# COMPILER DESIGN

## UNIT - I

## B.Tech (CSE) -VI SEM

# Syllabus

**UNIT-I: Introduction**: Language Processors, the Structure of a Compiler. Lexical Analysis: The Role of the Lexical Analyzer, Specification of Tokens, Recognition of Tokens and the Lexical-Analyzer Generator Lex.

**UNIT-II: Syntax Analysis**: Definition of CFG, Lexical Versus Syntactic Analysis, Writing a Grammar- Elimination of Left Recursion, Left Factoring. Top Down Parsing: Recursive Descent Parsing, First and Follow, LL(1) Grammars, Non recursive Predictive Parsing, Error Recovery in Predictive Parsing.

**UNIT-III: Bottom-Up Parsing**: Bottom Up Parser Classification, Reductions, Handle Pruning, Shift-Reducing, Conflicts During Shift Reduce Parsing. Introduction to LR Parsing: Difference between LR and LL Parsers, Why LR Parsers?, Items and the LR(0) automaton, The LR-Parsing Algorithm, Constructing SLR Parsing Tables

**UNIT-IV: More powerful LR parsers**: construction of CLR (1), LALR Parsing tables, Comparison of all Bottom Up approaches. Semantic Analysis: Syntax Directed Definitions, Evaluation Orders for SDD's, Applications of SDT.

**UNIT-V: Intermediate Code Generation**: Variants of Syntax Trees, Three-Address Code, Control Flow, Back-patching. Run-Time Environments: Storage Organization, Stack Allocation of Space, Heap Management.

**UNIT-VI: Code Generation**: Basic Blocks and Flow Graphs, Optimization of Basic Blocks, Peephole Optimization, Register Allocation and Assignment. Machine-Independent optimizations: The Principal Sources of Optimizations, Introduction to Data-Flow Analysis. .

**TEXT BOOKS**: 1. Compilers, Principles Techniques and Tools- Alfred V Aho, Monica S Lam, Ravi Sethi, Jeffrey D. Ullman,2nded, Pearson,2007

**REFERENCE BOOKS**: 1. Principles of compiler design, V. Raghavan, 2nd ed, TMH, 2011 2. Compiler Design, K. Muneeswaran, Oxford

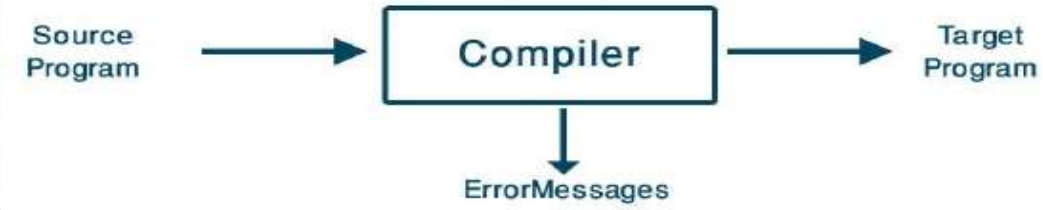
| S.No. | COURSE OUTCOMES (After completion of the course, The Learner is able to) | KNOWLEDGE LEVEL |
|---|---|---|
| CO1 | Describe the compilation process and lexical analyzer | K2 |
| CO2 | Construct top down parsing Techniques | K3 |
| CO3 | Construct bottom up parsing techniques | K3 |
| CO4 | Construct syntax directed translation | K3 |
| CO5 | Produce intermediate code generation process and run time environments | K3 |
| CO6 | Explain the code generation process. | K2 |

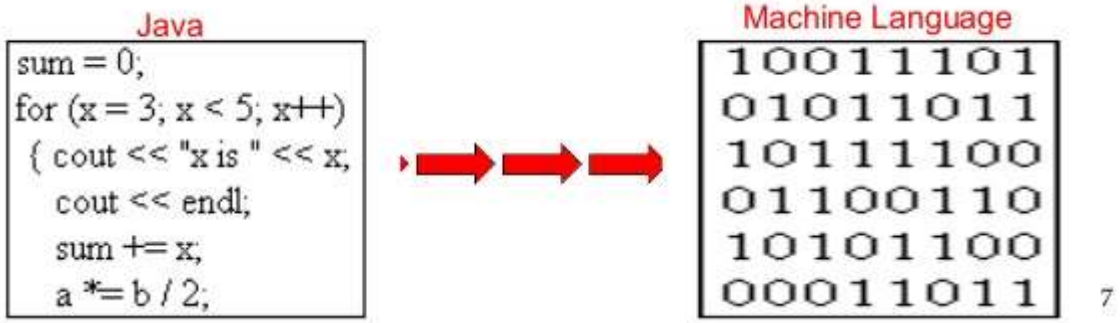
## UNIT-I: Introduction:

- Language Processors
- The Structure of a Compiler
- Lexical Analysis: The Role of the Lexical Analyzer
- Specification of Tokens
- Recognition of Tokens
- The Lexical-Analyzer Generator Lex.

