بسم اللّه الرّحمٰن الرّحيم

Programs Related to a Compiler

- These programs include
  - Interpreters
  - Assemblers
  - Linkers
  - Loaders
  - Preprocessors
  - Editors
  - Debuggers
  - Profilers
  - Project Managers

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

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

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

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

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

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

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

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

Project Managers

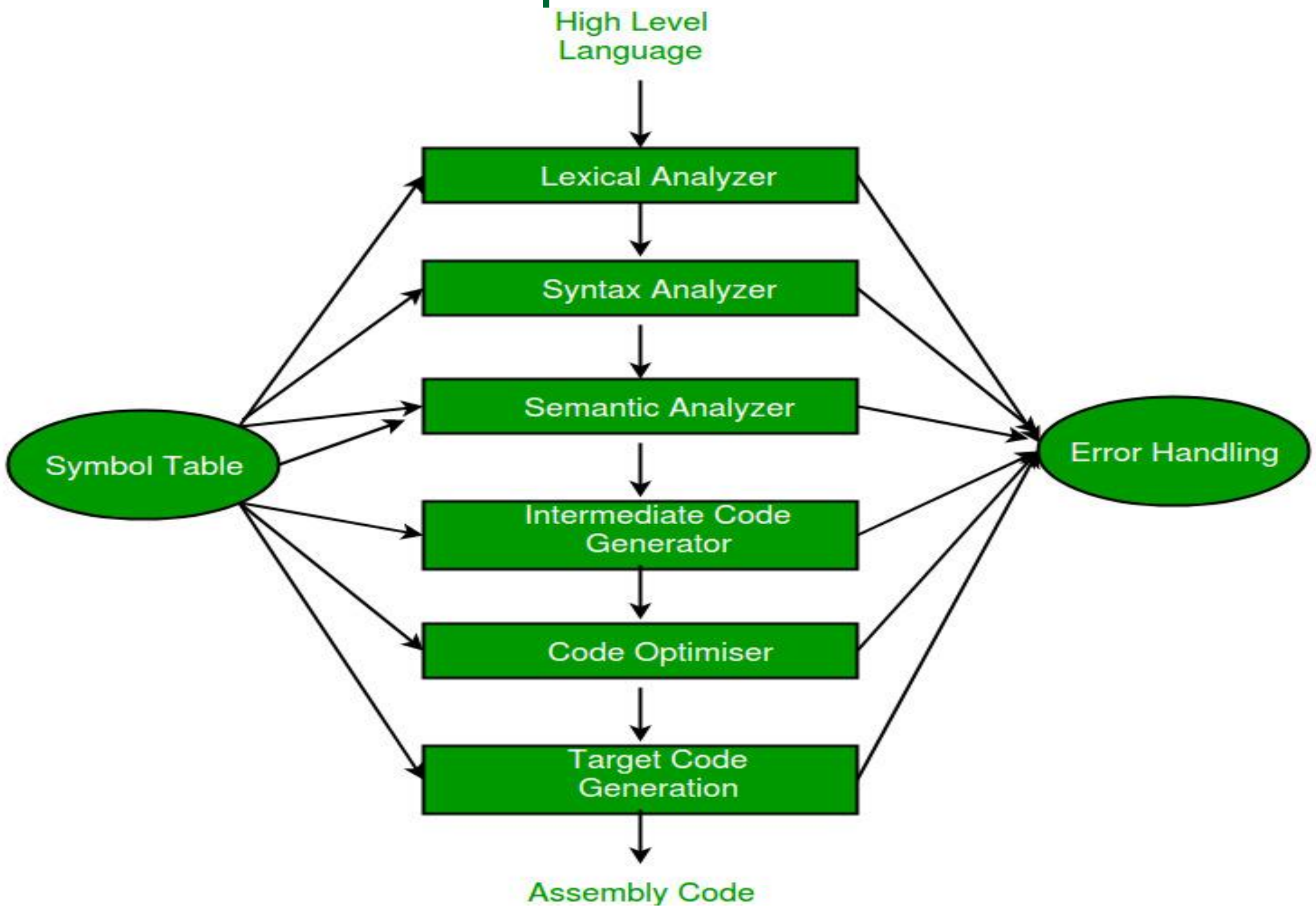
- Coordinate the files being worked on by different people, maintain coherent version of a program
- Language-independent or bundled together with a compiler
- Two popular project manager programs on Unix system
    - Sccs (Source code control system)
    - Rcs (revision control system)

