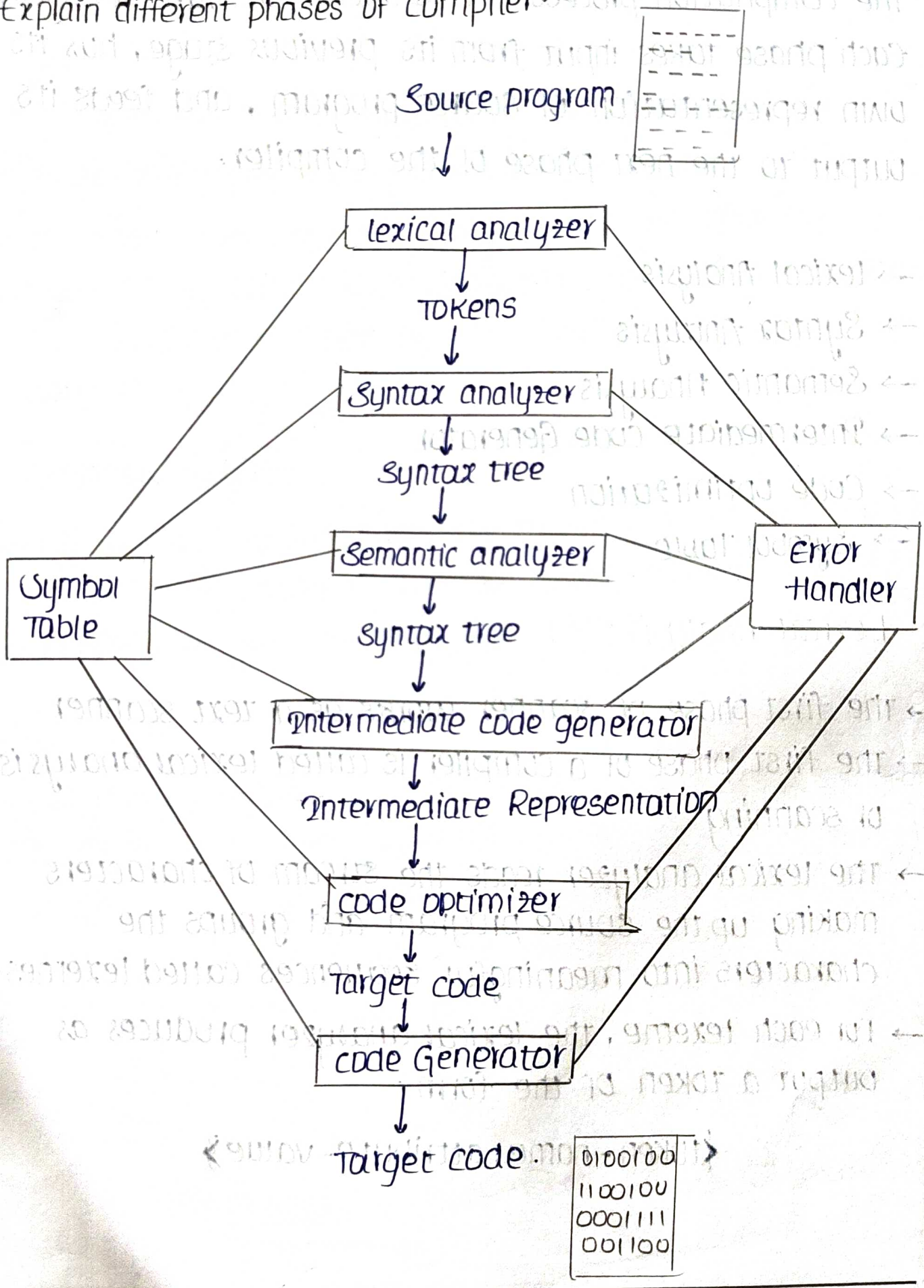
1. Explain different phases of compiler.



The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.

