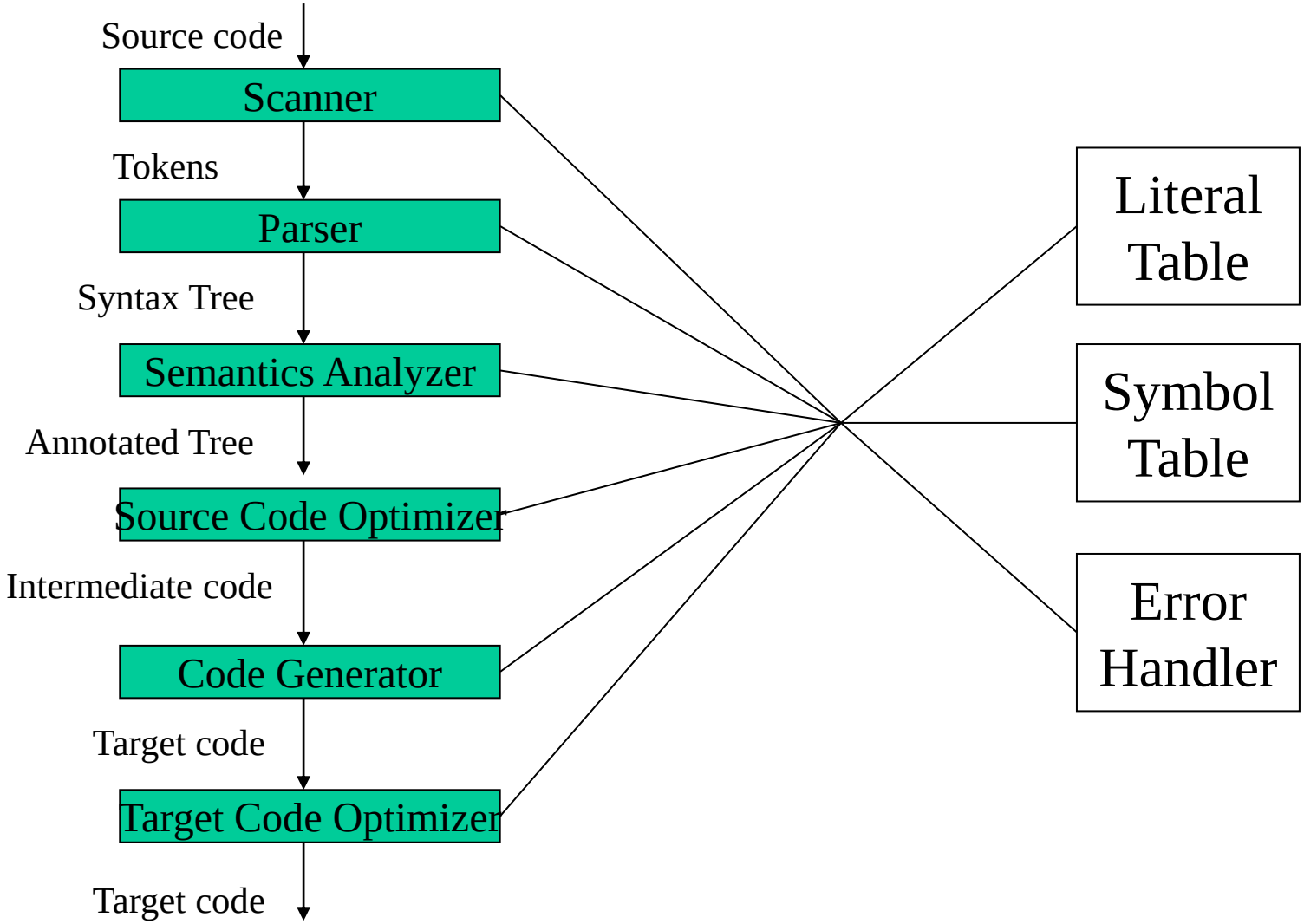
- Coordinate the files being worked on by different people, maintain coherent version of a program
- Language-independent or bundled together with a compiler
- Two popular project manager programs on Unix system
  - Sccs (Source code control system)
  - Rcs (revision control system)

# 1.3 The Translation Process

# The phases of a compiler

- Six phases
  - Scanner
  - Parser
  - Semantic Analyzer
  - Source code optimizer
  - Code generator
  - Target Code Optimizer
- Three auxiliary components
  - Literal table
  - Symbol table
  - Error Handler

# The Phases of a Compiler



# 1.3.1 The Scanner

- **Lexical analysis: it collects sequences of characters into meaningful units called tokens**
- **An example: a[index]=4+2**
  - **a** **identifier**
  - **[** **left bracket**
  - **index** **identifier**
  - **]** **right bracket**
  - **=** **assignment**
  - **4** **number**
  - **+** **plus sign**
  - **2** **number**
- **Other operations: it may enter literals into the literal table**

# The Role of Lexical Analyzer

- Lexical analyzer is the first phase of a compiler.
- Its main task is to read input characters and produce as output a sequence of tokens that parser uses for syntax analysis.

