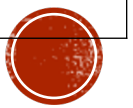
COURSE OUTCOMES (18CS61)

18CS61.1	Identify the steps involved in assembling for SIC/XE assembly language programs
18CS61.2	Utilize various techniques employed in designing and developing lexical analyzers
18CS61.3	Utilize various techniques and algorithms employed in designing and developing parsers
18CS61.4	Make use of lex and yacc tools for implementing different concepts of system software
18CS61.5	Analyze the implementation aspects compiler using Syntax -Directed Translation
18CS61.6	Analyze code generators and their functions



COURSE OUTCOMES (18CSL66)

18CS61.1	Implement programs using the LEX tool to demonstrate the Lexical Analysis phase of the compiler.
18CS61.2	Implement programs using YACC tool/C to demonstrate Syntax Analysis phase of the compiler.
18CS61.3	Implement a program using C to demonstrate the code generation phase of the compiler.
18CS61.4	Implement the different types of scheduling and memory management algorithms used in operating systems.
18CS61.5	Illustrate the deadlock handling algorithm in the operating system.
18CS61.6	Create programs using LEX and YACC to recognize C language constructs.

INTRODUCTION , LEXICAL ANALYSIS

Module 2

Text book 2:

Chapter 1:1.1-1.5

Chapter 3: 3.1 – 3.4

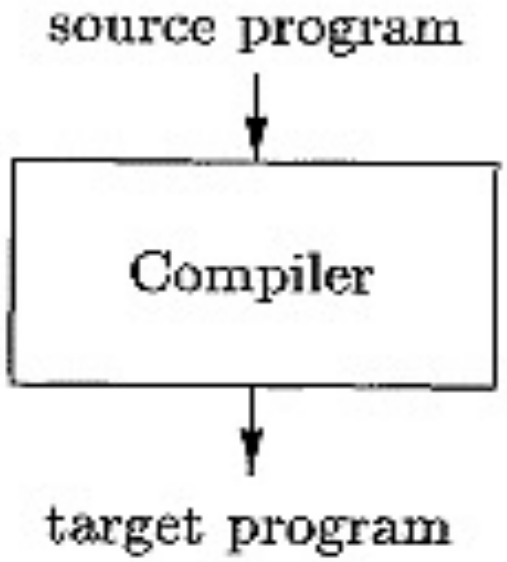
CONTENTS

Introduction

- Language processors
- The structure of a compiler
- The evolution of programming languages
- The science of building a compiler
- Applications of compiler technology

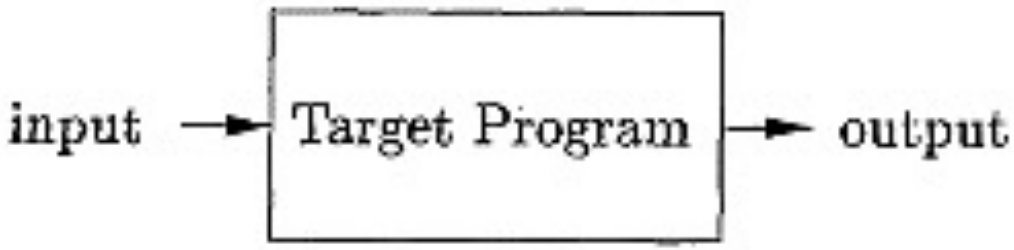
LANGUAGE PROCESSORS

- **Compiler :** Program that can read a program in one language (*source* language) and translate it into an equivalent program in another language (*target* language).



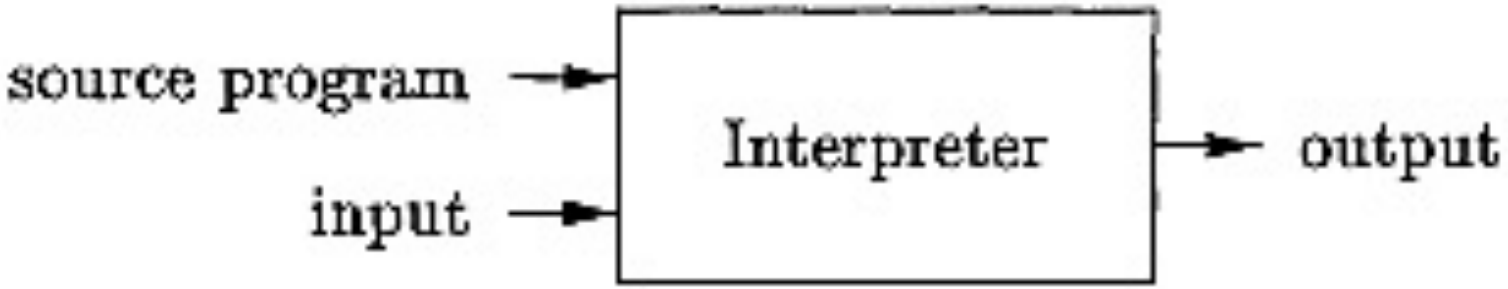
A compiler

- The target program is called by the user to process inputs and produce outputs.



Running the target program

- **Interpreter :** Executes the operations specified in the source program on inputs supplied by the user .

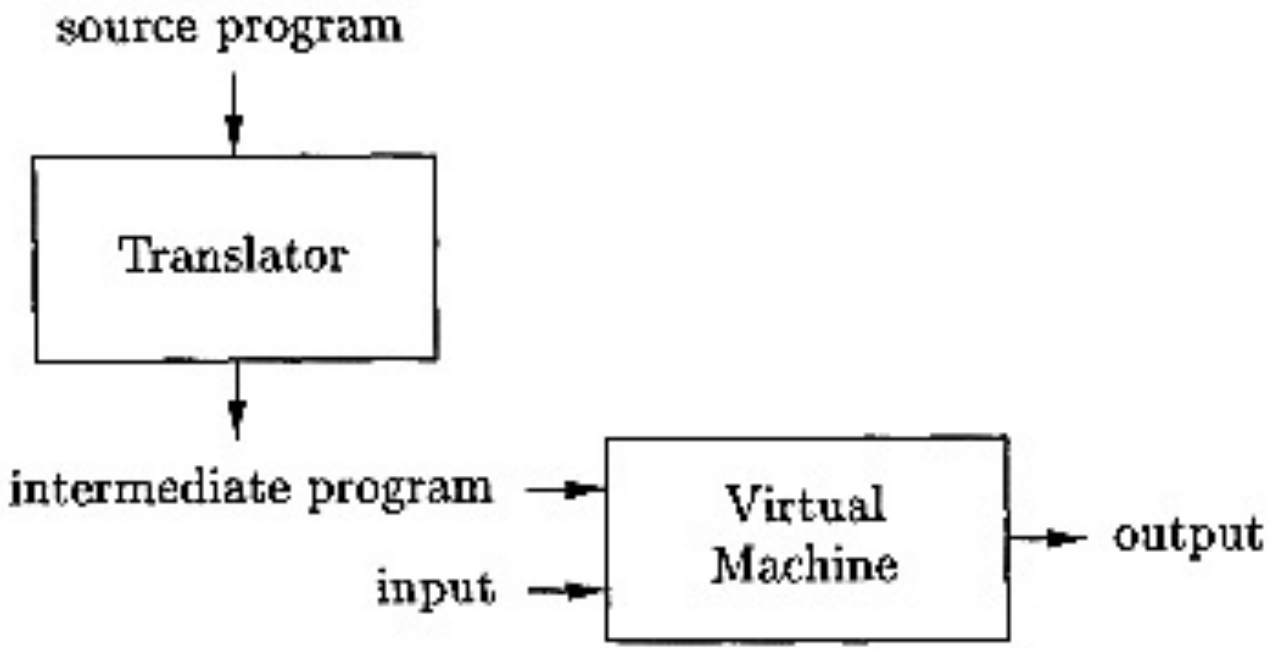


An interpreter

- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs .
- An interpreter, usually give better error diagnostics than a compiler.