→ lexical Analysis
→ Syntax Analysis
→ Semantic Analysis
→ Intermediate Code Generator
→ Code optimization
→ Symbol Table

Lexical Analysis :

→ The first phase of scanner works as a text scanner.
→ 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}

Syntax Analysis :

→ The second phase of the compiler is syntax analysis or parsing.

→ It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree).

→ In these phase, token arrangements are checked against the source code grammer, i.e. the parser checks if the expression made by the tokens is syntactically correct.

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

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

Intermediate Code Generator:

→ After semantic analysis the compiler generates an interm-ediate code of the source code for the target machine.
→ It represents a program for some abstract machine.
→ It is in between the high-level language and the machine language.
→ This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine.

Code Optimization:

→ The next phase does code optimization of intermediate code.
→ Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resource (CPU, memory)

Code Generator:

→ In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.
→ The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code.
→ Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table :

→ It is a data-structured maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here.

→ The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

2. Discuss the role of the lexical analyzer?

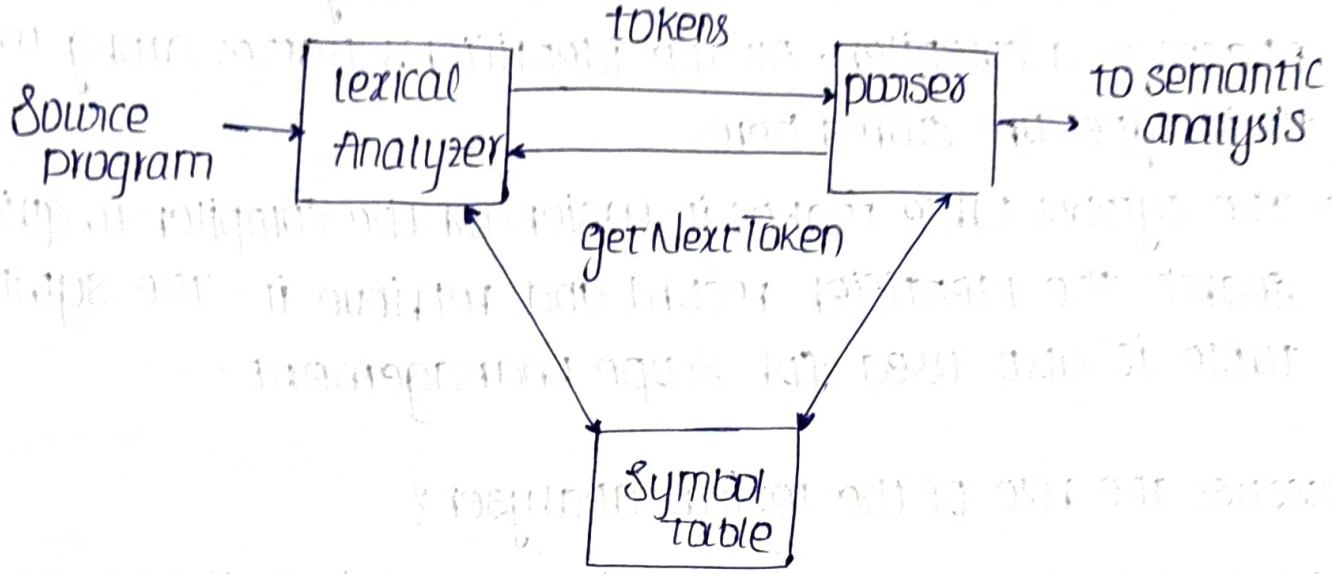
As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.

→ The stream of tokens is sent to the passer for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well.

→ 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 assit it in determining the proper token it must pass to the parser.

fig: Interactions between the lexical analyzer & the parser.



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.

→ Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes.

→ One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to seperate tokens in the input)

→ Another task is correlating error messages generated by the compiler with the source program.

