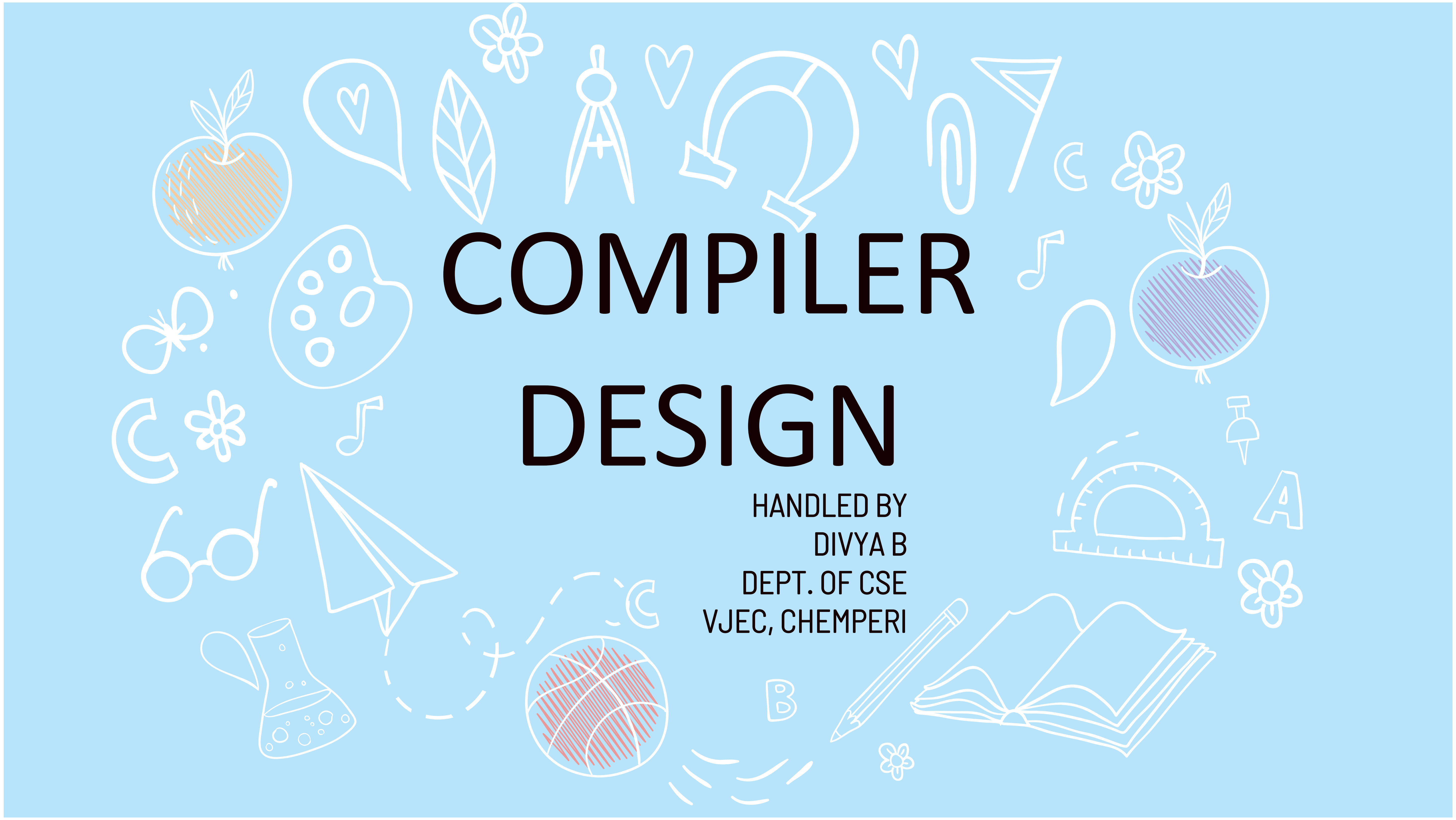


COMPILER DESIGN

HANDLED BY
DIVYA B
DEPT. OF CSE
VJEC, CHEMPERI

BOOTSTRAPPING

- Bootstrapping is widely used in the compiler 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.