EXAMPLE

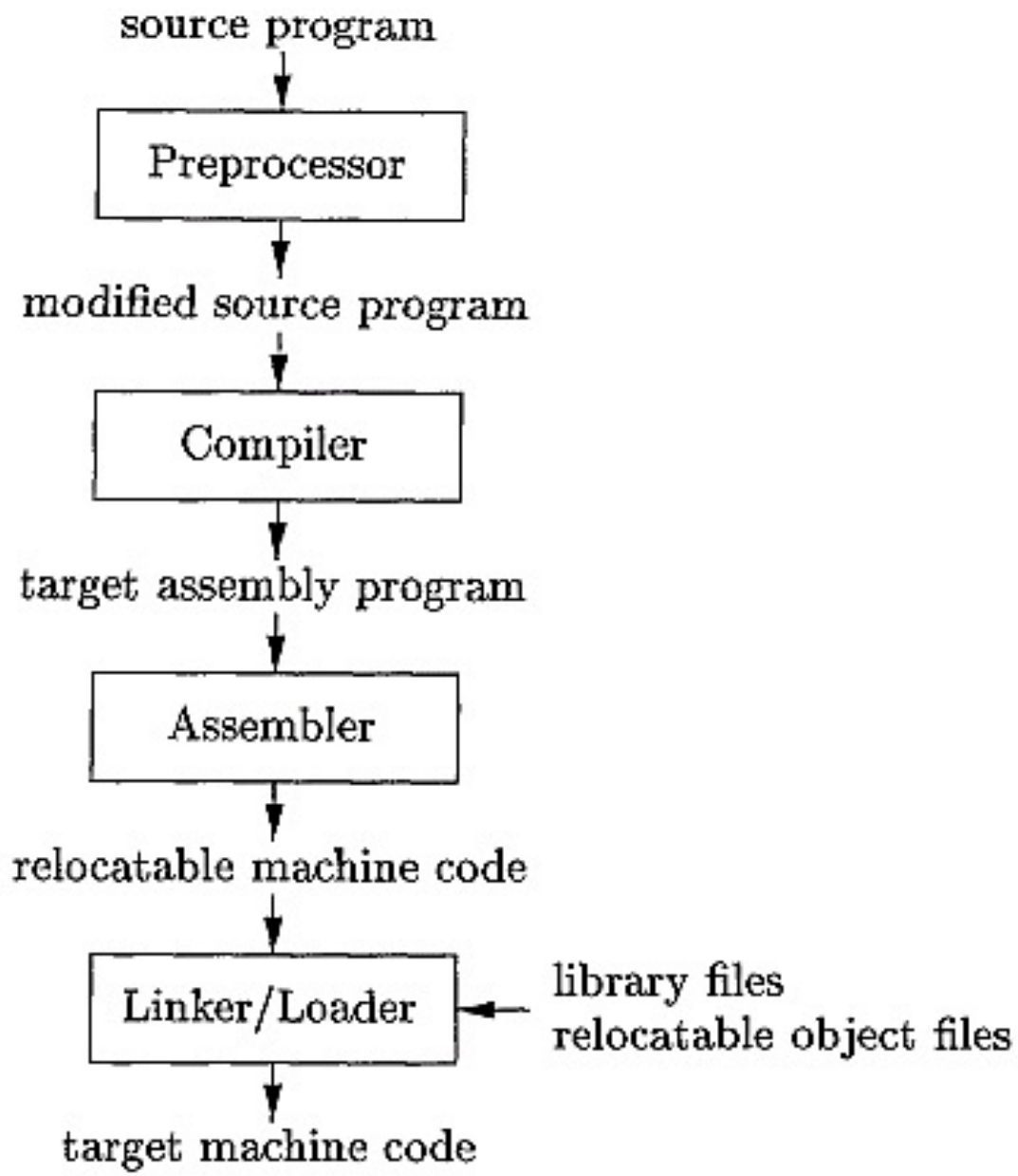
- Java language processors : Combines compilation and interpretation