3. Explain Input Buffering.

Input Buffering:

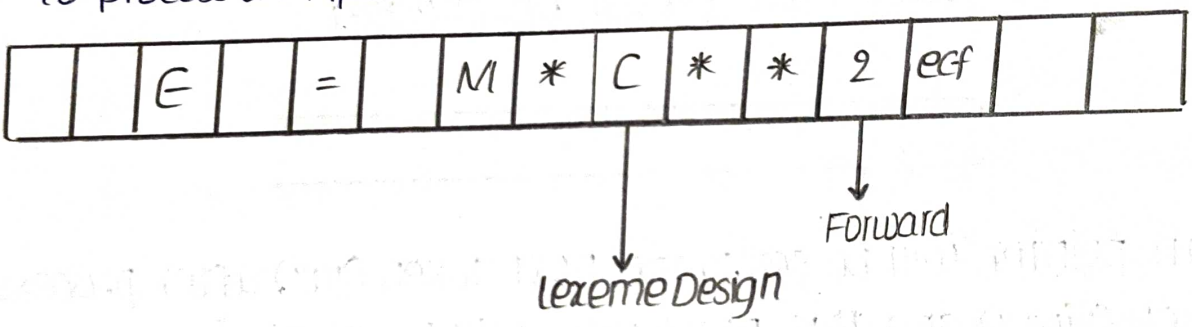
One or more characters beyond the following lexeme must be searched up to guarantee that the correct lexeme is identified.

→ As a result, a two buffer system is used to safely manage huge lookaheads.

→ Using sentinels to mark the buffer end has been embraced as a technique for speeding up the lexical analyzer process.

Buffer Pairs:

Specialized buffering techniques decrease the overhead required to process an input character in moving characters.



→ It consists of two buffers, each of which has an N-character size and is alternately reloaded

→ There are two pointers:

1. lexemeBegin
2. forward

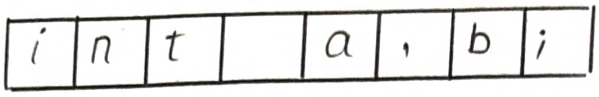
→ LexemeBegin denotes the start of the current lexeme, which has yet to be discovered.

→ Forward scans until it finds a match for a pattern.

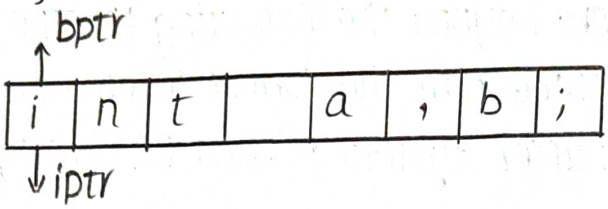
→ When a lexeme is discovered, lexeme begin is set to the character immediately after the newly discovered lexeme, and forward is set to the character at the right end of the lexeme

→ The collection of characters between two points is the current lexem

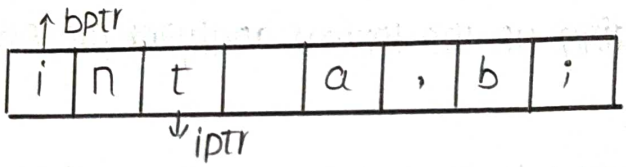
Sample example: For the statement int a,b;



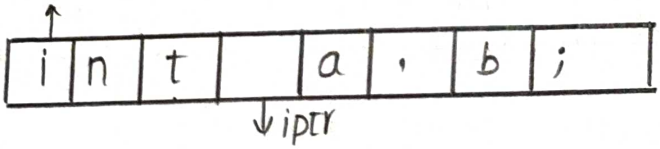
- Both points begin at the start of the string that is saved in the Buffer



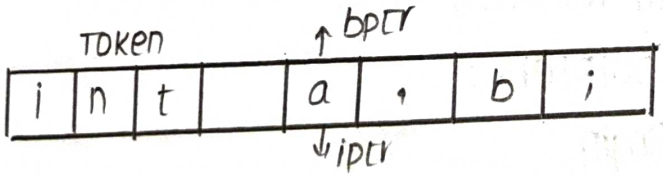
- The Look Ahead Pointer examines the buffer until it finds the token.



