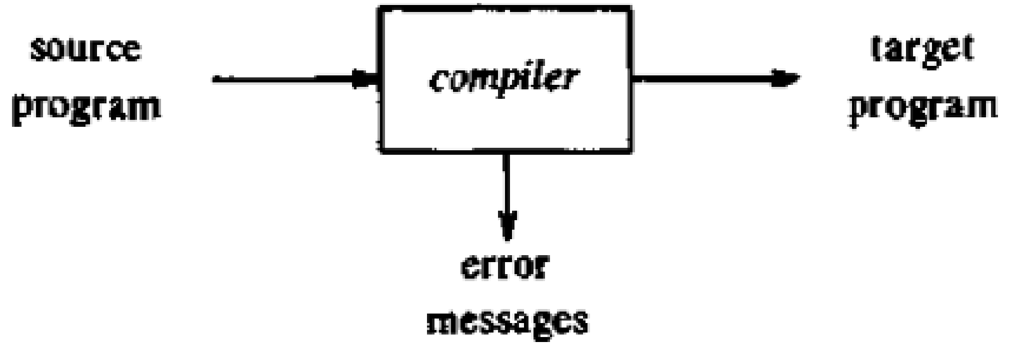
Compiler Design

Introduction

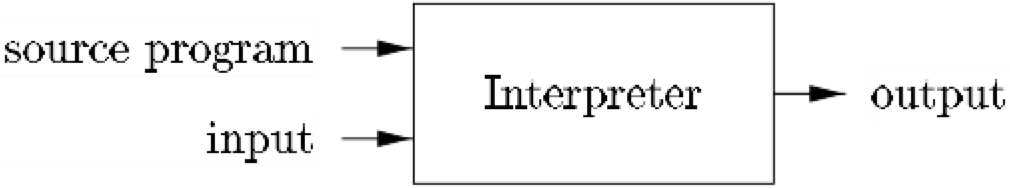
- Compiler
  - Program that can read a program in one language (the source language) and translate it into an equivalent program in another language (the target language)
  - Report any errors in the source program that it detects during the translation process



Introduction

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

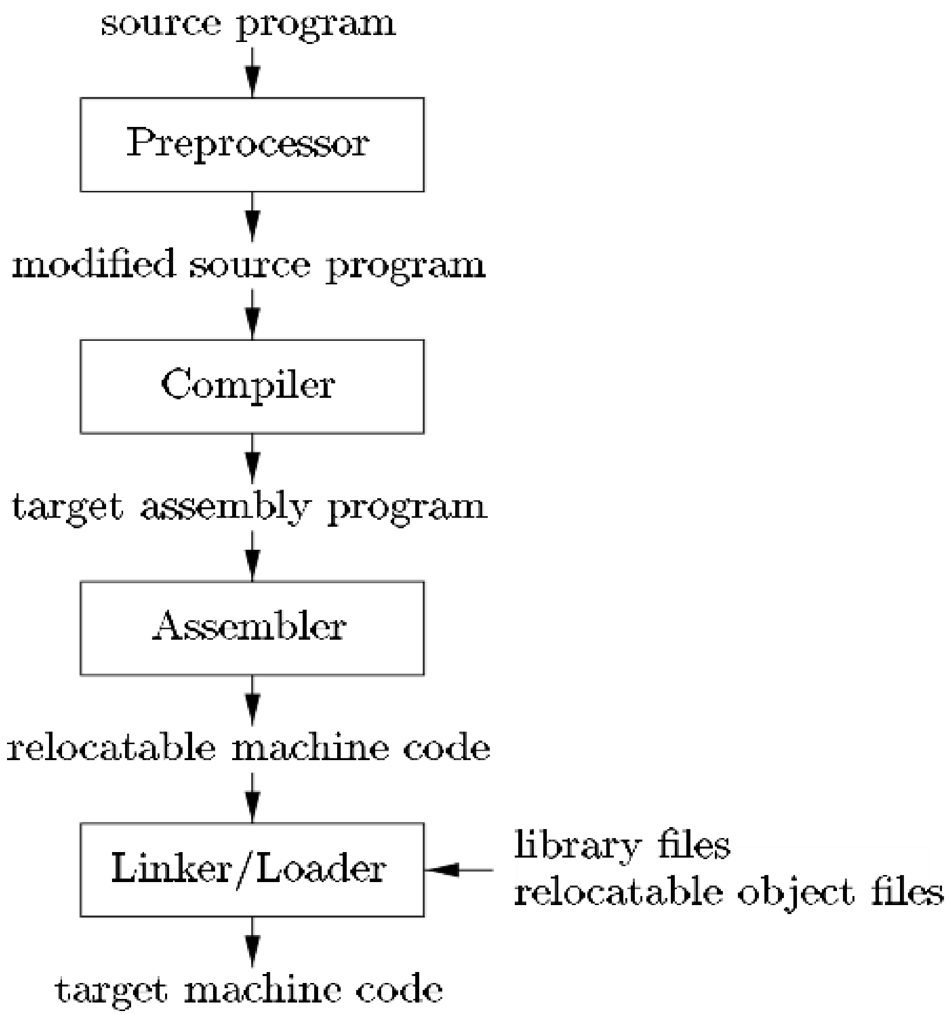
- Interpreter
  - Language processor
  - an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user



Introduction

- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs .
- An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement
- Java language processors

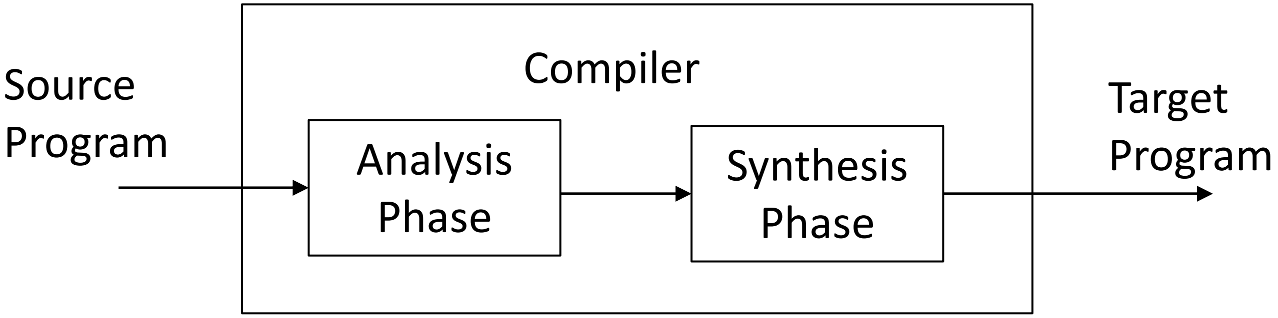
Context of a Compiler



A language-processing system

Analysis-Synthesis Model of Compilation

- There are two major parts of a compiler: **Analysis** and **Synthesis**
- **Analysis:** determines meaning of a source string
- **Synthesis:** constructs an equivalent target string