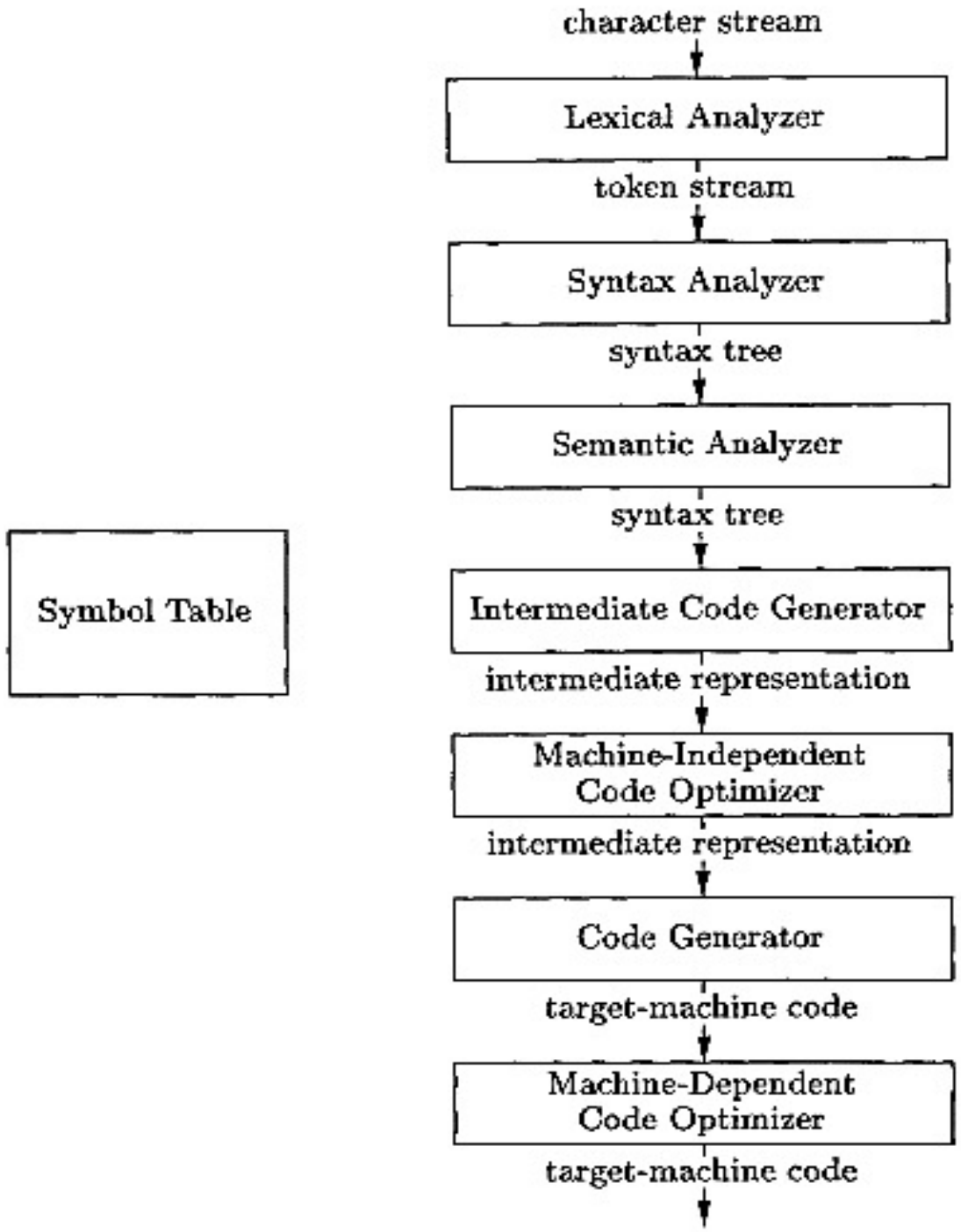
A hybrid compiler

- Several other programs may be required to create an executable target program.
  - **Preprocessor :** The task of collecting the source program stored in separate files.
  - **Assembler:** Program that processe's the asssembly level program to produce relocatable machine code as its output.

- **Linker :** 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, resolves external memory addresses.
- **Loader:** Puts together all of the executable object files into memory for execution.



A language-processing system



THE STRUCTURE OF A COMPILER

- Phases of a compiler

THE STRUCTURE OF A COMPILER

- Two parts of mapping : Analysis and Synthesis
- The Analysis part breaks the source program into constituent pieces and imposes a grammatical structure on them.
- Also collects information about the source program and stores it in a data structure called symbol table.
- The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.
- The analysis part is often called as front end of the compiler ; the synthesis part is the back end.

LEXICAL ANALYSIS/SCANNING

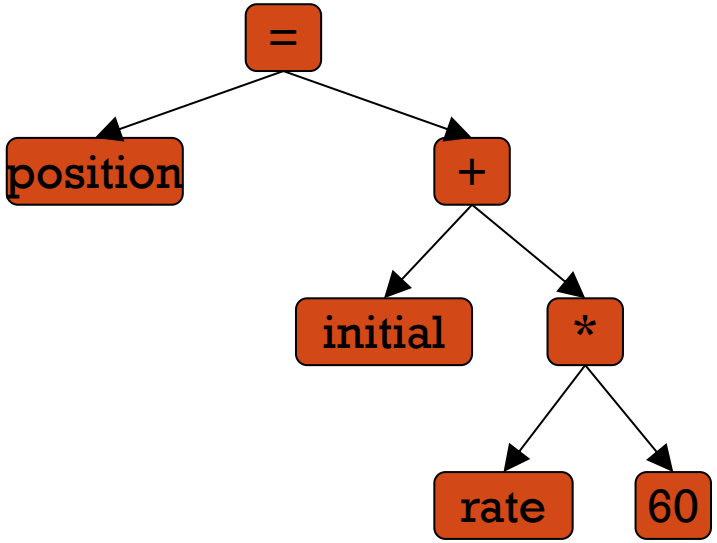
- Reads stream of characters making up the source program.
- Groups the characters into meaningful sequences called **lexemes.**
- For each lexeme, the lexical analyzer produces as output a token of the form **<token-name, attribute-value>**
  - Token-name is an abstract symbol that is used during syntax analysis
  - Attribute-value points to an entry in the symbol table for this token.
- Ex : position = i n i t i a l + r a t e * 60

  Token stream : <id,1> <=> <id,2> <+> <id,3> <*> <60>

SYNTAX ANALYSIS/PARSING

- Groups tokens to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- Representation : Syntax tree
  - Interior node represents an operation and the children of the node represent the arguments of the operation.
- Ex : position = i n i t i a l + r a t e * 60

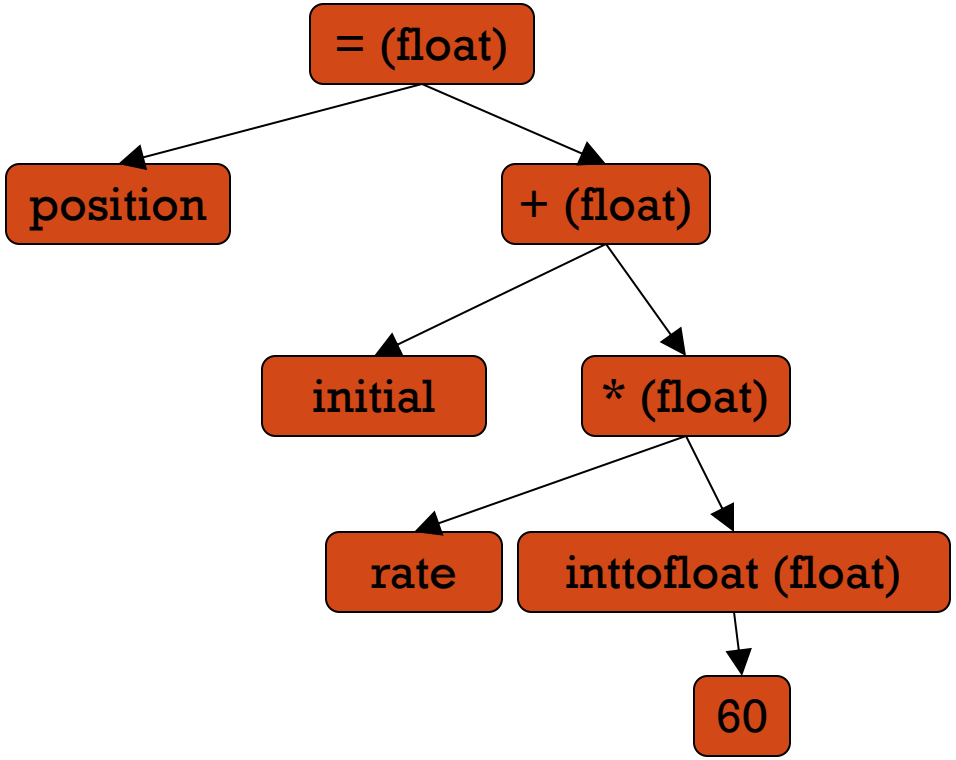
Syntax Tree:



SEMANTIC ANALYSIS

- Checks for semantic errors ,uses the syntax tree and information in the symbol table.
- Gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- **Type checking** : The compiler checks that each operator has matching operands.

- The language specification may permit some type conversions called **coercions.**



INTERMEDIATE CODE GENERATION

- Compilers generate an explicit low-level or machine-like intermediate representation, as program .
- Two important properties of the intermediate representation :
  - Easy to produce
  - Easy to translate into the target machine.
- Intermediate form called **three-address code**: Consists of a sequence of assembly-like instructions with three operands per instruction.

