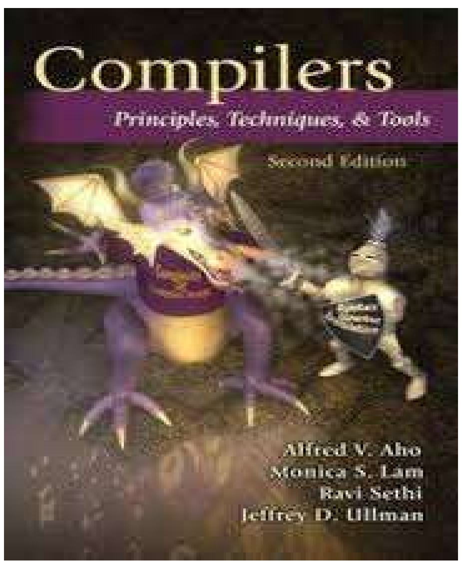
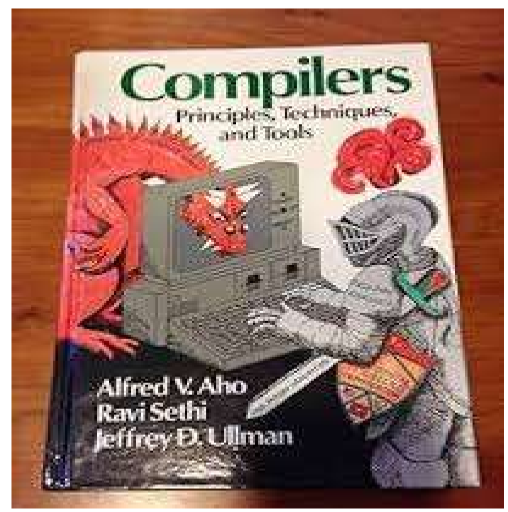
Addis Ababa Institute of Technology - AAiT

School of Information Technology and Engineering - SiTE

Compiler Design

Henock Mulugeta (PhD)





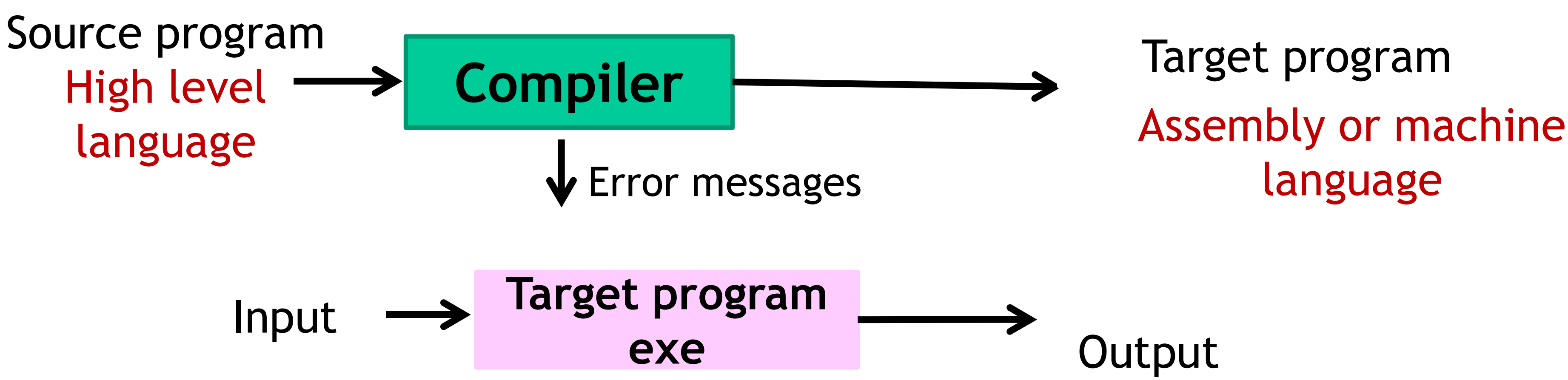
Outline

- Introduction
- Programs related to compiler
- The translation process
  - **Analysis**
    - Lexical analysis
    - Syntax analysis
    - Semantic analysis
  - **Synthesis**
    - IC generator
    - IC optimizer
    - Code generator
    - Code optimizer
- Major data and structures in a compiler
- Types of compilers
- Compiler construction tools

Introduction

What is a compiler?

- a program that reads a program written in one language (the source language) and translates it into an equivalent program in another language (the target language).
- Why we design compiler?
- Why we study compiler construction techniques?
  - Compilers provide an essential interface between applications and architectures
  - Compilers embody a wide range of theoretical techniques

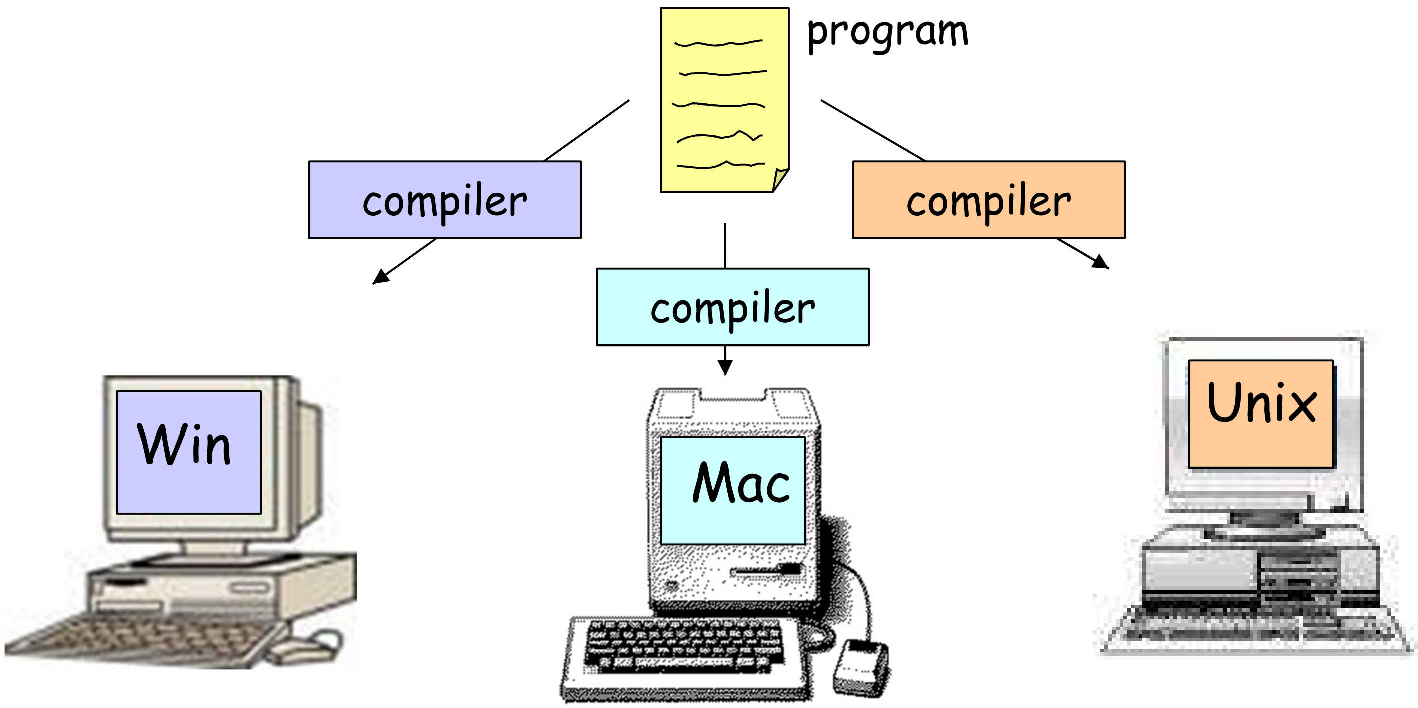


Introduction…

- Using a high-level language for programming has a large impact on how fast programs can be developed.
- **The main reasons for this are:**
  - Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
  - The compiler can spot some obvious programming mistakes.
  - Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.
  - The same program can be compiled to many different machine languages
    - and, hence, be brought to run on many different machines.

Introduction…

- Since different *platforms*, or hardware architectures along with the operating systems (Windows, Macs, Unix), require different machine code, you must compile most programs separately for each platform.