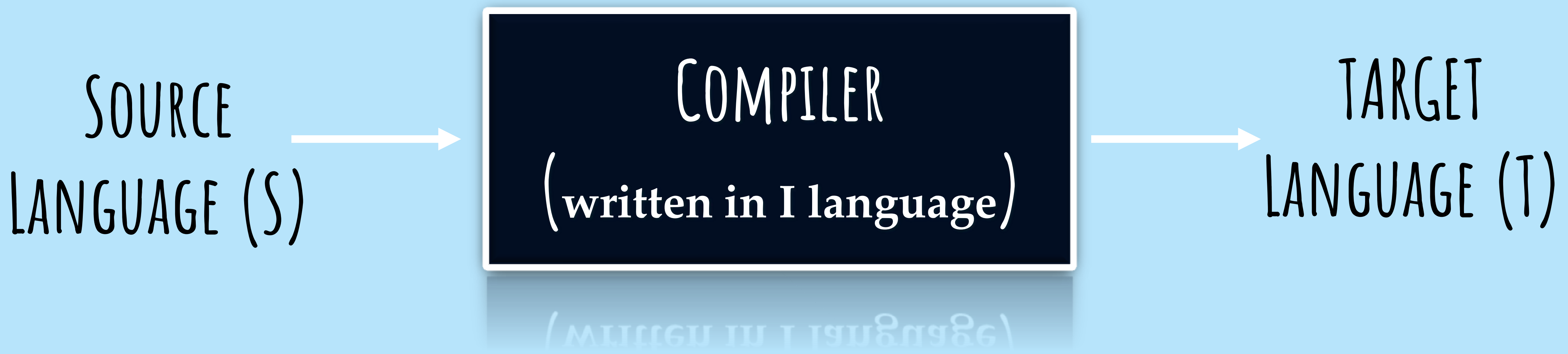
BOOTSTRAPPING

- An initial core version of the compiler (bootstrap compiler) is generated in a different language (which could be assembly language).
- Successive expanded versions of the compiler are developed using this minimal subset of the language.

BOOTSTRAPPING

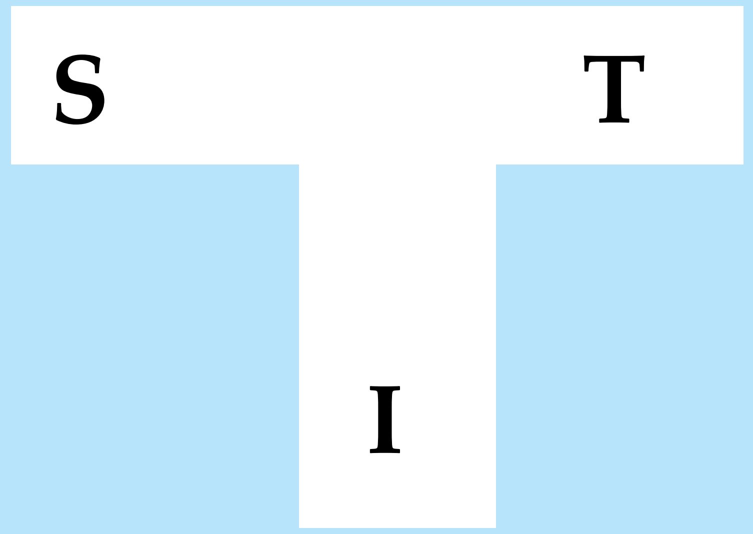
- For bootstrapping purposes, a compiler is characterized by three languages
  - Source Language (S) is the language that it compiles.
  - Target Language (T) is the language that it generates code for.
  - Implementation Language (I) is the language that it is written in.

BOOTSTRAPPING



BOOTSTRAPPING

- This can be represented by using T-diagram (Tombstone diagram).



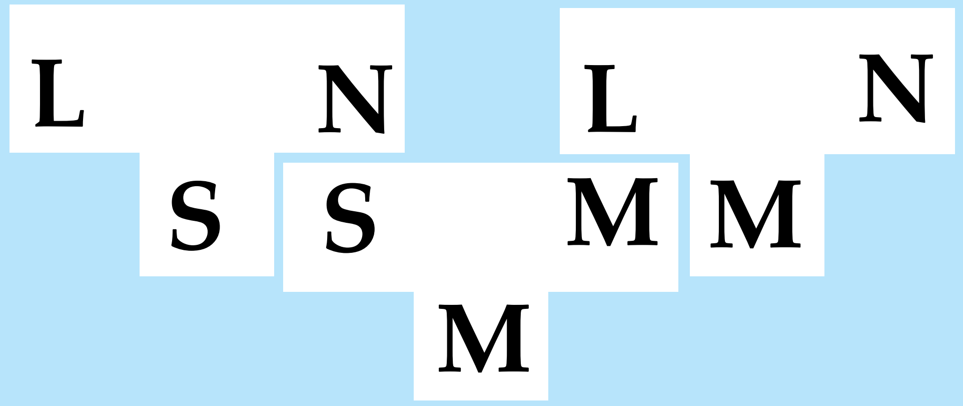
- It can be written as $\mathbf{S_I T}$

BOOTSTRAPPING

- Compilers can be ***native*** or ***cross compilers***.
- Native compilers are compilers written in the same language as the target language.
- A compiler may run on one machine and produce target code for another machine. Such a compiler is called as ***cross compiler***.
- A compiler that runs on Windows machine and produces code that runs on Android smartphone is a cross compiler.

BOOTSTRAPPING

- Write a cross compiler for a new language L in implementation language S to generate code for machine N.
- Existing compiler for S runs on different language M and generates code for M.