# Compiler :

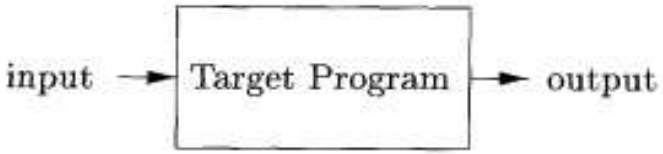
A compiler is a program that can read a program in one language i.e. source language and translate it into an equivalent program in another language i.e. target language



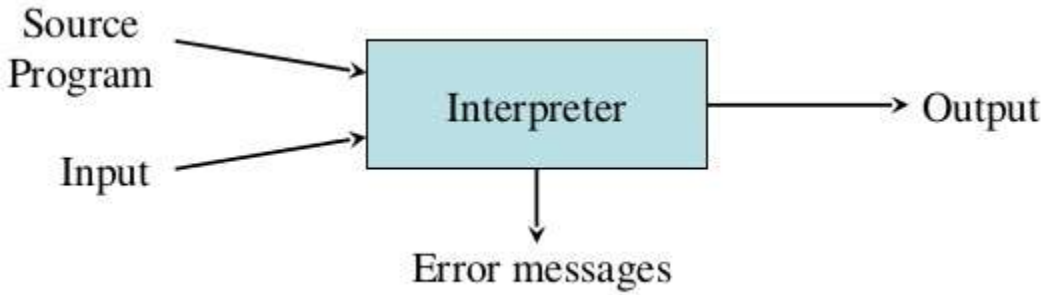

7

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs



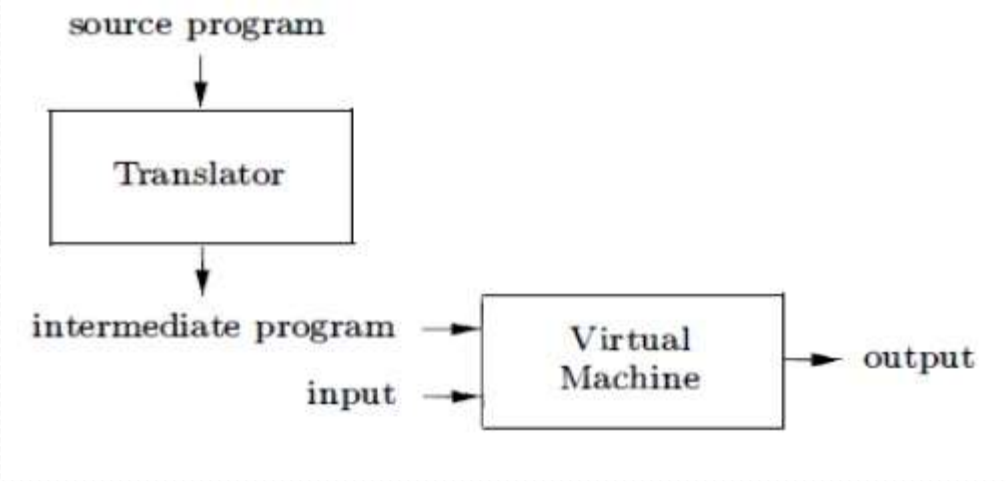

## Interpreter :

An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user



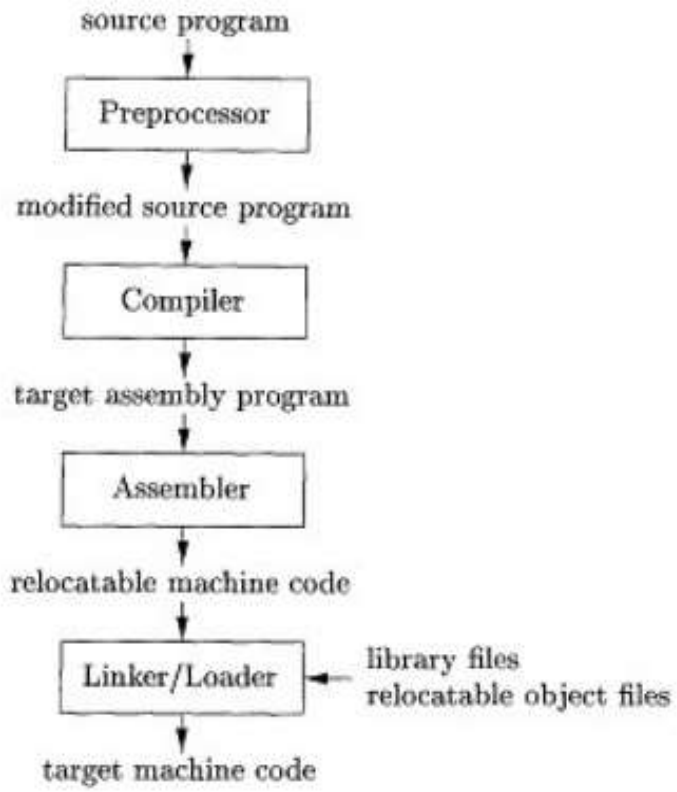
For example Java language processors combine compilation and interpretation A Java source program may first be compiled into an intermediate form called bytecodes. The bytecodes are then interpreted by a virtual machine.