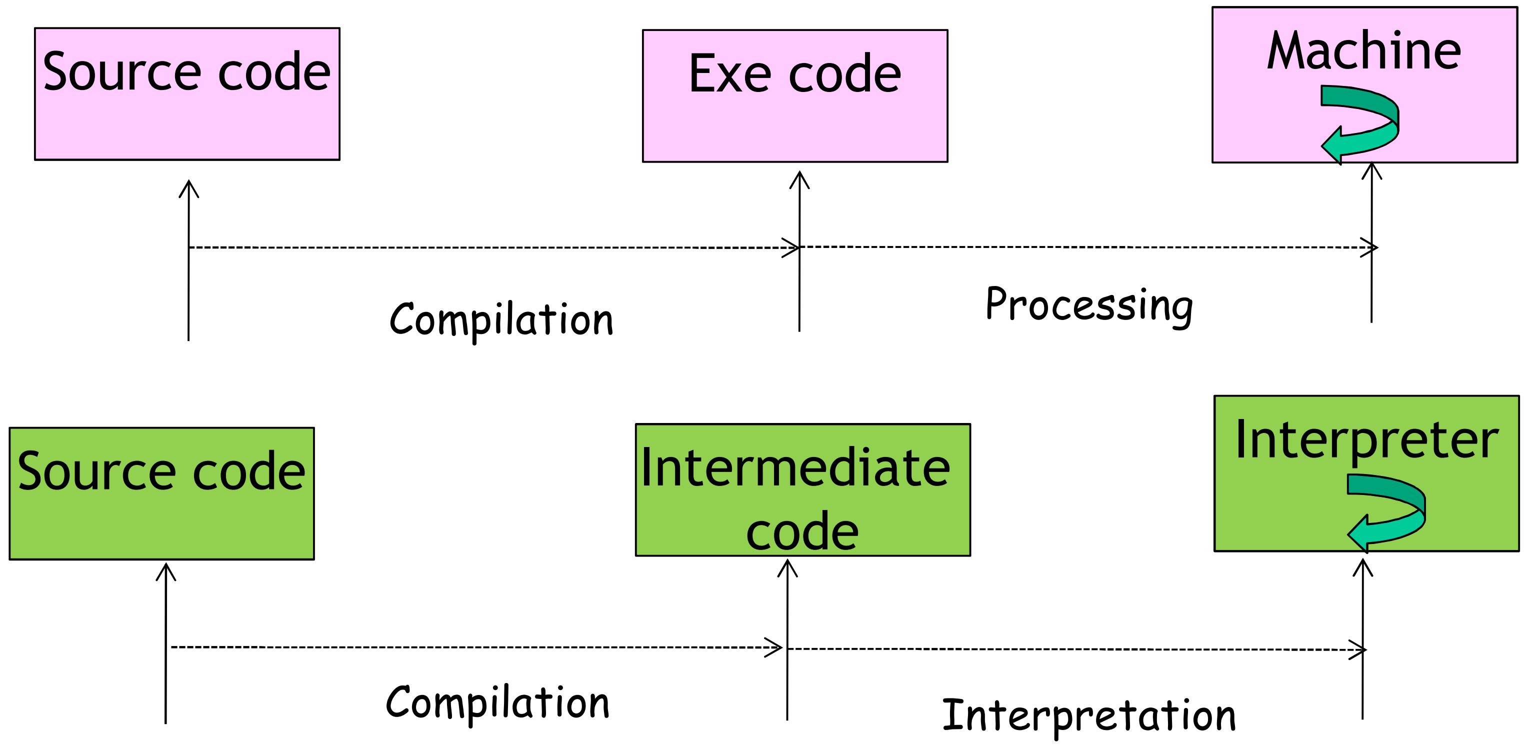


Programs related to compilers

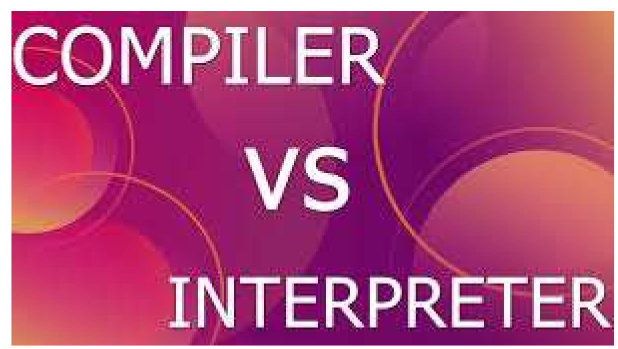
- **Interpreter**
  - Is a program that reads a source program and executes it
  - Works by analyzing and executing the source program commands *one at a time*
  - Does not translate the whole source program into object code

- Interpretation is important when:
  - Programmer is working in interactive mode and needs to view and update variables
  - Running speed is not important
  - Commands have simple formats, and thus can be quickly analyzed and executed
  - Modification or addition to user programs is required as execution proceeds

Programs related to compilers…



Interpreter and compiler



Interpreter and compiler differences

- *Interpreter* takes one statement then translates it and executes it and then takes another statement.
- *Interpreter* will stop the translation after it gets the first error.
- *Interpreter* takes less time to analyze the source code.
- Over all execution speed is less.

- While *compiler* translates the entire program in one go and then executes it.
- *Compiler* generates the error report after the translation of the entire program.
- *Compiler* takes a large amount of time in analyzing and processing the high level language code.
- Overall execution time is faster.

