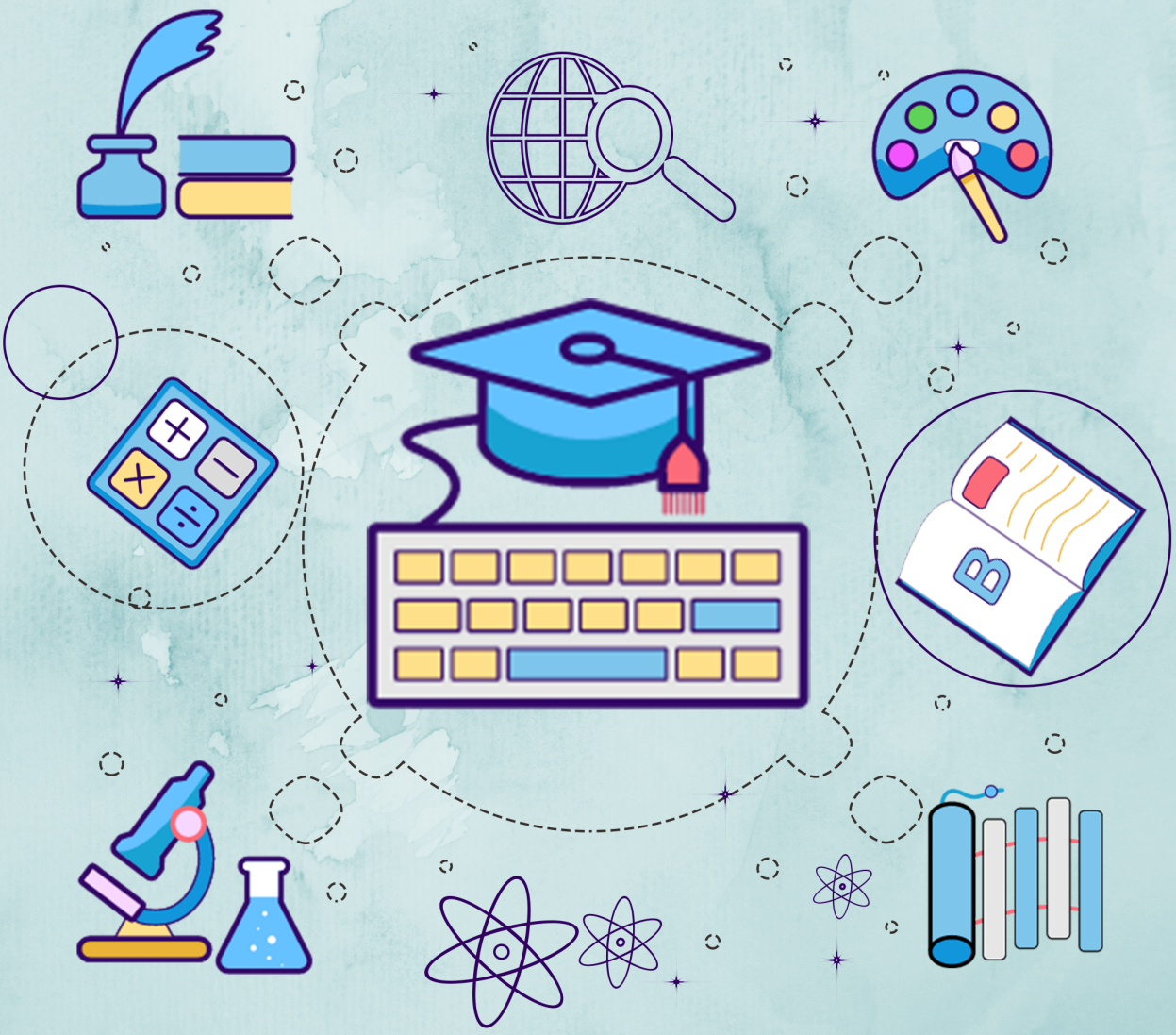
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

COMPILER DESIGN

CST 302

Module 1

Related Link :

- KTU S6 CSE NOTES 2019 SCHEME
- KTU S6 SYLLABUS CSE COMPUTER SCIENCE
- KTU PREVIOUS QUESTION BANK S6 CSE SOLVED
- KTU CSE TEXTBOOKS S6 B.TECH PDF DOWNLOAD
- KTU S6 CSE NOTES | SYLLABUS | QBANK | TEXTBOOKS DOWNLOAD

MODULE 1

Introduction to compilers – Analysis of the source program, Phases of a compiler, Grouping of phases, compiler writing tools – bootstrapping

Lexical Analysis:

The role of Lexical Analyzer, Input Buffering, Specification of Tokens using Regular Expressions, Review of Finite Automata, Recognition of Tokens.

1.1 INTRODUCTION TO COMPILERS

- A compiler is a 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).
- An important role of the compiler is to report any errors in the source program that it detects during the translation process.