- Ex : position = i n i t i a l + r a t e * 60

Three-address code sequence

tl = i n t t o f l o a t (60)

t 2 = id3 * tl

t 3 = id2 + t 2

id1 = t 3

CODE OPTIMIZATION

- Attempts to improve the intermediate code so that better target code will result.
- Objectives of better code includes faster, shorter code, or target code that consumes less power.
- Ex : intermediate code :

```
t1 = inttofloat (60)
t2 = id3 * t1
t3 = id2 + t 2
id1 = t3
```

- Optimizer transform intermediate code into the shorter sequence:

```
t1= id3 * 60.0
id1= id2 + t1
```

CODE GENERATION

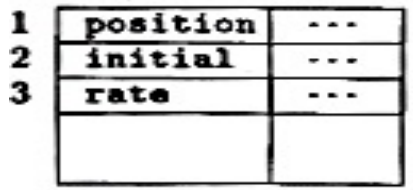
- The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- Target language as machine code.
  - Registers or memory locations are selected for each of the variables used by the program.
- Intermediate code : t1= id3 * 60.0

  id1= id2 + t1
- Using registers R1 and R2, the intermediate code might get translated into the machine code

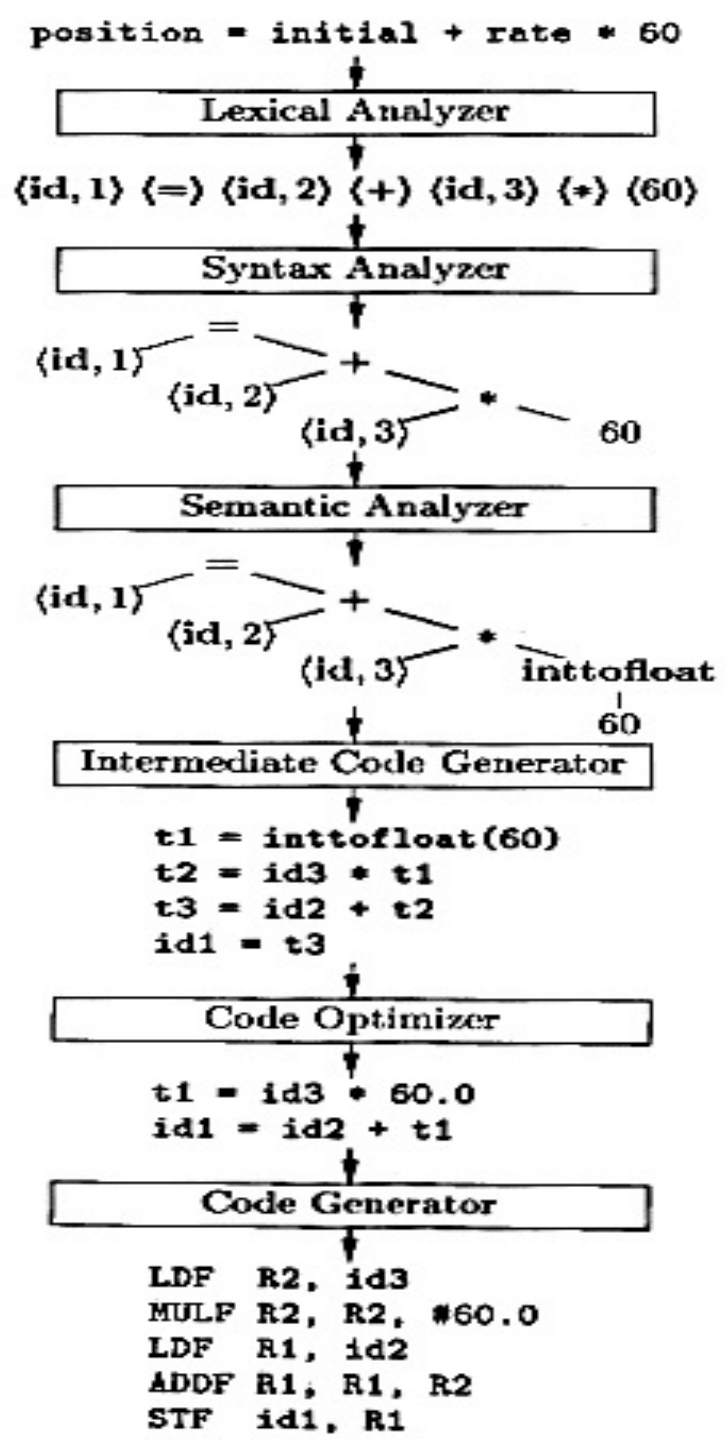
```
LDF   R2, id3
MULF  R2 , R2 , #60.0
LDF   R l , id2
ADDF  Rl , Rl , R2
STF   idl , Rl
```

SYMBOL-TABLE MANAGEMENT

- Data structure containing a record for each variable name, with fields for the attributes of the name.
- Compiler records the variable names used in the source program and collect information about various attributes of each name.
  - Attributes may provide information about the storage allocated for a name, its type, its scope .
  - In the case of procedure names, such things as the number and types of its arguments, the method of passing each argument and the type returned.



SYMBOL TABLE



THE GROUPING OF PHASES INTO PASSES

- Logical organization of a compiler
- Activities from several phases may be grouped together into a pass that reads an input file and writes an output file.
  - **Front-end phases** : lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass.
  - Code optimization might be an optional pass.
  - **Back-end pass** consisting of code generation for a particular target machine.

COMPILER CONSTRUCTION TOOLS

1. Scanner generators - regular-expression description of the tokens of a language.
2. Parser generators - Grammatical description of a programming language.
3. Syntax-directed translation engines - For walking a parse tree and generating intermediate code.
4. Code-generator generators - Collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. Data-flow analysis engine - Gathering of information about how values are transmitted from one part of a program to each other part
6. Compiler-construction toolkits - For constructing various phases of a compiler.

THE EVOLUTION OF PROGRAMMING LANGUAGES

The Move to higher level language

- Major step towards the high level languages was made in the latter half of 1950s : Fortran-scientific computation, Cobol-business data processing .
- **Classification based on generation**
  - First generation languages - machine languages
  - Second generation languages - assembly languages
  - Third generation languages - higher level languages
  - Fourth generation languages- designed for specific application(SQL)
  - Fifth generation languages - applied to logic and constraint based languages(Prolog)

Imperative language Vs Declarative language:

- Languages in which a program specifies how a computation is to be done.
- Ex: C, C++
- Languages in which a program specifies what computation is to be done.
- Ex: Prolog

- **Von Nuemann language :**
  - Languages whose computational model is based on Von Nuemann Architecture.
  - Ex: Fortan, C

- **Object Oriented Language:**
  - Languages which support object- oriented programming style.
  - Ex: C++, C#, Java, Ruby
- **Scripting Language:**
  - Languages that are interpreted with high level operators designed for gluing together computations .
  - Ex: JavaScript, PHP, Perl, Python ,Ruby

THE SCIENCE OF BUILDING A COMPILER

- The science behind the compiler:
  - Take a problem
  - Formulate a mathematical abstraction that captures the key characteristics.
  - Solve it using the mathematical techniques.
- A compiler must accept all source programs that conform to the specification of the language.
- Must preserve the meaning of the program being compiled.

Modeling in compiler design and implementation

- Finite state machines and regular expressions
  - For describing lexical units of programs
  - For describing the algorithms used by the compiler to recognize those units.
- Context free grammars:
  - Used to describe the syntactic structure of programming languages.
- Trees
  - Used for representing the structure of programs and their translation into object code.