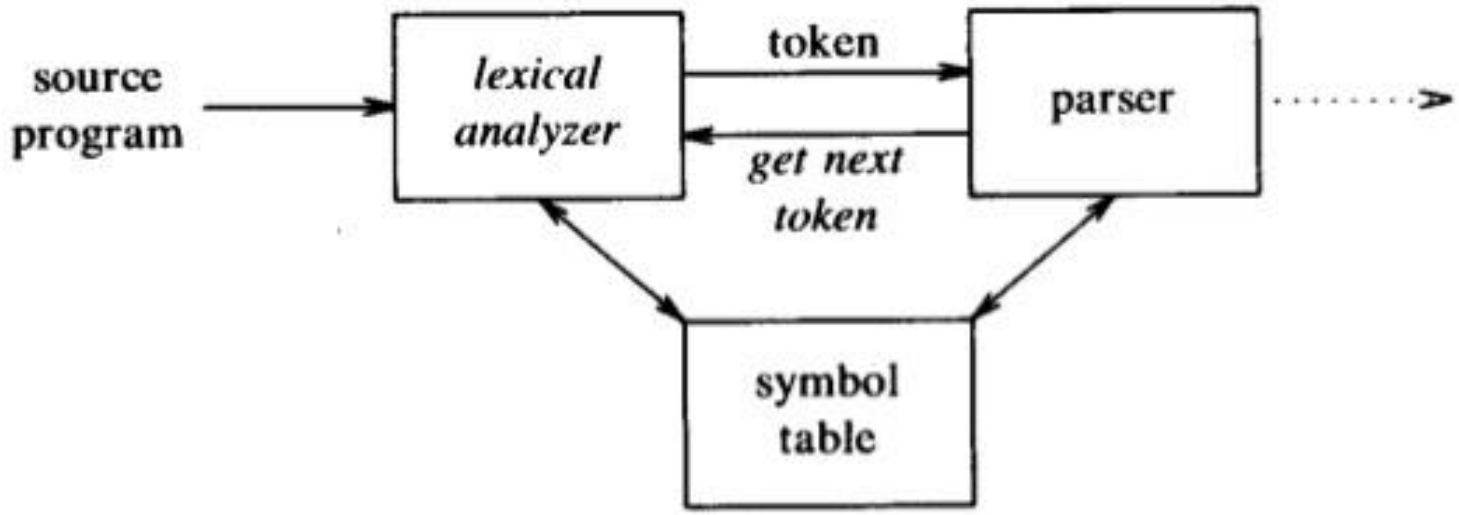
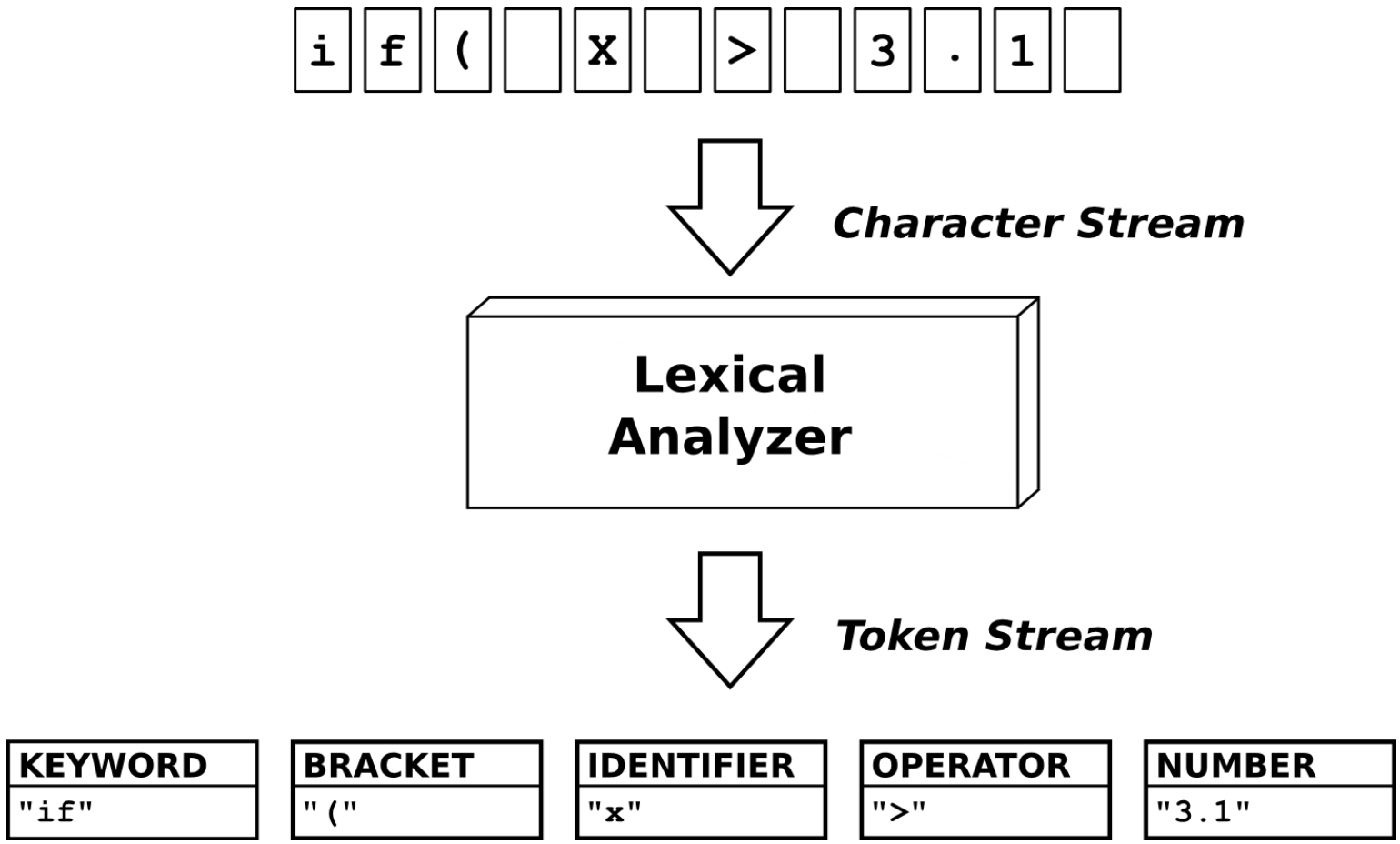