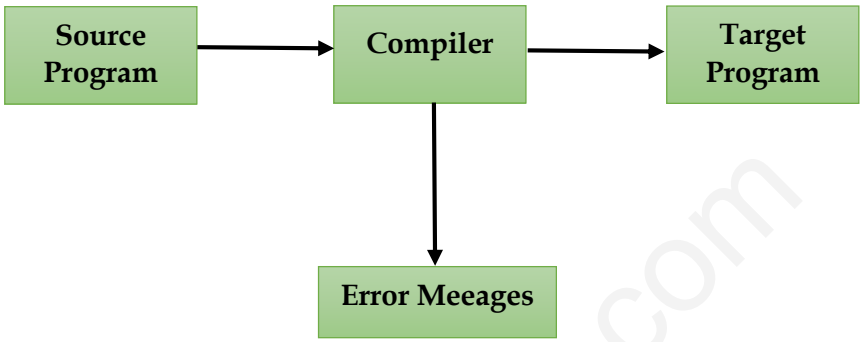


Fig: Compiler

- Compilers are sometimes classified as single pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform.

1.1.1 ANALYSIS OF THE SOURCE PROGRAM

- In compiling, analysis consists of three phases:
  - **Lexical Analysis**
  - **Syntax Analysis**
  - **Semantic Analysis**

Lexical Analysis

- In a compiler linear analysis is called lexical analysis or scanning. The lexical analysis phase reads the characters in the source program and grouped into tokens that are sequence of characters having a collective meaning.

EXAMPLE

position : = initial + rate * 60

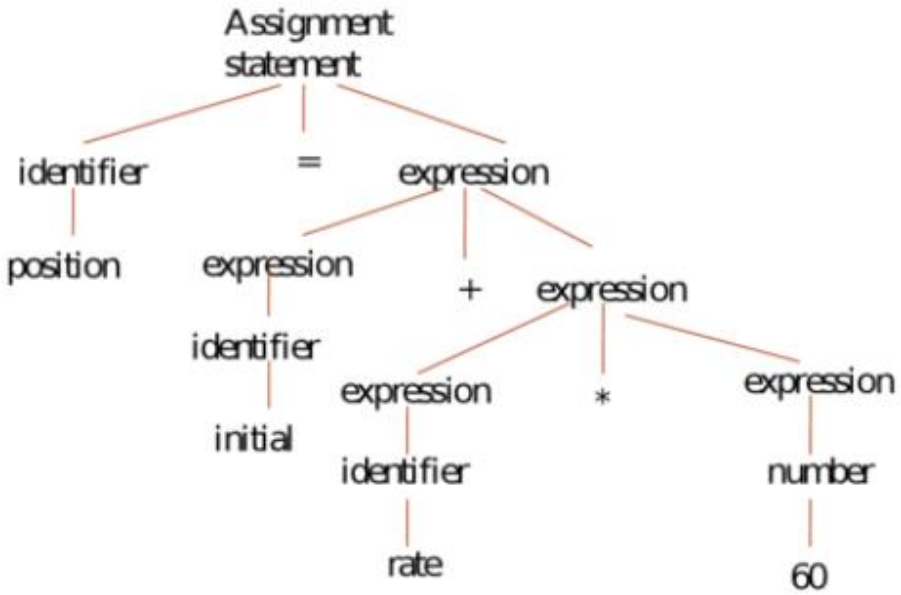
This can be grouped into the following tokens;

1. **The identifier position.**
2. **The assignment symbol : =**
3. **The identifier initial**
4. **The plus sign**
5. **The identifier rate**
6. **The multiplication sign**
7. **The number 60**

- Blanks separating characters of these tokens are normally eliminated during lexical analysis.

Syntax Analysis

- Hierarchical Analysis is called parsing or syntax analysis.
- It involves grouping the tokens of the source program into grammatical phrases that are used by the complier to synthesize output. They are represented using a syntax tree.
- A syntax tree is the tree generated as a result of syntax analysis in which the interior nodes are the operators and the exterior nodes are the operands. This analysis shows an error when the syntax is incorrect.