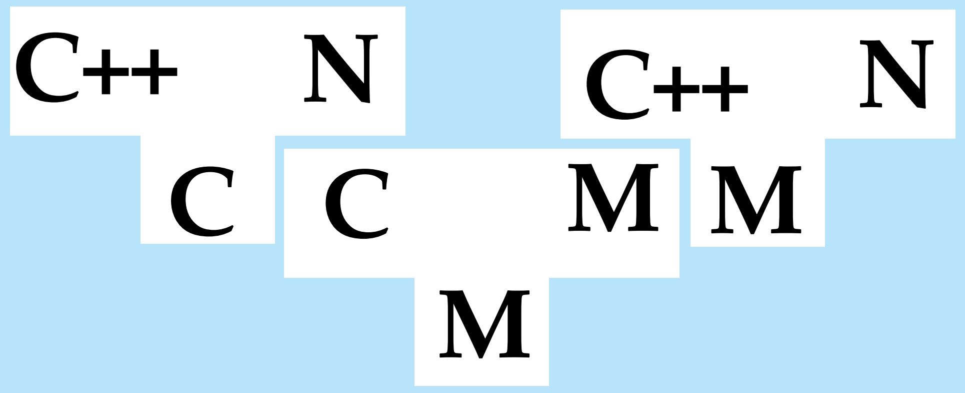


This can be summarized by the equation,

$L_SN+S_MM = L_MN$

BOOTSTRAPPING

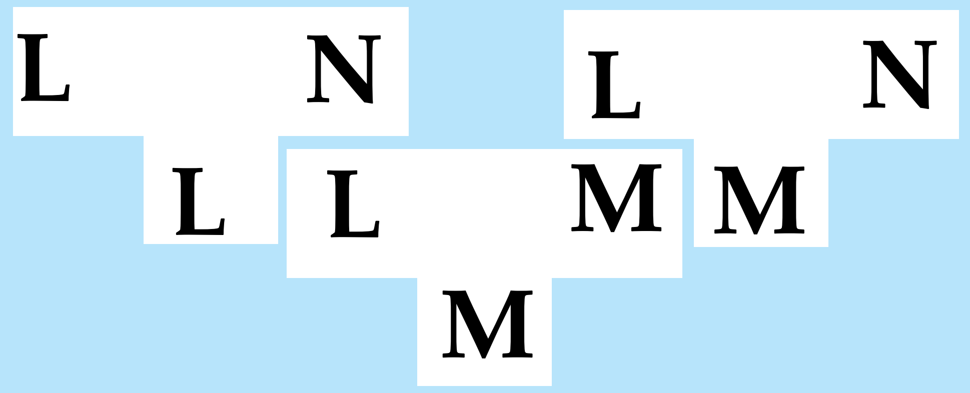
- Write a cross compiler for C++ in implementation language C to generate code for machine N.
- Existing compiler for C runs on different language M and generates code for M.



BOOTSTRAPPING

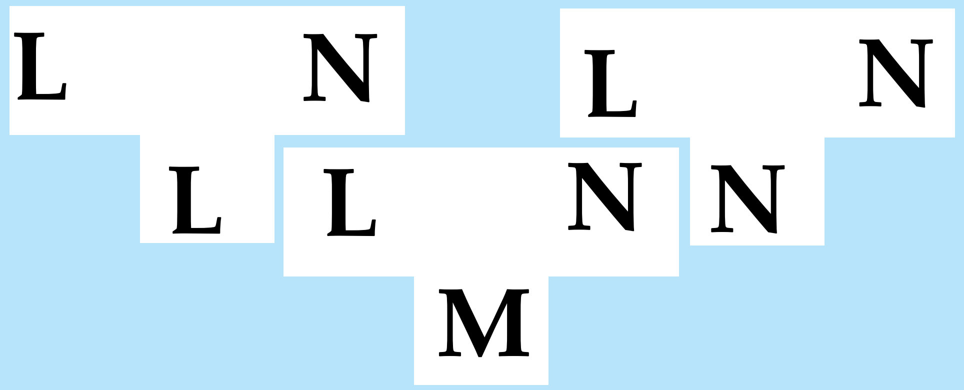
- Suppose we write a compiler $L_L N$ for language L in L to generate code for N.
- Development takes place on a machine M, where an existing compiler $L_M M$ for L runs and generates code for M.
- By first compiling $L_L N$ with $L_M M$, we obtain a cross compiler $L_M N$ that runs on M, but produces code for N.

BOOTSTRAPPING



BOOTSTRAPPING

- The compiler L_LN can be compiled a second time by using the generated cross compiler.