The science of code optimization

- Attempts that a compiler makes to produce code that is more efficient than the obvious code.
- Compiler optimization must meet the following design objectives:
  - The optimization must be correct.
  - Must improve the performance of many programs.
  - The compilation time must be kept reasonable.
  - The engineering effort required must be manageable.

APPLICATIONS OF COMPILER TECHNOLOGY

Implementation of High-Level Programming Languages

- A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language.
- Higher-level programming languages are easier to program in, but are less efficient, that is, the target programs run more slowly .
- Low-level language have more control over a computation and produce more efficient code.

- Optimizing compilers include techniques to improve the performance of generated code
- Ex: usage of **register** keyword in the earlier programming language.
- Common programming languages, support user-defined aggregate data types, such as arrays and structures, and high-level control flow, such as loops and procedure invocations.
- A body of compiler optimizations, known as **data-flow optimizations**, has been developed to analyze the flow of data through the program and removes redundancies across these constructs.

- The key ideas behind object orientation are
  - 1. Data abstraction and
  - 2. Inheritance of properties,

both of which have been found to make programs more modular and easier to maintain.

Optimizations for Computer Architectures

- Evolution of computer architectures has also led to an insatiable demand for new compiler technology.
- All high-performance systems take advantage of the same two basic techniques: **parallelism** and **memory hierarchies.**

Design of New Computer Architectures

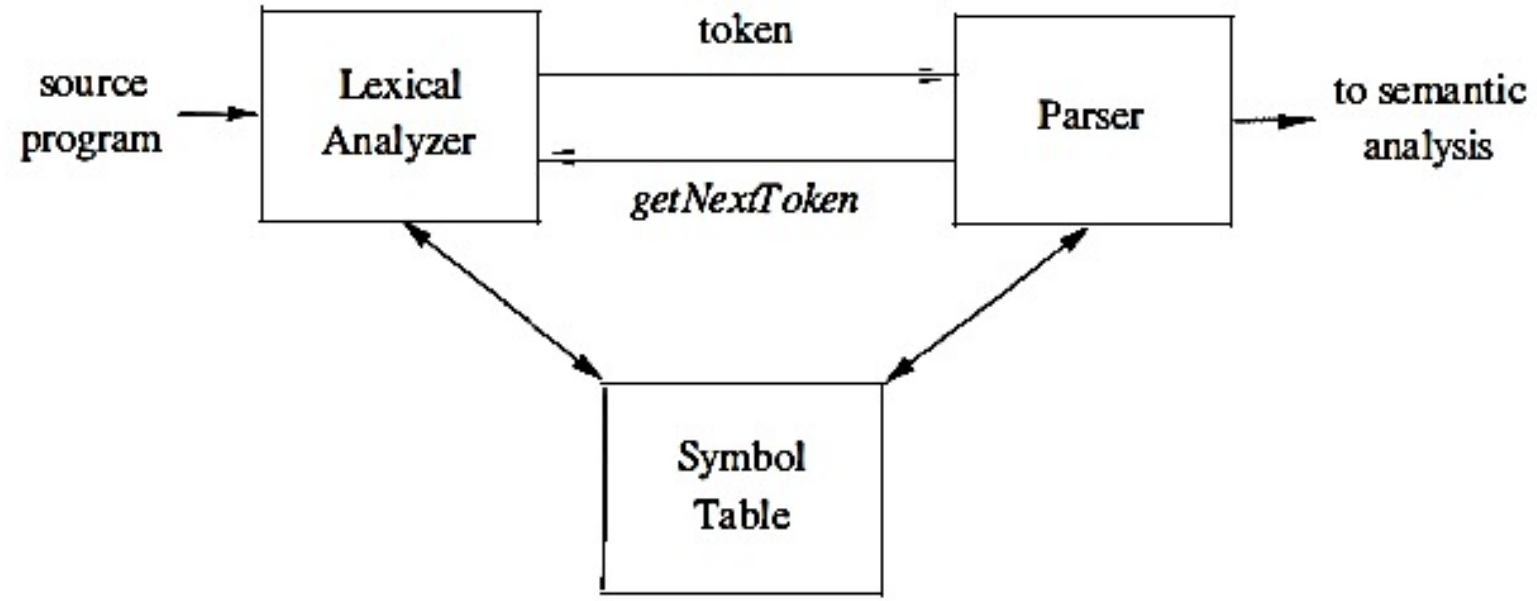
- The performance of a computer system is determined not by its raw speed but also by how well compilers can exploit its features.
- In modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features.
- Ex : Invention of the RISC (Reduced Instruction-Set Computer) architecture

Program translations

- Compiling as a translation from a high-level language to the machine level, the same technology can be applied to translate between different kinds of languages.
- The following are some of the important applications of program-translation techniques.
- Binary Translation
  - Compiler technology can be used to translate the binary code for one machine to that of another, allowing a machine to run programs originally compiled for another instruction set.

LEXICAL ANALYSIS : The Role of the Lexical Analyzer

- Lexical analyzer reads the input characters from the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.



Interactions between the lexical analyzer and the parser

- Lexical analyser performs certain other tasks :
  - Stripping out comments and whitespace.
  - Correlating error messages generated by the compiler with the source program.
    - Keeps track of the number of newline characters seen, so it can associate a line number with each error message.
- The expansion of macros may also be performed by the lexical analyzer.

- Lexical analyzers are sometimes divided into a cascade of two processes.
  - Scanning
    - Do not require tokenization of the input, deletion of comments and compaction of whitespace character into one.
  - Lexical analysis
    - Complex portion, which produces tokens from the output of the scanner.

LEXICAL ANALYSIS VERSUS PARSING

- Simplicity of design
- Improving compiler efficiency
- Enhancing compiler portability

TOKENS, PATTERNS AND LEXEMES

- **Tokens :** Pair consisting of a token name and an optional attribute value.
- Token names are the abstract symbol representing a kind of lexical unit.
  - Keywords, operators, identifiers, constants etc.
- **Patterns :** Description of the form that the lexemes of a token may take.
  - Keyword as a token, the pattern is just the sequence of characters that form the keyword.
- **Lexeme :** Sequence of characters in the source program that matches the pattern for a token.

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 letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Examples of tokens

- Most of the tokens belong to the following classes
  - One token per keyword
  - Tokens for the operators
  - One token representing all identifiers
  - Tokens representing constants (e.g. numbers)
  - Tokens for punctuation symbols

EXAMPLE

- C statement :

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

- printf and score are lexemes matching the pattern for token id.
- " Total = %d\n" is a lexeme matching literal.