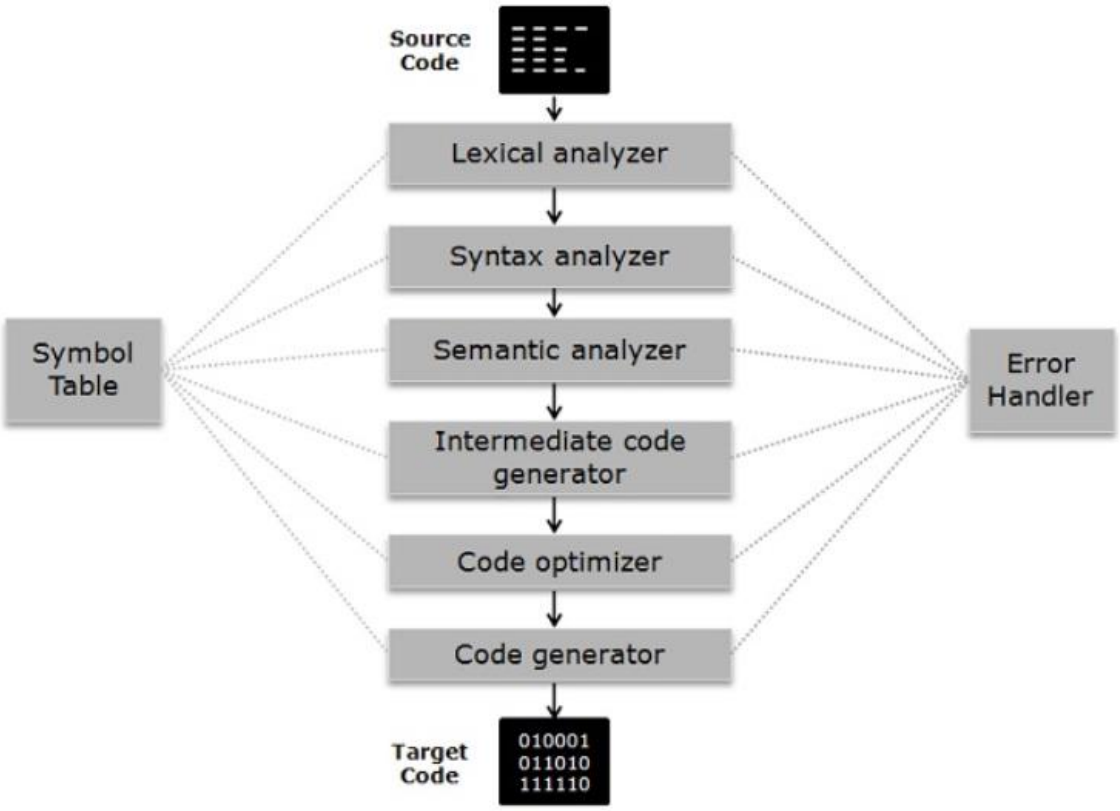
Semantic Analysis

- This phase checks the source program for semantic errors and gathers type information for subsequent code generation phase.
- An important component of semantic analysis is type checking.
- Here the compiler checks that each operator has operands that are permitted by the source language specification.

1.1.2 PHASES OF A COMPILER

The phases include:

- **Lexical Analysis**
- **Syntax Analysis**
- **Semantic Analysis**
- **Intermediate Code Generation**
- **Code Optimization**
- **Target Code Generation**



Lexical Analysis

- The first phase of a compiler is called lexical analysis or scanning.
- The lexical analyzer reads the stream of characters making up the source program and 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 >

  that it passes on to the subsequent phase, syntax analysis.
- In the token, the first component token- name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.
- Information from the symbol-table entry 'is needed for semantic analysis and code generation.
- For example, suppose a source program contains the assignment statement

  position = initial + rate * 60
- The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

  1. position is a lexeme that would be mapped into a token <id, 1>, where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol- table entry for an identifier holds information about the identifier, such as its name and type.
  2. The assignment symbol = is a lexeme that is mapped into the token < = >. Since this token needs no attribute-value, we have omitted the second component.
  3. initial is a lexeme that is mapped into the token < id, 2> , where 2 points to the symbol-table entry for initial .
  4. + is a lexeme that is mapped into the token <+>.
  5. rate is a lexeme that is mapped into the token < id, 3 >, where 3 points to the symbol-table entry for rate.
  6. * is a lexeme that is mapped into the token <* > .
  7. 60 is a lexeme that is mapped into the token <60>
- Blanks separating the lexemes would be discarded by the lexical analyzer. The representation of the assignment statement position = initial + rate * 60 after lexical analysis as the sequence of tokens as:

  < id, l > < = > <id, 2> <+> <id, 3> < * > <60>

Token : Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

- Identifiers
- keywords
- operators
- special symbols
- constants

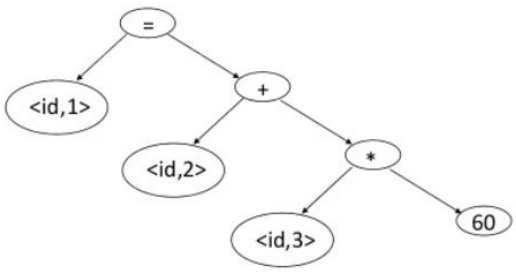
Pattern : A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme : A lexeme is a sequence of characters in the source program that is matched by 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"**

Syntax Analysis

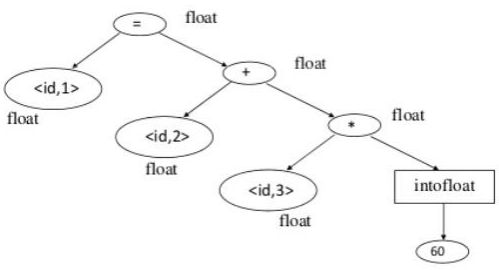
- The second phase of the compiler is syntax analysis or parsing.
- The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.
- The syntax tree for above token stream is:



- The tree has an interior node labeled with (id, 3) as its left child and the integer 60 as its right child.
- The node (id, 3) represents the identifier rate.
- The node labeled * makes it explicit that we must first multiply the value of rate by 60.
- The node labeled + indicates that we must add the result of this multiplication to the value of initial.
- The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier position.