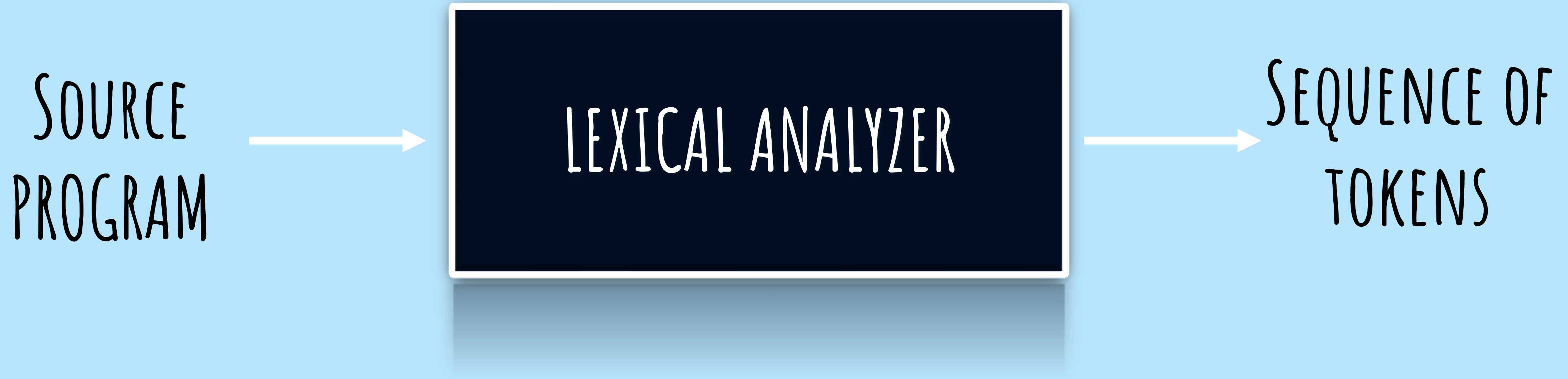


The result is a compiler L_NN that generates code for N and runs on N.

ROLE OF LEXICAL ANALYSIS

- 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.

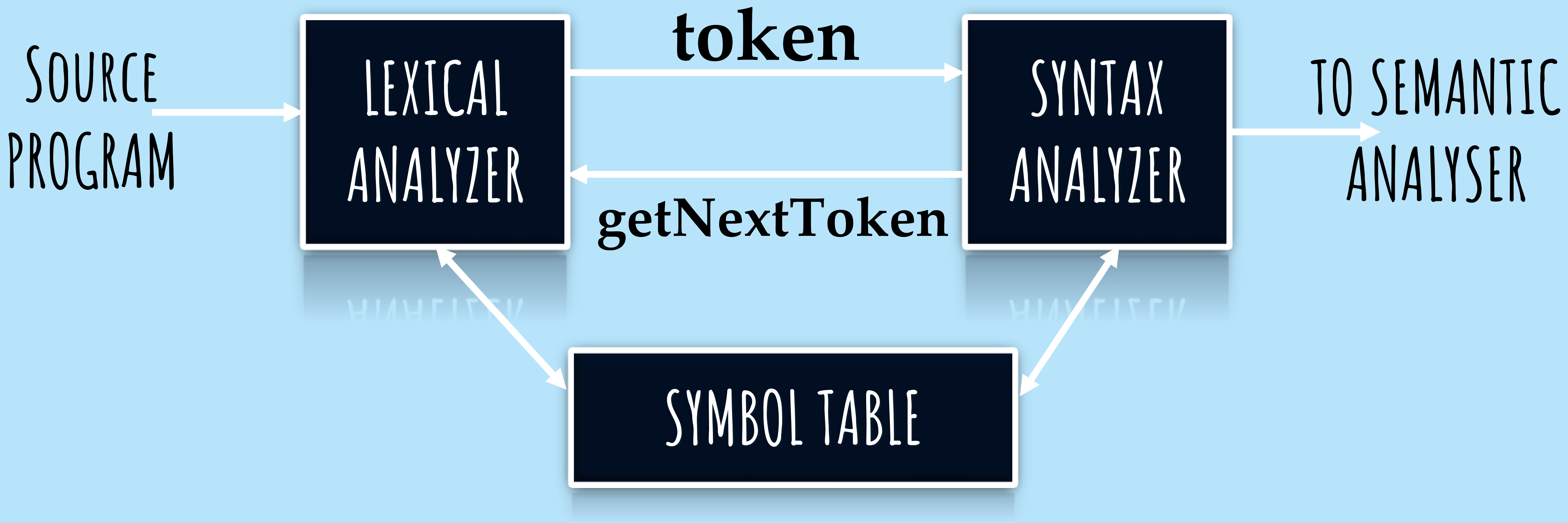
ROLE OF LEXICAL ANALYSIS



ROLE OF LEXICAL 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.

ROLE OF LEXICAL ANALYSIS



ROLE OF LEXICAL ANALYSIS

- Lexical analyzer strips out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- Correlates 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.
- If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

ISSUES IN LEXICAL ANALYSIS

- **Simplicity of design**
  - The separation of lexical analysis and syntax analysis often allows us to simplify at least one of these tasks. The syntax analyzer can be smaller and cleaner by removing the low level 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.

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.
- The successive characters are deleted 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.

LEXICAL ERRORS

- 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.

LEXICAL ERRORS

- 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.

INPUT BUFFERING

- Ways to speed up the process of reading the source program is difficult because the lexical analyser has to look one or more characters beyond the next lexeme to ensure that the right lexeme is found.
- For e.g. for finding the end of an identifier lexeme a character that is not a letter or digit needs to be found.
- In C, single character operators like = or < could also be the beginning of two character operators like = = or < =

INPUT BUFFERING

- Two-buffer scheme is introduced to handle large lookaheads safely.
- Sentinels to mark buffer end have also been adopted to speed up the process of lexical analysis.