Compiler Components

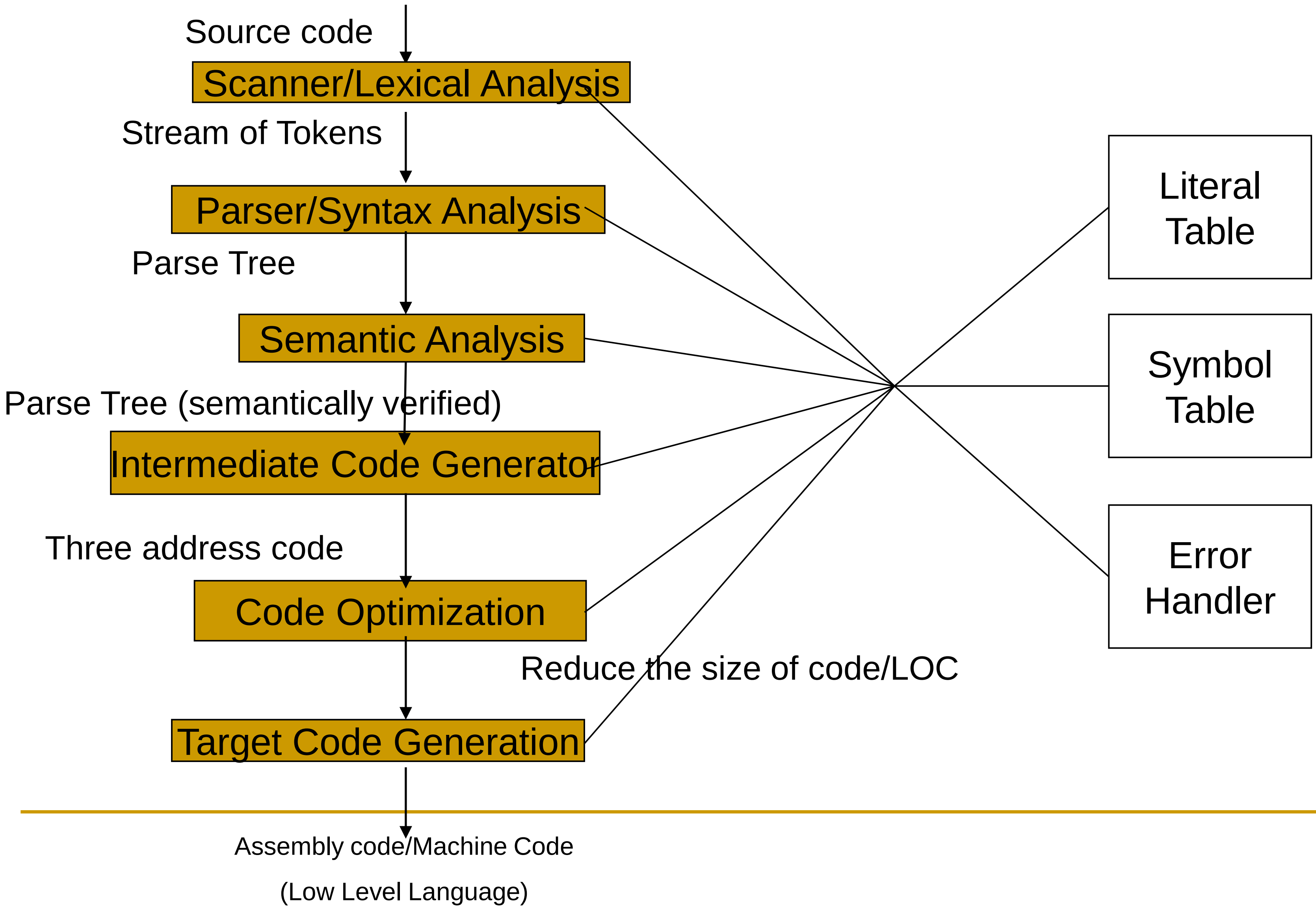
- Six Components
  1. Lexical Analyzer
  2. Syntax Analyzer
  3. Semantic Analyzer
  4. Intermediate Code Generator
  5. Code Optimizer
  6. Target Code Generator

- Three Auxiliary Components
  1. Literal Table
  2. Symbol Table
  3. Error Handler

Phases of Compiler



The Compilation Process



Notes

Lexical Analyzer:

Divide into tokens,

Removes comments, white spaces/blank spaces.