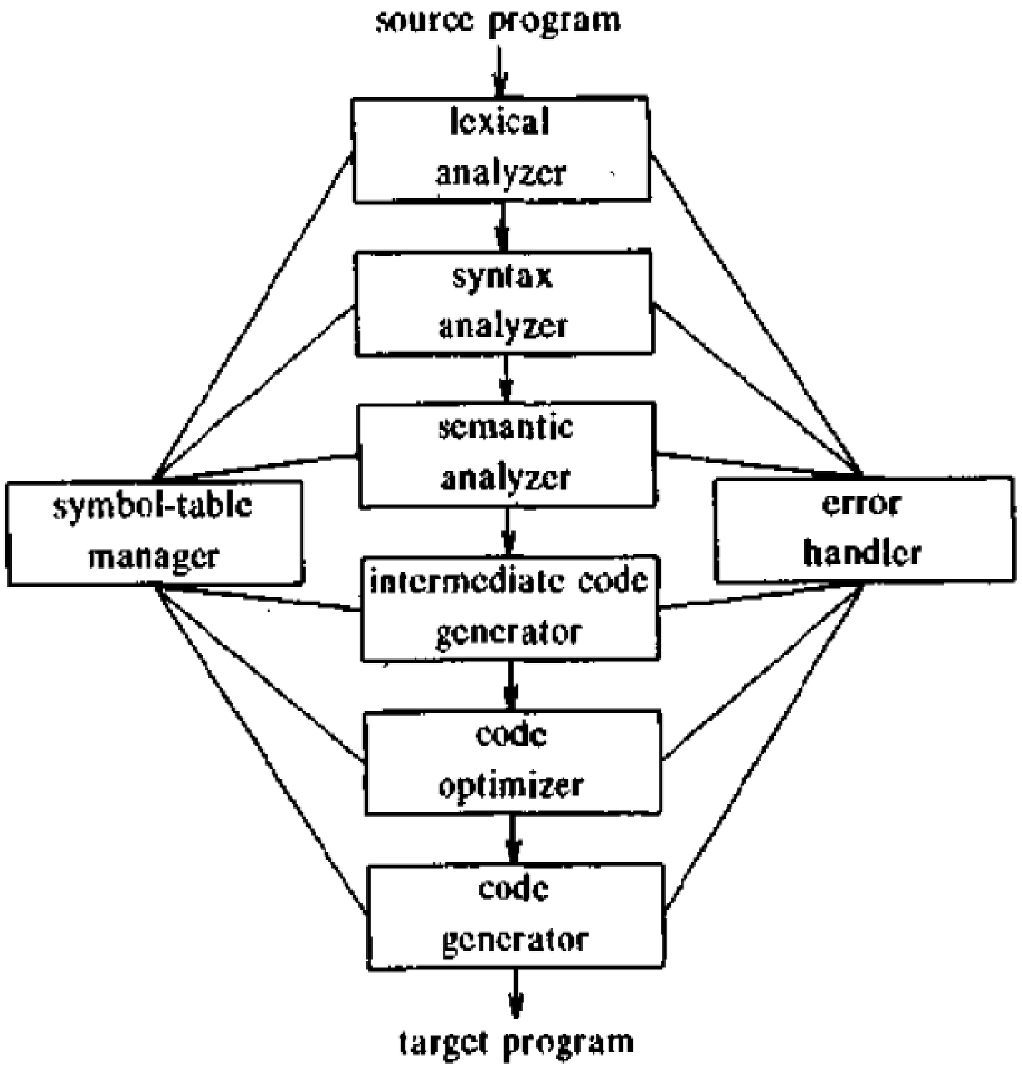
Analysis-Synthesis Model of Compilation

- Analysis phase
  - Breaks source program into constituent parts
  - Builds an intermediate representation from the source program.
- Analysis consists of three phases:
  - Determines the lexical constituents in a source string (lexical analysis)
  - Determines the structure of the source string (syntax analysis)
  - Determines the meaning of the source string (semantic analysis)
- The operations implied by the source program are determined and recorded in a hierarchical structure called the syntax tree.
- Collects information about the source program and stores it in a data structure called a symbol table
- Actions in lexical, syntax and sematic analysis phase are uniquely defined for a given language

Analysis-Synthesis Model of Compilation

- Synthesis phase
  - the equivalent target program is created from this intermediate representation and the information in the symbol table.
- The synthesis step consists of many instances where the actions depend on pragmatic aspects (aspects concerning execution environment)
  - OS interfaces
  - Target machine features such as instruction set, addressing modes, no. of registers, storage size etc.)
  - Design decisions taken by compiler designer
  - Number of passes
  - Whether optimization is to be performed
- Pragmatic aspects are unique for a given (source language, compiler) pair.
- The analysis part is often called the front end of the compiler; the synthesis part is the back end.

Phases of a Compiler



Lexical Analysis

- Lexical Analyzer reads the source program character by character and groups them into meaningful characters sequences called **lexemes**.
- For each lexeme, the lexical analyzer produces as output a token that it passes on to the subsequent phase, syntax analysis
- A token describes a pattern of characters having same meaning in the source program
  - identifiers, operators, keywords, numbers, delimiters etc.
- The lexical analyzer builds a descriptor for each token

Category Code	Attribute

eg: (id, #7)

- Puts information about identifiers into the symbol table.

Lexical Analysis

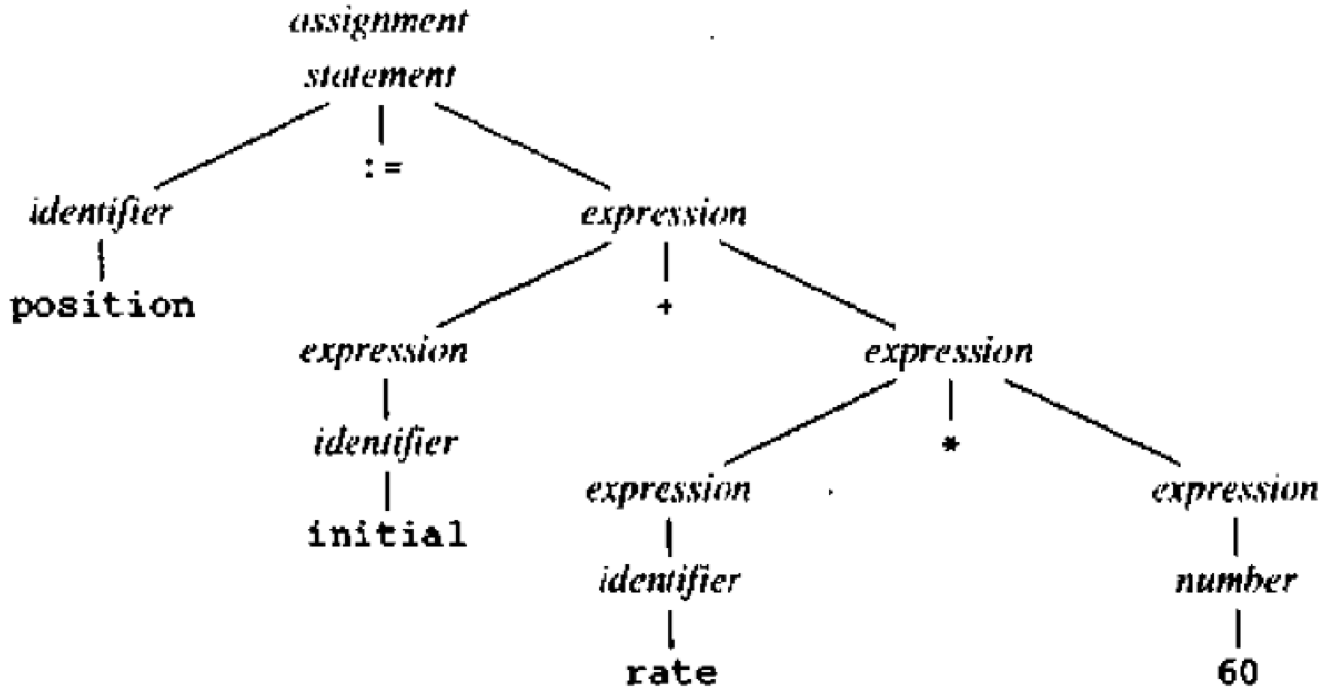
- *Eg: position := initial +rate*60*
- **Tokens:**
  1. The identifier ***position*** *(id, #1)*
  2. The assignment symbol ***:=***
  3. The identifier ***initial*** *(id, #2)*
  4. The ***+*** symbol
  5. The identifier ***rate*** *(id, #3)*
  6. The ***** symbol
  7. The number ***60***
- Blanks and comments are eliminated

Syntax Analysis

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- The tokens of the source program are grouped into grammatical phrases that are used by the compiler to synthesize output.