ATTRIBUTES FOR TOKENS

- A pointer to the symbol-table entry in which the information about the token is kept

E.g: E=M*C**2

<**id**, pointer to symbol-table entry for E>

<**assign_op**>

<**id**, pointer to symbol-table entry for M>

<**multi_op**>

<**id**, pointer to symbol-table entry for C>

<**exp_op**>

<**num**,integer value 2>

LEXICAL ERRORS

- Some errors are out of power of lexical analyzer to recognize:

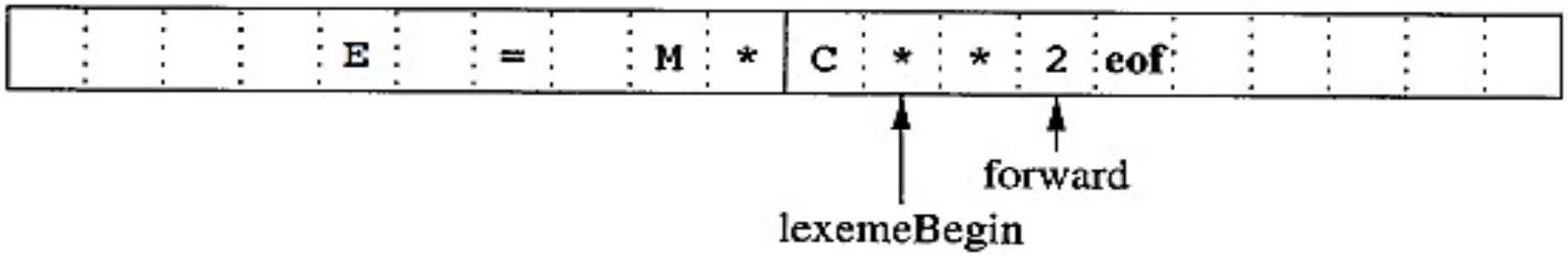
  fi (a == f(x)) …

- Situation arises in which the lexical analyser is unable to proceed because none of the patterns for the tokens matches any prefix of the remaining input.
- Error recovery strategy: **Panic mode recovery**
- **Panic mode:** Delete successive characters from the remaining input, until the lexical analyser finds a well-formed token at the beginning of what input is left.

- Other possible error-recovery actions are:
  - 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.

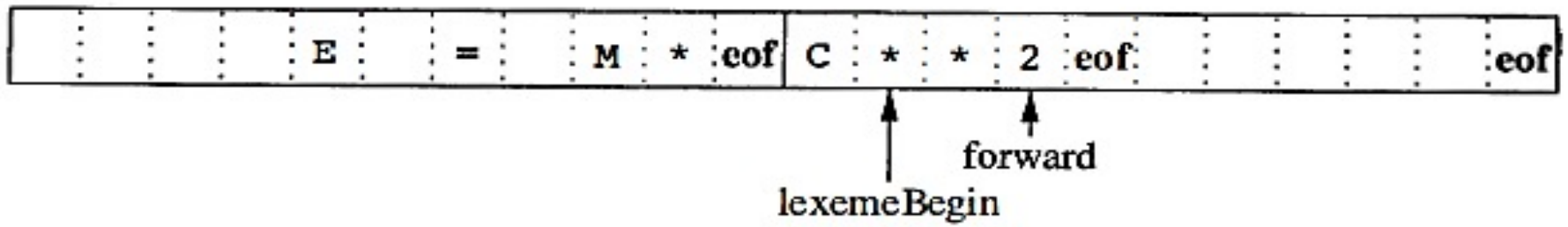
INPUT BUFFERING

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
- Two buffer scheme to handle large look-aheads safely



Using a pair of input buffers

SENTINELS



Sentinels at the end of each buffer

ALGORITHM TO MOVE FORWARD POINTER

Lookahead code with sentinels

```
Switch(*forward++)
 {
 case eof:
            if (forward is at end of first buffer)
            {  reload second buffer;
               forward = beginning of second buffer;
             }
     else if (forward is at end of second buffer)
            {  reload first buffer;
               forward = beginning of first buffer;
             }
     else /* eof within a buffer marks the end of input terminate lexical analysis*/
     break;
      case for the other characters
 }
```

SPECIFICATION OF TOKENS

- **Regular expressions** : An important notation for specifying lexeme patterns.
- **Alphabet :** A finite set of symbols. E.g : {a, b, c} {0,1}
- **String** : Finite sequence of symbols drawn from alphabet (sentence / word) E.g : 000, 0011, ….
- **Length of string:** |s| is the number of occurrences of symbols.
- **Empty string :** denoted by ε
- Prefix, suffix, substring, proper (Prefix, suffix, substring), subsequence of string
- **Language :** A countable set of strings over some fixed alphabet.

OPERATIONS ON LANGUAGES

- Operation on languages (a set):
  - **Union** of L and M, L U M = {s | s is in L or s is in M}
  - **Concatenation** of L and M

    LM = {st | s is in L and t is in M}
  - **Kleene closure** of L, $L^* = \bigcup_{i=0}^{\infty} L^i$
  - **Positive closure** of L, $L^+ = \bigcup_{i=1}^{\infty} L^i$

EXAMPLE

- L = {A, B , . . . , Z , a, b, . . . , z}
- D = {0,1 , . . . 9}

languages that can be constructed from languages L and D

- L U D is the set of letters and digits (62 strings)
- LD is the set of strings of length two.
- L^4 is the set of all 4-letter strings.
- L^* is the set of all strings of letters.
- L(L U D)* is the set of all strings of letters and digits beginning with a letter.
- D^+ is the set of all strings of one or more digits

REGULAR EXPRESSIONS

- Notation used for describing all the languages that can be built from operators applied to the symbols of some alphabet.
- Example:
  - letter_(letter_ | digit)* , language of C identifiers
- The regular expressions are built recursively out of smaller regular expressions, using the rules:
- Each regular expression r denotes a language L(r).
- The rules that define the regular expressions over some alphabet Σ

Basis

Regular expressions can be defined as follows:

- If Ɛ is R.E , then L(Ɛ)={Ɛ}
- If L(a)={a},then a is a R.E

Induction

Larger R.E are built from smaller ones

- Suppose r and s are regular expression denoting the languages L(r) and L(s) then,
  - (r|s) is a regular expression denoting L(r)U L(s)
  - rs is a regular expression denoting L(r)L(s)
  - (r)* is a regular expression denoting (L(r))*
  - (r) is a regular expression denoting L(r)

- If two regular expressions r and s denote the same regular set ,then they are equivalent and r = s.
- Some of the algebraic laws that hold for arbitrary regular Expressions r, s, and t.

LAW	DESCRIPTION
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over |
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Algebraic laws for regular expressions

Regular expression set : Let Σ= {a, b}

a|b	{a, b}
(a|b)(a|b) or aa|ab|ba|bb	{aa, ab, ba, bb}
a*	{ε, a, aa, aaa,…}
(a|b)*	{ε, a,aa,b,bb,…}
a|a*b	{a, b, ab, aab, aaab,…}

REGULAR DEFINITION

- Process of giving a name to certain regular expressions and use those names in subsequent expressions.

- A *regular definition* is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in

. $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

$d_n \rightarrow r_n$

basic symbols

previously defined names

- Example : regular definition for the language of C identifiers

$$
\begin{array}{rcl}
letter\_ & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
id & \rightarrow & letter\_ \; (\; letter\_ \mid digit \;)^*
\end{array}
$$

- Example : Unsigned numbers (integer or floating point) are strings such as 5280, 0 . 0 1234, 6 . 336E4, or 1 . 89E-4, 123E4.