A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network. In order to achieve faster processing of inputs to outputs.



# Language Processors:

In addition to a compiler, several other programs may be required to create an executable target program as shown in Fig



**Preprocessor :**

The preprocessor may also expand shorthands, called macros, into source language statements. The modified source program is then fed to a compiler.

**Compiler :**

The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug.

**Assembler :**

The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output.

**Linkers and Loaders :**

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.

The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file.

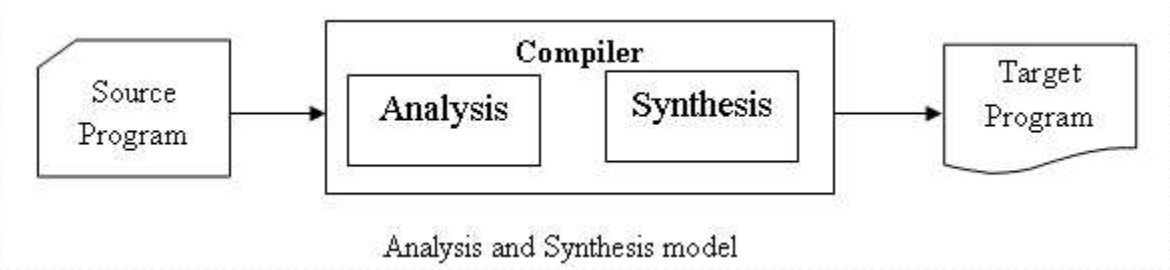
The *loader* then puts together all of the executable object files into memory for execution.

## Structure of a compiler :

There are two major parts of a compiler:

**Analysis**

**Synthesis**




Analysis and Synthesis model

In **analysis phase**, an intermediate representation is created from the given source program.

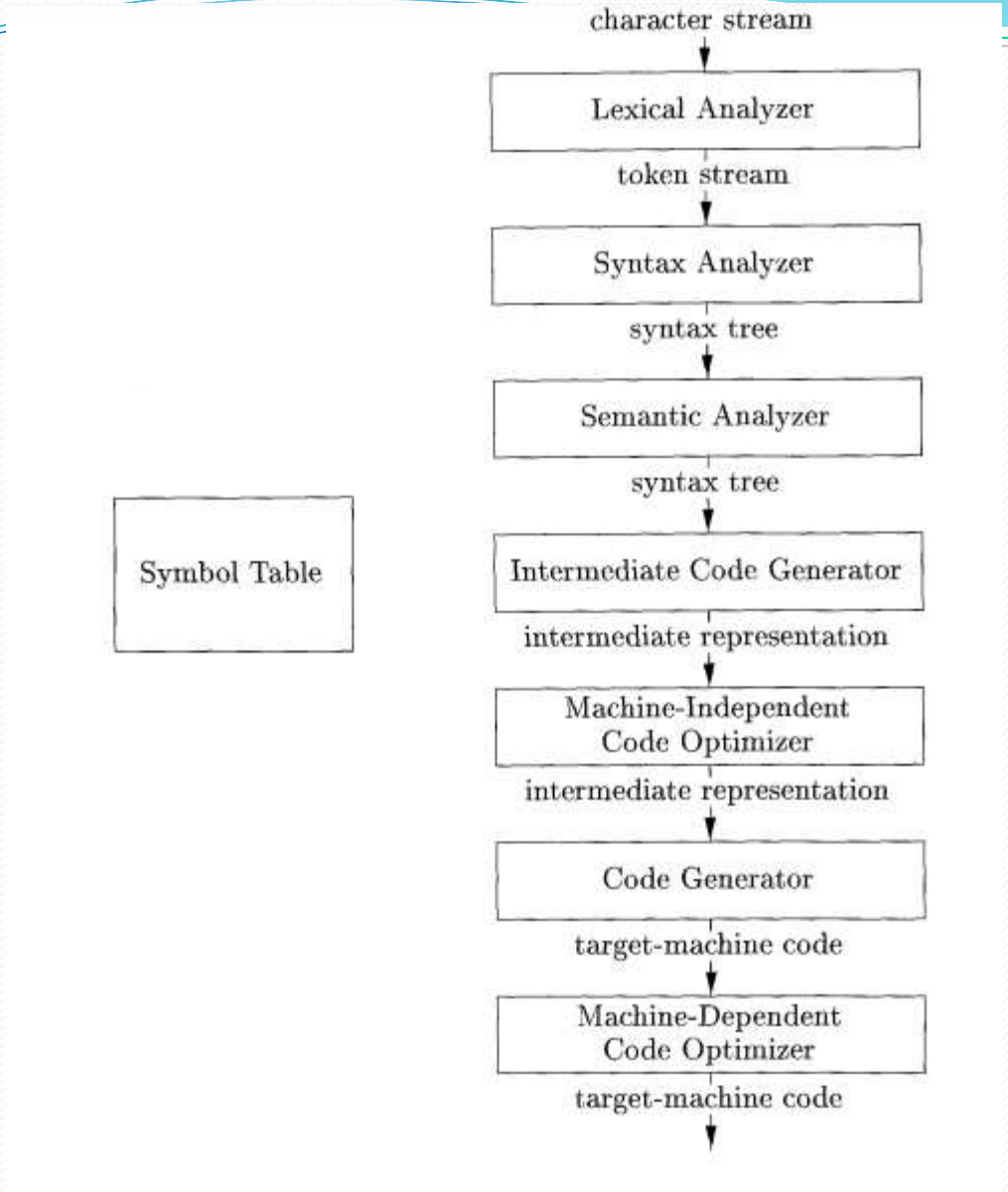
*Lexical Analyzer ,Syntax Analyzer and Semantic Analyzer are the parts of this phase.*