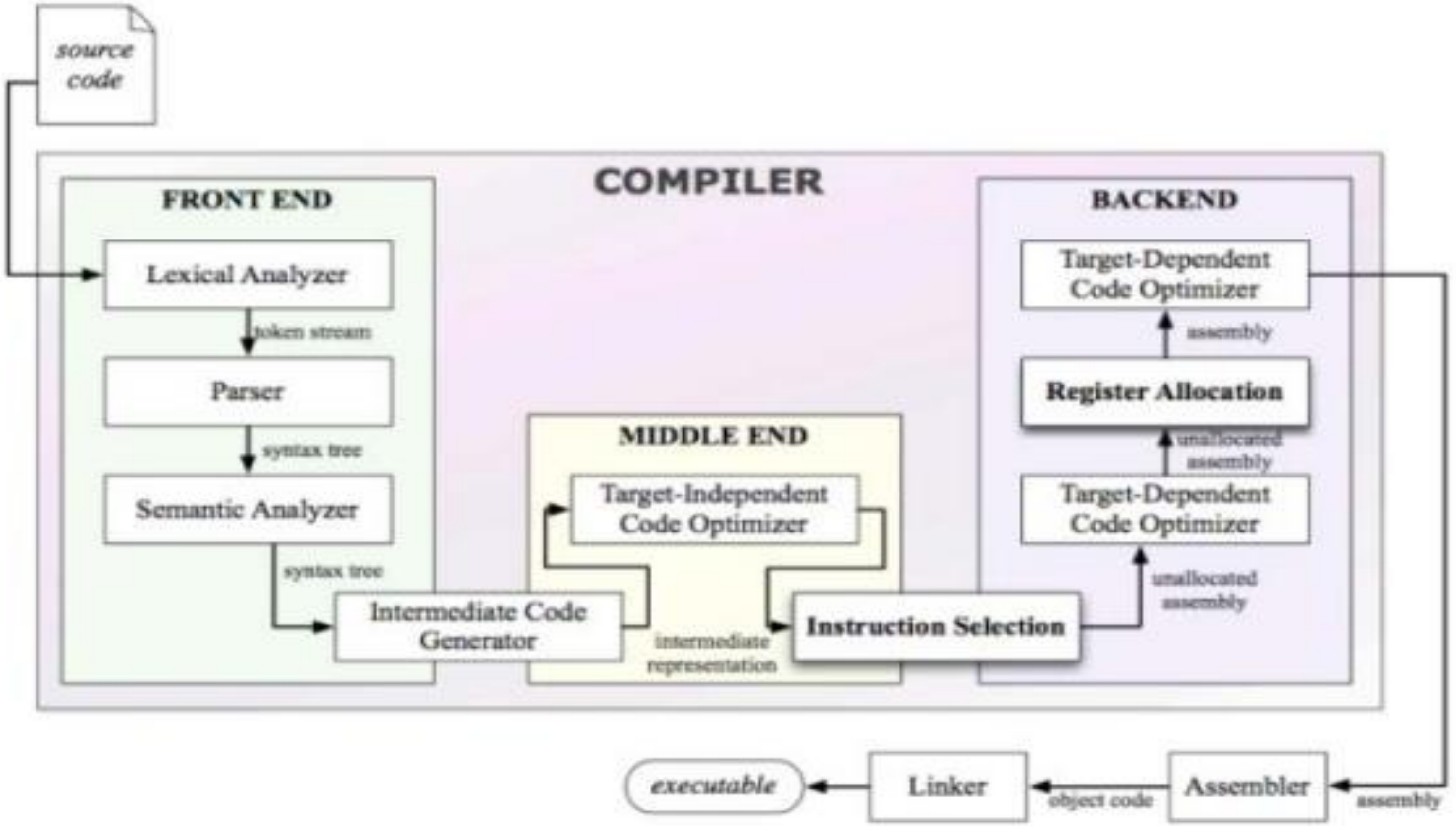
Also called tokenizer / scanner.

FA is used to create tokens i.e., to convert characters into tokens.

Symbol table stores detail about all data types being used in source code while converting source code to tokens.

Semantic analysis check logical kind of error only at compile time, whether the variables are properly declared or not, static scoping and dynamic scoping etc.

Syntax, logical, either variable and their types defined or not.


source code
COMPILER
FRONT END
Lexical Analyzer
token stream
Parser
syntax tree
Semantic Analyzer
syntax tree
Intermediate Code Generator
MIDDLE END
Target-Independent Code Optimizer
intermediate representation
BACKEND
Instruction Selection
unallocated assembly
Target-Dependent Code Optimizer
unallocated assembly
Register Allocation
assembly
Target-Dependent Code Optimizer
assembly
Assembler
object code
Linker
executable

1. The Scanner

- Lexical analysis: it converts 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