Semantic Analysis

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.
- For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.
- Some sort of type conversion is also done by the semantic analyzer.
- For example, if the operator is applied to a floating point number and an integer, the compiler may convert the integer into a floating point number.
- In our example, suppose that position, initial, and rate have been declared to be floating- point numbers, and that the lexeme 60 by itself forms an integer.
- The semantic analyzer discovers that the operator * is applied to a floating-point number rate and an integer 60.
- In this case, the integer may be converted into a floating-point number.
- In the following figure, notice that the output of the semantic analyzer has an extra node for the operator inttofloat , which explicitly converts its integer argument into a floating-point number.



Intermediate Code Generation

- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.
- Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.
- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine.
- This intermediate representation should have two important properties:
  - It should be simple and easy to produce
  - It should be easy to translate into the target machine.
- In our example, the intermediate representation used is three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction.

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

Code Optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- The objectives for performing optimization are: faster execution, shorter code, or target code that consumes less power.
- In our example, the optimized code is:

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

Code Generator

- The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

- A crucial aspect of code generation is the judicious assignment of registers to hold variables.
- If the target language is assembly language, this phase generates the assembly code as its output.
- In our example, the code generated is:

```
LDF   R2, id3
MULF  R2, #60.0
LDF   R1, id2
ADDF  R1, R2
STF   id1, R1
```

- The first operand of each instruction specifies a destination.
- The F in each instruction tells us that it deals with floating-point numbers.
- The above code loads the contents of address id3 into register R2, then multiplies it with floating-point constant 60.0.
- The # signifies that 60.0 is to be treated as an immediate constant.
- The third instruction moves id2 into register Rl and the fourth adds to it the value previously computed in register R2.
- Finally, the value in register Rl is stored into the address of idl , so the code correctly implements the assignment statement **position = initial + rate * 60.**

Symbol Table

- An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.
- These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.
- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.
- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

Error Detection And Reporting

- Each phase can encounter errors.
- However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.

- A compiler that stops when it finds the first error is not a helpful one.

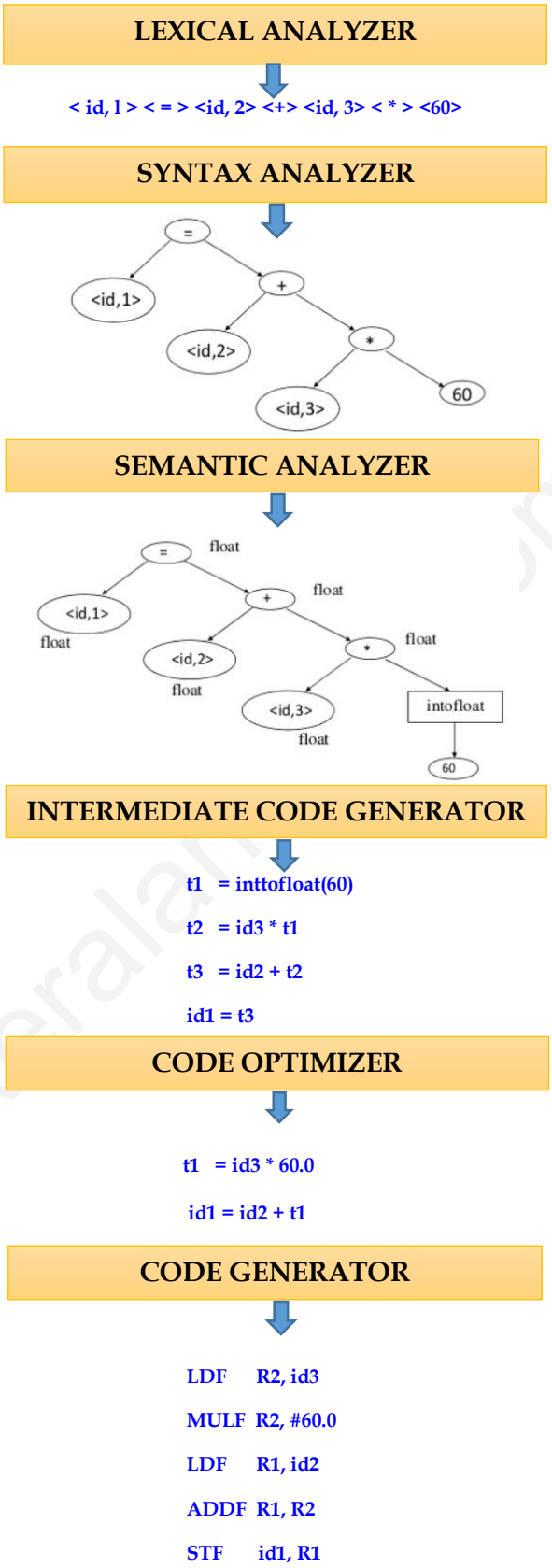


Figure : ***Translation of an assignment statement***

1.1.3 GROUPING OF PHASES

- The process of compilation is split up into following phases:
  - **Analysis Phase**
  - **Synthesis phase**

Analysis Phase

Analysis Phase performs 4 actions namely:

a. **Lexical analysis**

b. **Syntax Analysis**

c. **Semantic analysis**

d. **Intermediate Code Generation**

- The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them.
- It then uses this structure to create an intermediate representation of the source program.
- If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.
- The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.
- The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them.
- It then uses this structure to create an intermediate representation of the source program.
- If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.
- The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

Synthesis Phase

Synthesis Phase performs 2 actions namely:

a. **Code Optimization**

b. **Code Generation**

- 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 the front end of the compiler; the synthesis part is the back end.