In **synthesis phase**, the equivalent target program is created from this intermediate representation.

*Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.*

**Phases of a compiler:**

- Compiler consists of 6 phases
- Each phase transforms the source program from one representation into another representation.
- They communicate with error handlers.
- They communicate with the symbol table.


character stream
Lexical Analyzer
token stream
Syntax Analyzer
syntax tree
Semantic Analyzer
syntax tree
Symbol Table
Intermediate Code Generator
intermediate representation
Machine-Independent
Code Optimizer
intermediate representation
Code Generator
target-machine code
Machine-Dependent
Code Optimizer
target-machine code


## Lexical Analysis :

- Lexical analyzer phase is the first phase of compilation process.

- Lexical Analyzer reads the stream of characters making up the source program ang group the characters into meaningful sequence s called Lexeme

- For each lexeme, the lexical analyzer produces a token of the form that it passes on to the subsequent phase, syntax analysis

<token-name, attribute-value>

Token-name: an abstract symbol is used during syntax analysis, an
attribute-value: points to an entry in the symbol table for this token

- Puts information about identifiers into the symbol table.

- **Example:** position =initial + rate * 60

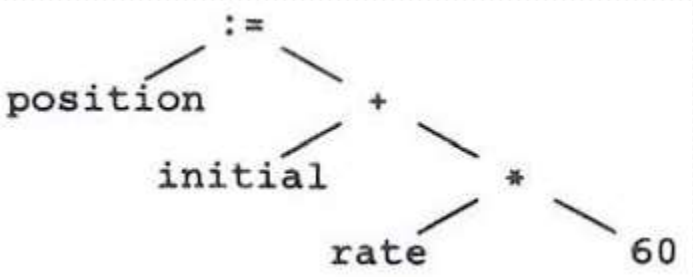
$$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$$

**Syntax analysis :**

- Syntax analysis is the second phase of compilation process.

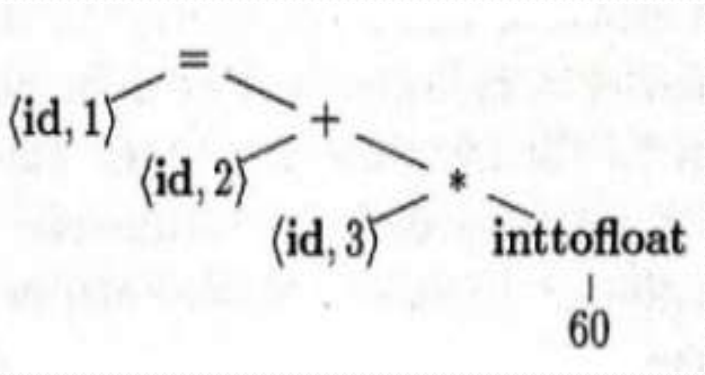
- It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

- A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation



**Semantic analysis :**

- Semantic analysis is the third phase of compilation process.
- It checks whether the parse tree follows the rules of language.
- Semantic analyzer keeps track of identifiers, their types and expressions.
- The output of semantic analysis phase is the annotated tree syntax.



**Intermediate Code Generation :**

- In the intermediate code generation, compiler generates the source code into the intermediate code.

- Intermediate code is generated between the high-level language and the machine language.

