MCA 18 302
PRINCIPLES OF COMPILERS

MODULE 1

1. **Introduction to compiling**
    1. Definition of compiler, translator, interpreter
    2. Analysis of the source program
    3. The phases of a compiler
    4. Compiler construction tools
2. **Programming language basics**
3. **Lexical analysis**
    1. Role of lexical analyzer
    2. Input buffering
    3. Specification of tokens
    4. Recognition of tokens using finite automata
    5. Regular expressions and finite automata
    6. From NFA to DFA
    7. Regular expression to an NFA

Introduction to compiling

The phases of a compiler

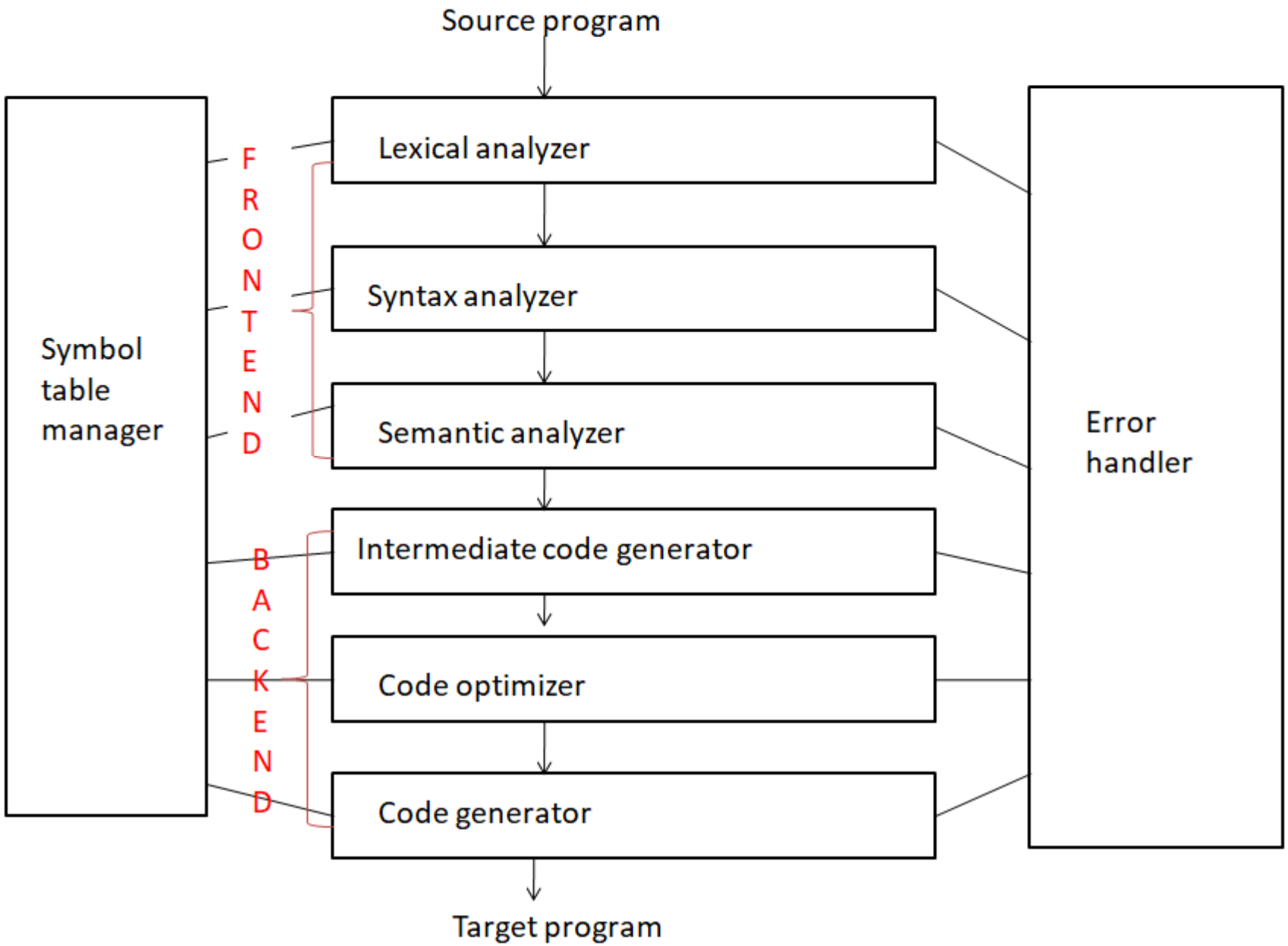
- The compiler phases are:

I. Analysis phase(Front end)

1. Lexical analyzer
2. Syntax analyzer
3. Semantic analyzer

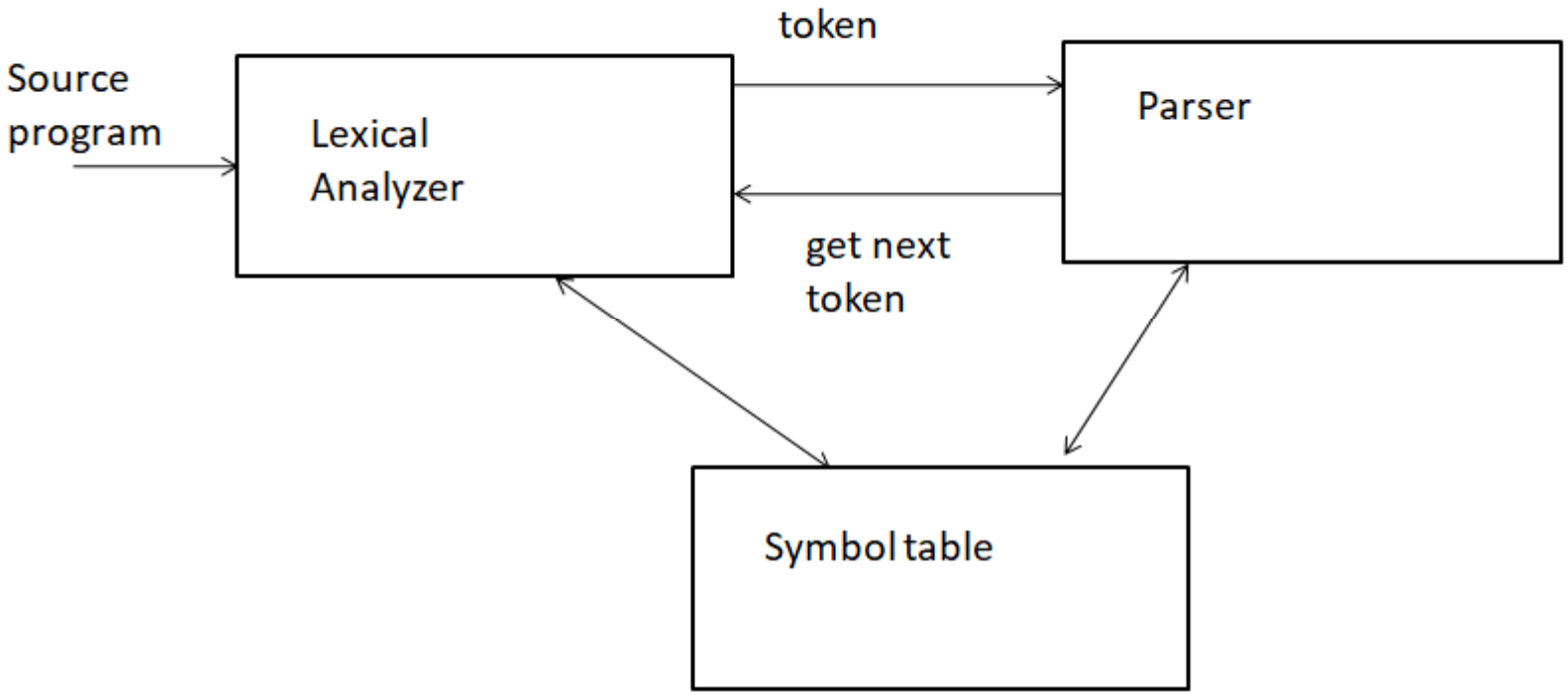
II. Synthesis phase(Back end)

4. Intermediate code generator
5. Code optimizer
6. Code generator


Source program
Lexical analyzer
Syntax analyzer
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Target program
Symbol table manager
Error handler
FRONTEND
BACKEND