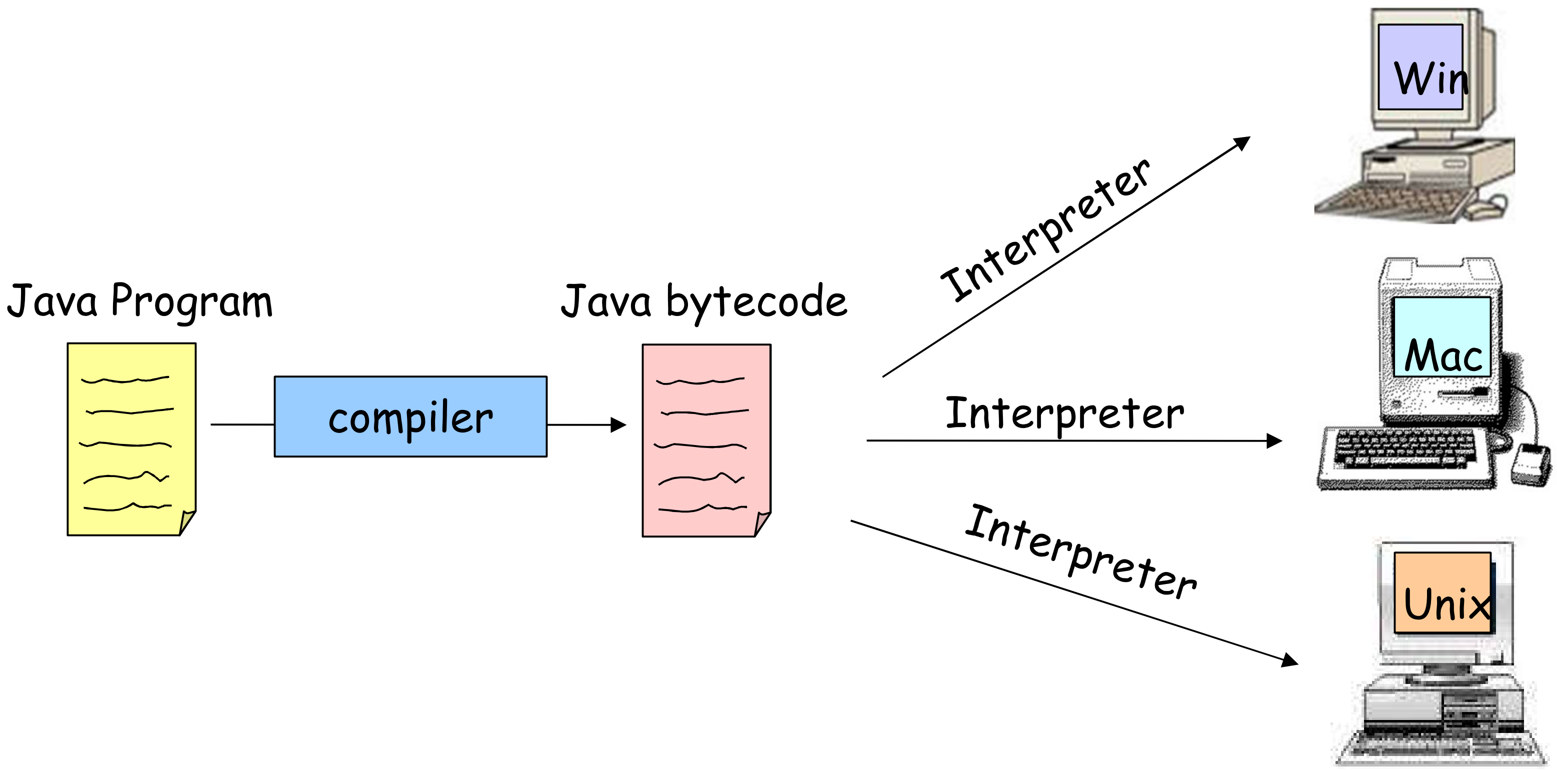
Programs related to compilers…

- **Interpreter…**

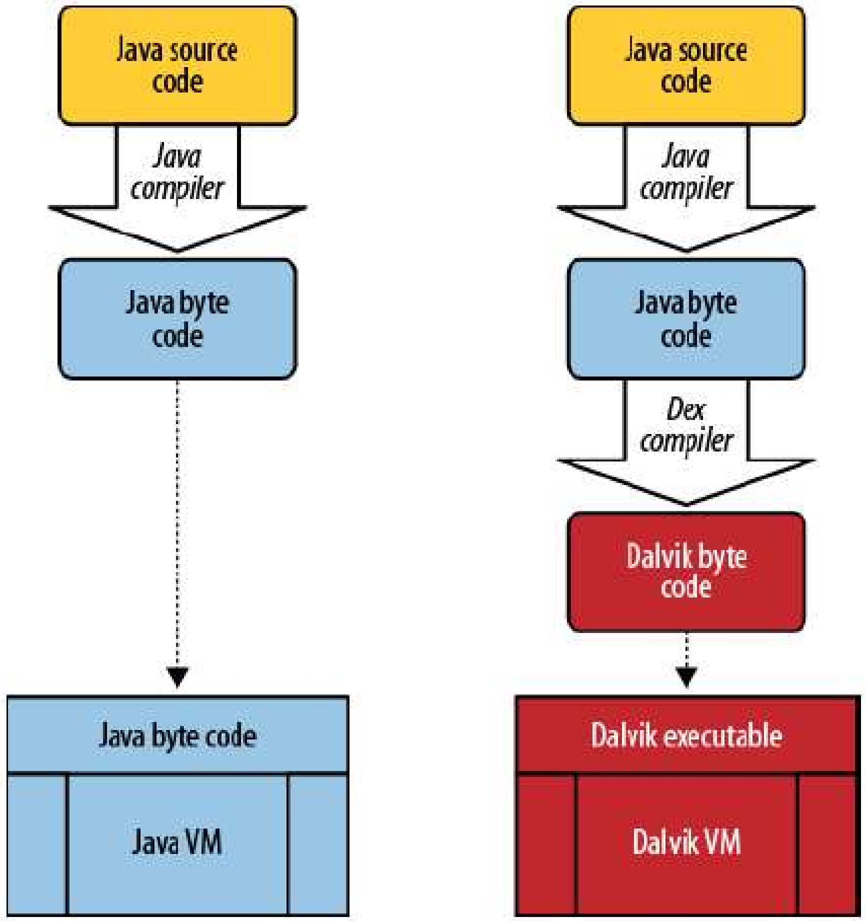
- Well-known examples of interpreters:
  - Basic interpreter, Lisp interpreter, UNIX shell command interpreter, SQL interpreter, java interpreter…
- In principle, any programming language can be either interpreted or compiled:
  - Some languages are designed to be interpreted, others are designed to be compiled
- Interpreters involve large overheads:
  - Execution speed degradation can vary from 10:1 to 100:1
  - Substantial space overhead may be involved

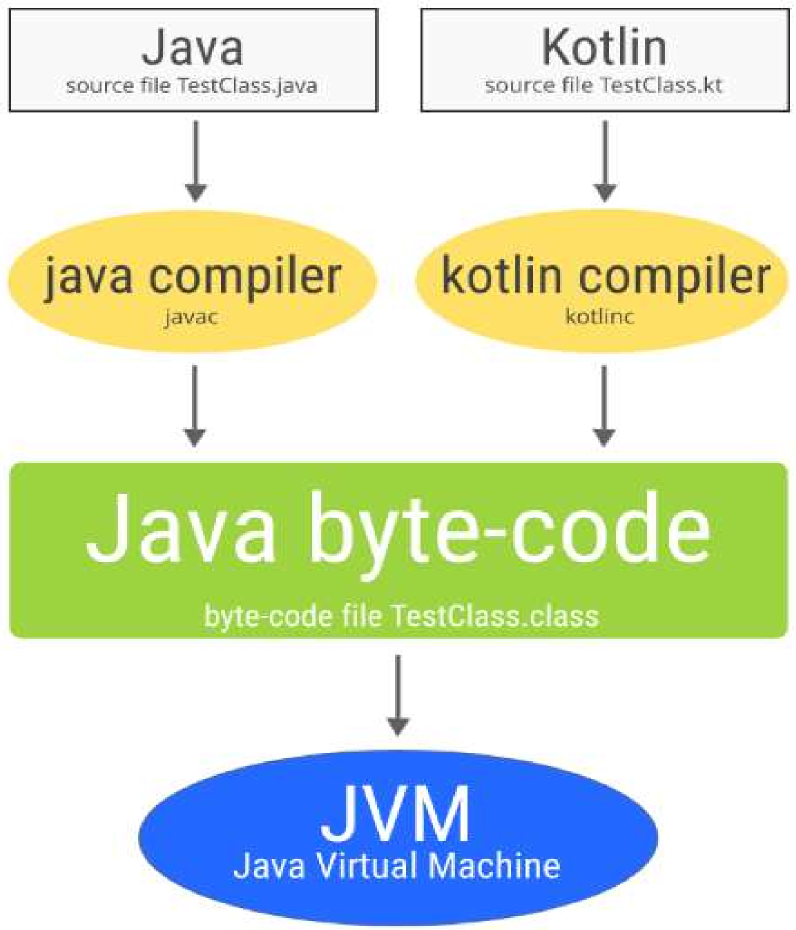
E.g., Compiling Java Programs

- The Java compiler produces *bytecode* not machine code
- Bytecode is converted into machine code using a *Java Interpreter*
- You can run bytecode on any computer that has a Java Interpreter installed



Android: Java and Kotlin





Programs related to compiler…

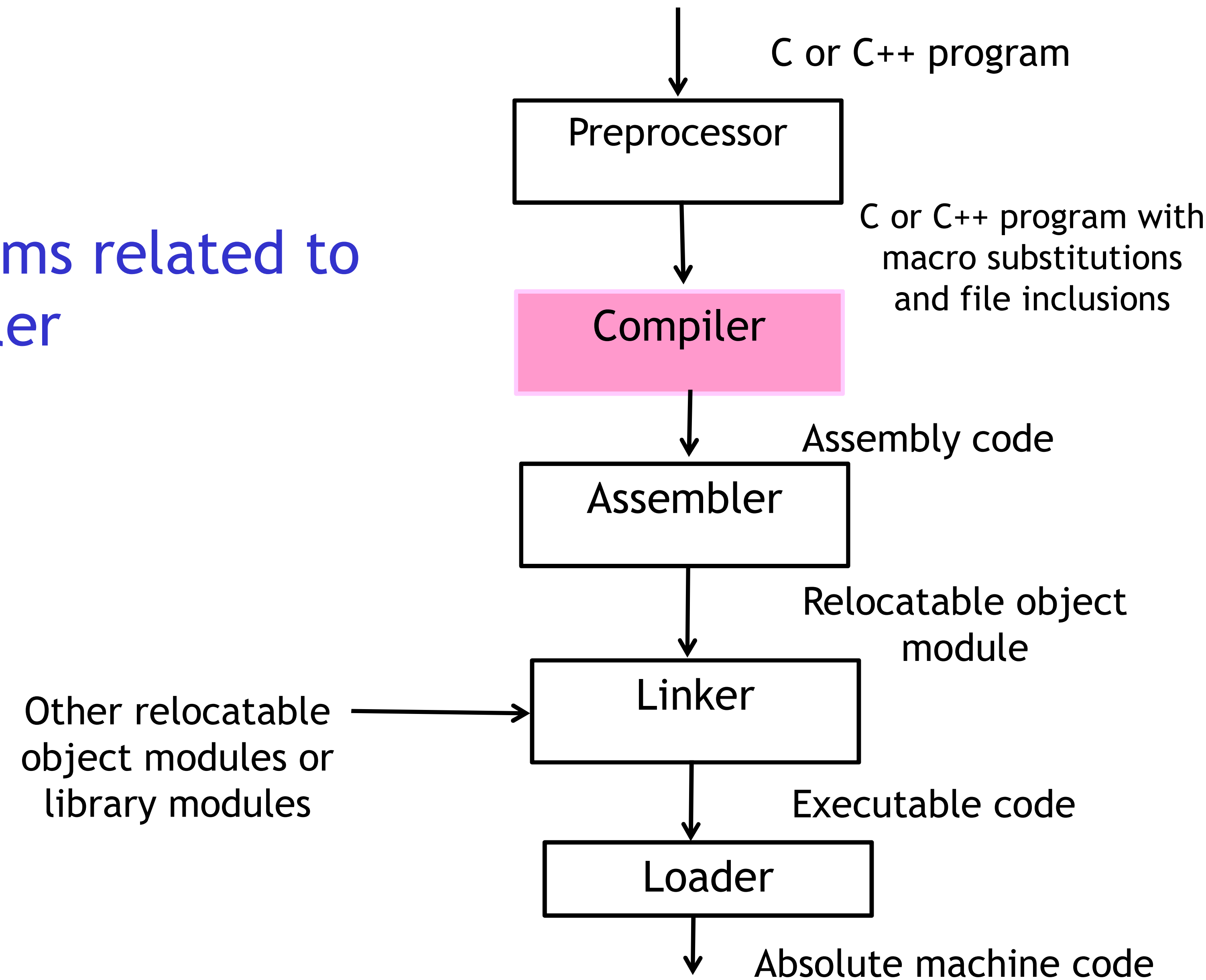
- **Assemblers**
  - Translator for the assembly language.
  - Assembly code is translated into machine code
  - Output is relocatable machine code.