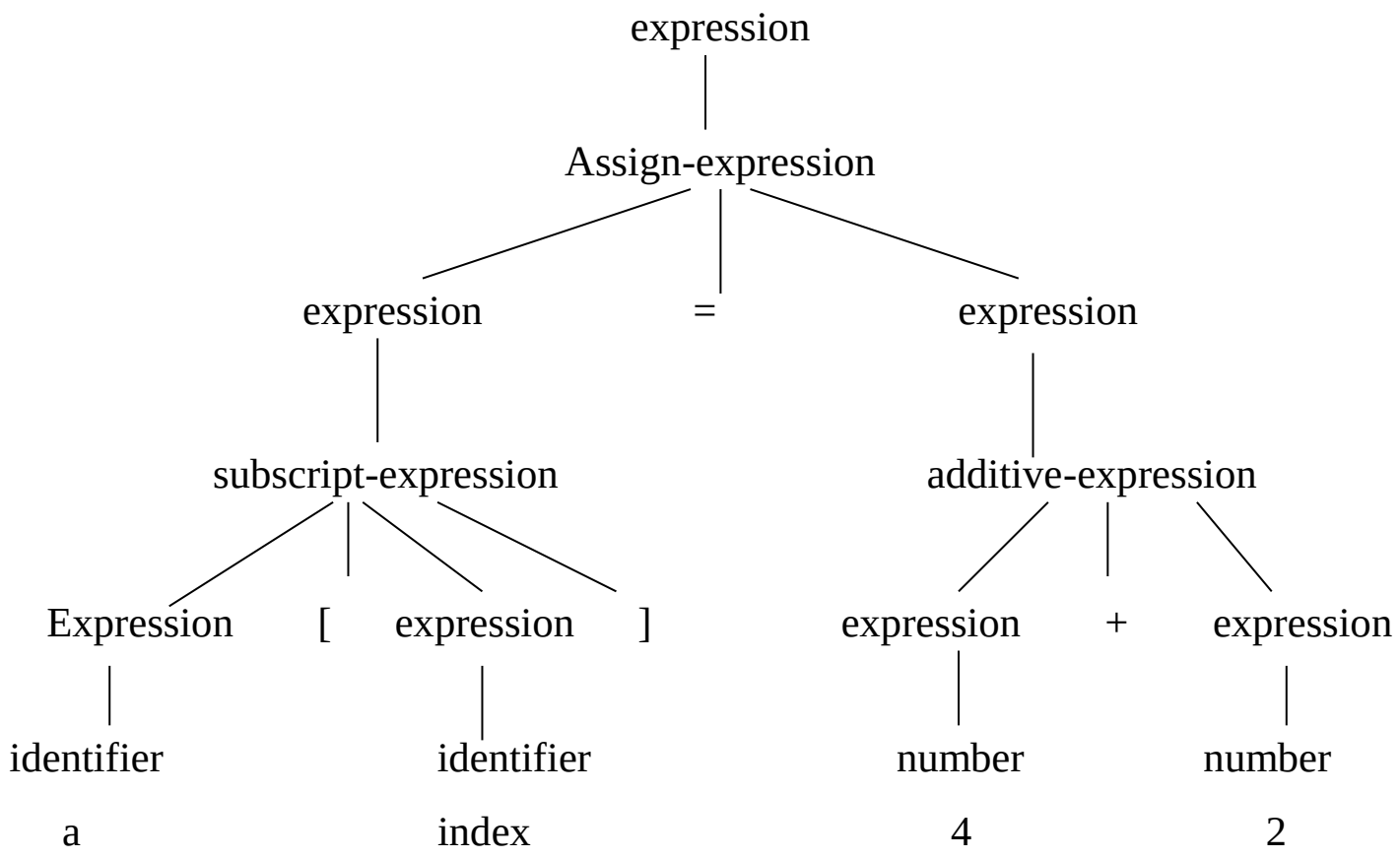

**Fig. 3.1. Interaction of lexical analyzer with parser.**


i f ( x > 3 . 1
Character Stream
Lexical Analyzer
Token Stream
KEYWORD
"if"
BRACKET
"("
IDENTIFIER
"x"
OPERATOR
">"
NUMBER
"3.1"

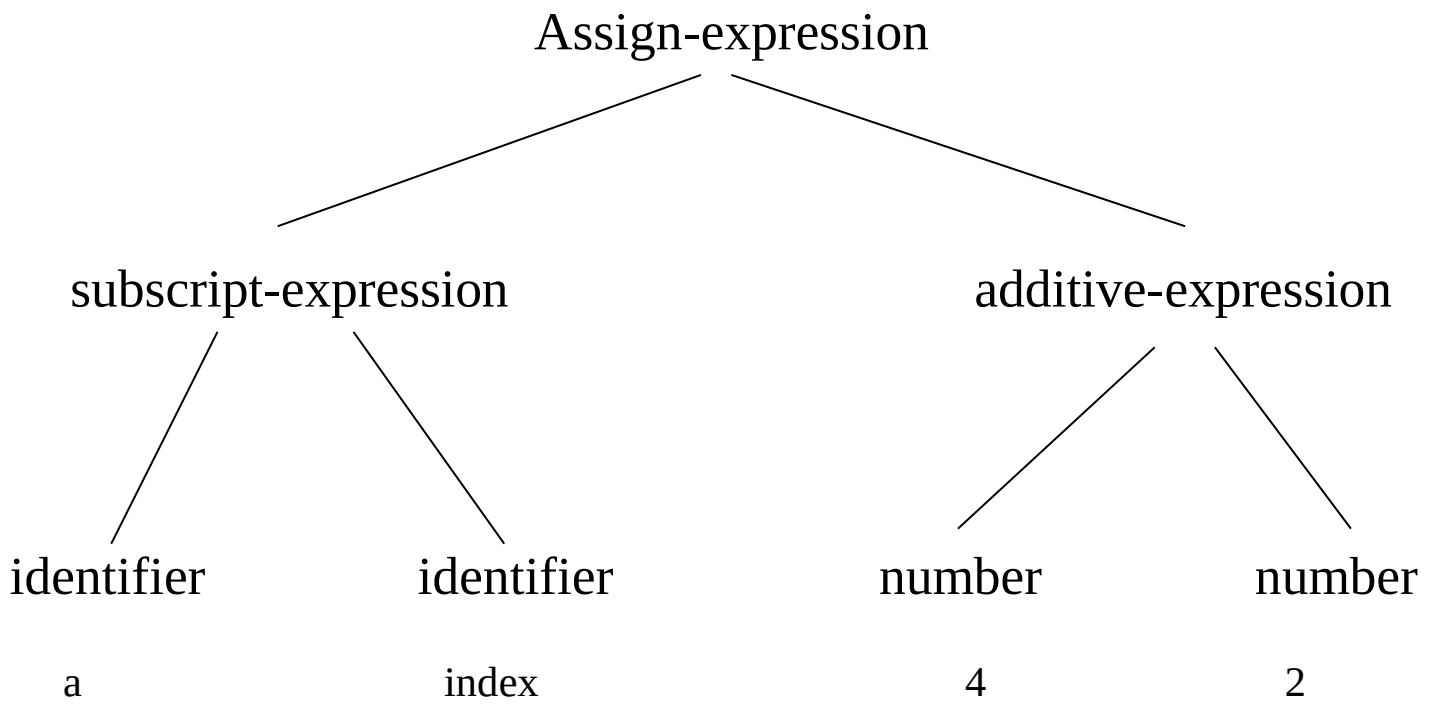
# 1.3.2 The Parser

- Syntax analysis: it determines the structure of the program
- The results of syntax analysis are a parse tree or a syntax tree
- An example: a[index]=4+2
  - Parse tree
  - Syntax tree ( abstract syntax tree)