INPUT BUFFERING

- **Buffer Pairs**
  - Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.

INPUT BUFFERING

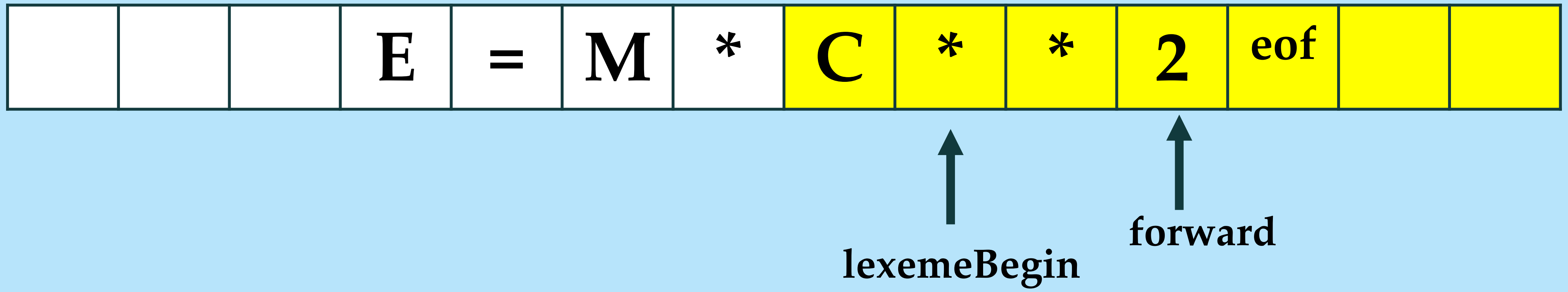
- **Buffer Pairs**
  - An important scheme involves two buffers that are alternatively reloaded.
  - Each buffer is of the same size N, and N is usually the size of a disk block, e.g. 4096 bytes.
  - Using one system read command we can read N characters into a buffer, rather than using one system call per character.

INPUT BUFFERING

- **Buffer Pairs**
  - If fewer than N characters remain in the input file, then a special character represented by **eof** marks the end of the source file.
  - Two pointers to the input are maintained
    1. ***lexemeBegin*** marks the beginning of the current lexeme.
    2. ***forward*** scans ahead until a pattern match is found.

INPUT BUFFERING

- **Buffer Pairs**



INPUT BUFFERING

- **Buffer Pairs**
  - Once the next lexeme is determined, ***forward*** is set to the character at its right end.
  - After the lexeme is recorded as an attribute value of a token returned to the parser, ***lexemeBegin*** is set to the character immediately after the lexeme just found.
  - In the diagram, ***forward*** has passed the end of the next lexeme (exponentiation operator) and must be retracted one position to its left.

INPUT BUFFERING

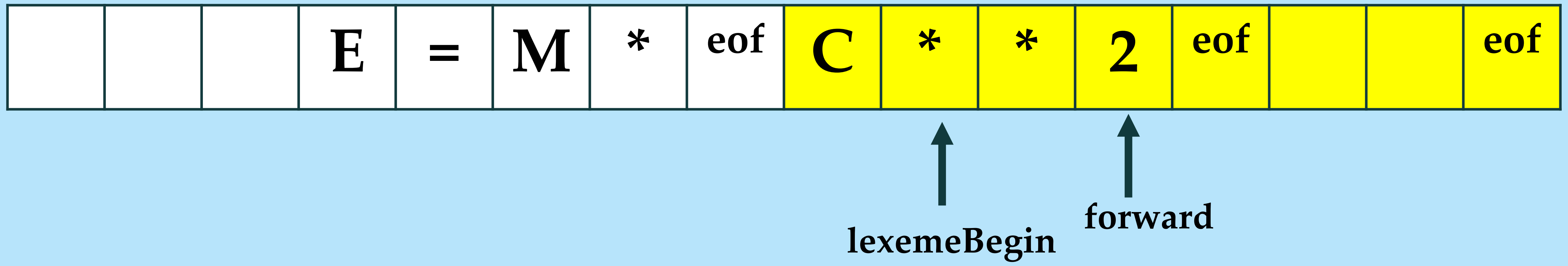
- **Buffer Pairs**
  - Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer with the input and move forward to the beginning of the newly loaded buffer.

INPUT BUFFERING

- **Sentinels**
  - In the previous scheme, two tests are required every time the forward pointer is moved
    1. For end of the buffer.
    2. To determine what character is read.
  - The use of sentinels reduced the tests to one by having each buffer half to hold one sentinel character at the end.

INPUT BUFFERING

- **Sentinels**
  - Sentinel is a special character that is not part of the source program. Here, it is eof which marks the end of the buffer.



Lookahead code with sentinels

```
switch (*forward++)
{
    case eof : if(forward is at the end of first buffer)  {
                     reload second buffer;
                     forward = beginning of second buffer ; }
               else if (forward is at the 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;
      cases for other characters
}
```

SPECIFICATION OF TOKENS

- There are three specification of tokens
  - Strings
  - Languages
  - Regular expressions

SPECIFICATION OF TOKENS

- **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."