- **Linker**
  - Links object files separately compiled or assembled
  - Links object files to standard library functions
  - Generates a file that can be loaded and executed

Programs related to compiler…

- **Loader**
  - Loading of the executable codes, which are the outputs of linker, into main memory.
- **Pre-processors**
  - A pre-processor is a separate program that is called by the compiler before actual translation begins.
  - Such a pre-processor:
    - Produce input to a compiler
    - can delete comments,
    - Macro processing (substitutions)
    - include other files...

Programs related to compiler



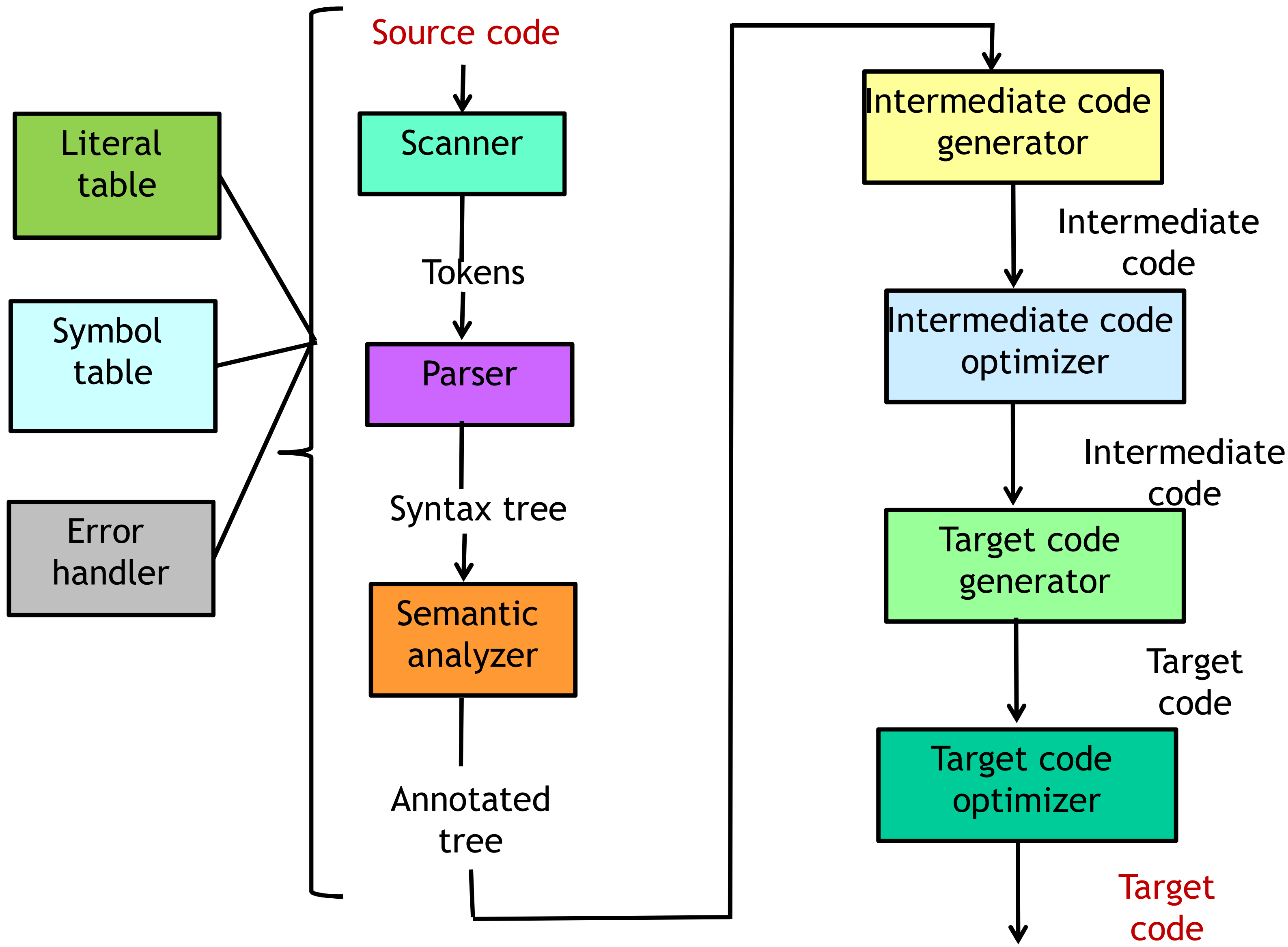
The translation process

- A compiler consists of internally of a number of steps, or phases, that perform distinct logical operations.

- The phases of a compiler are shown in the next slide, together with *three auxiliary components* that interact with some or all of the phases:
  - The symbol table,
  - the literal table,
  - and error handler.

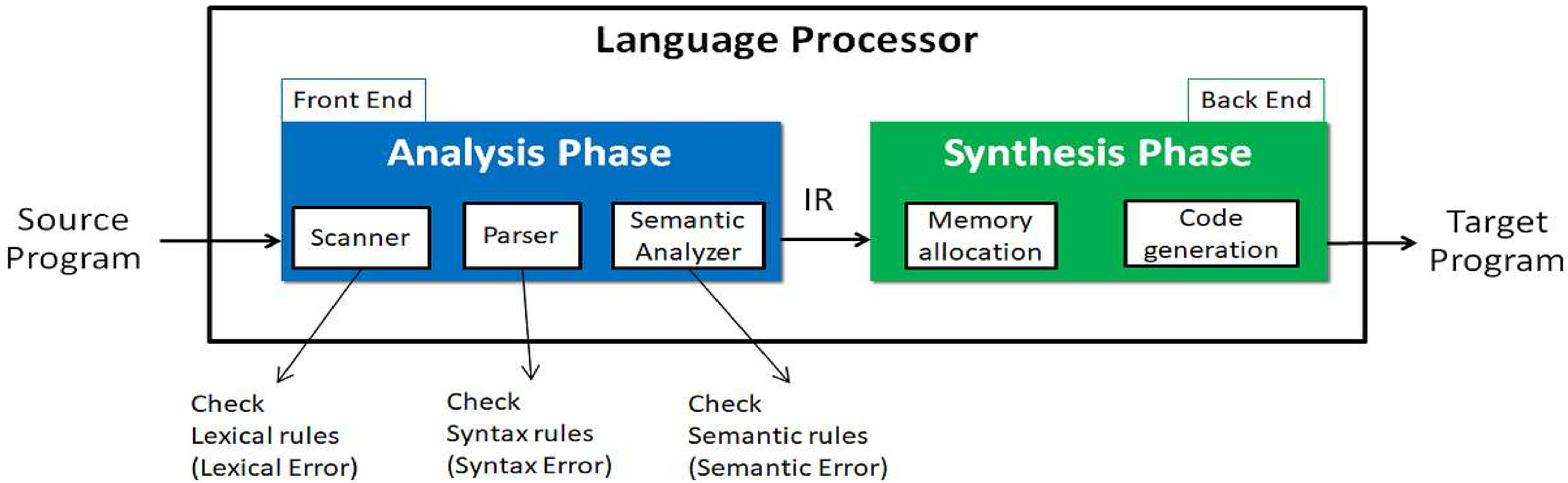
- There are two important parts in compilation process:
  - Analysis and
  - Synthesis.

The translation process…



Analysis and Synthesis

- **Analysis (front end)**
  - Breaks up the source program into constituent pieces and
  - Creates an intermediate representation of the source program.
  - During analysis, the operations implied by the source program are determined and recorded in hierarchical structure called a *tree*.
- **Synthesis (back end)**
  - The synthesis part constructs the desired program from the intermediate representation.