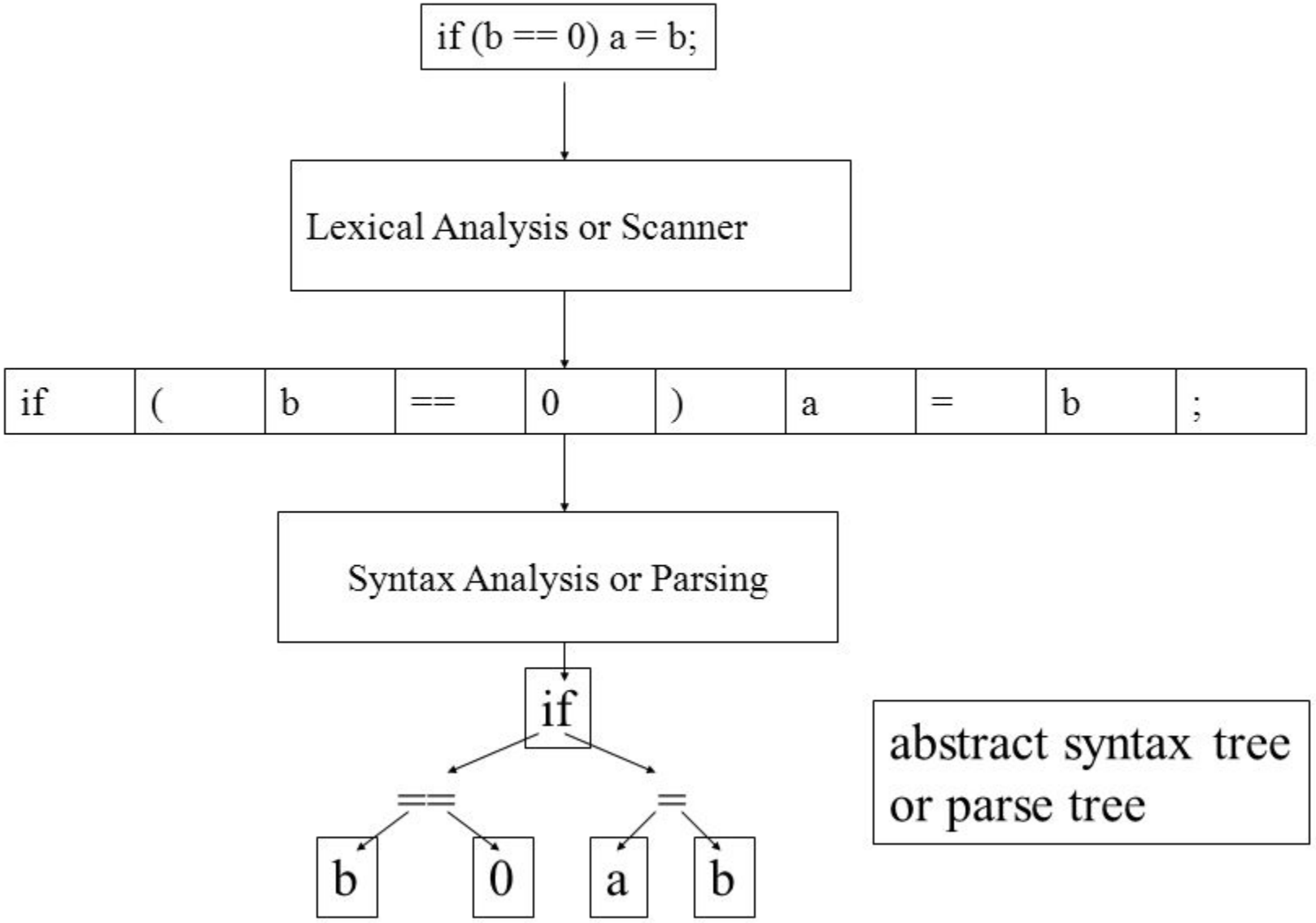
# The Parse Tree



# The Syntax Tree

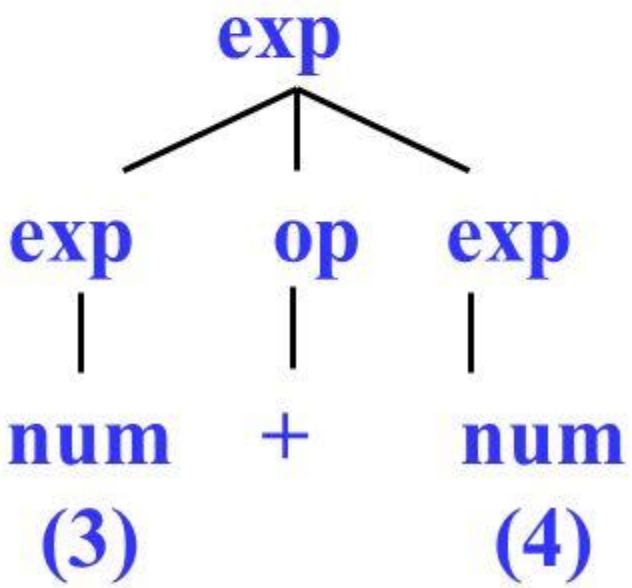


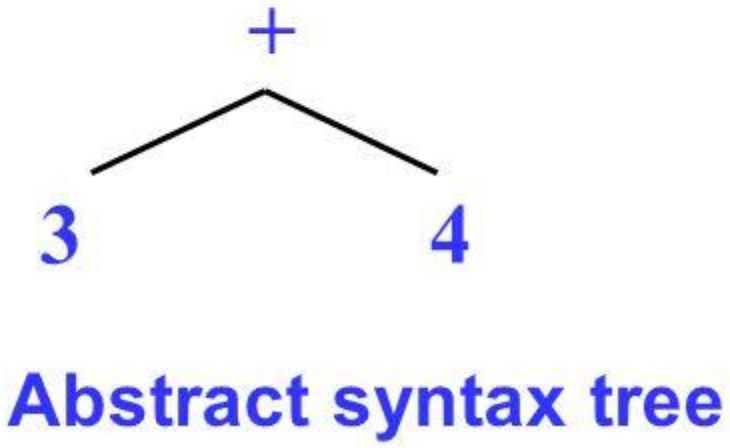
# Where is Syntax Analysis?