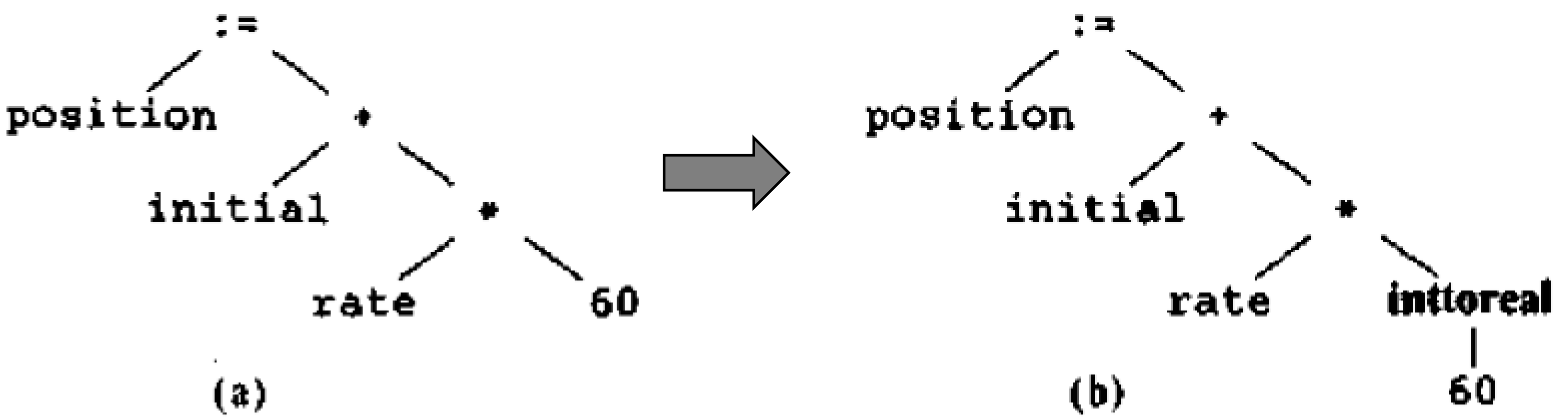
- A syntax analyzer is also called as a **parser**.

- A parse tree describes a syntactic structure



Semantic Analysis

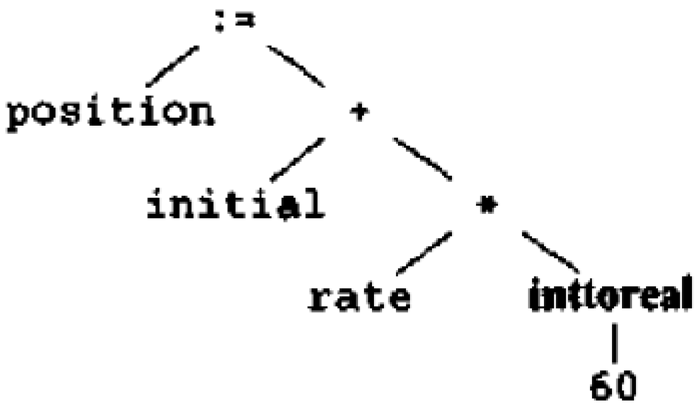
- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- It uses the hierarchical structure determined by syntax analysis phase to identify the operators and operands of the expressions and statements
- Type-checking is an important part of semantic analyzer.



Semantic analysis inserts a conversion from integer to real.

Intermediate Code Generation

- A compiler may produce explicit intermediate codes representing the source program.
- Properties:
  - Easy to produce
  - Easy to translate to target program



Syntax Tree

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Three address Code

Code Optimization

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Intermediate Code

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

Optimized Code

Code generation

- Produces the target language in a specific architecture.
- The target program is normally a relocatable machine code or assembly code
- Memory locations are selected for each of the variables
- Then intermediate instructions are each translated into a sequence of machine instructions that performs the same task

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

→

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

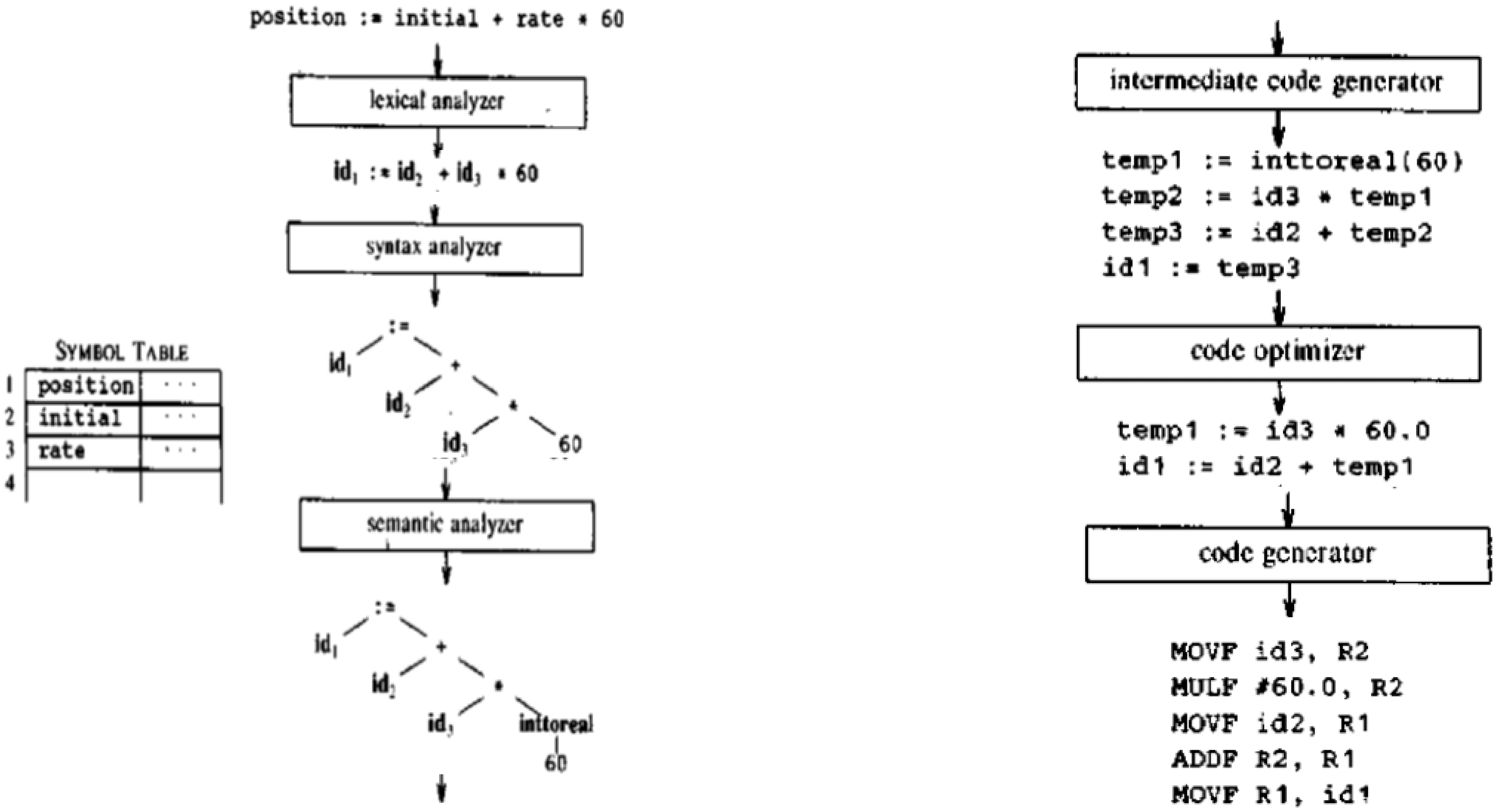
Symbol Table Management

- Records identifiers used in the source program and stores information about the attributes of the identifier
  - Identifier: type, scope
  - Procedure names: number and type of arguments, method of passing each argument, return type
- Data structure
  - Contains a record for each identifier with fields for the attributes of the identifier

Error Detection and Reporting

- Phases should deal with the errors so that compiler can proceed
- Large fraction of errors are handled by syntax and semantic analysis phases
- Lexical analysis phase can detect errors when the characters remaining in the input do not form any token
- Syntax error: token stream violates the syntax (rules) of the language
- Semantic error: Constructs have the right syntactic structure, but no meaning to operations
    - Real numbers to index an array