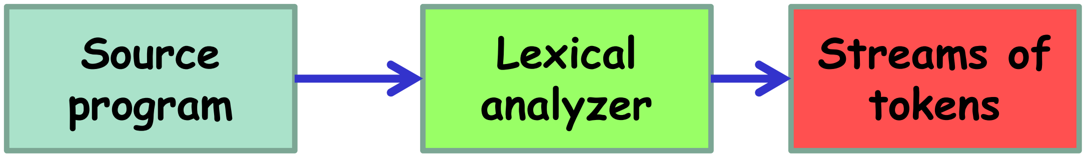


Analysis of the source program

- **Analysis consists of three phases:**
  - Linear/Lexical analysis
  - Hierarchical/Syntax analysis
  - Semantic analysis

1. Lexical analysis or Scanning

- A **lexical analyzer**, also called a **lexer** or a **scanner**,
  - receives a stream of characters from the source program and
  - groups them into **tokens**.
- A **token** is a sequence of characters having a collective meaning.
- Examples:
  - Identifiers
  - Keywords
  - Symbols (+, -, …)
  - Numbers …



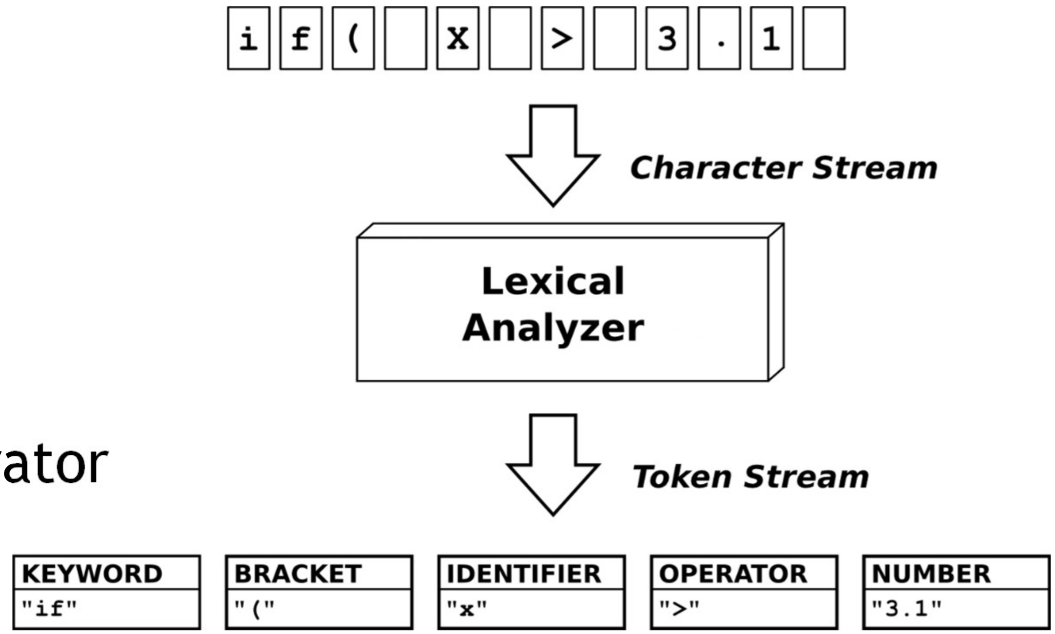
- Blanks, new lines, tabulation marks will be removed during lexical analysis.

Lexical analysis or Scanning…

- Example

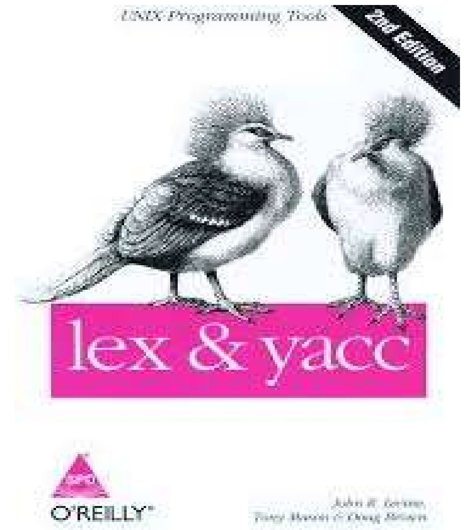
  a[index] = 4 + 2;

a	identifier
[	left bracket
index	identifier
]	right bracket
=	assignment operator
4	number
+	plus operator
2	number
;	semicolon



- A scanner may perform other operations along with the recognition of tokens.
  - It may inter identifiers into the symbol table, and
  - It may inter literals into literal table.

Lexical Analysis Tools



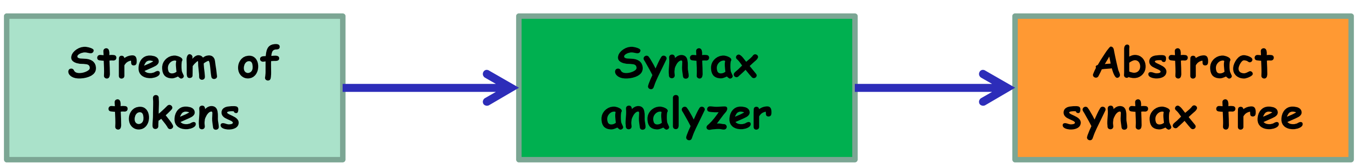
- There are tools available to assist in the designing of lexical analyzers.
  - lex - produces C source code (UNIX/linux).
  - flex - produces C source code (gnu).
  - JfLex - produces Java source code.

- We will use Lex.

JFlex
The Fast Lexical Analyser Generator for Java (TM)
© 1998-2001 by Gerwin Klein

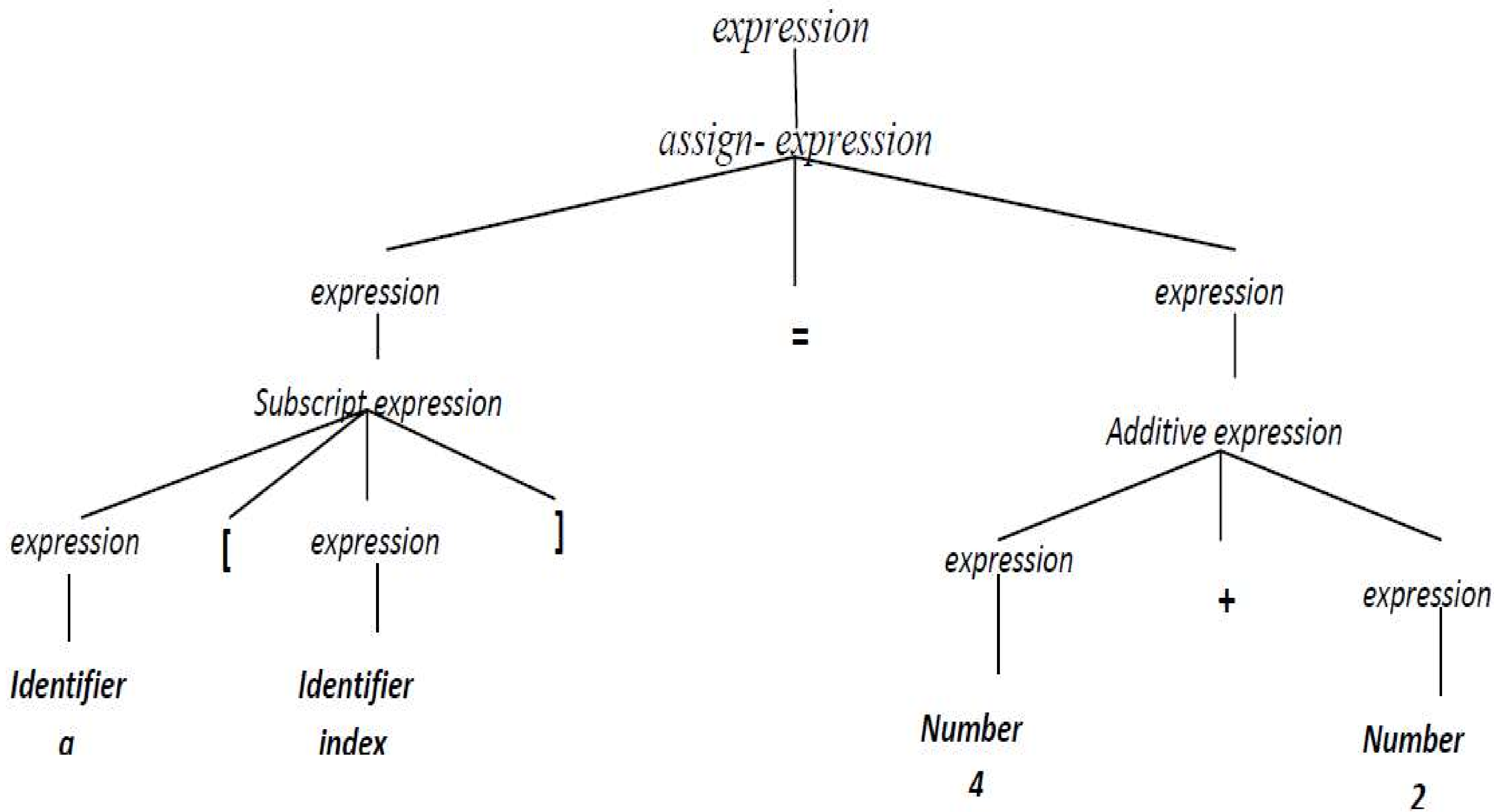
2. Syntax analysis or Parsing

- The parser receives the source code in the form of **tokens** from the scanner and performs syntax analysis.
- The results of syntax analysis are usually represented by a parse tree or a syntax tree.
  - Syntax tree → each interior node represents an operation and the children of the node represent the arguments of the operation.
- The syntactic structure of a programming language is determined by context free grammar (CFG).