# 3.3.2 Abstract Syntax Trees

- The need of abstract syntax tree
  - A parse tree contains much more information than is absolutely necessary for a compiler to produce executable code
  - For example

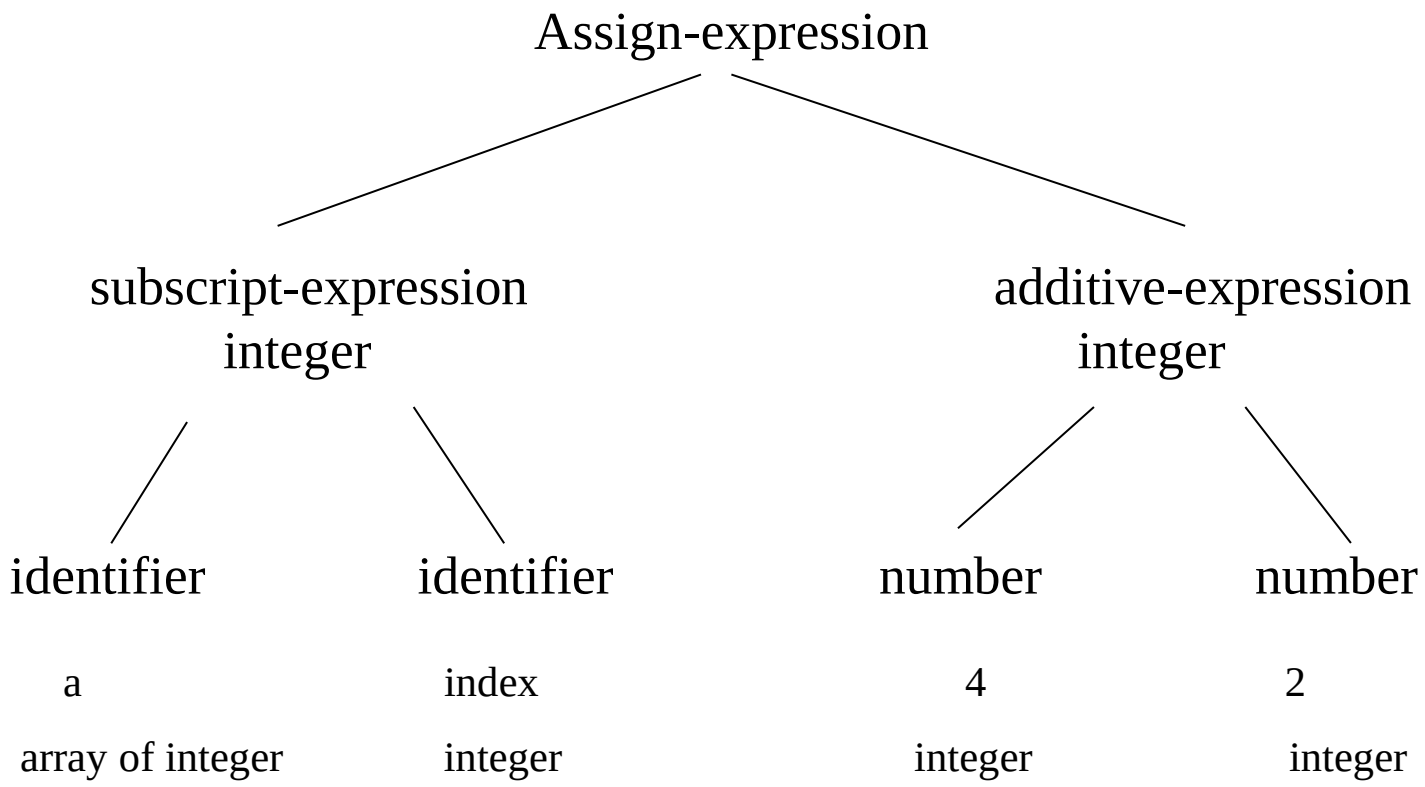


# 1.3.3 The Semantic Analyzer

- The semantics of a program are its “meaning”, as opposed to its syntax, or structure, that
  - determines some of its running time behaviors prior to execution.
- Static semantics: declarations and type checking
- Attributes: The extra pieces of information computed by semantic analyzer
- An example: a[index]=4=2
  - The syntax tree annotated with attributes

# The Annotated Syntax Tree



# 3. Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars
- Ex:

  newval := oldval + 12

  - The type of the identifier *newval* must match with type of the expression *(oldval+12)*

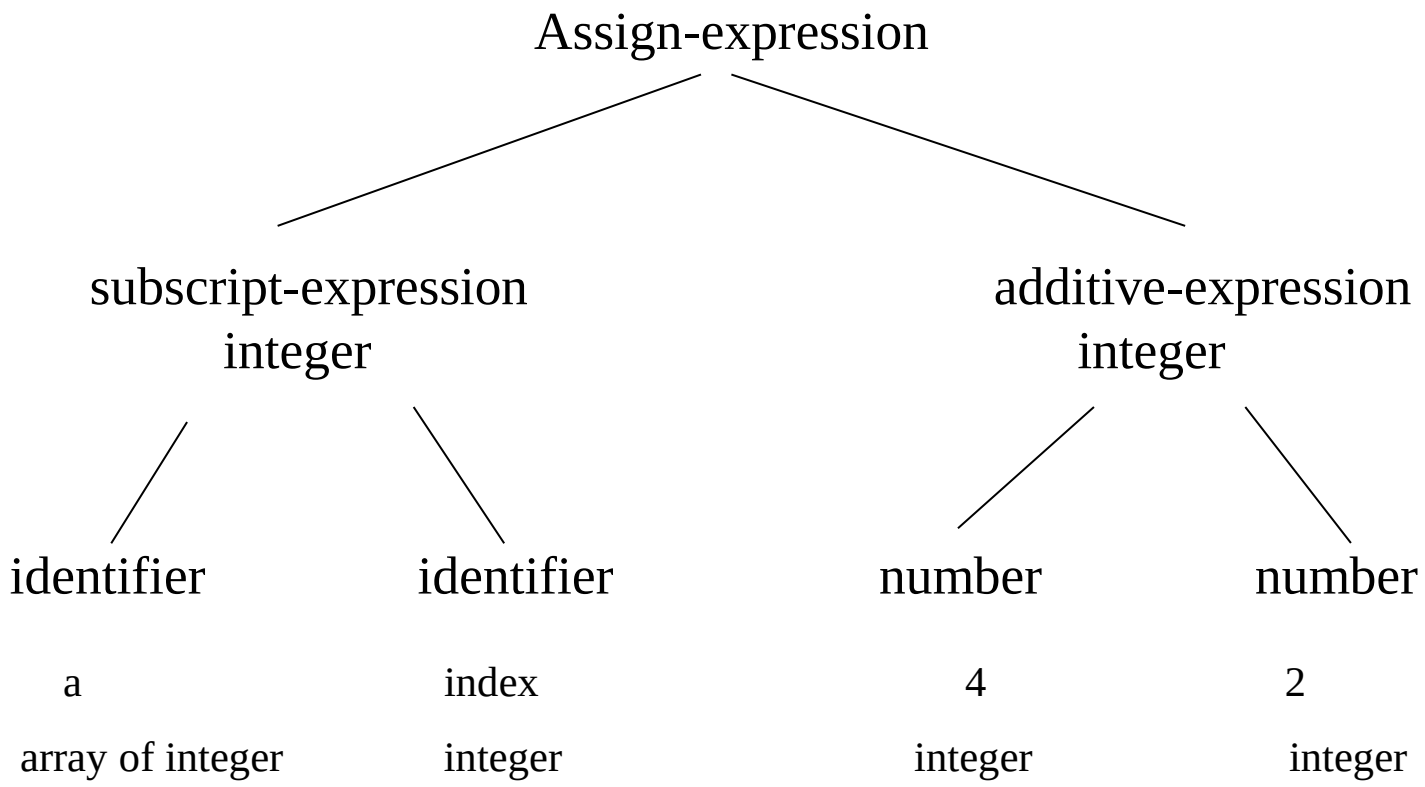
# Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.
- Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