SPECIFICATION OF TOKENS

- **Strings and Languages**
  - The length of a string s, usually written |s|, is the number of occurrences of symbols in s.
  - E.g. banana is a string of length six.
  - The empty string, denoted ε, is the string of length zero.

SPECIFICATION OF TOKENS

Operations on Strings

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 obtained by removing zero or more leading symbols of string 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.

SPECIFICATION OF TOKENS

Operations on Strings

Term	Definition
Proper prefix, suffix or substring of s	Any non empty string x that is a prefix, suffix or substring such that it is not s itself.
Subsequence of s	Any string formed by deleting zero or more not necessarily consecutive positions of s. e.g. baaa is a subsequence of banana.

SPECIFICATION OF TOKENS

Operations on Languages

Operation	Definition
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	$L^* = \cup_{i=0}^{\infty} L^i$ L^* denotes zero or more concatenations of L
Positive Closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$ L^+ denotes one or more concatenations of L

SPECIFICATION OF TOKENS

- **Regular Expressions**
  - It allows defining the sets to form tokens precisely.
  - E.g. Identifier is denoted as letter (letter | digit)*
  - 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).

SPECIFICATION OF TOKENS

- **Regular Expressions**
  - **The Rules That Define Regular Expressions Over Alphabet ∑** (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*.

SPECIFICATION OF TOKENS

- **Regular Expressions**
  - **The Rules That Define Regular Expressions Over Alphabet ∑**
    (Associated with each rule is a specification of the language denoted by the regular expression being defined)

3. Suppose r and s are regular expressions denoting the languages L(r) and L(s).
   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).
   Indicates that we can add additional pairs of parentheses around expressions without changing the language they denote.

SPECIFICATION OF TOKENS

- **Regular Expressions**
  - 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.

SPECIFICATION OF TOKENS

- **Regular Expressions – Algebraic properties**

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|ε)*	ε is guaranteed in a closure
r** = r*	* is idempotent

SPECIFICATION OF TOKENS

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

$$\begin{aligned} &\mathbf{d1 \rightarrow r1} \\ &\mathbf{d2 \rightarrow r2} \\ &\ldots \\ &\mathbf{dn \rightarrow rn} \end{aligned}$$

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.

SPECIFICATION OF TOKENS

- **Regular Definitions**
  - E.g. Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set is

    letter → A | B | … | Z | a | b | … | z
    digit → 0 | 1 | … | 9
    id → letter (letter | digit) *

SPECIFICATION OF TOKENS

- **Notational Shorthand**
  1. **One or more instances (+)**
  2. **Zero or one instance (?)**
  3. **Character Classes**

SPECIFICATION OF TOKENS

- **Notational Shorthand**

  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 *.

SPECIFICATION OF TOKENS

- **Notational Shorthand**

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 L(r)? is a regular expression that denotes the language.

SPECIFICATION OF TOKENS

- **Notational Shorthand**

  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]*

SPECIFICATION OF TOKENS

- **Non Regular Set**
  - A language which cannot be described by any regular expression is a non-regular set.
  - E.g. 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.

RECOGNITION OF TOKENS

- Assume the following grammar fragment to generate a specific language,

$$stmt \rightarrow \boldsymbol{if}\ expr\ \boldsymbol{then}\ stmt \mid \boldsymbol{if}\ expr\ \boldsymbol{then}\ stmt\ \boldsymbol{else}\ stmt \mid \varepsilon$$
$$expr \rightarrow term\ \boldsymbol{relop}\ \text{term} \mid \text{term}$$
$$term \rightarrow \boldsymbol{id} \mid \mathbf{number}$$

RECOGNITION OF TOKENS

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

if → if

then → then

else → else

relop → <|<=|< >|> |> =

id → letter (letter|digit)*

num → digit optional-fraction optional-exponent

RECOGNITION OF TOKENS

- 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.
- num represents the unsigned integer and real numbers.
- Assume that lexemes are separated by white space, consisting of non-null sequences of blanks, tabs, and newlines.

RECOGNITION OF TOKENS

- 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

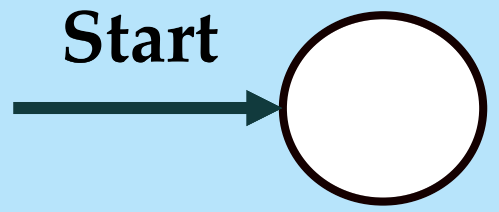
If a match for **ws** is found, the lexical analyzer does not return a token to the parser.

RECOGNITION OF TOKENS

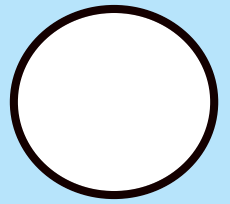
- **Transition Diagram**
  - Transition diagram depict the actions that take place when a lexical analyzer is called by the parser to get the next token.
  - The TD keeps track of information about characters that are seen as the forward pointer scans the input.