Translation of a Statement



Grouping of Phases

- **Front End and Back End**
- **Front end:** Phases that depend primarily on the source language and largely independent of the target machine
  - Lexical and syntax analysis phase, symbol table creation, semantic analysis, generation of intermediate code, error handling
- **Back end:** Portions of the compiler that depend on the target machine
  - Code optimization, code generation along with necessary error handling and symbol table operations
- To produce a compiler for the same source language on a different machine
  - Take front end and redo the back end

Grouping of Phases

- Passes
  - Several phases of compilation are usually grouped in a single pass
    - The activities of these phases are interleaved during the pass
  - The lexical analysis, syntax analysis, semantic analysis and intermediate code generation may be grouped in one pass
- Reducing the Number of Passes
  - Desirable to have relatively few passes since it takes time to read and write intermediate files
  - Grouping several phases requires keeping the entire program in memory because one may need information in a different order than the previous phase produces it.
    - Internal form of the program may be considerably larger than either the source program or target program

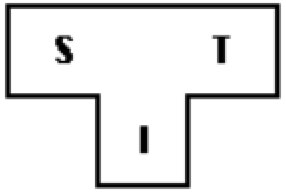
Bootstrapping

- Bootstrapping is widely used in the compilation development.
- Bootstrapping is the concept of obtaining a compiler for a language by using compilers for less powerful version(s), i.e. subsets of the same language
- Bootstrapping is used to produce a self-hosting compiler.
  - Self-hosting compiler is a type of compiler that can compile its own source code.
- A compiler can be characterized by three languages:

1. Source Language (S)
2. Target Language (T)
3. Implementation Language (I)

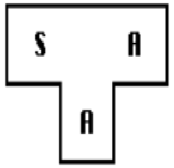
Bootstrapping

- The T- diagram shows a compiler ${}^{S}C_{I}^{T}$ for Source S, Target T, implemented in I
- To create a new language, L, for machine A:

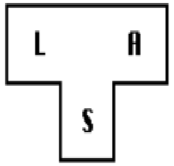


To create a new language, L, for machine A:

1. Create ${}^{S}C_{A}^{A}$, a compiler for a subset, S, of the desired language, L, using language A, which runs on machine A. (Language A may be assembly language.)

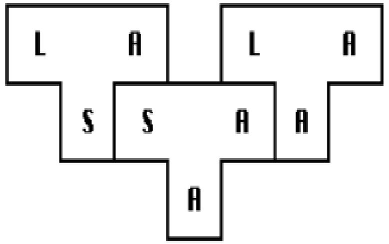


2. Create ${}^{L}C_{S}^{A}$, a compiler for language L written in a subset of L.



3. Compile ${}^{L}C_{S}^{A}$ using ${}^{S}C_{A}^{A}$ to obtain ${}^{L}C_{A}^{A}$, a compiler for language L, which runs on machine A and produces code for machine A.

$${}^{L}C_{S}^{A} \rightarrow {}^{S}C_{A}^{A} \rightarrow {}^{L}C^{A}$$

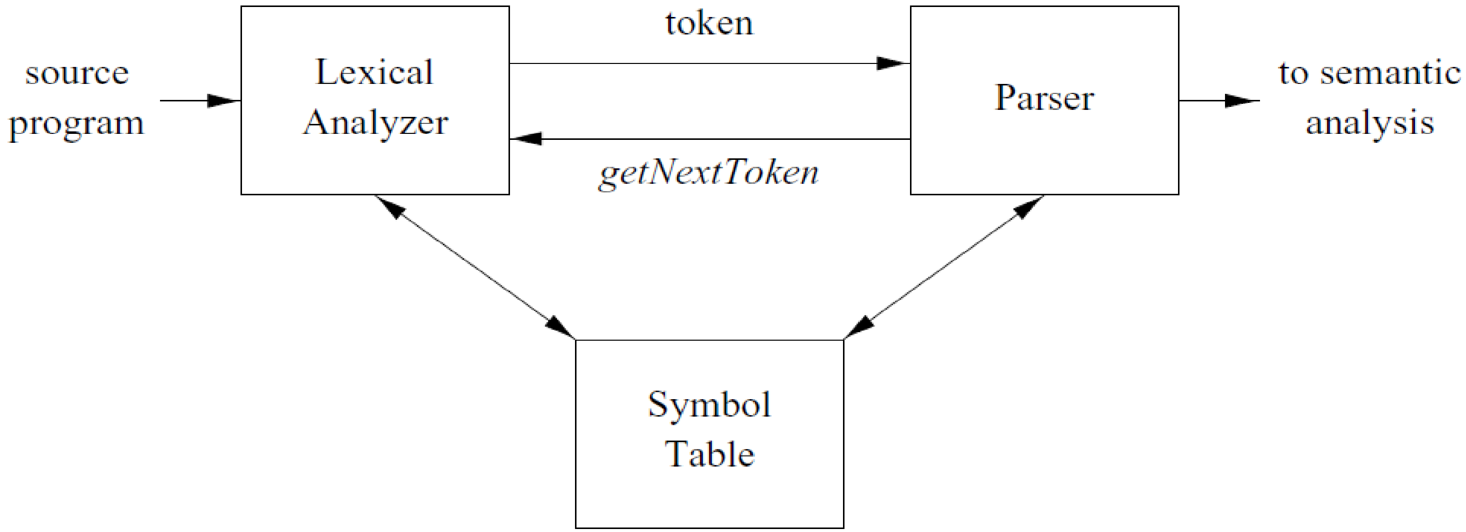


The process illustrated by the T-diagrams is called *bootstrapping* and can be summarized by the equation:

$$L_S A + S_A A = L_A A$$

Lexical Analysis

- First phase of compiler
- 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
- Lexical analyzer interacts with the symbol table as well.



Lexical Analysis

- Stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- Correlating error messages generated by the compiler with the source program.
  - For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
- In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions.
- If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Tokens, Patterns and Lexemes

- **Lexeme**-smallest logical units of a program such as A, B, +, - etc.
- **Tokens**- classes of similar lexemes such as identifier, constant, operator etc.
- **Pattern**-An informal or formal description of a token
  - Identifier-string of characters in which the first character is an alphabet followed by alphabets or digits
- *A **lexeme** is a sequence of characters in the source program that is matched by the **pattern** for a **token***
- A pattern serves two purposes
  - Precise description of specification of tokens
  - This description can be used to automatically generate a lexical analyzer

Tokens, Patterns and Lexemes

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	`const`	`const`
if	`if`	`if`
relation	`<, <=, =, <>, >, >=`	`<` or `<=` or `=` or `<>` or `>=` or `>`
id	`pi, count, D2`	letter followed by letters and digits
num	`3.1416, 0, 6.02E23`	any numeric constant
literal	`"core dumped"`	any characters between " and " except "

- Token: keywords, identifiers, operators, literal strings, constants, punctuation symbols

Attributes for Tokens

- Information about tokens are collected in associated attributes
  - Tokens influence parsing decisions
  - Attributes influence translation of tokens

- A token usually has a single attribute- a pointer to the symbol table entry, where information about the token is kept

Attributes for Tokens

```
E = M * C ** 2
```

- Token names and associated attribute values

<**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

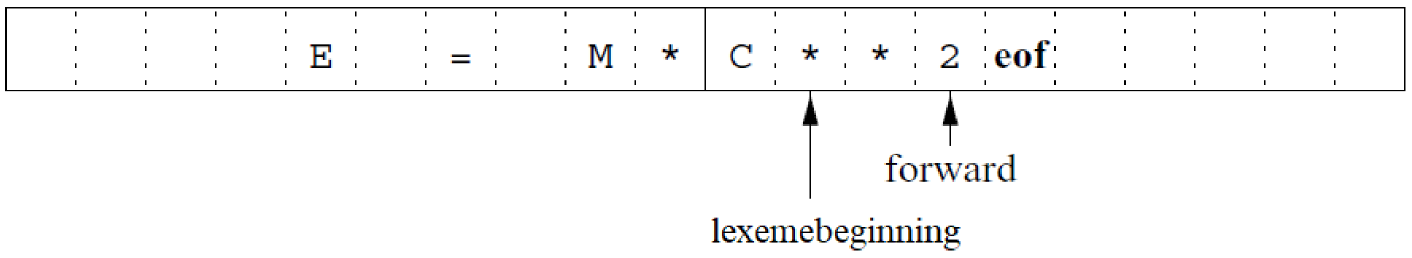
- Lexical Errors
  - Numeric literals too long
  - Identifiers that are too long (warning)
  - Ill-formed numeric literals
  - Input characters not in the source language
- Recovery actions
  - Delete one character from the remaining input.
  - Insert a missing character into the remaining input.
  - Replace a character by another character.
  - Transpose two adjacent characters