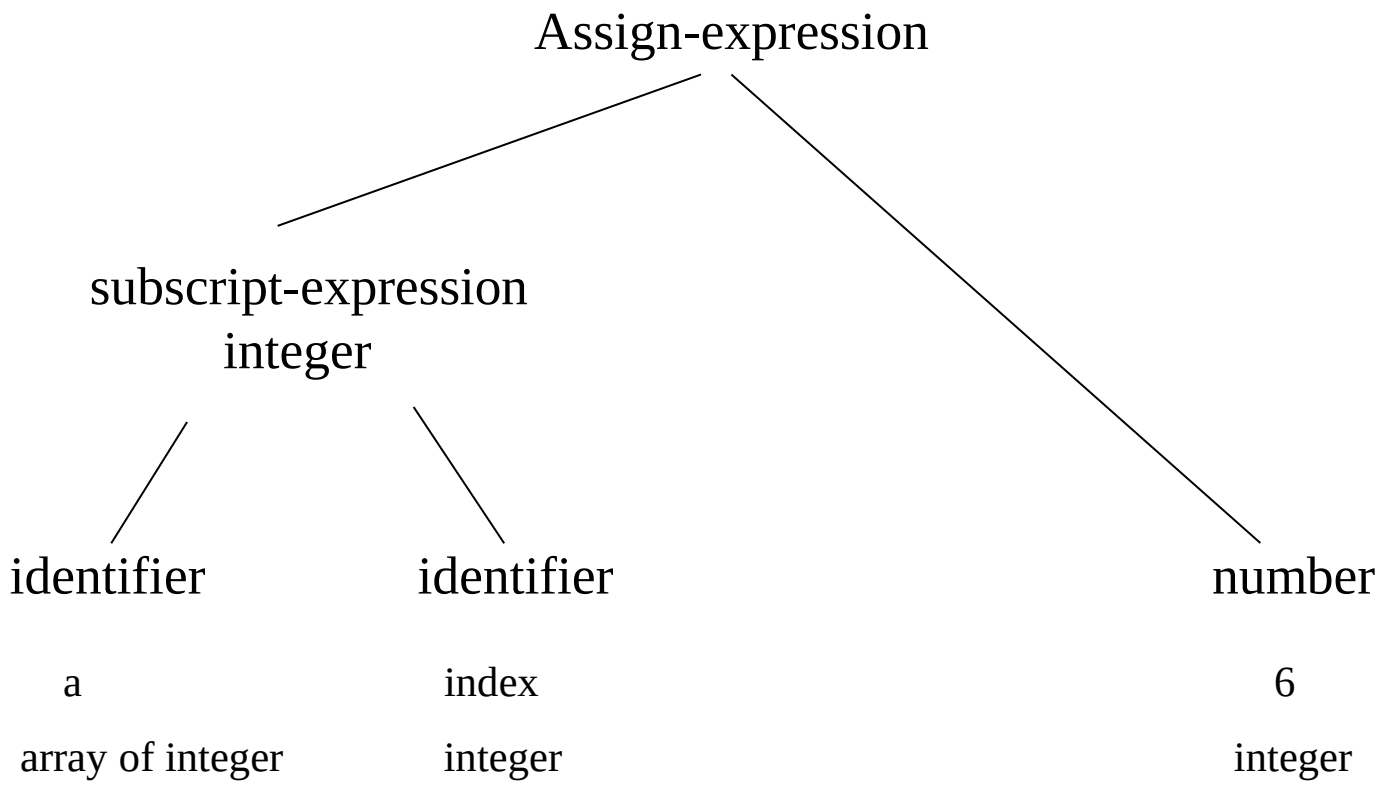
# 1.3.4 The Source Code Optimizer

- The earliest point of most optimization steps is just after semantic analysis
- The code improvement depends only on the source code, and as a separate phase
- Individual compilers exhibit a wide variation in optimization kinds as well as placement
- An example: a[index]=4+2
  - Constant folding performed directly on annotated tree
  - Using intermediate code: three-address code, p-code