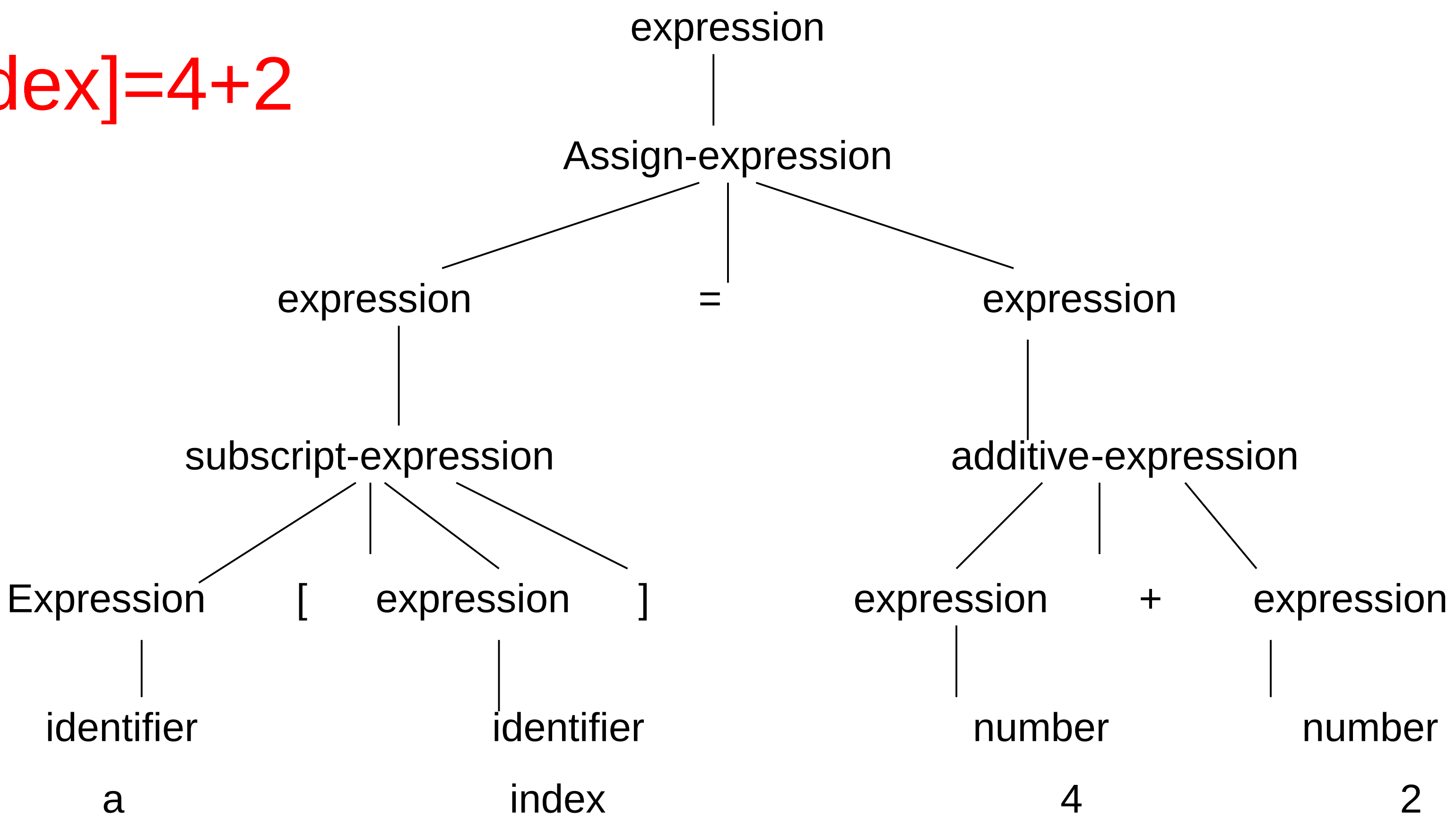
- Other operations: it may enter literals into the literal table

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)