1.1.4 COMPILER WRITING TOOLS

- Compiler writers use software development tools and more specialized tools for implementing various phases of a compiler. Some commonly used compiler construction tools include the following.
    - **Parser Generators**
    - **Scanner Generators**
    - **Syntax-directed translation engine**
    - **Automatic code generators**
    - **Data-flow analysis Engines**
    - **Compiler Construction toolkits**

Parser Generators.

Input : Grammatical description of a programming language

Output : Syntax analyzers.

- These produce syntax analyzers, normally from input that is based on a context-free grammar.
- In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler.
- This phase is one of the easiest to implement.

Scanner Generators

Input : Regular expression description of the tokens of a language

Output : Lexical analyzers.

- These automatically generate lexical analyzers, normally from a specificalion based on regular expressions.
- The basic organization of the resulting lexical analyzer is in effect a finite automaton.

Syntax-directed Translation Engines

Input : Parse tree.

Output : Intermediate code.

- These produce collections of routines that walk the parse tree, generating intermediate code.

- The basic idea is that one or more "translations" are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbour nodes in the tree.

Automatic Code Generators

Input : Intermediate language.

Output : Machine language.

- Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.
- The rules must include sufficient detail that we can handle the different possible access methods for data.

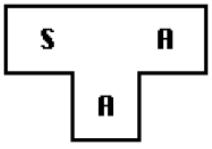
Data-flow Analysis Engines

- Data-flow analysis engine gathers the Information that is, the values transmitted from one part of a program to each of the other parts.
- Data-flow analysis is a key part of code optimization.

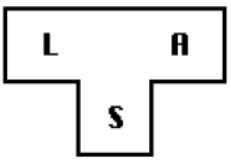
1.1.4.1 BOOTSTRAPPING

- Bootstrapping is widely used in the compilation development.
- Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.
- Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler to compile everything else as well as future versions of itself.
- A compiler is characterized by three languages:
  - **Source Language**
  - **Target Language**
  - **Implementation Language**
- Notation : ${}^{S}C_{I}^{T}$ represents a compiler for Source **S** , Target **T** , implemented in **I** . The **T**-diagram shown above is also used to depict the same compiler.
- **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.)

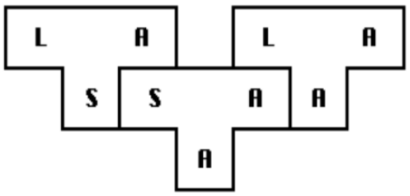


2. Create ${}^{L}C_{S}^{A}$, a compiler for language L written in a subset of L.



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.

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

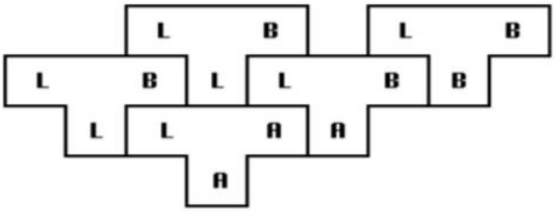


- The process illustrated by the T-diagrams is called bootstrapping and can be summarized by the equation:

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

To produce a compiler for a different machine B:

1. Convert ${}^{L}C_{S}^{A}$ into ${}^{L}C_{L}^{B}$ (by hand, if necessary). Recall that language S is a subset of language L.
2. Compile ${}^{L}C_{L}^{B}$ to produce ${}^{L}C_{A}^{B}$, a *cross-compiler* for L which runs on machine A and produces code for machine B.
3. Compile ${}^{L}C_{L}^{B}$ with the cross-compiler to produce ${}^{L}C_{B}^{B}$, a compiler for language L which runs on machine B.