Input Buffering

- Used to read input characters and process tokens
- Two methods
  - Buffer pairs and sentinels
- Buffer pairs
- Use buffer divided into two N char halves
- N is the no. of characters on the disk block- 1024 or 4096
- Using one system read command we can read N characters into a buffer
- If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file
  - eof is different from any possible character of the source program

Input Buffering

- Two pointers to the input are maintained:
  - Pointer ***lexemebeginning***, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  - Pointer ***forward*** scans ahead until a pattern match is found
- The string of characters in between the two pointers is the current lexeme

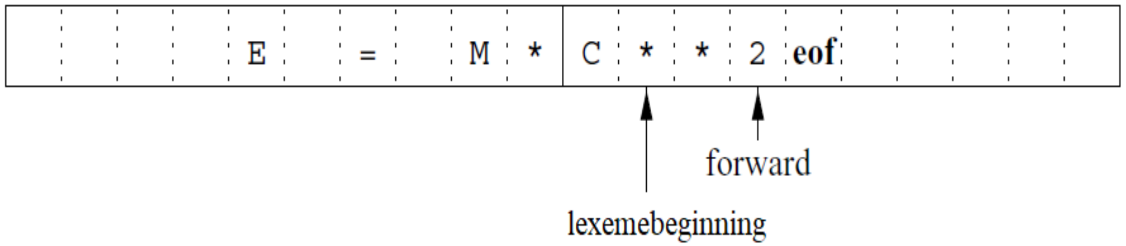


Input Buffering

- Initially both pointers point to the first character
- The forward pointer scans ahead until a match is found
- Once the next lexeme is determined, the forward pointer is set to the character at its right end.
- After the lexeme is processed, both the characters are set to the character immediately past the lexeme
- Comments and white spaces are treated as tokens that yield no token
- If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters
- If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new input characters and the forward pointer wraps around to the beginning of the buffer

Input Buffering

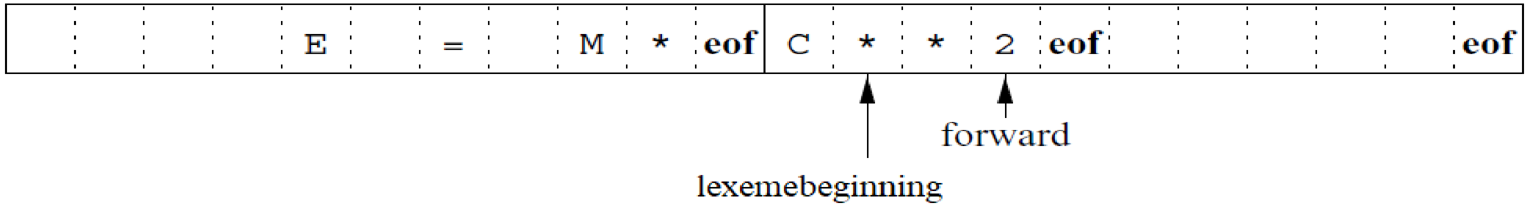
```
if forward at end of first half then begin
        reload second half;
        forward := forward + 1
end
else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half
end
else forward := forward + 1;
```



- Advancing forward requires that
  - we have to test whether we have reached the end of one of the buffers, and
  - if so, we must reload the other buffer from the input, and
  - move forward to the beginning of the newly loaded buffer.

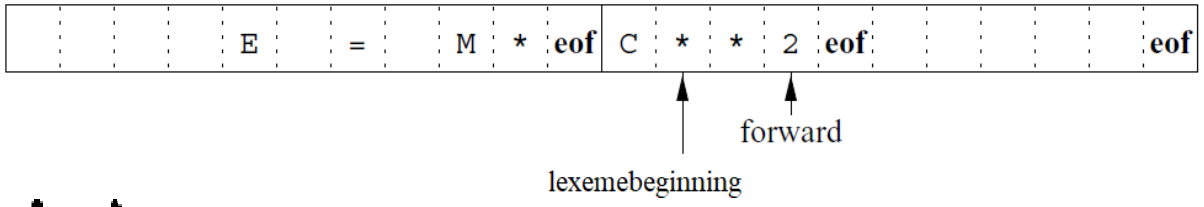
Input Buffering - Sentinels

- We must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer.
- Except at the end of buffer halves, the code requires two tests for each advance of forward pointer
- We can reduce the two tests to one if we extend each buffer to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.



Input Buffering - Sentinels

```
forward := forward + 1;
if forward↑ = eof then begin
        if forward at end of first half then begin
              reload second half;
              forward := forward + 1
        end
        else if forward at end of second half then begin
              reload first half;
              move forward to beginning of first half
        end
        else  /* eof within a buffer signifying end of input */
              terminate lexical analysis
end
```



Specification of Tokens

- Regular expressions
- **Strings and Languages**
- Alphabet or character class is any finite set of symbols
  - {0,1} binary alphabet
- **String over some alphabet** is some finite sequence of symbols drawn from that alphabet
- **Length of string s**, denoted by |s| is the number of occurrences of symbols in s
- **Empty string** denoted by ε is a string of length zero
- **Language** denotes any set of symbols over some alphabet
- **Concatenation of strings x and y**, denoted xy is the string formed by appending y to x
- Empty string is the **identity under concatenation**, εy=y

Strings and Languages

TERM	DEFINITION
prefix of s	A string obtained by removing zero or more trailing symbols of string s; e.g., **ban** is a prefix of **banana**.
suffix of s	A string formed by deleting zero or more of the leading symbols of s; e.g., **nana** is a suffix of **banana**.
substring of s	A string obtained by deleting a prefix and a suffix from s; e.g., **nan** is a substring of **banana**. Every prefix and every suffix of s is a substring of s, but not every substring of s is a prefix or a suffix of s. For every string s, both s and ϵ are prefixes, suffixes, and substrings of s.
proper prefix, suffix, or substring of s	Any nonempty string x that is, respectively, a prefix, suffix, or substring of s such that $s \neq x$.
subsequence of s	Any string formed by deleting zero or more not necessarily contiguous symbols from s; e.g., **baaa** is a subsequence of **banana**.

Operations on Languages

- Let L={A, B,…, Z, a, b,…, z} and D={0, 1,2, …, 9}

1. $L \cup D$ is the set of letters and digits.
2. LD is the set of strings consisting of a letter followed by a digit.
3. L^4 is the set of all four-letter strings.
4. L^* is the set of all strings of letters, including ϵ, the empty string.
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

Operations on Languages

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Regular Expressions

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.

2. If a is a symbol in Σ, then $\mathbf{a}$ is a regular expression, and $L(\mathbf{a}) = \{a\}$, that is, the language with one string, of length one, with a in its one position.

3. $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.

4. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.

5. $(r)^*$ is a regular expression denoting $\big(L(r)\big)^*$.

6. (r) is a regular expression denoting $L(r)$. This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

Regular Expressions

- A language denoted by a regular expression is a regular set

a) The unary operator $*$ has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

c) $|$ has lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression $(\mathbf{a})|((\mathbf{b})^*(\mathbf{c}))$ by $\mathbf{a}|\mathbf{b}^*\mathbf{c}$. Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.

If two regular expressions r and s denote the same regular set, we say they are *equivalent* and write $r = s$. For instance, $(\mathbf{a}|\mathbf{b}) = (\mathbf{b}|\mathbf{a})$.

Regular Expressions

Let $\Sigma = \{a, b\}$.