RECOGNITION OF TOKENS

- **Transition Diagram – Components**
  - **Start state** - It is the initial state of transition diagram where control resides when we begin to recognize a token.

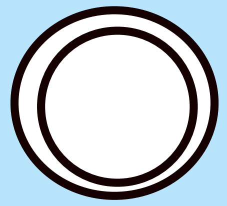


  - Position is a transition diagram are drawn as circles and are called states.



RECOGNITION OF TOKENS

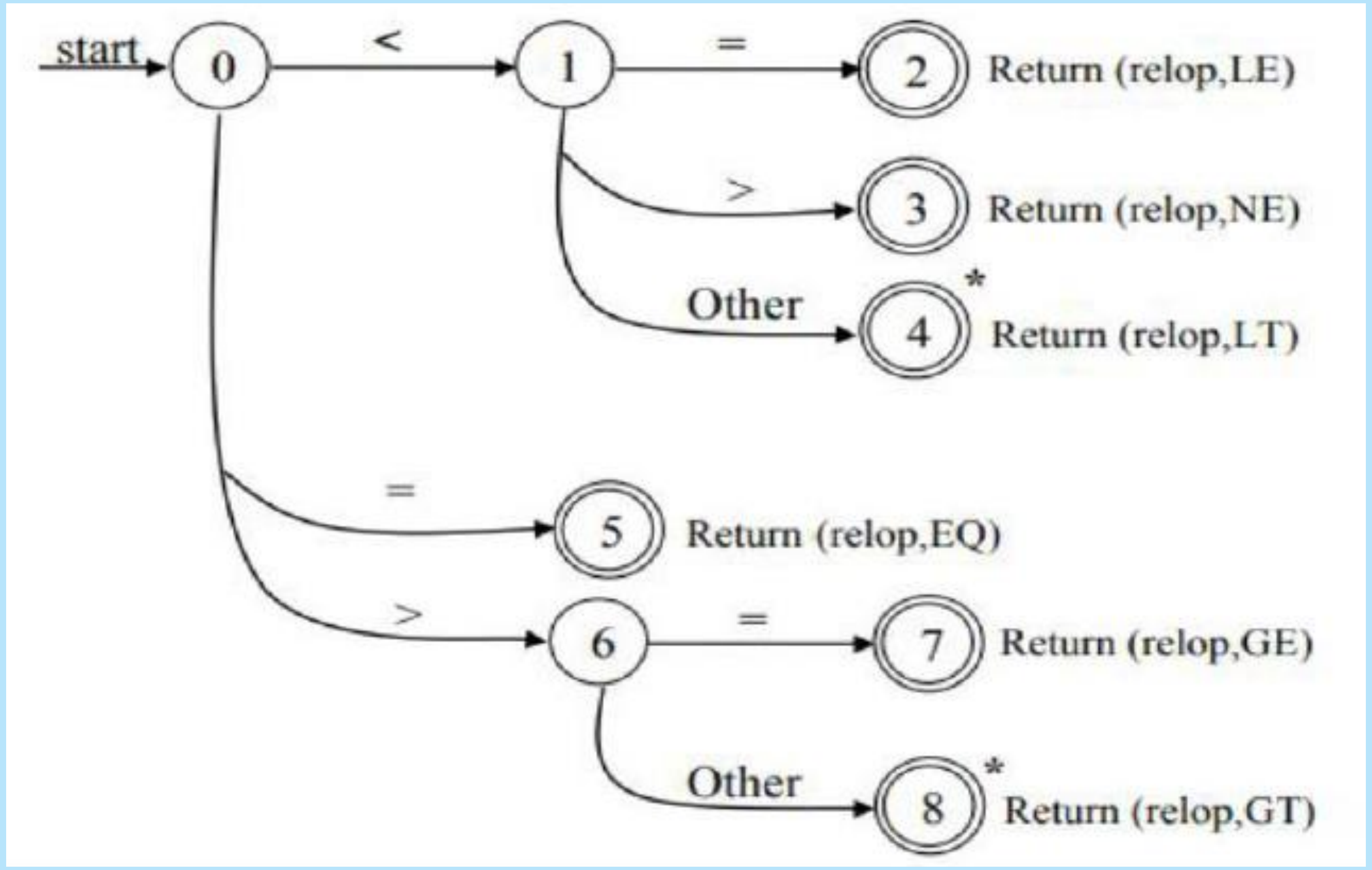
- **Transition Diagram – Components**
  - The states are connected by **arrows** called edges. Labels on edges indicate the input characters.
  - The **accepting** states are in which the tokens has been found.



  - * is used to indicate states on it needs to **retract**.