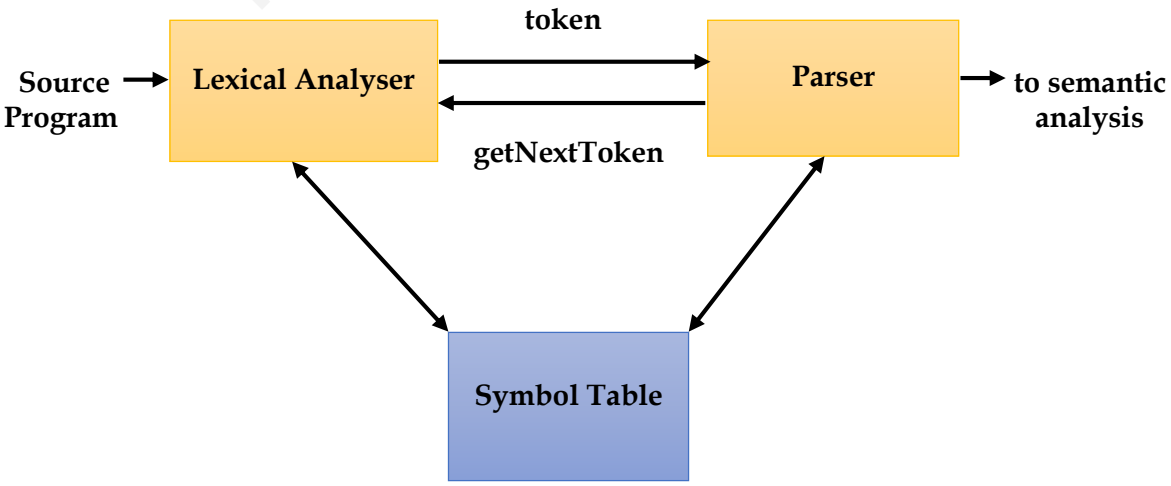


1.2 LEXICAL ANALYSIS

1.2.1 ROLE OF LEXICAL ANALYSIS

- As 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.

- Lexical Analyzer also interacts with the symbol table.
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.
- These interactions are given in following figure.
- Commonly, the interaction is implemented by having the parser call the lexical analyzer.
- The call, suggested by the **getNextToken** command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.



Interactions between lexical analyser and parser

Other tasks of Lexical Analyzer

1. Stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
2. Correlatingerrormessagesgeneratedbythecompilerwiththesourceprogram.Forinstance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
3. If the source program uses a macro-pre-processor, the expansion of macros may also be performed by the lexical analyzer.

Issues In Lexical Analysis

Following are the reasons why lexical analysis is separated from syntax analysis

Simplicity Of Design

The separation of lexical analysis and syntactic analysis often allows us to simplify at least one of these tasks. The syntax analyzer can be smaller and cleaner by removing the lowlevel details of lexical analysis

Efficiency

Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

Portability

Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

Attributes For Tokens

- Sometimes a token need to be associate with several pieces of information.
- The most important example is the token id, where we need to associate with the token a great deal of information.
- Normally, information about an identifier - e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table.
- Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Lexical Errors

- A character sequence that can't be scanned into any valid token is a lexical error.

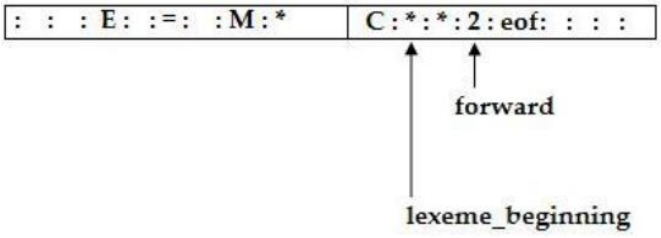
- Suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.
- The simplest recovery strategy is "panic mode" recovery.
- We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.
- This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.
- Other possible error-recovery actions are:
    1. **Delete one character from the remaining input.**
    2. **Insert a missing character into the remaining input.**
    3. **Replace a character by another character.**
    4. **Transpose two adjacent characters.**
- Transformations like these may be tried in an attempt to repair the input.
- The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation.
- A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes, but this approach is considered too expensive in practice to be worth the effort.

1.2.2 INPUT BUFFERING

- To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.
- Hence a two-buffer scheme is introduced to handle large lookaheads safely.
- Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer end have been adopted.
- There are three general approaches for the implementation of a lexical analyzer:
    a. By using a lexical-analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.
    b. By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.
    c. By writing the lexical analyzer in assembly language and explicitly managing the reading of input.
- The three choices are listed in order of increasing difficulty for the implementer

Buffer Pairs

- Because of large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.
- Fig shows the buffer pairs which are used to hold the input data.



An input buffer in two halves

Scheme

- Consists of two buffers, each consists of N-character size which are reloaded alternatively.
- N-Number of characters on one disk block.
- N characters are read from the input file to the buffer using one system read command.
- *eof* is inserted at the end if the number of characters is less than N.

Pointers

- Two pointers ***lexemeBegin*** and ***forward*** are maintained.
- lexeme Begin points to the beginning of the current lexeme which is yet to be found.
- forward scans ahead until a match for a pattern is found.
- Once a lexeme is found, ***lexemebegin*** is set to the character immediately after the lexeme which is just found and ***forward*** is set to the character at its right end.
- Current lexeme is the set of characters between two pointers.

```
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 second half;
        move forward to beginning of first half
end
else forward := forward + 1;
```

Code to advance forward pointer

Disadvantages Of This Scheme

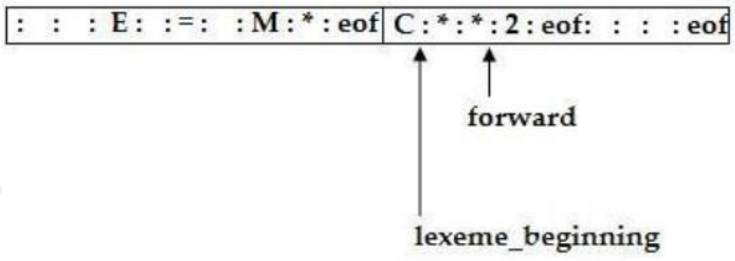
- This scheme works well most of the time, but the amount of lookahead is limited.
- This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.