1. The regular expression $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.
2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.
3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \dots\}$.
4. $(\mathbf{a}|\mathbf{b})^*$ denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$. Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.
5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \dots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

Algebraic Laws for Regular Expressions

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

Regular Definitions

- For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols.
- If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$\begin{array}{lcl} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \cdots & \\ d_n & \rightarrow & r_n \end{array}$$

where each d_i is a distinct name and each r_i is a regular expression over the symbols $\Sigma U(d_1, d_2, \ldots d_{i-1})$

Regular Definitions

$$
\begin{aligned}
\textbf{letter} &\rightarrow \mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{Z} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{z} \\
\textbf{digit} &\rightarrow \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\
\textbf{id} &\rightarrow \textbf{letter} \ (\ \textbf{letter} \mid \textbf{digit} \ )^* \\
\textbf{digit} &\rightarrow \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\
\textbf{digits} &\rightarrow \textbf{digit} \ \textbf{digit}^* \\
\textbf{optional\_fraction} &\rightarrow \ . \ \textbf{digits} \mid \epsilon \\
\textbf{optional\_exponent} &\rightarrow (\ \mathtt{E} \ (\ + \mid - \mid \epsilon \ ) \ \textbf{digits} \ ) \mid \epsilon \\
\textbf{num} &\rightarrow \textbf{digits} \ \ \textbf{optional\_fraction} \ \ \textbf{optional\_exponent}
\end{aligned}
$$

Extensions of Regular Expressions

- One or more instances: the unary prefix operator +
  - If r is a regular expression denoting language L(r), the r^+ denotes the language $(L(r))^+$
  - Regular expression a^+ represents set of all strings of one or more a's
- Zero or one instance: the unary prefix operator ?
  - The notation r? is the shorthand for r|ε
- Character class:
  - The notation [abc] represents the regular expression a|b|c
  - An abbreviated character class [a-z] denotes a|b|…….|z
  - Identifier regular expression: [A-Za-z] [A-Za-z0-9]*

Recognition of Tokens

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

- Regular Definitions

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits\ (\texttt{.}\ digits)?\ (\ \texttt{E}\ [\texttt{+-}]?\ digits\ )? \\
letter & \rightarrow & \texttt{[A-Za-z]} \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^{*} \\
if & \rightarrow & \texttt{if} \\
then & \rightarrow & \texttt{then} \\
else & \rightarrow & \texttt{else} \\
relop & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

Recognition of Tokens

- The lexical analyzer will
  - recognize the keywords *if, then, else*
  - the lexemes denoted by *relop, id* and *num*
- Assume lexemes are separated by white space consisting of non null sequence of tabs, blanks and newline.
- The lexical analyzer will strip out white spaces

$$ws \rightarrow (\ \mathbf{blank} \mid \mathbf{tab} \mid \mathbf{newline}\ )^{+}$$

- If a match for *ws* is found, no token is returned to the parser.
- The lexical analyzer proceeds to find a token following the white space and returns that to the parser

Recognition of Tokens

Tokens, their patterns, and attribute values

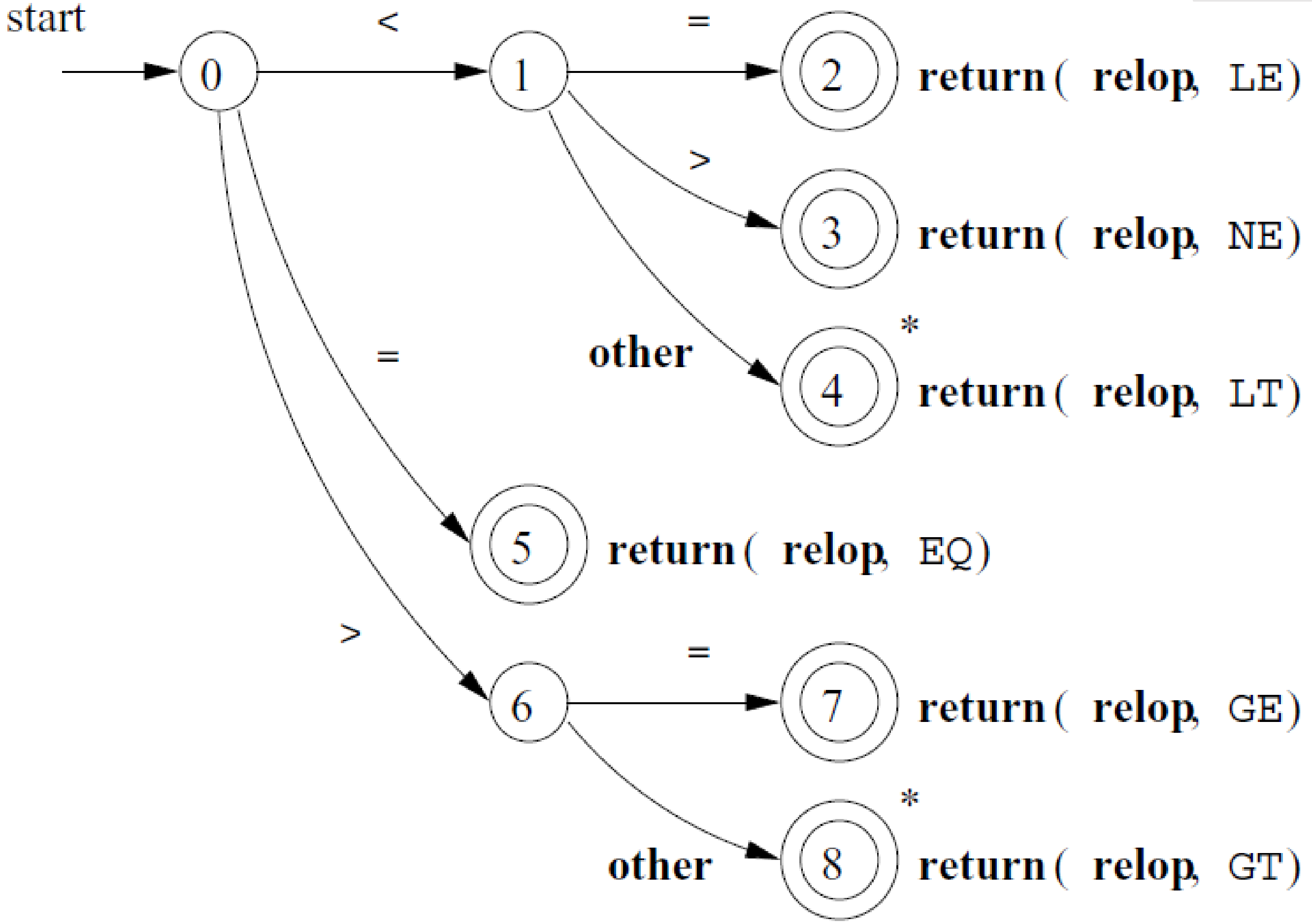
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	—	—
if	**if**	—
then	**then**	—
else	**else**	—
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Transition Diagrams

- Depict the actions taken place when lexical analyzer is called by the parser to get the next token.
- **States**- represented with circles
- **Labeled edges** connect the states
- Transition diagram is **deterministic**

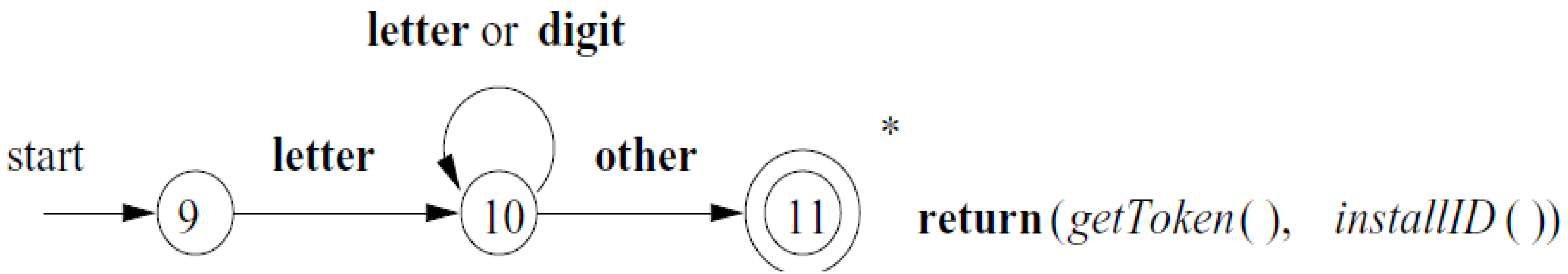
Transition Diagram for *relop*

$relop \rightarrow < \mid > \mid <= \mid >= \mid = \mid <>$



Recognition of Reserved Words and Identifiers

- To separate keywords from identifiers, initialize the symbol table.
  - Enter keywords into symbol table along with a token to be returned when one of these strings is recognized.
- The return state next to the accepting state uses *gettoken()* and *installID()* to obtain the token and attribute values to be returned.

