Chapter 1: Introduction

1.1 What is a Compiler?

- A computer program that 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

1.2 Programs Related to Compilers

1) Interpreters
- Execute the source program immediately rather than generating object code
- Speed of execution is slower than compiled code by a factor of 10 or more
- Share many of their operations with compilers

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

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

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

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

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

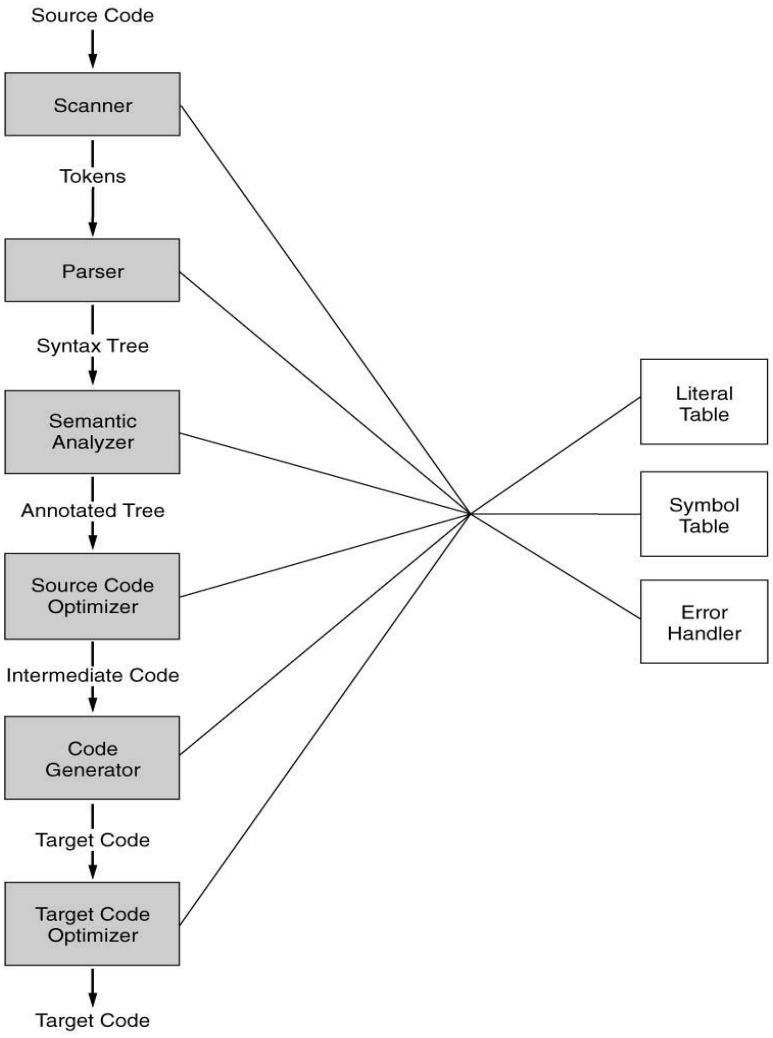
8) Profilers
- Collect statistics on the behavior of an object program during execution
  - Called Times for each procedures
  - Percentage of execution time
- Used to improve the execution speed of the program

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

1.3 The Translation Process

The phases of a compiler can be described and implemented in many ways, but here is a representative picture of the process:



These phases transform the source code from a sequence of characters created by the programmer into an abstract representation of the program, and then into the desired target code. Each stage adds some new information to the representation.

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

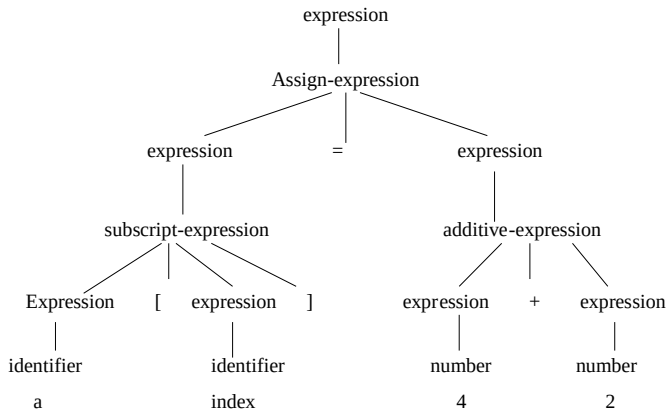
1.3.1 The Scanner

- 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, and identifiers into the symbol table