(eg.) DECLARE (ARGl, ARG2, . . . , ARGn) in PL/1 program;

- It cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

Sentinels

- In the previous scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer
  - **Test 1: For end of buffer.**
  - **Test 2: To determine what character is read.**
- The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program. *(eof* character is used as sent



Sentinels at end of each buffer half

Lookahead code with sentinels

Advantages

- Most of the time, It performs only one test to see whether forward pointer points to an eof.
- Only when it reaches the end of the buffer half or eof, it performs more tests.
- Since N input characters are encountered between eofs, the average number of tests per input character is very close to 1.

```
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
```

Lookahead code with sentinels

1.2.3 SPECIFICATION OF TOKENS

There are 3 specifications of tokens:

- **Strings**
- **Language**
- **Regular expression**

Strings and Languages

- An alphabet or character class is a finite set of symbols
- A string over an alphabet is a finite sequence of symbols drawn from that alphabet.
- A language is any countable set of strings over some fixed alphabet.
- In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s. For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

Operations On Strings

- The following string-related terms are commonly used:

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.
Proper prefix, suffix, or substring of s	Any nonempty string x that is a prefix, suffix or substring of s that s <> 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.

Terms for parts of a string

Operations On Languages:

- The following are the operations that can be applied to languages:
  - **Union**
  - **Concatenation**
  - **Kleene closure**
  - **Positive closure**

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L and M written LM	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes "zero or more concatenations of" L.
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of" L.

Regular Expressions

- It allows defining the sets to form tokens precisely.

  Eg, **letter (letter | digit)***

- Define a Pascal identifier- which says the identifier is formed by a letter followed by zero or more letters or digits.
- A regular expression is built up out of simpler regular expression using a set of defining rules.
- Each regular expression r denotes a language L(r).

The Rules That Define Regular Expressions Over Alphabet Σ

- (Associated with each rule is a specification of the language denoted by the regular expression being defined)

1. ε is a regular expression that denotes {ε}, i.e. the set containing the empty string.
2. If *a* is a symbol in Σ, then *a* is a regular expression that denotes {a}, i.e. the set containing the string*a*.
3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then
   - **a.** (r) | (s) is a regular expression denoting the languages L(r) U L(s).
   - **b.** (r)(s) is a regular expression denoting the languages L(r)L(s).
   - **c.** (r)* is a regular expression denoting the languages (L(r))*.
   - **d.** (r) is a regular expression denoting the languages L(r).

- A language denoted by a regular expression is said to be a regular set.
- The specification of a regular expression is an example of a recursive definition.
- *Rule (1) and (2)* form the basis of the definition.
- *Rule (3)* provides the inductive step.

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 |
εr = rε = r	ε is the identity for concatenation
$r^* = (r\|\varepsilon)^*$	ε is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Algebraic properties of regular expressions

Regular Definition

- If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

d1 → r1

d2 → r2

…

dn → rn

- – Where each di is a distinct name, and each ri is a regular expression over the symbols in Σ U {d1, d2, … , di-1}, i.e., the basic symbols and the previously defined names.
- Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:

letter → A | B | … | Z | a | b | … | z

digit → 0 | 1 | … | 9

id → letter (letter | digit) *

Notational Shorthand

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational short hands for them.

1. *One or more instances (+)*

- The unary postfix operator + means " one or more instances of" .
- If r is a regular expression that denotes the language L(r), then (r)+ is a regular expression that denotes the language (L (r))+
- Thus the regular expression a+ denotes the set of all strings of one or more a's.
- The operator + has the same precedence and associativity as the operator *.

2. Zero or one instance (?)

- The unary postfix operator ? means "zero or one instance of".
- The notation r? is a shorthand for r | ε.
- If 'r' is a regular expression, then (r)? is a regular expression that denotes the language

3. Character Classes

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.
- We can describe identifiers as being strings generated by the regular expression, [A–Za–z][A– Za–z0–9]*

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

1.2.4 REVIEW OF FINITE AUTOMATA

- Refer from TOC
- Finite Automata
    1. Deterministic Finite automata
    2. Non Deterministic Finite automata
- NFA to DFA Conversion

1.2.5 RECOGNITION OF TOKENS

- The question is how to recognize the tokens?

  EXAMPLE

  Assume the following grammar fragment to generate a specific language

  stmt → **if** ***expr*** **then** ***stmt*** | **if** ***expr*** **then** ***stmt*** **else** ***stmt*** | **ε**
  expr → ***term*** **relop** ***term*** | ***term***
  term → **id** | **number**

  where the terminals if, then, else, relop, id and num generates sets of strings given by following regular definitions.

  if → if
  then → then
  else → else
  rebop→ < | <= | < > | > | > =
  id→ letter (letter | digit)*
  num→ digits optional-fraction optional-exponent

- where letter and digits are defined previously
- For this language, the lexical analyzer will recognize the keywords i f , then,and else, as well as lexemes that match the patterns for ***relop***, ***id***, and ***number***.
- To simplify matters, we make the common assumption that keywords are also *reserved words:* that is they cannot be used as identifiers.
- The num represents the unsigned integer and real numbers of Pascal.
- In addition, we assume lexemes are separated by white space, consisting of nonnull sequences of blanks, tabs, and newlines.
- Our lexical analyzer will strip out white space. It will do so by comparing a string against the regular definition **ws,** below.