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
digits & \rightarrow & digit \; digit^* \\
optionalFraction & \rightarrow & \texttt{.} \; digits \mid \epsilon \\
optionalExponent & \rightarrow & (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; digits \;) \mid \epsilon \\
number & \rightarrow & digits \; optionalFraction \; optionalExponent
\end{array}
$$

EXTENSIONS OF REGULAR EXPRESSIONS

- Example: Using shorthands, rewrite the regular definition

$$
\begin{aligned}
\textit{letter\_} &\rightarrow \texttt{[A-Za-z\_]} \\
\textit{digit} &\rightarrow \texttt{[0-9]} \\
\textit{id} &\rightarrow \textit{letter\_}\ (\ \textit{letter} \mid \textit{digit}\ )^*
\end{aligned}
$$

$$
\begin{aligned}
\textit{digit} &\rightarrow \texttt{[0-9]} \\
\textit{digits} &\rightarrow \textit{digit}^+ \\
\textit{number} &\rightarrow \textit{digits}\ (\texttt{.}\ \textit{digits})?\ (\ \texttt{E}\ \texttt{[+-]}?\ \textit{digits}\ )?
\end{aligned}
$$

RECOGNITION OF TOKENS

- Consider the following grammar/regular definitions:

Stmt→ **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| ε

Expr→ *term* **relop** *term*

| *term*

Term→ **id**

| **num**

digit->[0-9]

digits->digit $^{+}$

num → digits (.digits+)? (E[+-]?digits) ?

letter ->[A-Za-z]

id → letter(letter|digit)*

if → if

then → then

else → else

relop → <|<=|>|>=|=|<>

- Lexical analyzer also has the job of striping out whitespace, by recognition the token "ws" defined by:

ws->(blank|tab|newline) $^{+}$

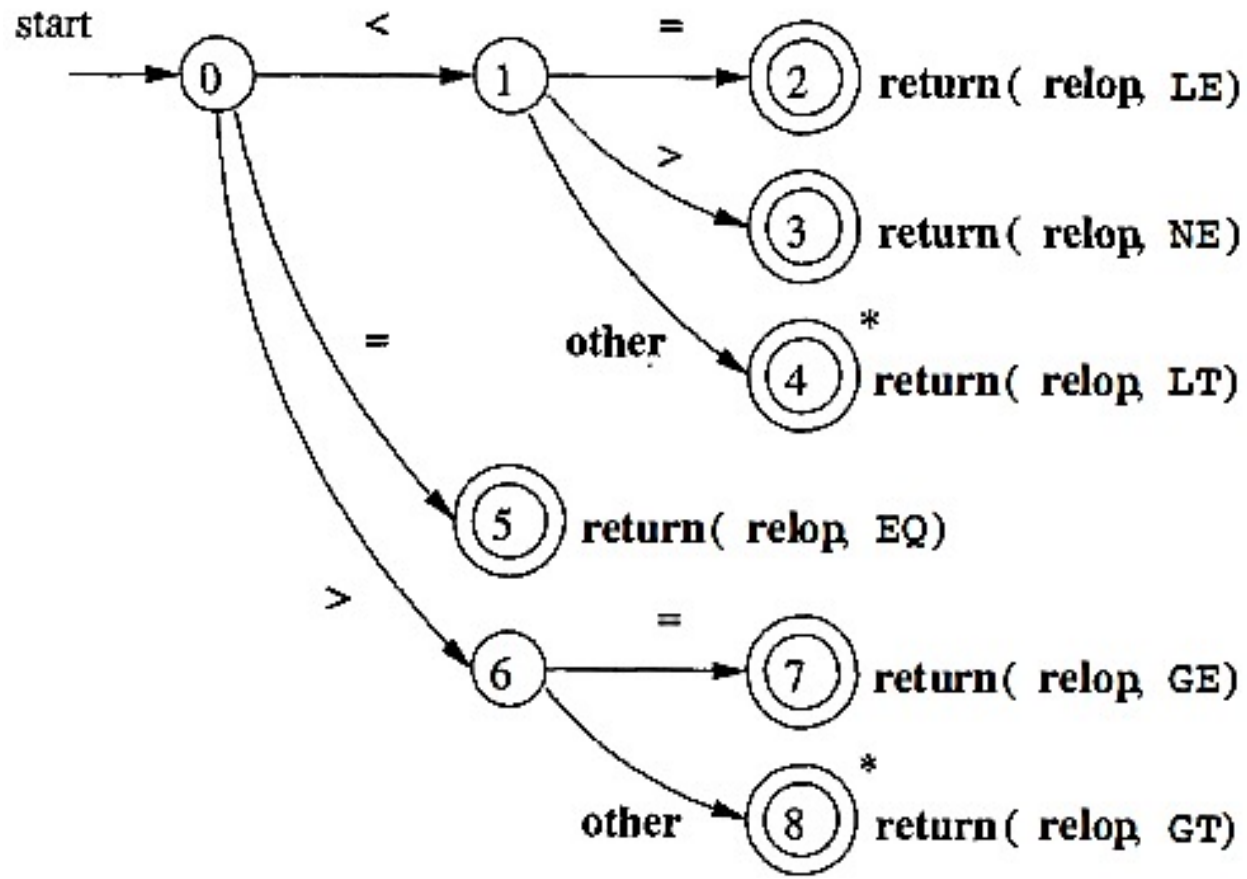
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Tokens, their patterns, and attribute values

TRANSITION DIAGRAMS

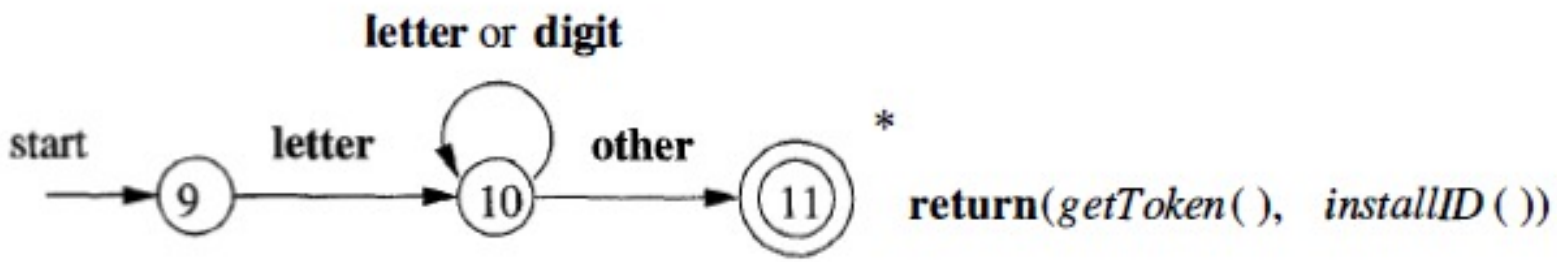
- Collection of nodes or circles, called **states**.
- Edges are directed from one state of the transition diagram to another, edge is labeled by a symbol.
- Certain states are said to be accepting, or final.
- One state is designated the start state, or initial state.
- Retract the forward pointer one position: denoted by *

- Transition diagram **relop**



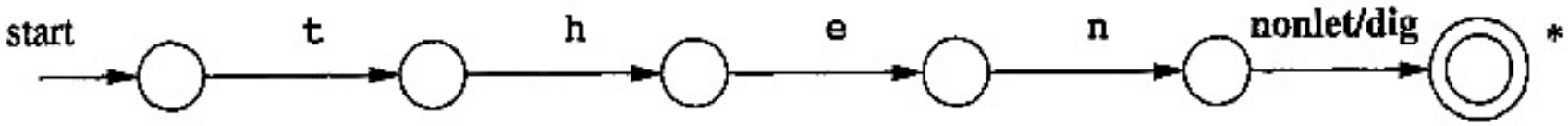
RECOGNITION OF RESERVED WORDS AND IDENTIFIERS

1) Install the reserved words in the symbol table initially.
- installID( ): places it in the symbol table if it is not already there.
- getToken( ): examines the symbol table entry for the lexeme found