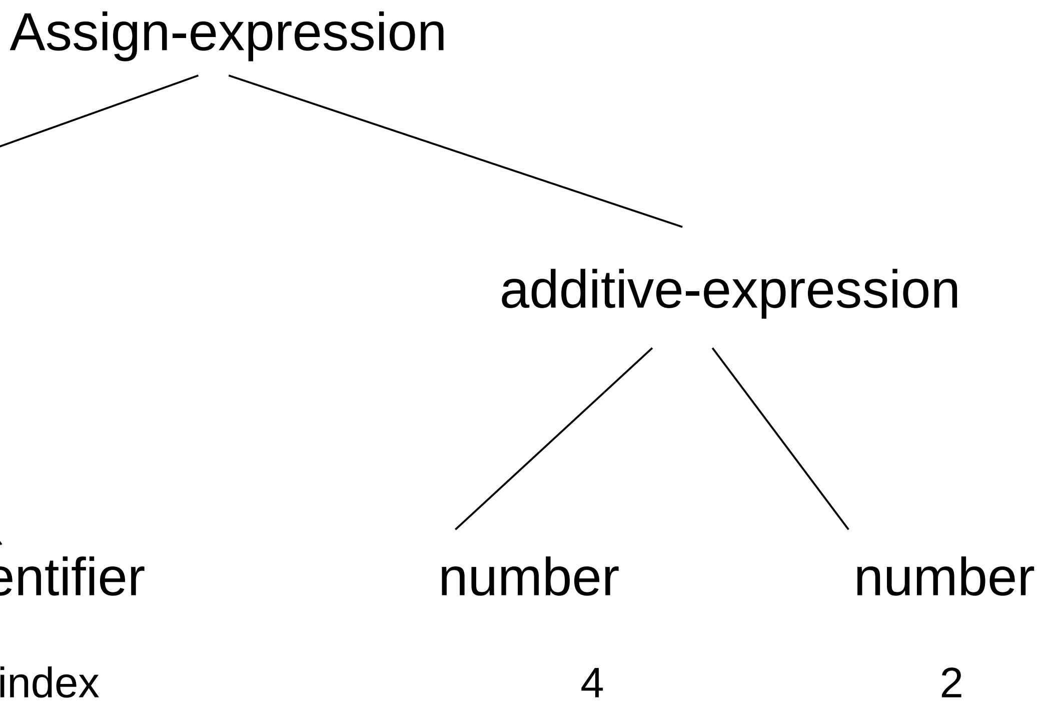
The Parse Tree

a[index]=4+2



The Syntax Tree

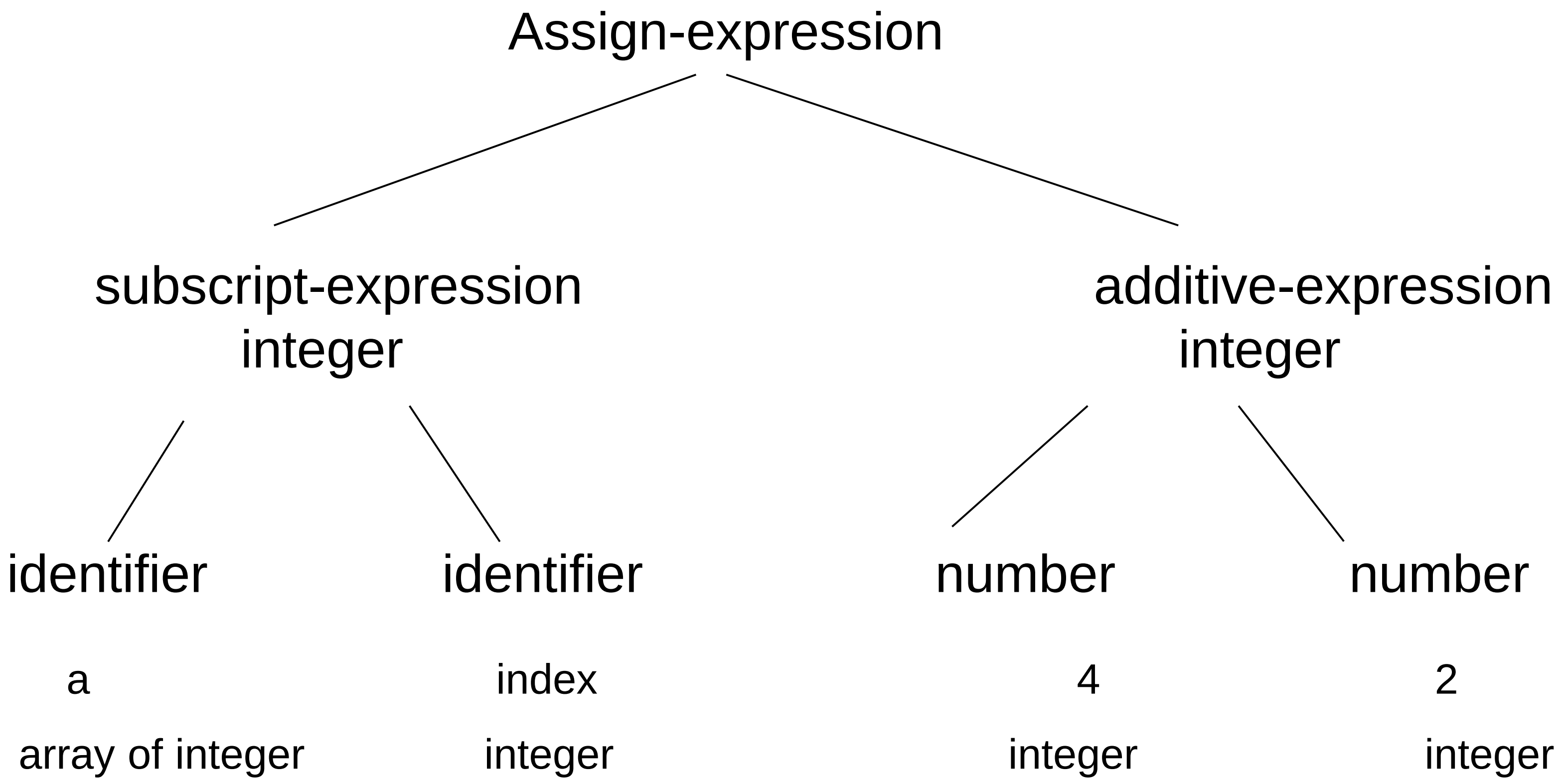
a[index]=4+2



3. The Semantic Analyzer

- The semantics of a program are its “meaning”, as opposed to its syntax, or structure
  - It determines some of its running time behaviors prior to execution
- Semantic analyzer keeps track of identifiers, their types and expressions.
- 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