  Delim → blank | tab | newline
  ws → delim

- If a match for **ws** is found, the lexical analyzer does not return a token to the parser.

Transition Diagram

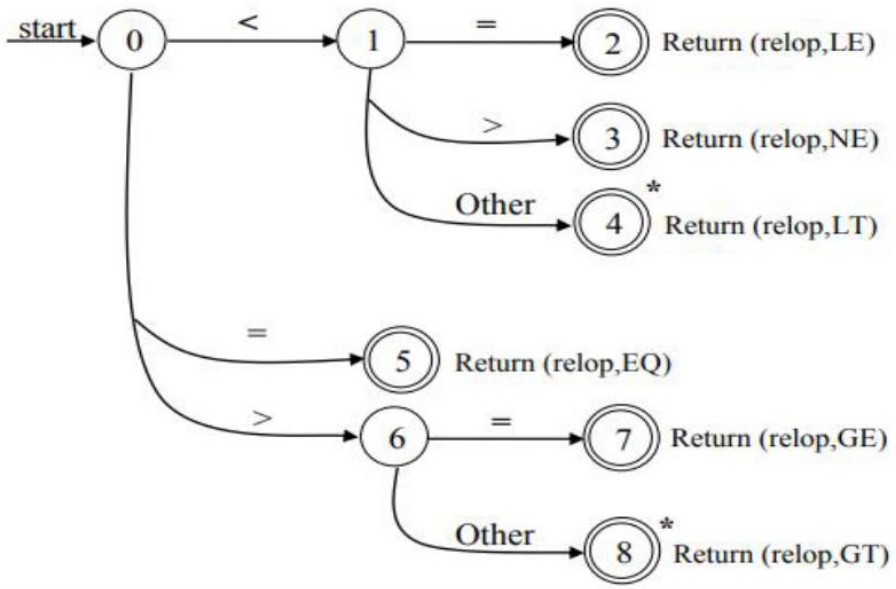
- As an intermediate step in the construction of a lexical analyzer, we first produce a flowchart, called a r diagram. Transition diagrams.
- Transition diagram depict the actions that take place when a lexical analyzer is called by the parser to get the next token.

- The TD uses to keep track of information about characters that are seen as the forward pointer scans the input.
- It does that by moving from position in the diagram as characters are read.

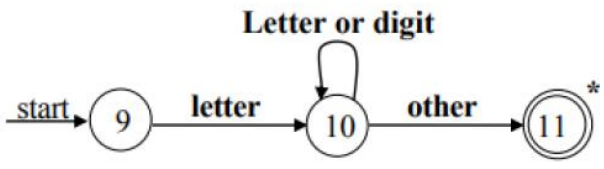
COMPONENTS OF TRANSITION DIAGRAM

1. One state is labelled the **Start State** start ◯ It is the initial state of transition diagram where control resides when we begin to recognize a token.
2. Position is a transition diagram are drawn as circles ◯ and are called states.
3. The states are connected by **Arrows** ⟶ called edges. Labels on edges are indicating the input characters.
4. The **Accepting** states in which the tokens has been found ◎
5. **Retract** one character use * to indicate states on which this input retraction.

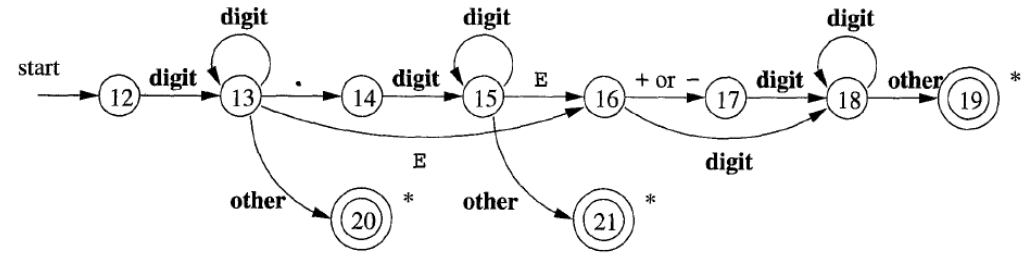
Example: A Transition Diagram for the token relation operators **"relop"** is shown in Figure below:

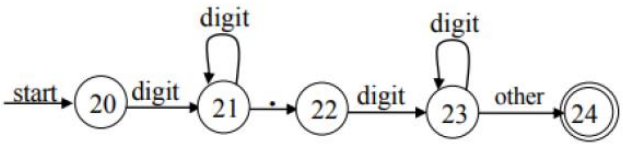


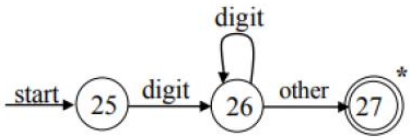
Example: A Transition Diagram for the **identifiers** and **keywords**



Example: A Transition Diagram for **Unsigned Numbers** in Pascal







num ⟶ digit⁺(. digit⁺| ∈)(E(+|-|∈)digit⁺|∈)

****