- The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```


## Code Optimization :

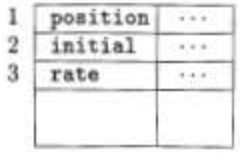
- Code optimization is used to improve the intermediate code so that the output of the program could run faster and take less space.
- It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

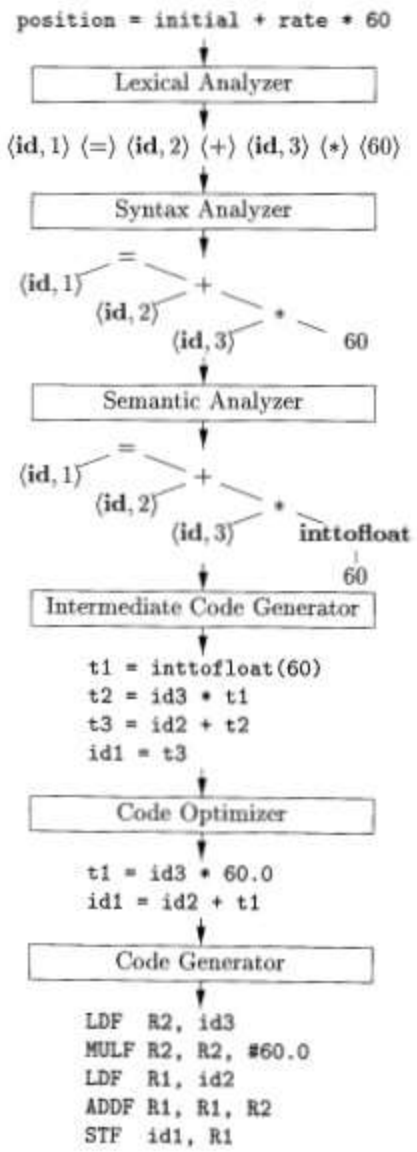
## Code Generation :

- Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language.
- Code generator translates the intermediate code into the machine code of the specified computer.

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```



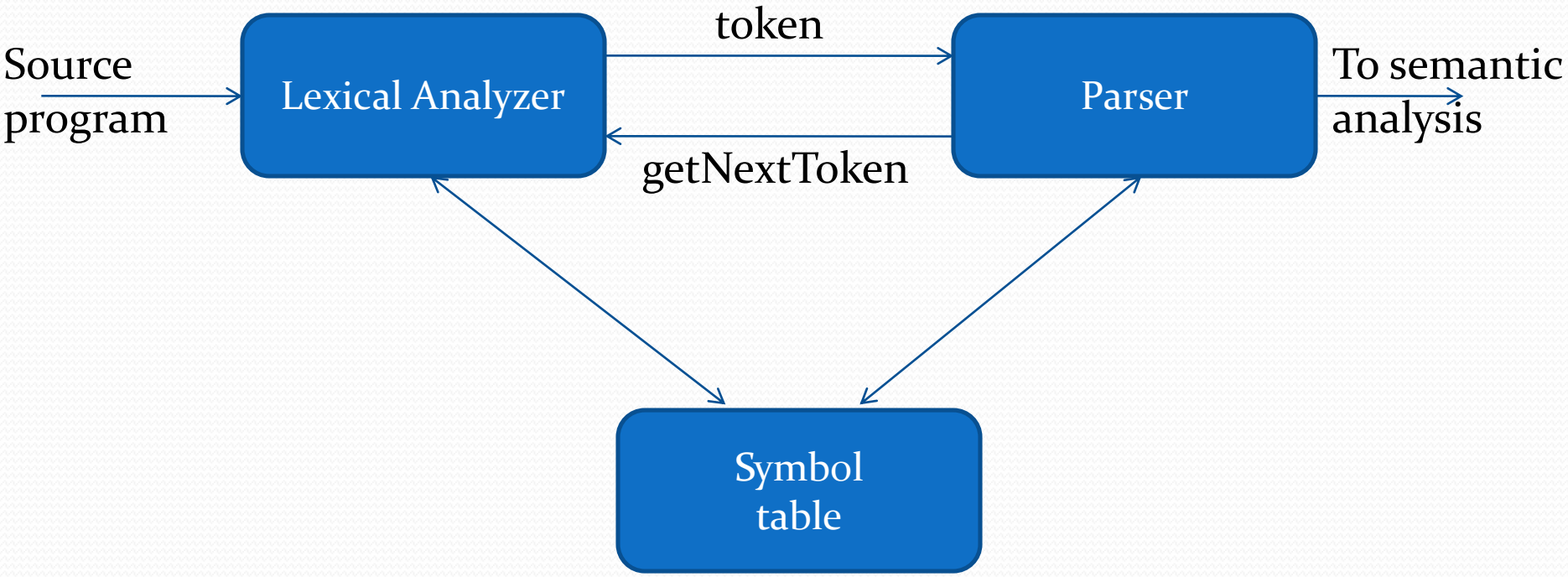

SYMBOL TABLE



# Lexical Analysis :

- The first phase of a compiler
- The main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes , and produce as output a sequence of tokens for each lexeme in the source program.
- The stream of tokens is sent to the parser for syntax analysis
- The lexical analyzer to interact with the symbol table
- One such task is stripping out comments and *whitespace* (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).