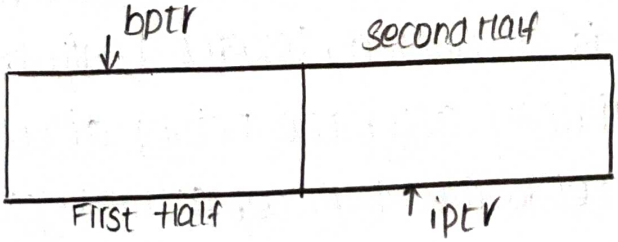
- Before the token ("int") can be identified, the character ("blank space") beyond the token ("int") must be checked.



- Both pointer will be set to the next token ("a") after processing the token ("int") and this procedure will be continued throughout the program.



TWO portions of a buffer can be separated. If you move the look ahead cursor halfway through the first half, the second half will be filled with fresh characters to read.



TWO pointers to the input are maintained:

1. pointer lexemeBegin
2. pointer forward

→ pointer lexemeBegin, marks the begining of the current lexeme, whose extent we are attempting to determine.

→ pointer forward scans ahead until a pattern match is found; [the exact strategy].

Sentinels:

→ It is a special symbol that are added to the input buffer to help identify the end of buffer.

→ They are used to avoid the need for boundary checks when reading the input buffers, which can slow down the lexeme analysis phase of the compiler.

4 How tokens recognized?

Example : assume the following grammer 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 | num

Where the terminals if, then, else, relop, id, and num generate sets of strings given by the following regular definitions:

if → if

then → then

else → else

relop → < | <= | = | <> | > | >=

id → letter(letter|digit)*

num → digits optional-fraction optional-exponent

Where letter and digits are as defined previously.

For this Language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num. To simplify matters, we assume keywords are reserved; 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. The 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 Diagrams (TD):

→ As an intermediate step in the construction of a lexical analyzer we first produce flowchart, called a transition diagram.

→ Transition diagrams 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 to position in the diagram as characters are read.

Components of Transition Diagram:

1. One state is labeled the <u>start state</u> start ○

It is the initial state of the transition diagram where control resides when we begin to recognize a token.

2. <u>Positions</u> in a transition diagram are drawn as circles and are called state ○

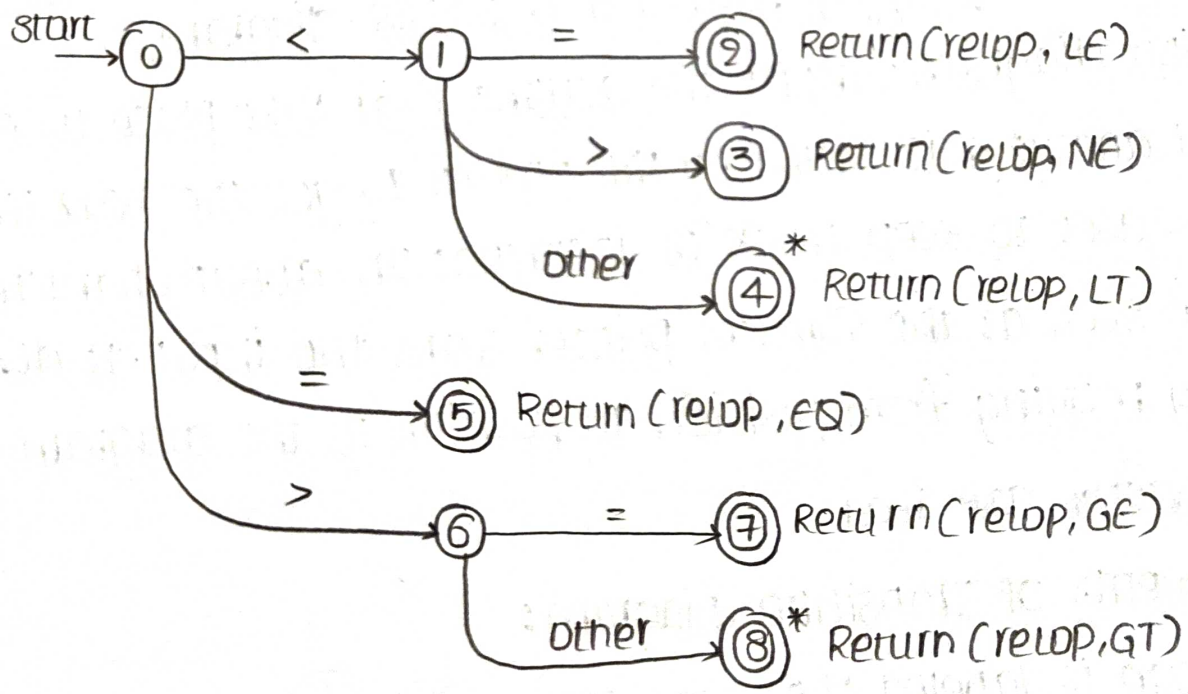
3. The states are connected by Arrows. ⟶ called edges. Labels on edges are indicating the input characters.