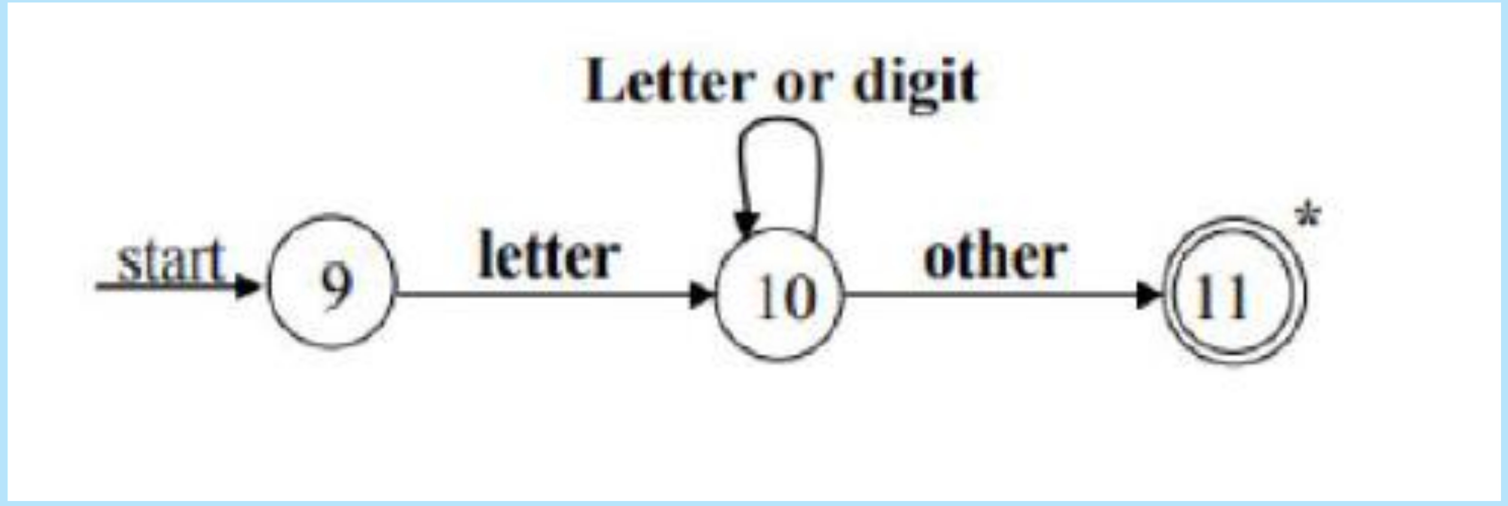
RECOGNITION OF TOKENS

- **Transition Diagram – Relational operators**



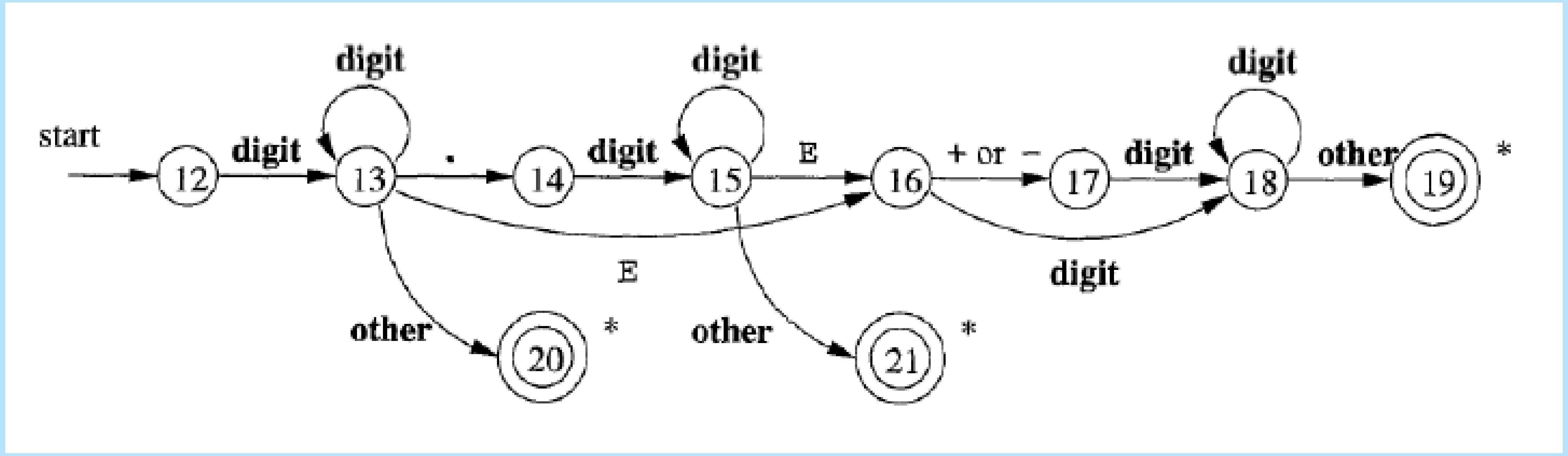
RECOGNITION OF TOKENS

- **Transition Diagram – Identifiers and keywords**



RECOGNITION OF TOKENS

- **Transition Diagram – unsigned numbers**


start
digit
digit
.
digit
digit
E
digit
+ or –
digit
digit
other
*
E
digit
other
*
other
*
12
13
14
15
16
17
18
19
20
21