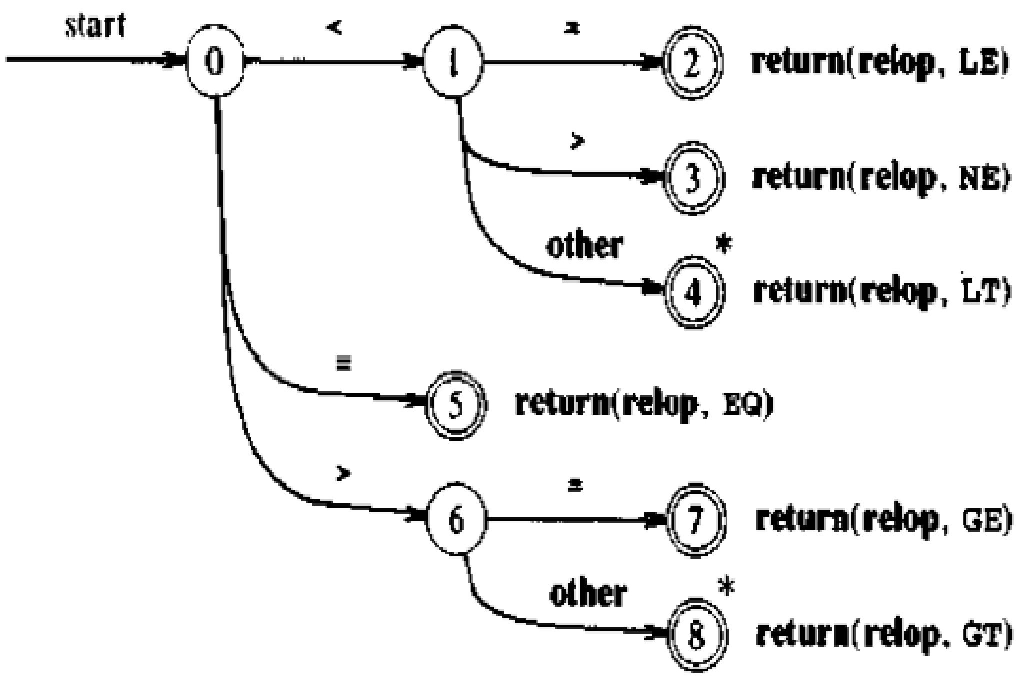


A transition diagram for **id**'s and keywords

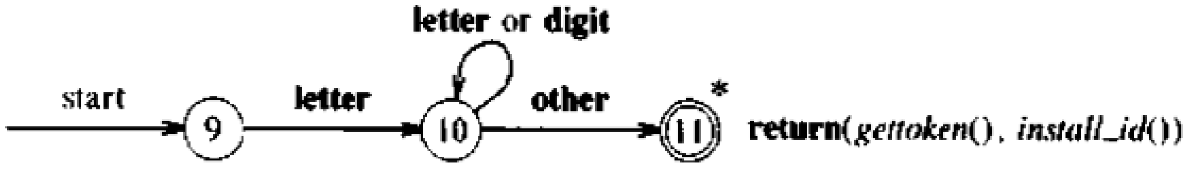
Recognition of Reserved Words and Identifiers

- *installID()* has access to the buffer where identifier lexeme has been located
  - The symbol table is examined and if the lexeme is marked as a **keyword** it returns zero
  - If the **lexeme is found** and is a **program variable***, installID()* returns a pointer to the symbol table entry
  - If the **lexeme is not found in the symbol table**, it is installed as a variable and a pointer to the newly created entry is returned
- The procedure *gettoken()* looks for the lexeme in the symbol table.
  - If the lexeme is a keyword, the corresponding token is returned, otherwise the token *id* is returned.

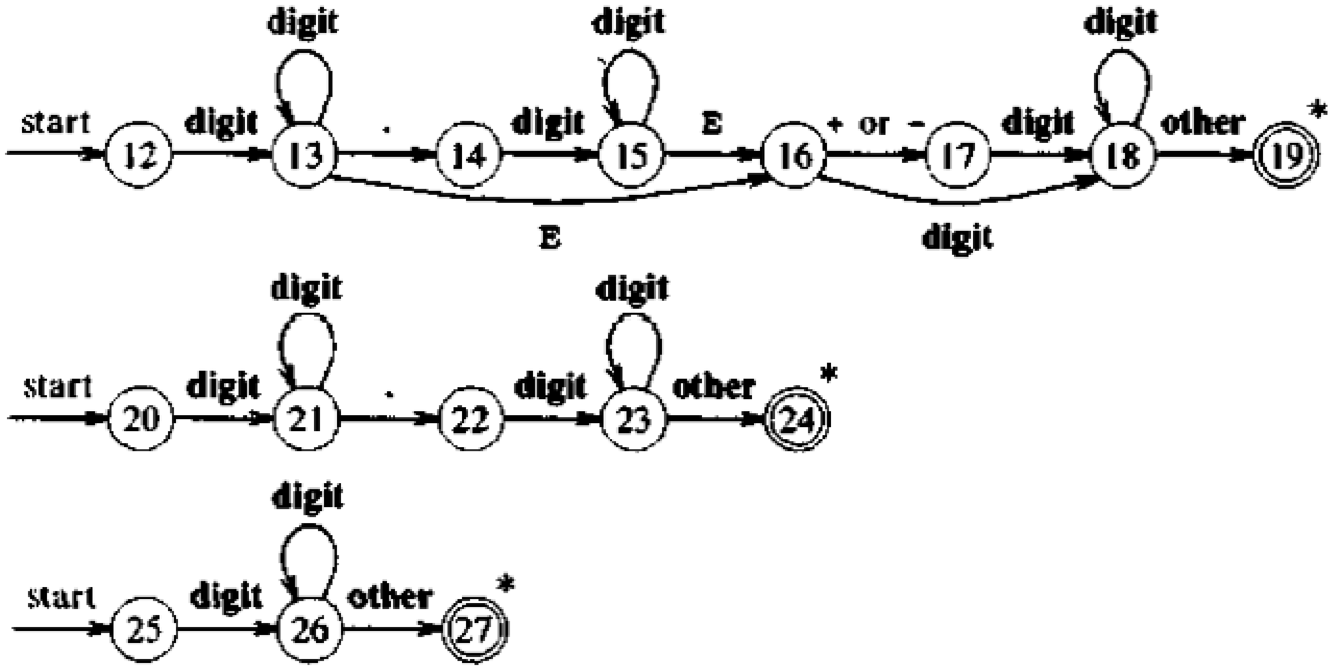
Implementing a Transition Diagram



Transition diagram for relational operators.



Transition diagram for identifiers and keywords.



Transition diagrams for unsigned numbers in Pascal.