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:

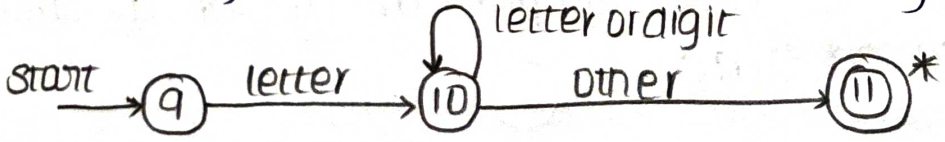
A Transition Diagram for the token relation operators "relop"



Transition diagram for relation operators.

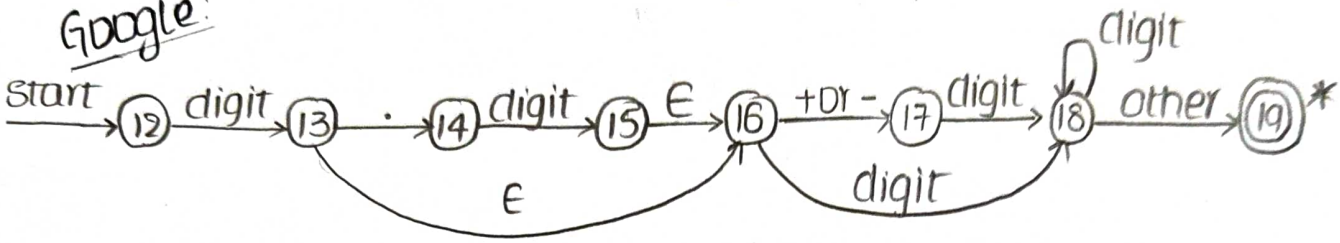
Recognition of Reseoved Words and identifiers:

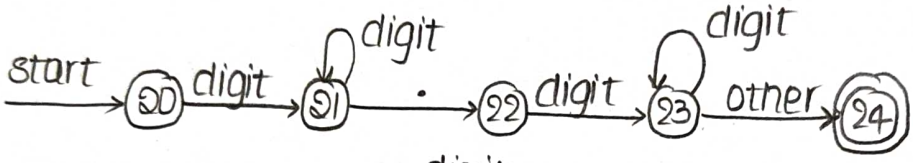
A transition Diagram for the identifiers and keywords

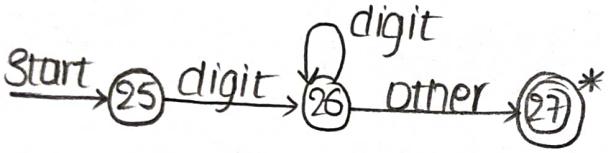


A Transition Diagram for Unsigned Numbers in Pascal:

Google:







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

textbook