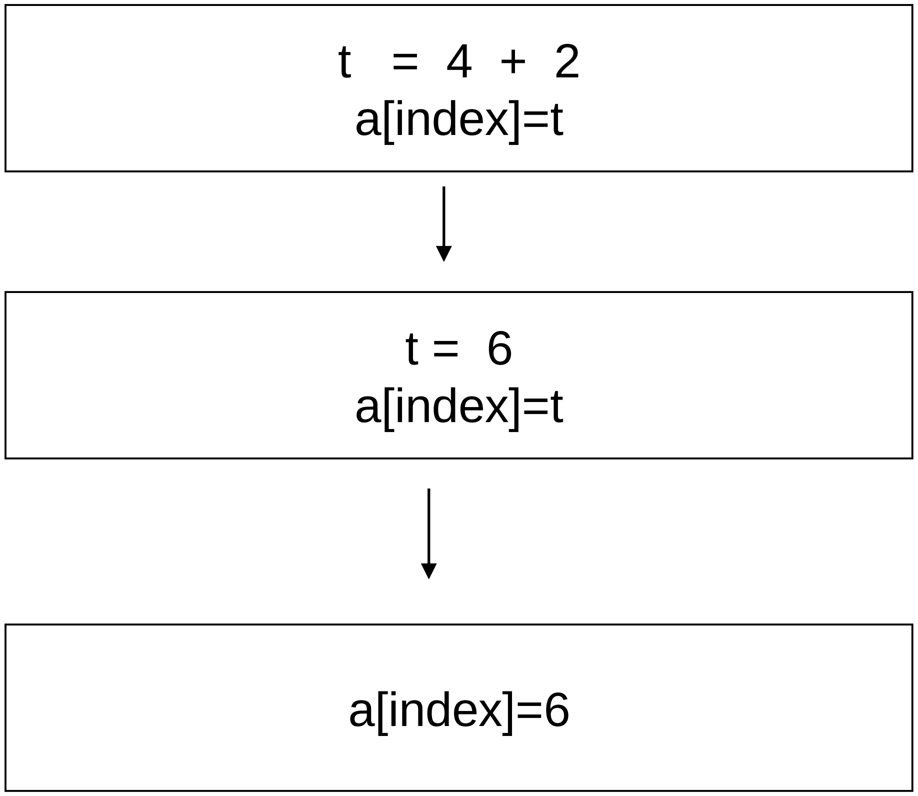


4. Intermediate Code Generator

- An example: a[index]=4+2
  - Using intermediate code: three-address code
  - Three address code is a sort of intermediate code that is simple to create and convert to machine code. It can only define an expression with three addresses and one operator.
  - It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler.

```
T1= 4+2
T2= a[index]
T3: T2=T1
```