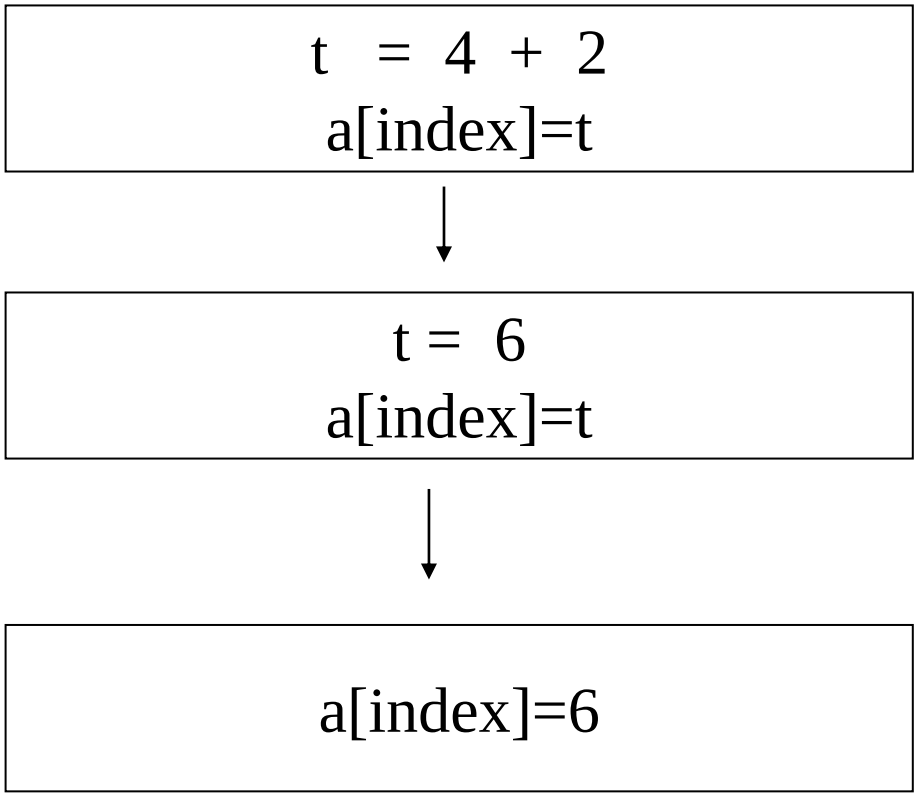
# Optimizations on Annotated Tree

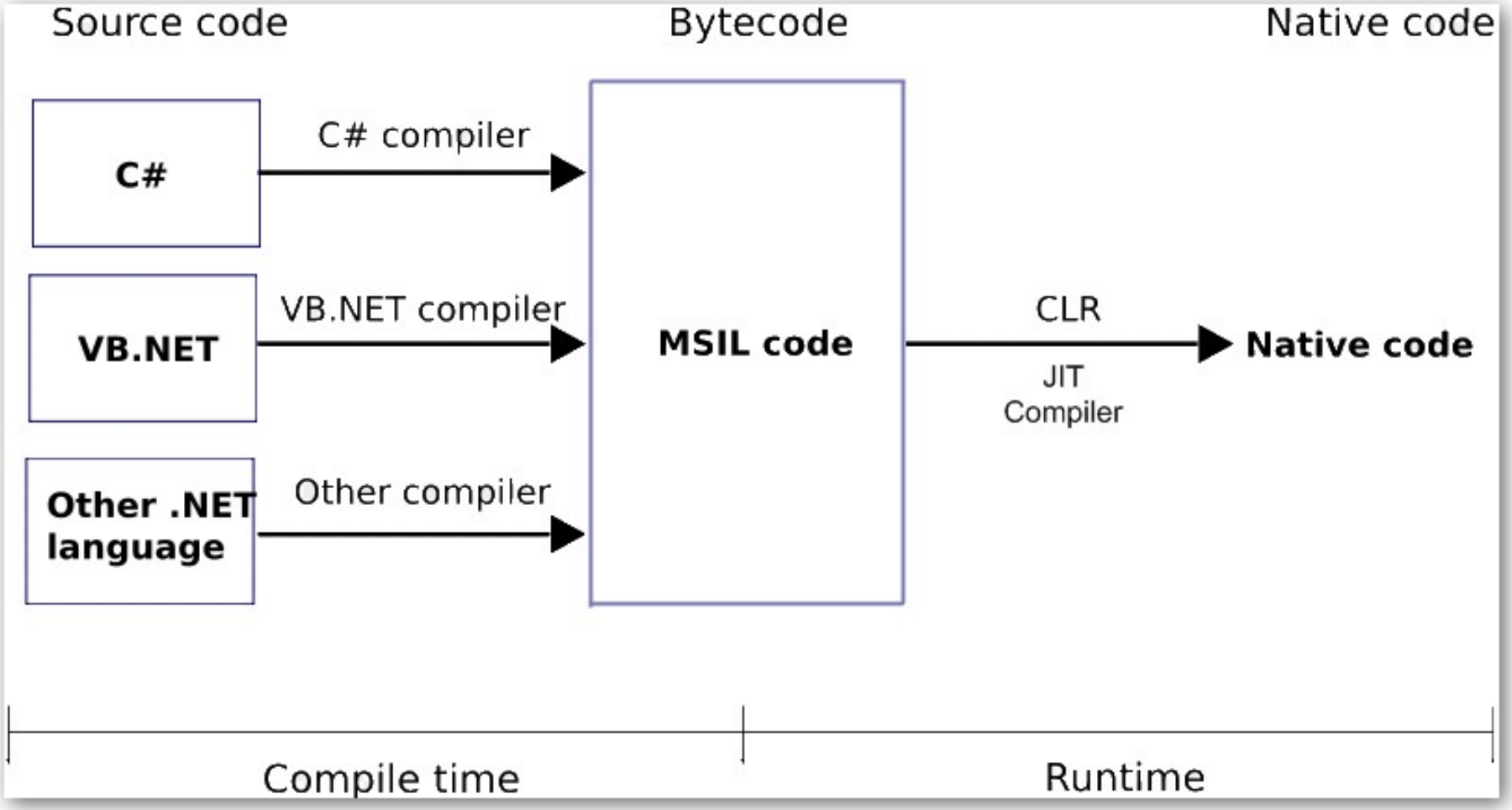


# Optimizations on Annotated Tree



# Optimization on Intermediate Code

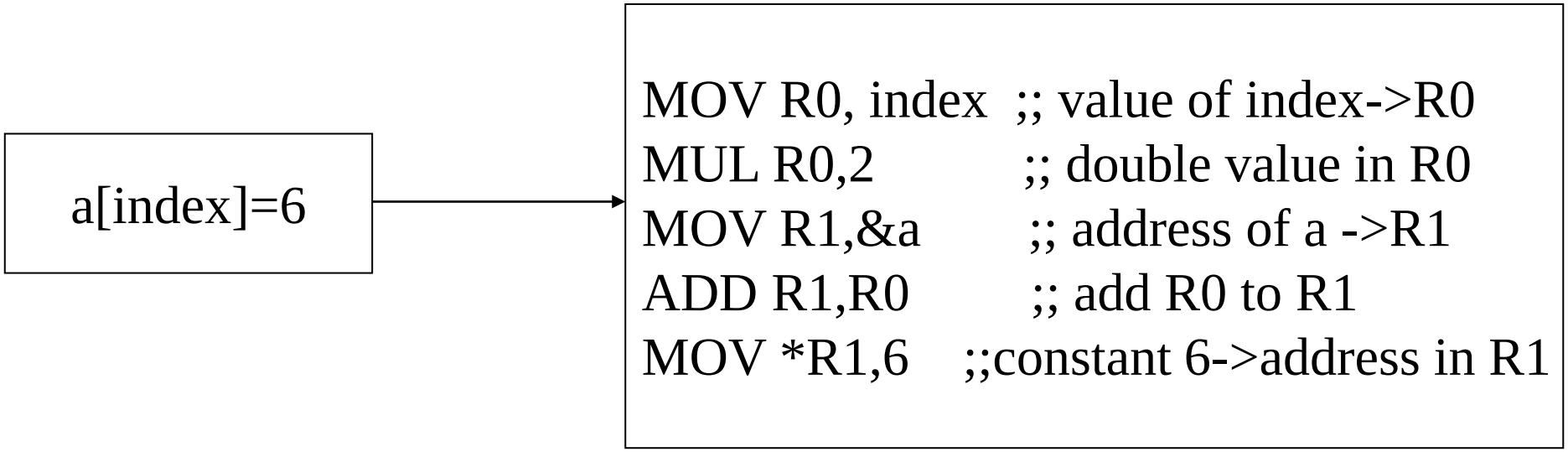



Source code
Bytecode
Native code
C#
C# compiler
VB.NET
VB.NET compiler
Other .NET language
Other compiler
MSIL code
CLR
JIT Compiler
Native code
Compile time
Runtime

# 1.3.5 The Code Generation

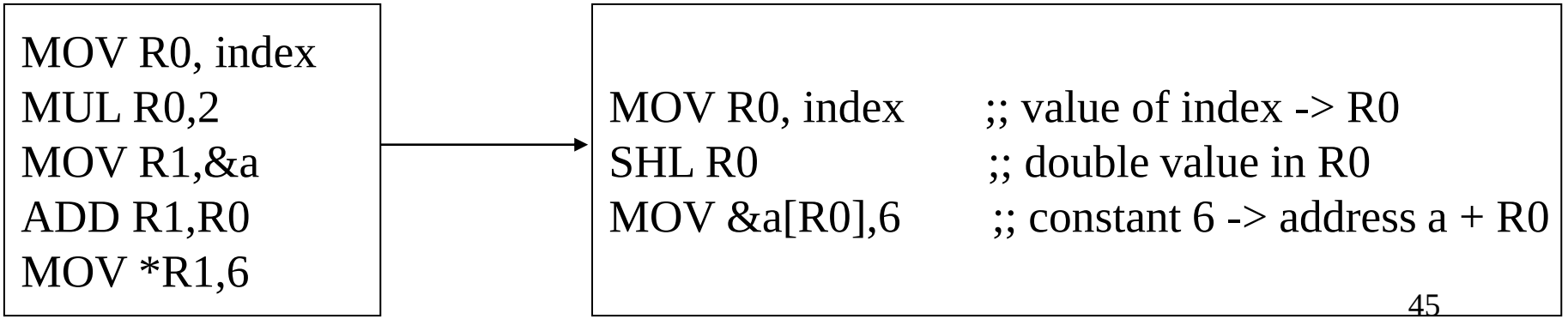
- It takes the intermediate code or IR and generates code for target machine
- The properties of the target machine become the major factor:
  - Using instructions and representation of data
- An example: a[index]=4+2
  - Code sequence in a hypothetical assembly language

# A possible code sequence



# 1.3.6 The Target Code Optimizer

- It improves the target code generated by the code generator:
  - Address modes choosing
  - Instructions replacing
  - As well as redundant eliminating



- Target code improvement include:
  - Allocation and use of registers
  - Selection of better (faster) instructions and addressing modes

# 1.4 Major Data Structure in a Compiler

# Principle Data Structure for Communication among Phases

- TOKENS
  - A scanner collects characters into a token, as a value of an enumerated data type for tokens
  - May also preserve the string of characters or other derived information, such as name of identifier, value of a number token
  - A single global variable or an array of tokens
- THE SYNTAX TREE
  - A standard pointer-based structure generated by parser
  - Each node represents information collect by parser or later, which maybe dynamically allocated or stored in symbol table
  - The node requires different attributes depending on kind of language structure, which may be represented as variable record.

# Principle Data Structure for Communication among Phases

- THE SYMBOL TABLE
  - Keeps information associated with identifiers: function, variable, constants, and data types
  - Interacts with almost every phase of compiler.
  - Access operation need to be constant-time
  - One or several hash tables are often used,
- THE LITERAL TABLE
  - Stores constants and strings, reducing size of program
  - Quick insertion and lookup are essential

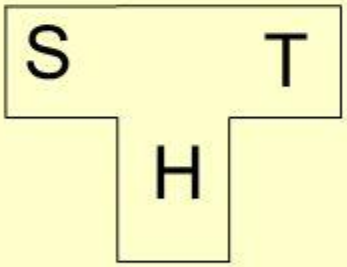
# Principle Data Structure for Communication among Phases

- INTERMEDIATE CODE