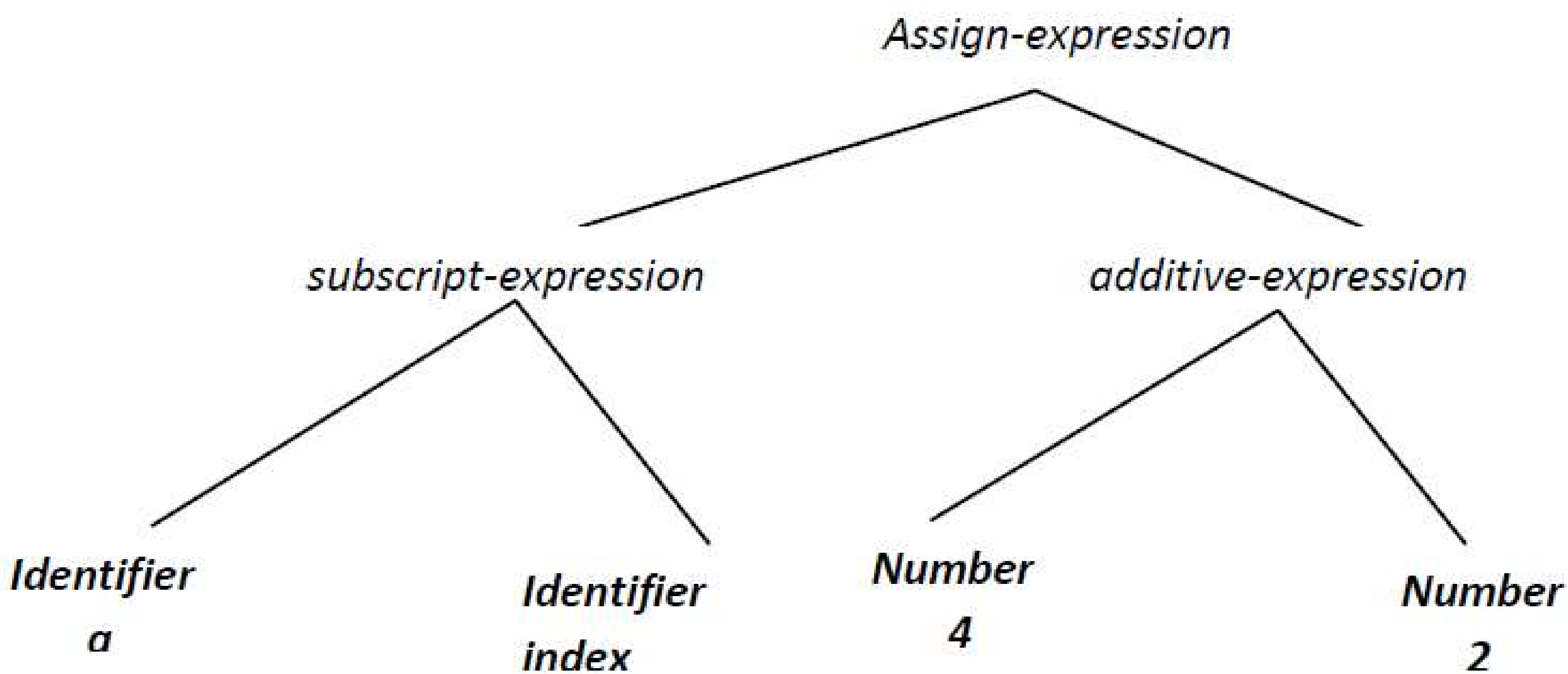
Syntax analysis or Parsing…

- Ex. Consider again the line of C code: **a[index] = 4 + 2**



Syntax analysis or Parsing…

- Sometimes syntax trees are called **abstract syntax trees**, since they represent a further abstraction from parse trees. Example is shown in the following figure.



Syntax Analysis Tools

- There are tools available to assist in the writing of parsers.
  - yacc - produces C source code (UNIX/Linux).
  - bison - produces C source code (gnu).
  - CUP - produces Java source code.

- We will use yacc.

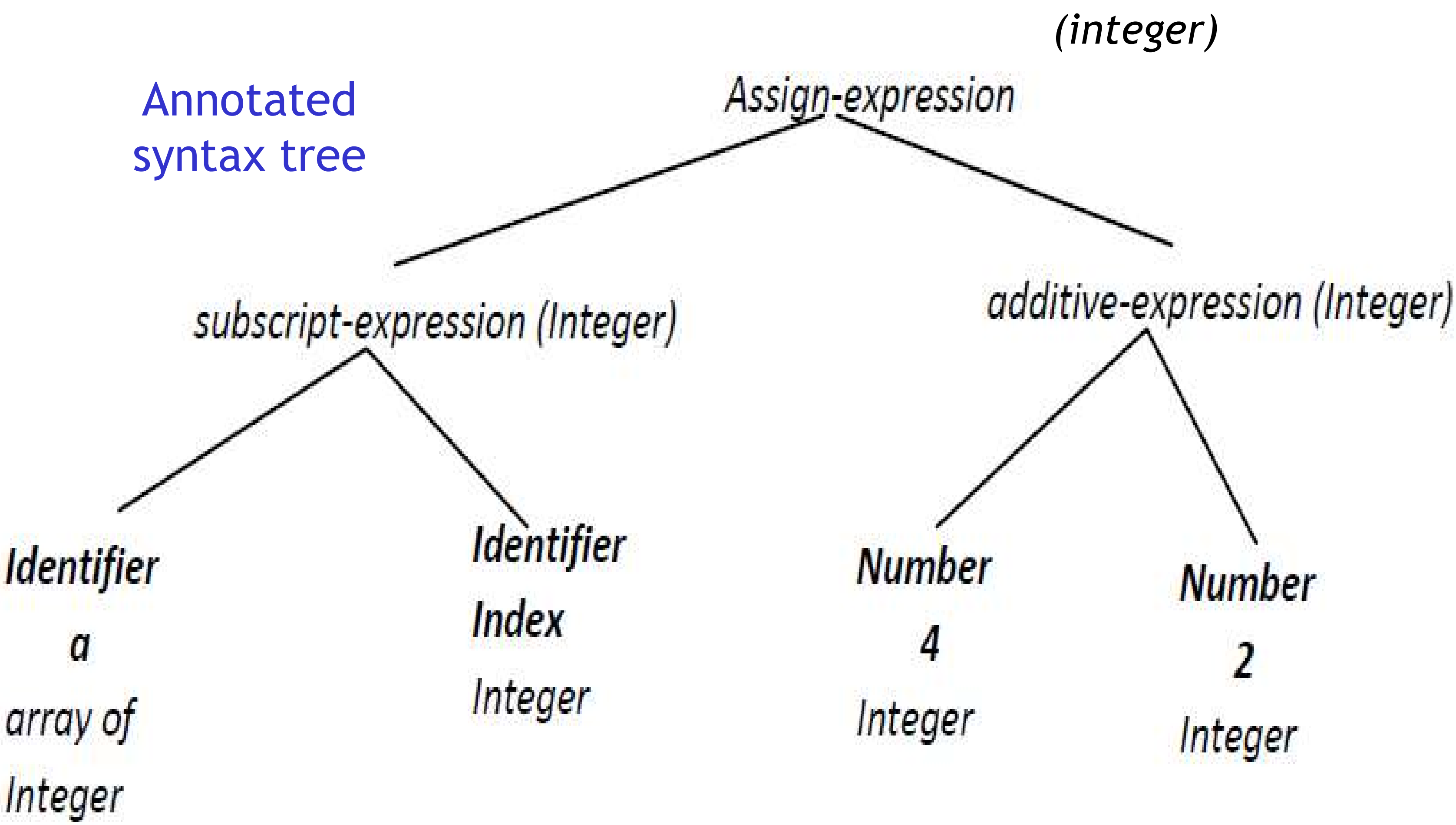
3. Semantic analysis

- The semantics of a program are its **meaning** as opposed to syntax or structure
- The semantics consist of:
  - **Runtime semantics** - behavior of program at runtime
  - **Static semantics** - checked by the compiler
- Static semantics include:
  - Declarations of variables and constants before use
  - Calling functions that exist (predefined in a library or defined by the user)
  - Passing parameters properly
  - Type checking.
- The semantic analyzer does the following:
  - Checks the static semantics of the language
  - Annotates the syntax tree with type information