Implementing a Transition Diagram

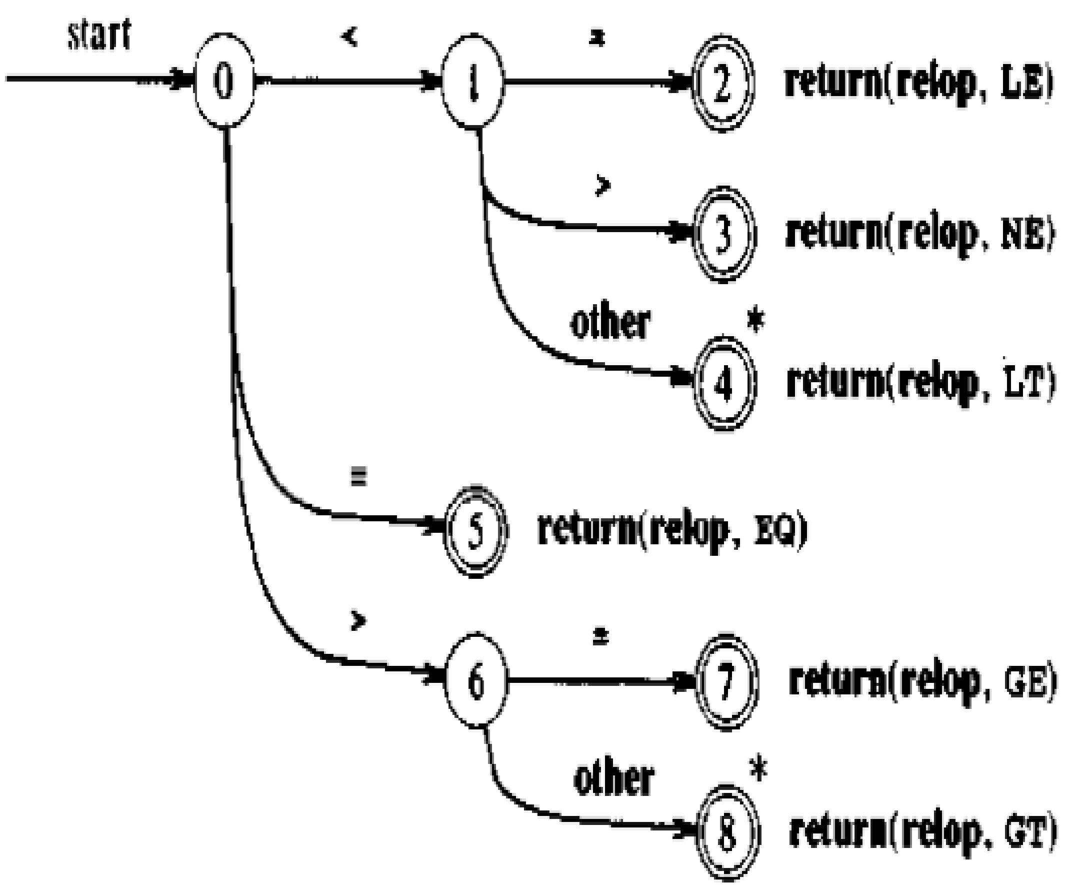
C code to find next start state.

```
int state = 0, start = 0;
int lexical_value;
    /* to "return" second component of token */

int fail()
{
    forward = token_beginning;
    switch (start) {
        case 0:   start = 9; break;
        case 9:   start = 12; break;
        case 12:  start = 20; break;
        case 20:  start = 25; break;
        case 25:  recover(); break;
        default:   /* compiler error */
    }
    return start;
}
```

- Transition diagrams are useful in two ways
  - They serve as a precise specification of tokens
  - They also aid in structuring the lexical analyzer program

Implementing a Transition Diagram

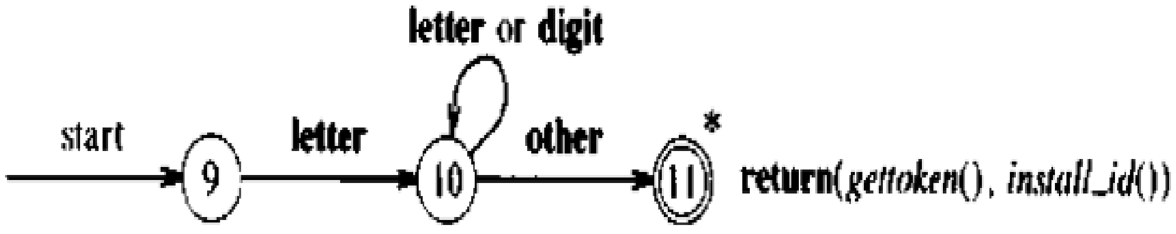
```
token nexttoken()
{   while(1) {
        switch (state) {
        case 0:   c = nextchar();
            /* c is lookahead character */
            if (c==blank || c==tab || c==newline) {
                state = 0;
                lexeme_beginning++;
                    /* advance beginning of lexeme */
            }
            else if (c == '<') state = 1;
            else if (c == '=') state = 5;
            else if (c == '>') state = 6;
            else state = fail();
            break;

            .../* cases 1-8 here */
```



Transition diagram for relational operators.

Implementing a Transition Diagram

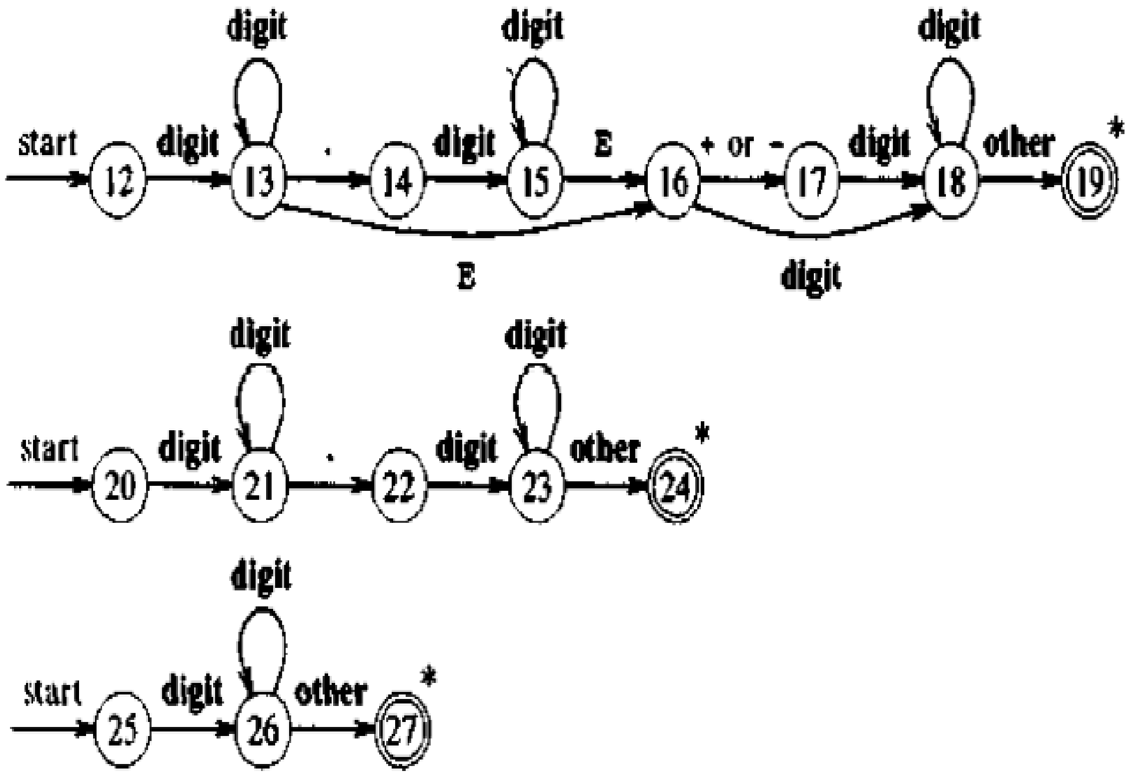
```
case 9:   c = nextchar();
    if (isletter(c)) state = 10;
    else state = fail();
    break;
case 10:  c = nextchar();
    if (isletter(c)) state = 10;
    else if (isdigit(c)) state = 10;
    else state = 11;
    break;
case 11:  retract(1); install_id();
    return ( gettoken() );

    .../* cases 12-24 here */
```



Transition diagram for identifiers and keywords.

Implementing a Transition Diagram

```
case 25:  c = nextchar();
    if  (isdigit(c)) state = 26;
    else state = fail();
    break;
case 26:  c = nextchar();
    if (isdigit(c)) state = 26;
    else state = 27;
    break;
case 27:  retract(1); install_num();
    return ( NUM );
  }
}
```



Transition diagrams for unsigned numbers in Pascal.