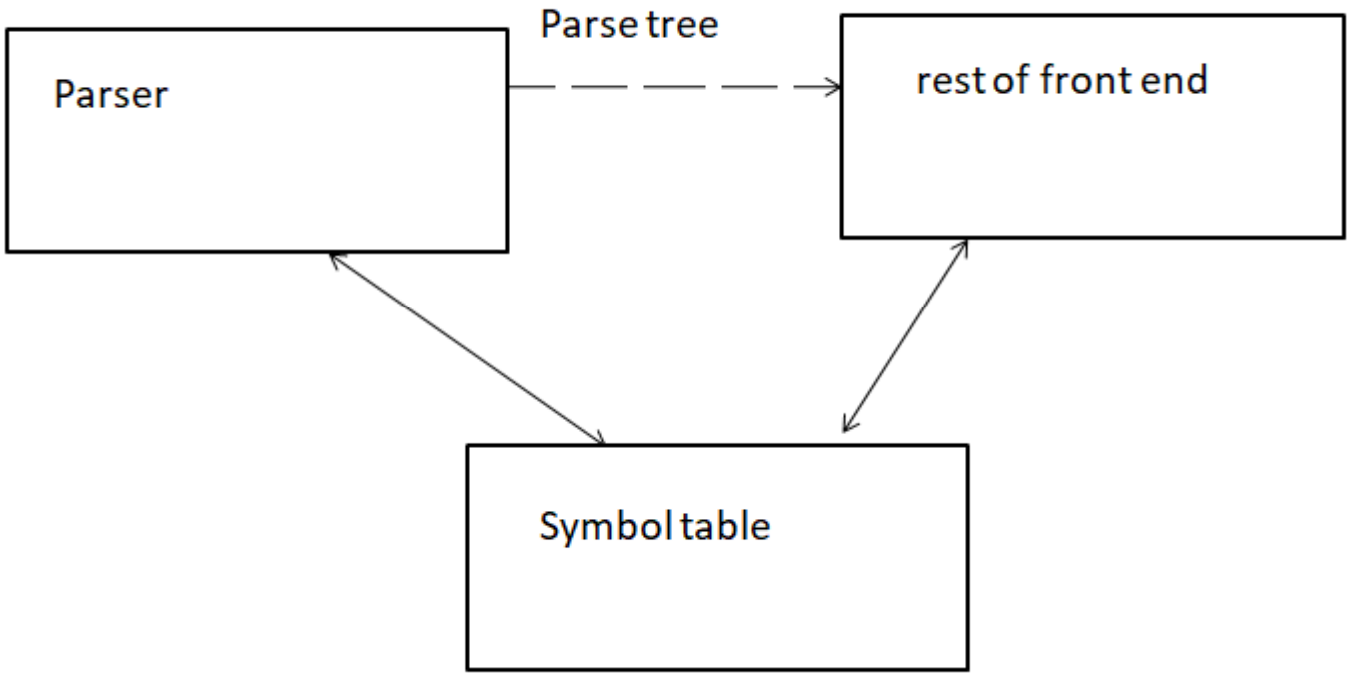
1. Lexical analyzer:

- Lexical analysis is the process of converting a sequence of characters into a sequence of tokens.
- A program or function which performs lexical analysis is called a lexical analyzer or lexer or scanner.
- The lexical analysis is the first phase of the compiler. It's main task is to read the input characters and produces as output a sequence of tokens, that the parser uses for syntax analysis.