5. Optimization on Intermediate Code

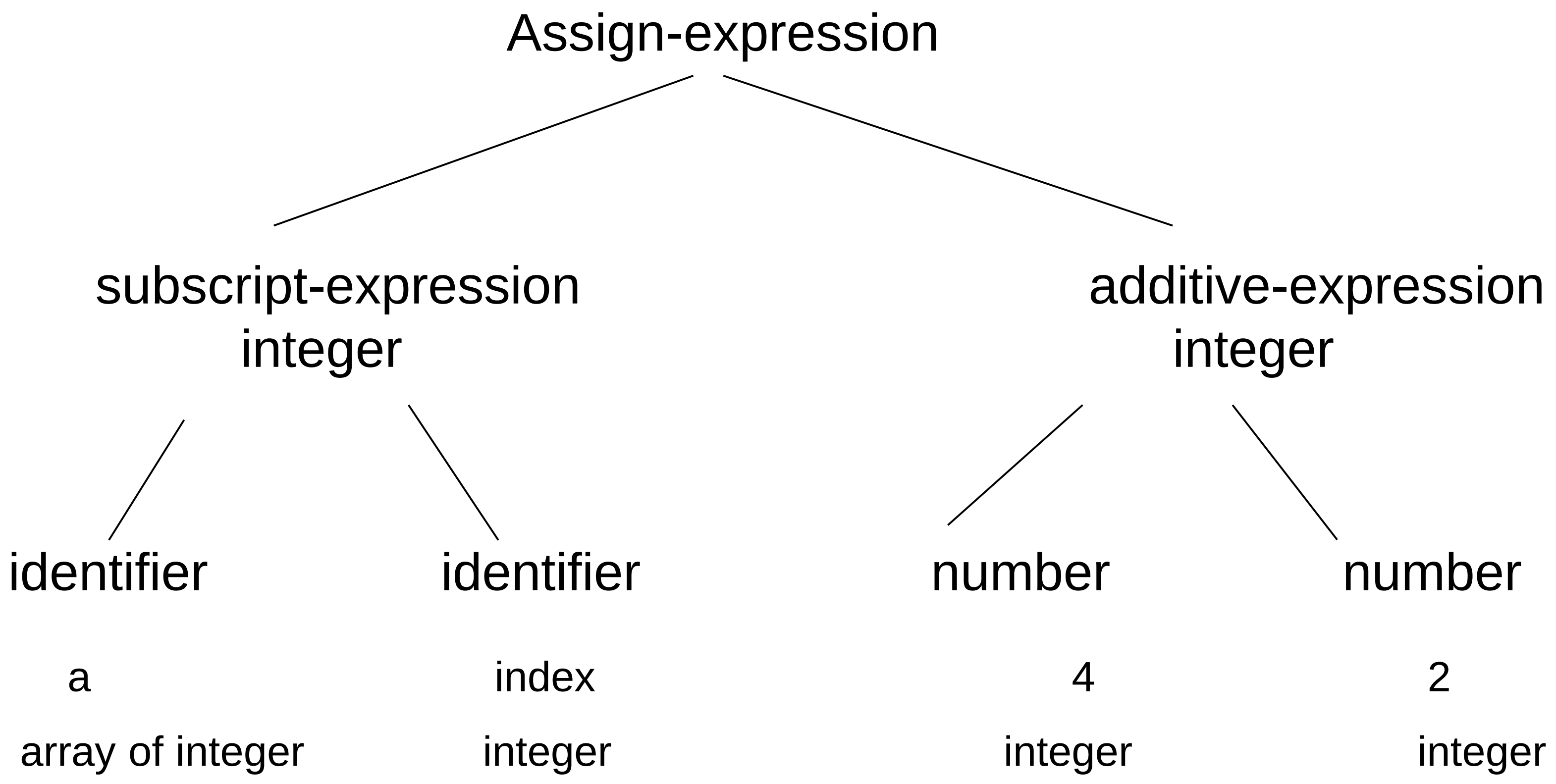


```
t  =  4  +  2
a[index]=t
```

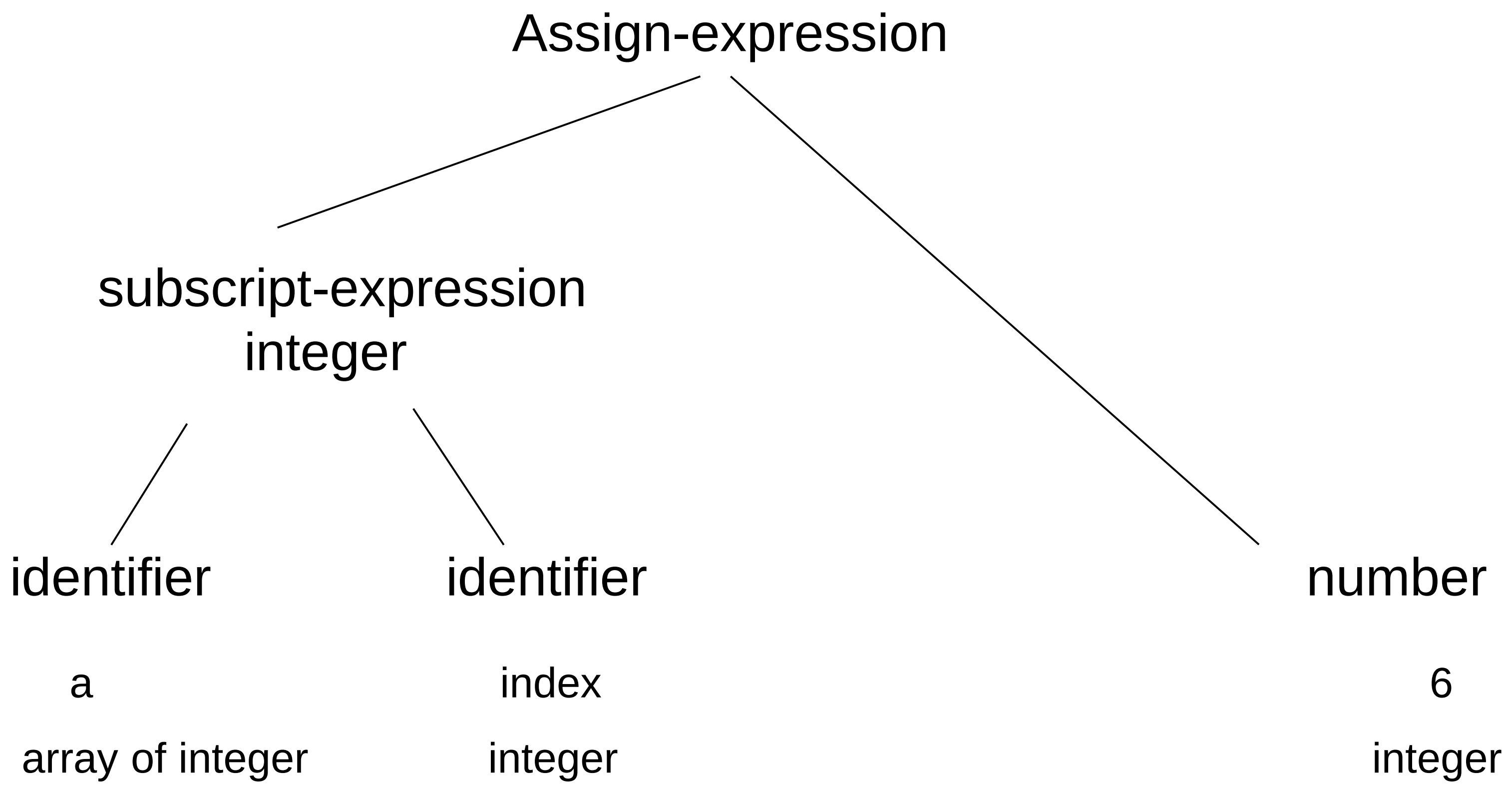
```
t =  6
a[index]=t
```

```
a[index]=6
```