# The role of lexical analyzer :



# Lexical Analysis Versus Parsing

- Simplicity of design is the most important consideration.
- Compiler efficiency is improved
- Compiler portability is enhanced.

## Tokens, Patterns and Lexemes

- A token is a pair a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

# Example:

In many programming languages, the following classes cover most or all of the tokens:

| Token | Informal description | Sample lexemes |
|---|---|---|
| **if** | Characters i, f | if |
| **else** | Characters e, l, s, e | else |
| **comparison** | < or > or <= or >= or == or != | <=, != |
| **id** | Letter followed by letter and digits | pi, score, D2 |
| **number** | Any numeric constant | 3.14159, 0, 6.02e23 |
| **literal** | Anything but " sorrounded by " | "core dumped" |

printf("total = %d\n", score);

# Attributes for tokens

- E = M * C ** 2
  - <id, pointer to symbol table entry for E>
  - <assign-op>
  - <id, pointer to symbol table entry for M>
  - <mult-op>
  - <id, pointer to symbol table entry for C>
  - <exp-op>
  - <number, integer value 2>

# Lexical errors

- Some errors are out of power of lexical analyzer to recognize:
  - fi (a == f(x)) …
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

# Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

# Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
  - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

# Regular expressions

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) ∪ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L9r))*
- (r) is a regular expression denting L(r)

# Regular definitions

d1 -> r1

d2 -> r2

…

dn -> rn

Example:

letter_ -> A | B | … | Z | a | b | … | Z | _

digit -> 0 | 1 | … | 9

id -> letter_ (letter_ | digit)*

# Extensions

- One or more instances: (r)+
- Zero of one instances: r?
- Character classes: [abc]

- Example:
  - letter_ -> [A-Za-z_]
  - digit -> [0-9]
  - id -> letter_(letter|digit)*

# Recognition of tokens

Starting point is the language grammar to understand the tokens:

stmt -> **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| ε

expr -> term **relop** term
| term

term -> **id**
| **number**

# Recognition of tokens (cont.)

- The next step is to formalize the patterns:

  *digit* -> [0-9]

  *Digits* -> digit+

  *number* -> digit(.digits)? (E[+-]? Digit)?

  *letter* -> [A-Za-z_]

  *id* -> letter (letter|digit)*

  *If* -> if

  *Then* -> then

  *Else* -> else

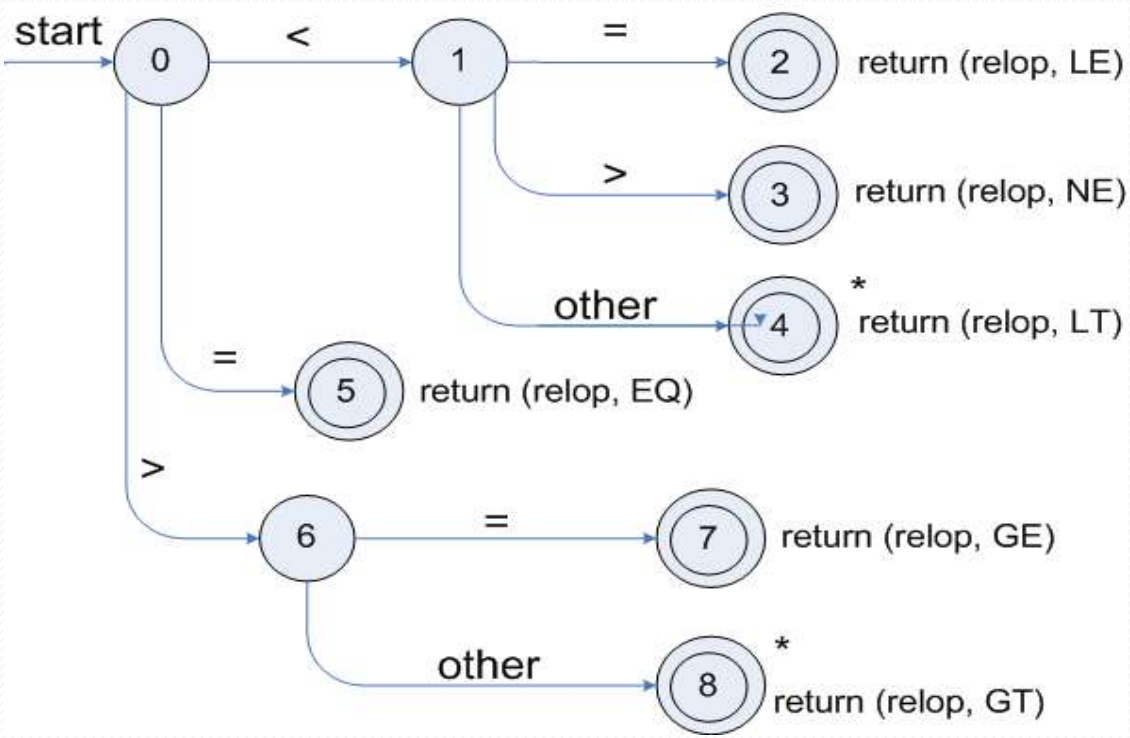
  *Relop* -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