Source
program
Lexical
Analyzer
token
get next
token
Parser
Symbol table

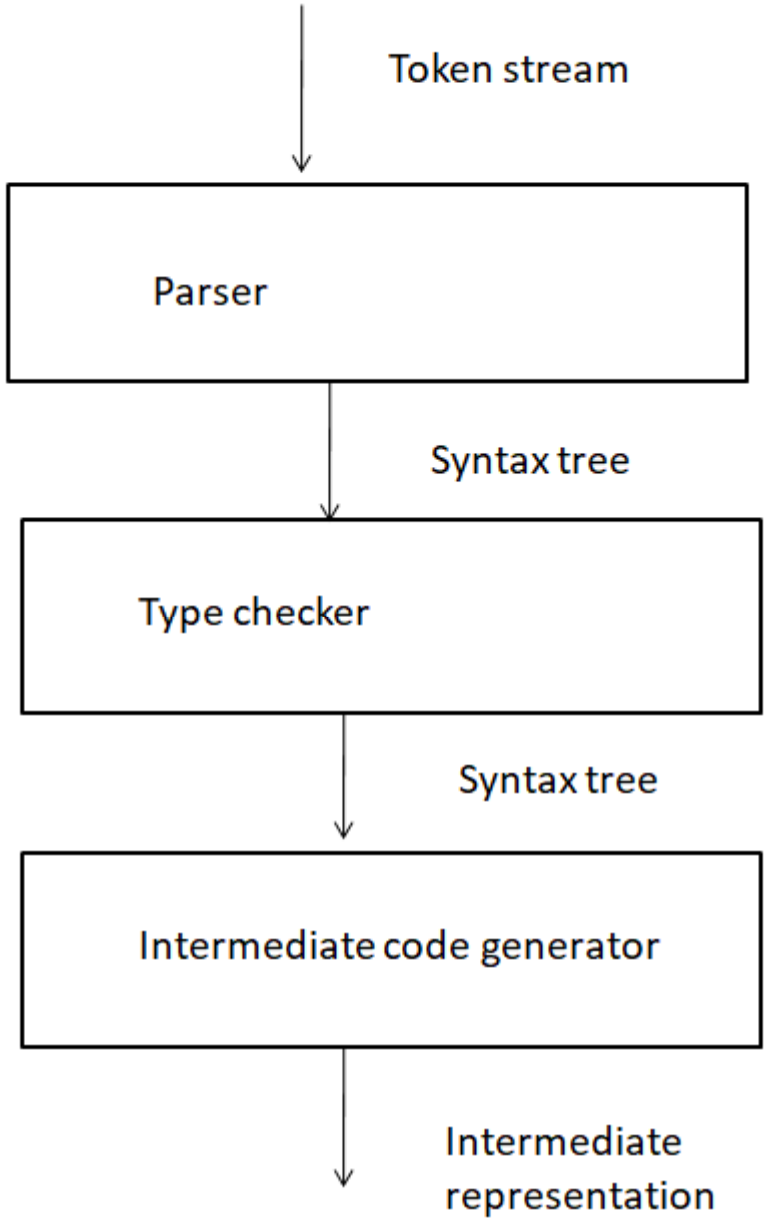
2. Syntax analyzer:

- The parser obtains a string of token from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.
- It should also discover from commonly occurring error so that it can continue the remainder of its input.


Parse tree
Parser
rest of front end
Symbol table

3. Semantic analyzer:

- The semantic analysis phase checks the source program for semantic errors.
- An important component of semantic analysis is type checking.
- The input of the semantic analysis is the syntax tree and output to also a syntax tree including type checking information.


Token stream
Parser
Syntax tree
Type checker
Syntax tree
Intermediate code generator
Intermediate
representation

4. Intermediate code generator:

- After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program.
- This intermediate representation should have two important properties:
  1. It should be easy to produce.
  2. Easy to translate into the target program.
- Three kinds of intermediate representation:
  a) Syntax tree
  b) Postfix notation
  c) Three addressed code

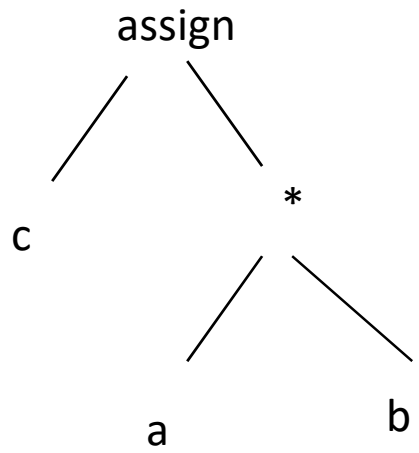
a) Syntax tree:

- ❖ It depicts the natural hierarchical structure of a source program.

b) Postfix notation:

- ❖ It is a lineraized representation of a syntax tree.

Example :c:=a*b



Syntax tree

cab*assign

Postfix notation

c) Three addressed code

- It is a sequence of statement.
- General form: x:=y op z
  - x,y,z are variable names, constants or compiler generated temporaries.
  - op stands for any operator such as fixed or floating point arithmetic operator or a logical operator on boolean valued data.