Semantic analysis…

- Ex. Consider again the line of C code: **a[index] = 4 + 2**

Annotated syntax tree



Synthesis of the target program

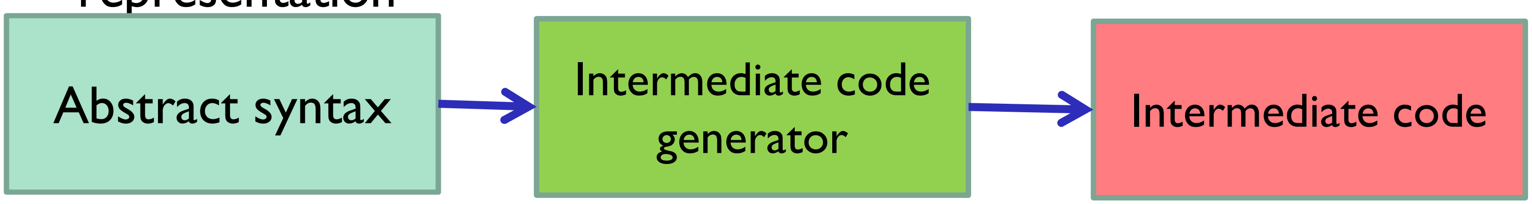
- Intermediate code generator
- Intermediate code optimizer
- The target code generator
- The target code optimizer

Code Improvement

- **Code improvement techniques** can be applied to:
  - Intermediate code – independent of the target machine
  - Target code – dependent on the target machine
- Intermediate code improvement include:
  - Constant folding
  - Elimination of common sub-expressions
  - Improving loops
  - Improving function calls
- Target code improvement include:
  - Allocation and use of registers
  - Selection of better (faster) instructions and addressing modes

Intermediate code generator

- Comes after syntax and semantic analysis
- Separates the compiler front end from its backend
- Intermediate representation should have 2 important properties:
  - Should be easy to produce
  - Should be easy to translate into the target program
- Intermediate representation can have a variety of forms:
  - Three-address code, P-code for an abstract machine, Tree or DAG representation



- Three address code for the original C expression a[index]=4+2 is:

$$t_1=2$$
$$t_2 = 4 + t_1$$
$$a[index] = t_2$$

IC optimizer

- An IC optimizer reviews the code, looking for ways to reduce:
  - the number of operations and
  - the memory requirements.
- A program may be optimized for speed or for size.
- This phase changes the IC so that the code generator produces a faster and less memory consuming program.
- The optimized code does the same thing as the original (non-optimized) code but with
  - less cost in terms of CPU time and memory space.

IC optimizer...

- There are several techniques of optimizing code and they will be discussed in the forthcoming chapters.

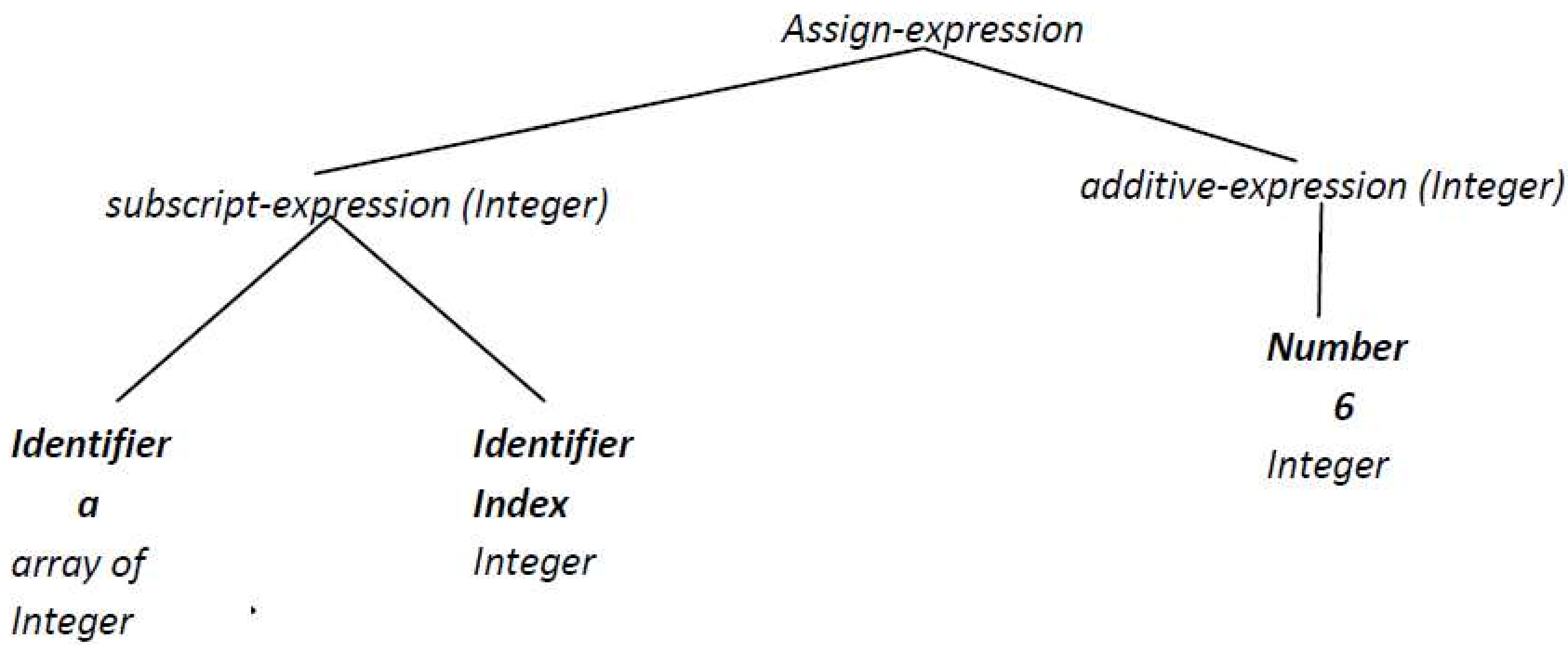
- Ex. Unnecessary lines of code in loops (i.e. code that could be executed outside of the loop) are moved out of the loop.

```
 for(i=1; i<10; i++){
     x = y+1;
     z = x+i; }

     x = y+1;
     for(i=1; i<10, i++)
     z = x+i;
```

IC optimizer…

- In our previous example, we have included an opportunity for source level optimization; namely, the expression **4 + 2** can be recomputed by the compiler to the result **6**(This particular optimization is called **constant folding**).
- This optimization can be performed directly on the syntax tree as shown below.



IC optimizer...

- Many optimizations can be performed directly on the tree.
- However, in a number of cases, it is easier to optimize a linearized form of the tree that is closer to assembly code.
- A standard choice is **Three-address code**, so called because it contains the addresses of up to **three locations in memory**.
- In our example, three address code for the original C expression might look like this:

$$t_1=2$$

$$\mathbf{t_2 = 4 + t_1}$$

$$\mathbf{a[index] = t_2}$$

- Now the optimizer would improve this code in two steps, first computing the result of the addition

$$t = 4+2$$

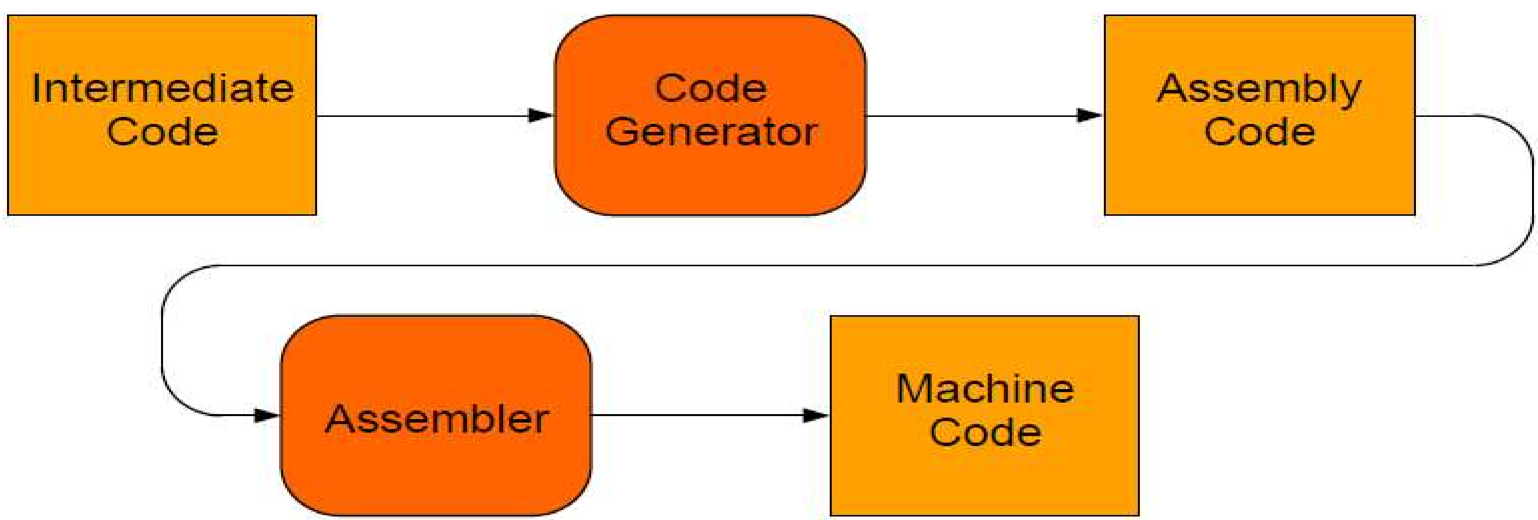
$$a[index] = t$$

- And then replacing t by its value to get the three-address statement

$$a[index] = 6$$

Code generator

- The machine code generator receives the (optimized) intermediate code, and then it produces either:
  - Machine code for a specific machine, or
  - Assembly code for a specific machine and assembler.
- **Code generator**
  - Selects appropriate machine instructions
  - Allocates memory locations for variables
  - Allocates registers for intermediate computations