  *ws* -> (blank | tab | newline)+

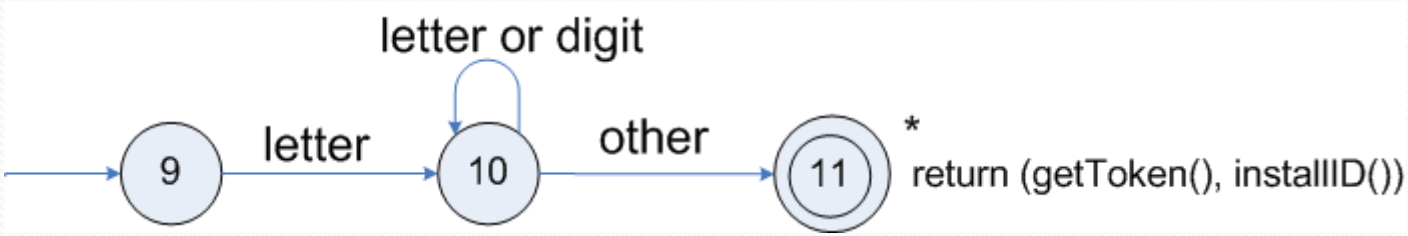
# Transition diagrams

Transition diagram for relop



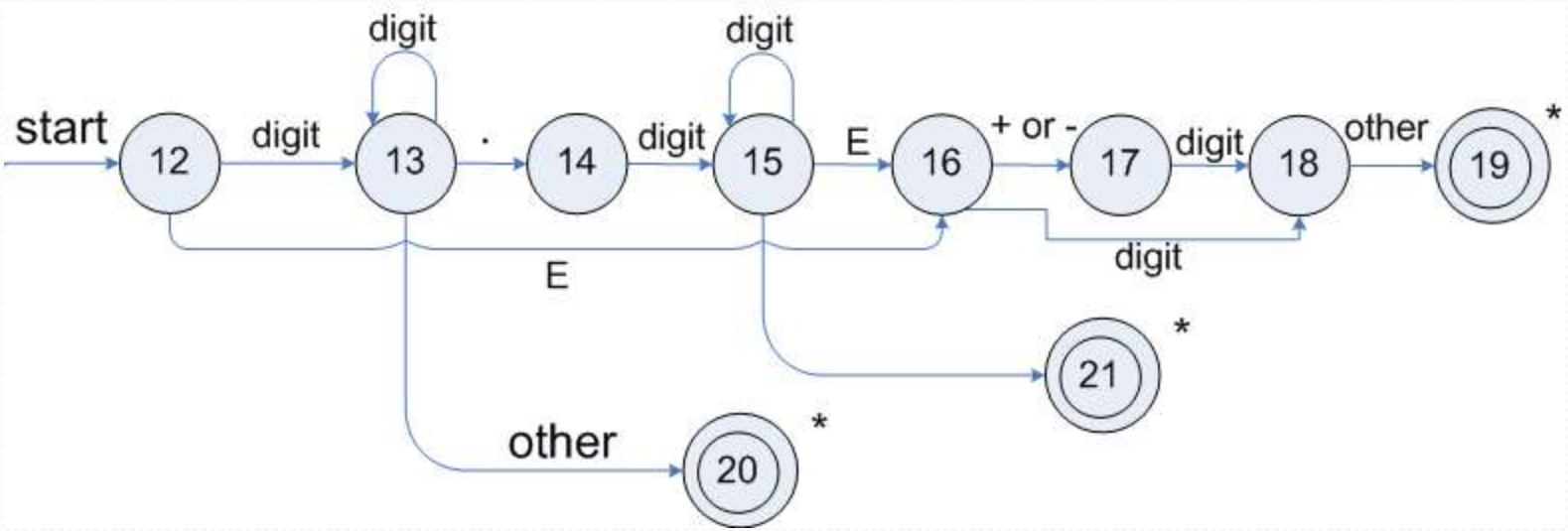
# Transition diagrams (cont.)

Transition diagram for reserved words and identifiers



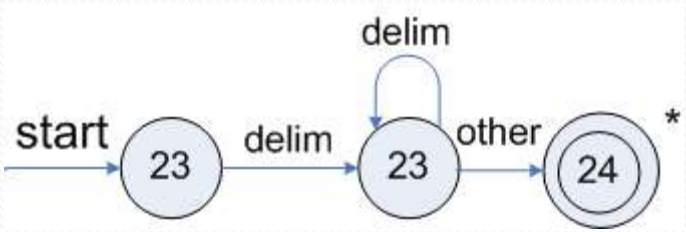
# Transition diagrams (cont.)