Annotated Tree Without Optimization



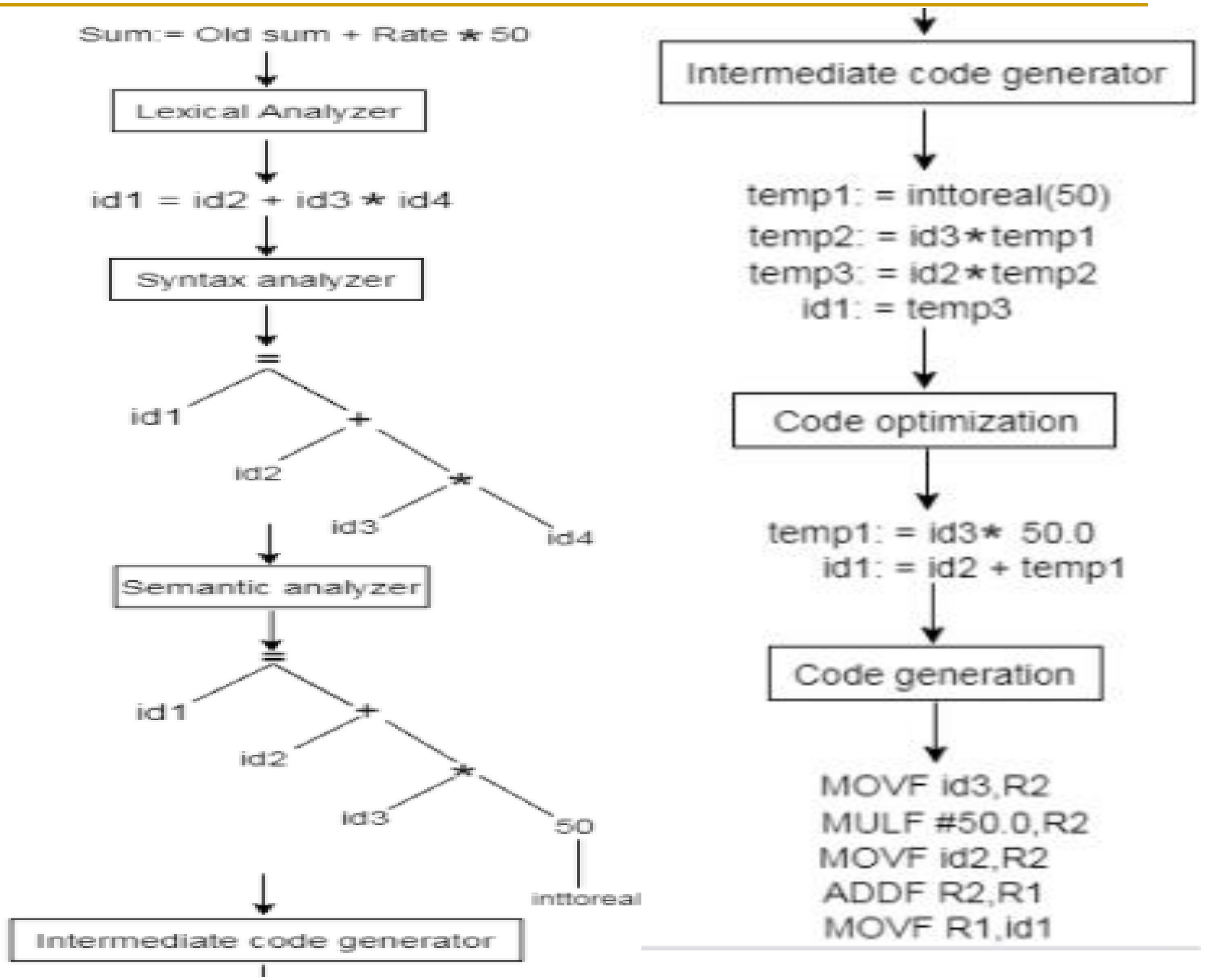
Optimized Annotated Tree



6. The Target Code Generator

- 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 that exist on target machines
  - How to make representation of data i.e. how many bytes and words will be allocated to a give type of a variable

Example:



```
temp1: = inttoreal(50)
temp2: = id3*temp1
temp3: = id2*temp2
  id1: = temp3
```

```
temp1: = id3*  50.0
  id1: = id2 + temp1
```

```
MOVF id3,R2
MULF #50.0,R2
MOVF id2,R2
ADDF R2,R1
MOVF R1,id1
```

Principal Data Structures Used in Compilers

- 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
  - A single global variable or an array of tokens
- THE SYNTAX TREE
  - A standard pointer-based structure generated by parser
  - Each node represents information collected by parser, which maybe dynamically allocated in symbol table
  - The node requires different attributes depending on kind of language structure, which may be represented as variable record

Principal Data Structures Used in Compilers (Continue…)

- 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, reduce size of program
  - Quick insertion and lookup are essential
  - If a literal is used more than once (as they often are in a program), we still want to store it only once

Principal Data Structures Used in Compilers (Continue…)

- 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

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
    - Must handle a wide range of errors
    - Must handle multiple errors.
    - Must not get stuck.
    - Must not get into an infinite loop

Error Handling (Continue…)

- Exception handling
  - Generate extra code to perform suitable runtime tests to guarantee all such errors to cause an appropriate event during execution

Summary

Any Questions?