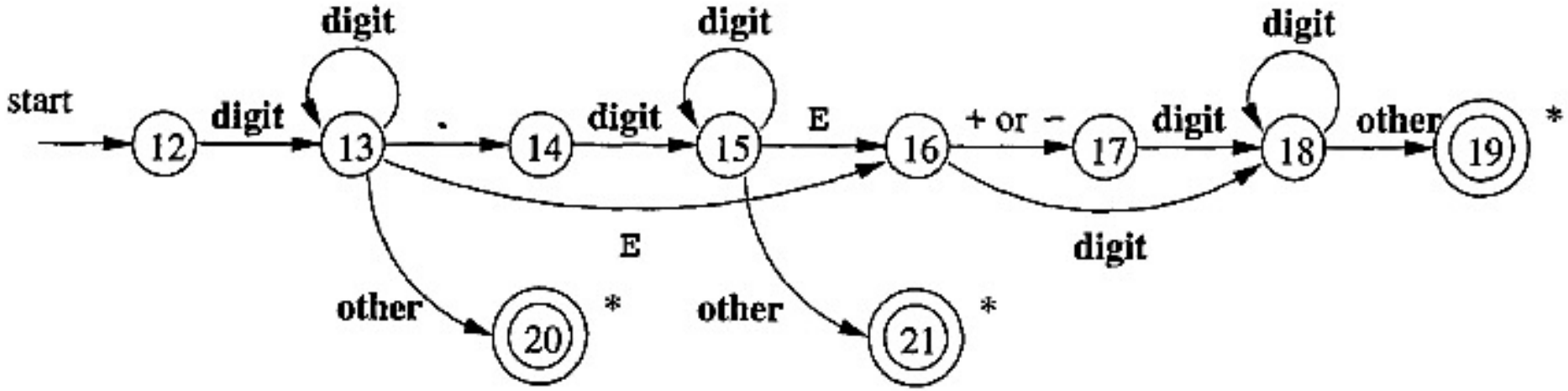
A transition diagram for **id**'s and keywords

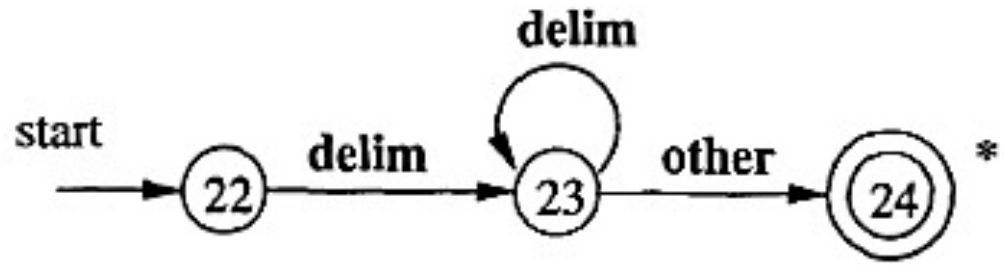
2) Create separate transition diagrams for each keyword



Hypothetical transition diagram for the keyword **then**

- Transition diagram for unsigned number:



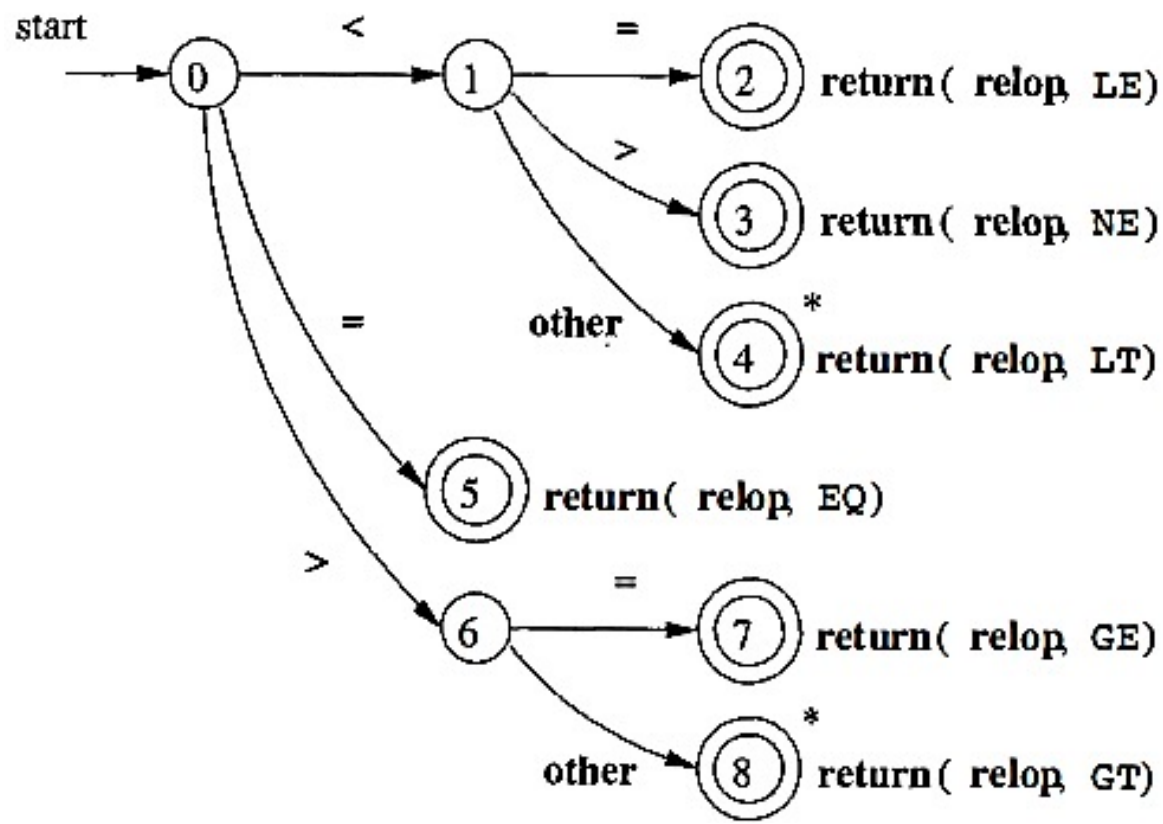


A transition diagram for whitespace

ARCHITECTURE OF A TRANSITION- DIAGRAM BASED LEXICAL ANALYSER

```
TOKEN getRelop ( ) {
TOKEN retToken = new(RELOP);
while(1) { /* repeat character processing until a return or failure occurs */
switch(state) {
case 0:
        c = nextChar( );
        if ( c == '<' ) state = 1;
       else if ( c == '=' ) state = 5;
       else if ( c == '>' ) state = 6;
       else fail(); /* lexeme is not a relop */
        break;
case 1: ……
```

```
case 2: ….
case 8: retract();
        retToken.attribute = GT;
        return(retToken);
}
}
```


start
0
<
1
=
2
return(relop, LE)
>
3
return(relop, NE)
other
*
4
return(relop, LT)
=
5
return(relop, EQ)
>
6
=
7
return(relop, GE)
other
*
8
return(relop, GT)