5. Code optimizer:

- The code optimization phase attempts to improve the intermediate code, so that faster running machine code will result.
- The code optimization is a program transformation technique, which tries to improve the intermediate code.
- The process of code optimization involves:
  1. Eliminating the unwanted code lines
  2. Rearranging the statements of the code

6. Code generation:

- The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or target code.
- Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine.
- A code generator is expected to generate the correct code.

Symbol table:

- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that quickly.

- When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table.
- Example : int a,b;

  float c;

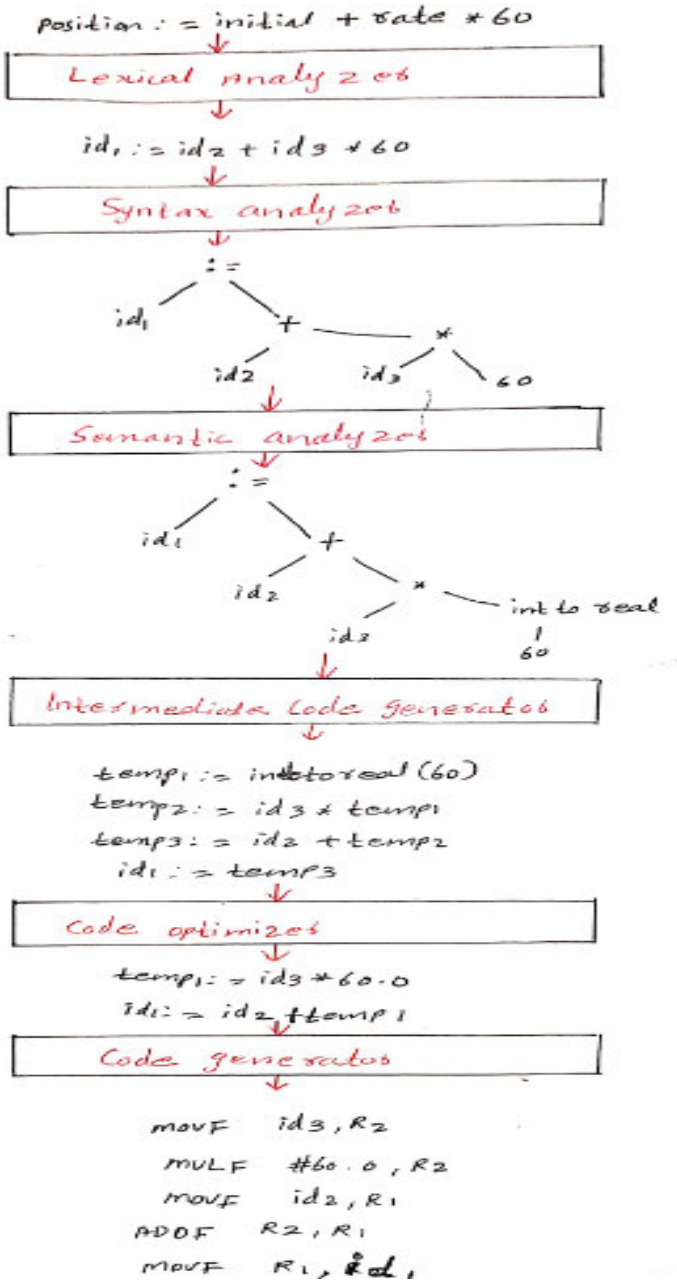
  Here a,b, and c are seen by the lexical analyzer.
- The remaining phases enter information about identifiers into the symbol table and then use this information in various ways.

Error detection and Reporting:

- Each phase can encounter errors.However,after detecting an error, a phase must some how deal with that errors, so that compilation can proceed ,allowing further errors in the source program to be detected.
- The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.
- The lexical phase can detect errors where the characters remaining in the input do not form any token of the language.

- Errors where the token stream violates the structure rules(syntax) of the language are determined by the syntax analysis phase.
- During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved.

Example:


position := initial + rate * 60
Lexical analyzer
id1 := id2 + id3 * 60
Syntax analyzer
:=
id1
+
id2
*
id3
60
Semantic analyzer
:=
id1
+
id2
*
id3
int to real
60
Intermediate Code Generator
temp1 := inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
Code optimizer
temp1 := id3 * 60.0
id1 := id2 + temp1
Code generator
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1