Transition diagram for unsigned numbers



# Transition diagrams (cont.)

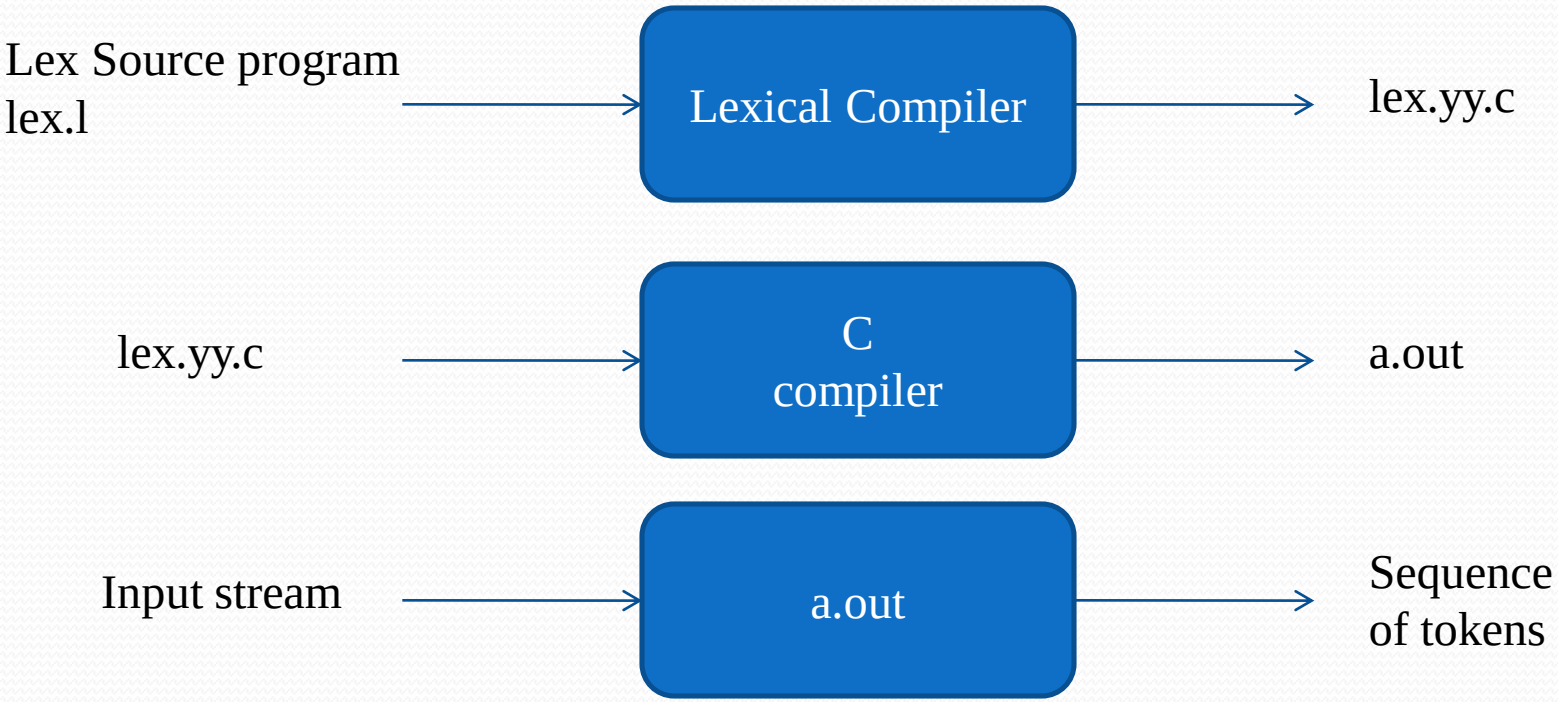
- Transition diagram for whitespace



# Lexical Analyzer Generator - Lex

- LEX is a tool that allows one to specify a Lexical Analyzer by specifying RE to describe patterns for tokens.
- Input Notation-Lex language(Specification)
- Lex Compiler-Transforms Input patterns into a Transition diagram and generates code in a file called lex.yy.c

# Lexical analyzer with LEX



# Structure of Lex programs :

Lex program has the following form:

declarations
%%
translation rules
%%
auxiliary functions

- The translation rules each have the form
  Pattern {Action}

- The declarations section includes declarations of variables, *manifest constants* (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions.
- The translation rules each have the form
  - Pattern { Action }

- pattern is a regular expression
- Action-Fragment of code written in C.

- Third Section-holds whatever additional functions are used in the actions.
- Alternatively, these functions can be compiled separately and loaded with the lexical analyser

## Conflict Resolution in Lex

There are two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer a longer prefix to a shorter prefix.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

# The Lookahead Operator

- Lex automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input.

- However, sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters. If so, we may use the **slash** in a pattern to indicate the end of the part of the pattern that matches the lexeme.

- What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

# Thank you