Code generator…

- The code generator takes the IR code and generates code for the target machine.
- Here we will write target code in assembly language: 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
```

- &a –the address of **a** (the base address of the array)
- *R1-indirect registers addressing (the last instruction stores the value 6 to the address contained in R1)

The target code optimizer

- In this phase, the compiler attempts to improve the target code generated by the code generator.
- Such improvement includes:
  - Choosing addressing modes to improve performance
  - Replacing slow instruction by faster ones
  - Eliminating redundant or unnecessary operations
- In the sample target code given, use a shift instruction to replace the multiplication in the second instruction.
- Another is to use a more powerful addressing mode, such as indexed addressing to perform the array store.
- With these two optimizations, our target code becomes:

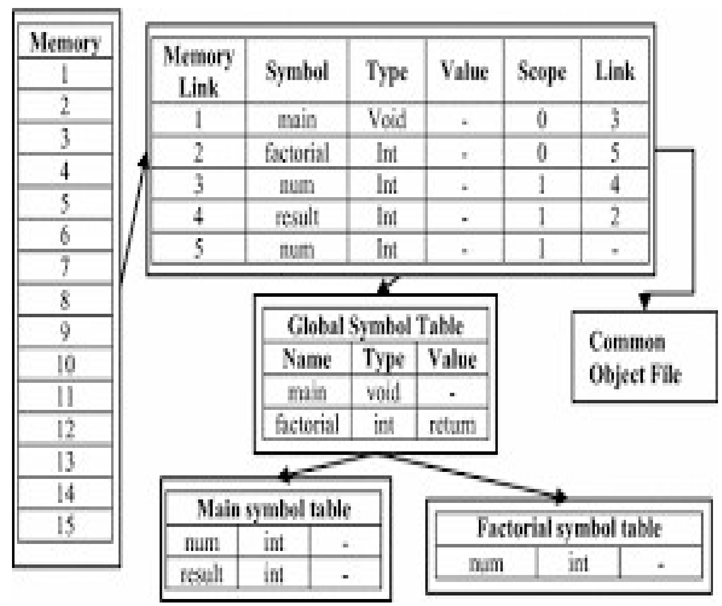
```
MOV R0, index        ;; value of index -> R0
SHL R0               ;; double value in R0
MOV &a [R0], 6       ;; constant 6 -> address a + R0
```

Major Data and Structures in a Compiler

- Token
  - Represented by an integer value or an enumeration literal
  - Sometimes, it is necessary to preserve the string of characters that was scanned
  - For example, name of an identifiers or value of a literal
- Syntax Tree
  - Constructed as a pointer-based structure
  - Dynamically allocated as parsing proceeds
  - Nodes have fields containing information collected by the parser and semantic analyzer

Major Data and Structures in a Compiler…

- ❒ **Symbol Table**
  - Keeps information associated with all kinds of tokens:
    - Identifiers, numbers, variables, functions, parameters, types, fields, etc.
  - Tokens are entered by the scanner and parser
  - Semantic analyzer adds type information and other attributes
  - Code generation and optimization phases use the information in the symbol table



- ❒ **Performance Issues**
  - Insertion, deletion, and search operations need to be efficient because they are frequent
  - More than one symbol table may be used

Major Data and Structures in a Compiler…

Literal Table

- Stores constant values and string literals in a program.
- One literal table applies globally to the entire program.
- Used by the code generator to:
  - Assign addresses for literals.
- Avoids the replication of constants and strings.
- Quick insertion and lookup are essential.

Literal Table

Name of the Literal	Value of the Literal	Address of Usage of Symbol	Address of Defination of Symbol
'4'	4	0003,0007	0014
'5'	5	0011	0015

Types of Compilers

- Based on the **number of passes**:
- Single-pass compilers: These compilers read the source code and generate the target code in a single pass.
  - They are generally faster than multi-pass compilers but can be less efficient in terms of code quality. Pascal
- Two-pass compilers: These compilers read the source code twice.
  - The first pass gathers information about the code, and the second pass generates the target code.
  - They can produce more efficient code than single-pass compilers but are slower. C, C++, and Java
- Multi-pass compilers: These compilers read the source code multiple times.
  - They can produce the most efficient code but are also the slowest. Ada

Types of Compilers…

- Based on the **target platform**:
- Cross compilers: These compilers run on one platform but generate code for a different platform.
  - They are used to develop software for embedded systems or other platforms where a compiler is not available.
  - SDCC (Small Device C Compiler)
- Native compilers: These compilers run on a platform and generate code for the same platform.
  - They are the most common type of compiler.
  - C# program on Windows using the .NET framework's

Types of Compilers…

- Based on the **type of code generated**

- Just-in-time (JIT) compilers: These compilers compile code at runtime, rather than ahead of time.
  - They are used in Java and other languages where code is often downloaded from the internet.
  - Java, JavaScript, .NET languages (C#, VB.NET, F#), Python, Ruby, PHP

- Ahead-of-Time (AOT) Compiler: It converts the entire source code into machine code before the program runs.
  - C and C++, Fortran, Go, Rust, Swift, Java (with GraalVM), .NET (with .NET Native)

- Hybrid Approaches: Modern runtime environments often use a combination of AOT and JIT techniques to optimize performance.

Other types of compilers

- Source-to-source compilers: These compilers translate code from one high-level language to another high-level language.
  - Haxe (pronounced "hex") is an open-source high-level multiplatform programming language and compiler that can produce code for many different target platforms from a single code-base.
  - Haxe can be compiler to C++, C#, Java, Lua, Python

- Decompilers: These programs attempt to reverse the compilation process, converting machine code back into a higher-level language.
  - Java (e.g., JAD) and .NET (e.g., .NET Reflector)

Key Considerations When Choosing a Compiler

- Compilation speed: How quickly can the compiler translate source code into executable code?
- Code efficiency: How well does the compiler optimize the generated code for performance?
- Error detection: How effectively does the compiler identify and report errors in the source code?
- Portability: Can the compiler generate code for multiple platforms?
- Features: Does the compiler offer advanced features, such as debugging support or code optimization tools?

Popular Compiler Examples by Language

- C/C++: GCC, G++ (GNU Compiler Collection), Clang/LLVM, Microsoft Visual C++ Compiler
- Java: OpenJDK, Oracle JDK, IBM SDK
- C#: Microsoft Visual Basic Compiler (VBC)
- Python: While Python is primarily interpreted, there are also compilers like MyPy (for type checking) and Cython (for performance optimization)
- Go: Go Compiler (gc)
- JavaScript: V8 (used in Chrome and Node.js), SpiderMonkey (used in Firefox)

Compiler construction tools

- Various tools are used in the construction of the various parts of a compiler.
- **Scanner generators**
  - Ex. Lex, flex, JLex
  - These tools generate a scanner /lexical analyzer/ if given a regular expression.
- **Parser Generators**
  - Ex. Yacc, Bison, CUP
  - These tools produce a parser /syntax analyzer/ if given a Context Free Grammar (CFG) that describes the syntax of the source language.

Compiler construction tools…

- **Syntax directed translation engines**
  - Ex. Cornell Synthesizer Generator
  - It produces a collection of routines that walk the parse tree and execute some tasks.
- **Automatic code generators**
  - Take a collection of rules that define the translation of the IC to target code and produce a code generator.
- **Data-Flow Analysis Engines**
  - It supports code optimization by analyzing the flow of values throughout different parts of the program.
- **Compiler Construction Toolkits:**
  - It provides integrated routines to facilitate the construction of various compiler components.

This completes our brief description of the compiler Design.

Thank u

?