  - Kept as an array of text string, a temporary text, or a linked list of structures, depending on kind of intermediate code (e.g. three-address code and p-code)
  - Should be easy for reorganization
- TEMPORARY FILES
  - Holds the product of intermediate steps during compiling
  - Solve the problem of memory constraints or back-patch addressed during code generation

# 1.5 Other Issues in Compiler Structure

# Cross Compiler

- a compiler which generates target code for a different machine from one on which the compiler runs.
- A host language is a language in which the compiler is written.
  - T-diagram




- Cross compilers are used very often in practice.

# Error Handling

- Static (or compile-time) errors must be reported by a compiler
  - Generate meaningful error messages and resume compilation after each error
  - Each phase of a compiler needs different kind of error handing
- Exception handling
  - Generate extra code to perform suitable runtime tests to guarantee all such errors to cause an appropriate event during execution.

A syntax error is an error in the source code of a program. Since computer programs must follow strict syntax to compile correctly, any aspects of the code that do not conform to the syntax of the programming language will produce a syntax error.

```
function testFunction()
{
echo "Just testing.";
}}
```

syntax errors are small grammatical mistakes, sometimes limited to a single **character.** For example, a missing semicolon at the end of a line or an extra bracket at the end of a **function** may produce a syntax error. In the **PHP** code below, the second closed bracket would result in a syntax error since there is only one open bracket in the function.

```
function testFunction()
{
echo "Just testing.";
}}
```

Unlike **logic errors**, which are errors in the flow or logic of a program,

During the lexical analysis phase this type of error can be detected. Lexical error is a sequence of characters that does not match the pattern of any token.

...

Example:

```
Void main()
{
int x=10, y=20;
char * a;
a= &x;
x= 1xab;
}
```

What is semantic error example?
semantic errors are a type of compile errors which are gramitically correct unlike syntax errors for example let us say the following declaration: int a; float a; here both the statements are syntactically correct but you cannot use both of. them simultaneously it is an example of semantic error.