1.3.2 The Parser

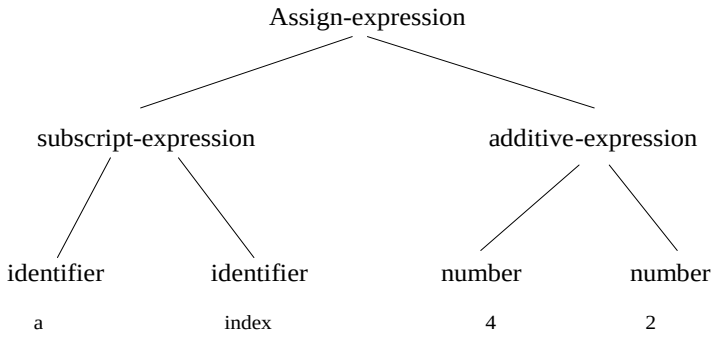
- Syntax analysis: it determines the structure of the program
- The results of syntax analysis are a parse tree or a syntax tree
- An example: a[index]=4+2
  - Parse tree

The Parse Tree



  - Syntax tree (abstract syntax tree)

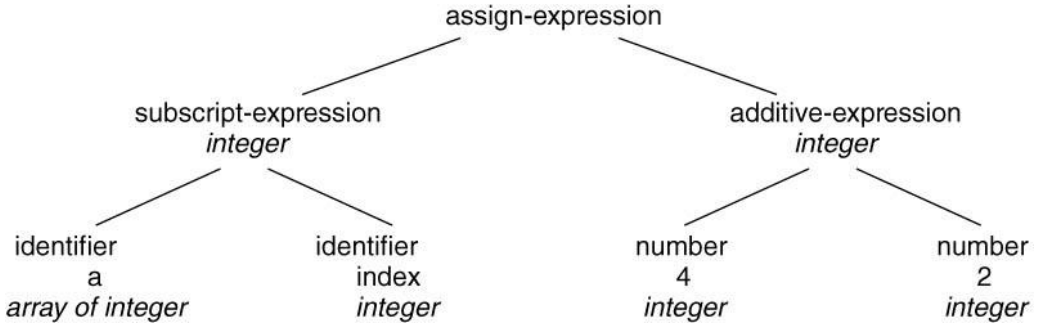
The Syntax Tree



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**

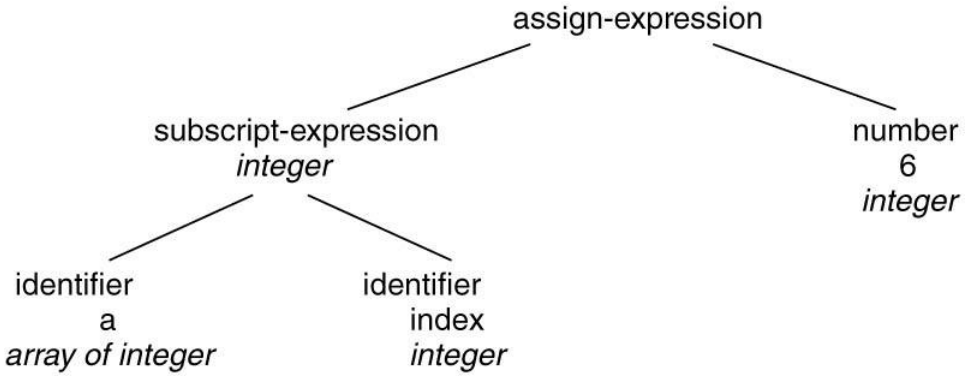
The Annotated Syntax Tree



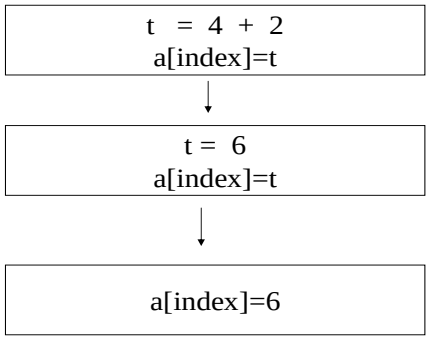
1.3.4 The Source Code Optimizer

- 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



Optimization on Intermediate Code



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

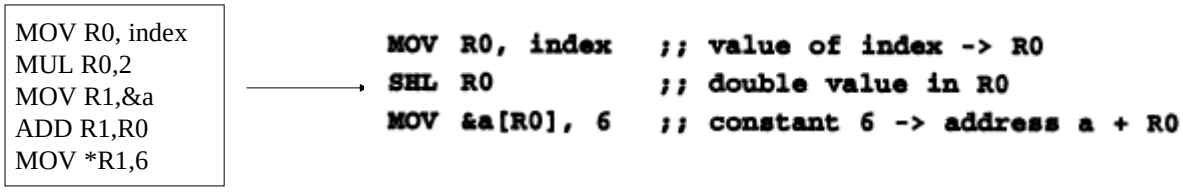
A possible code sequence

a[index]=6 ——

```
MOV   R0, index   ;; value of index -> R0
MUL   R0, 2       ;; double value in R0
MOV   R1, &a      ;; address of a -> R1
ADD   R1, R0      ;; add R0 to R1
MOV   *R1, 6      ;; constant 6 -> address in R1
```

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 eliminating redundant operations



1.4 Major Data Structures 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.
- **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
- **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