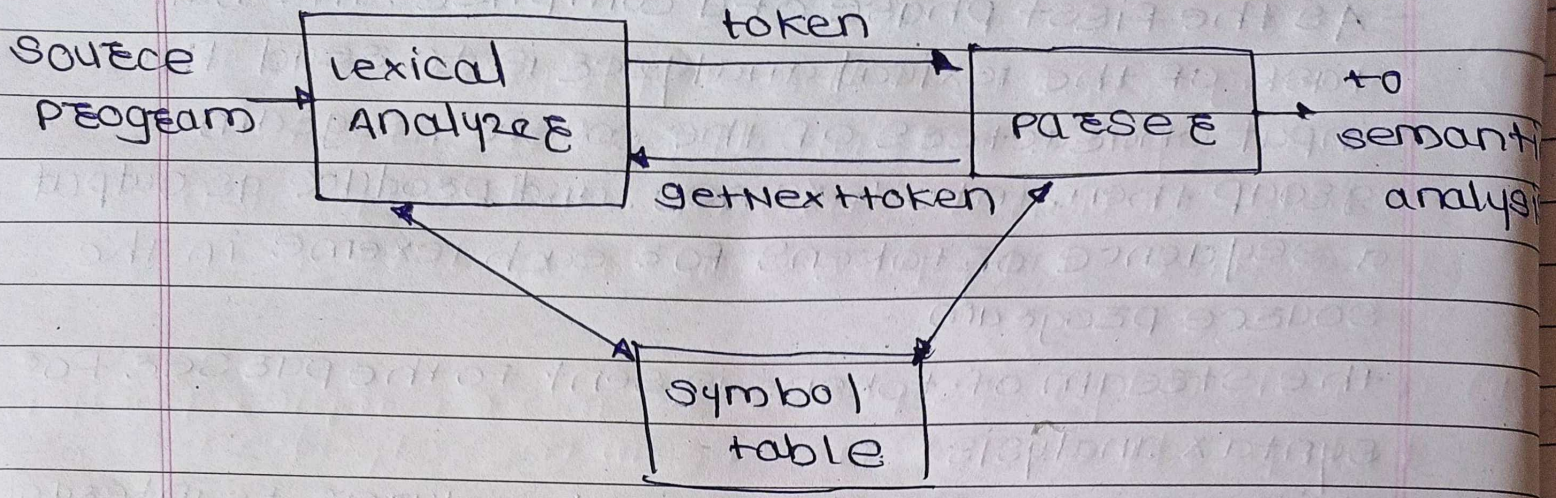
chapter 2 Lexical Analysis.

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 parser for syntax analysis
- It is common for the lexical analyzer to interact with the symbol table.
- when the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- 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, the interactive in implemented by having the parser call the lexical analyzer.
- The call, suggested by the getNextToken command, causes the lexical analyzer
The call by the getNextToken command, causes the lexical analyzer to read characters from it's input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.



fig; interactions between the lexical analyzer and the parser.

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.

Another task is 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 generated by the compiler with the source program.

- 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.

sometimes lexical analyzers are divided into a cascade of two processes.

a) scanning consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

b) lexical analysis proper is the more complex portion, which produces tokens, from the output of the scanner.

1) Tokens, Patterns, and lexemes.

① A token is a pair consisting of a token name and an optional attribute value the token name is an abstract symbol representing a kind of lexical unit.
e.g., a particular keyword or a sequence of input characters denoting an identifier.
the token names are the input symbols that the parser processes, In what follows,

② A pattern is a description of the form that the lexemes of a token may take.
in the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword
for identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

3) A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

* Input Buffering

① We often have to look one or more characters beyond the next lexeme.
e.g. end of an identifier until we see character that is not a letter or digit, and therefore is not part of the lexeme for id.
In C, single-character operator like -, =, or (could also be the begining of a two-character operator like ->, ==, or <=,
we shall introduce a two-buffer scheme that handles large lookaheads safely,
involving "sentinels" that saves time checking for the ends of buffers.

① 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.
An important scheme involves two buffers that are alternately reloaded, as suggested in.

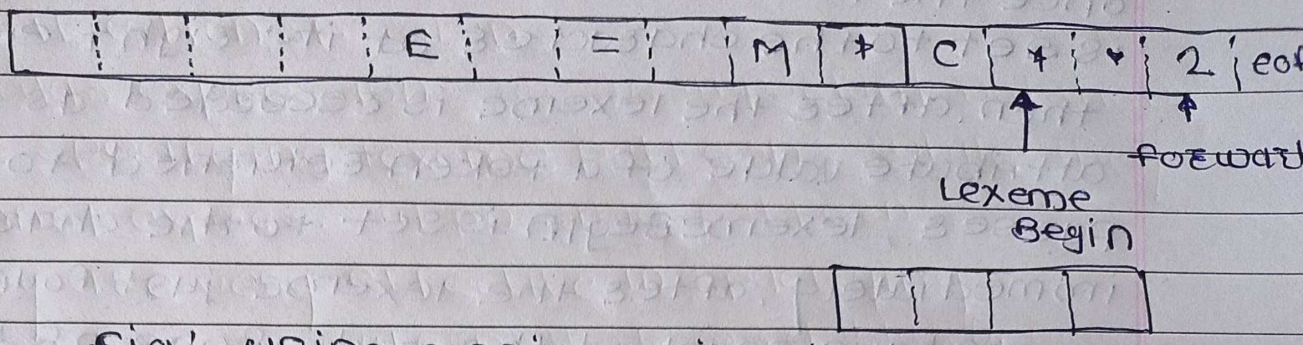


fig: using a pair of input buffers.

Each buffer is of the same size N, and N is usually the size of a disk block, eg., 4096 bytes,
using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained.

1) Pointer lexemeBegin, marks the beginning of the current lexeme, whose extent, we are attempting to determine.

2) Pointer forward scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

once the next lexeme is determined, forward is set to the character at its right end. then, 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.

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.

— Sentinels

Thus, for each character read, we make two tests:
one for the end of the buffer, and one to determine what character is read.
we can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
The sentinels is a special character that cannot be part of the source program, and a natural choice is the character eof.
- eof that appears other than at the end of a buffer means that the input is at an end.

specification and recognition of tokens :-

① Specification of tokens :-

Regular expressions are an important notation for specifying lexeme patterns while they cannot express all possible patterns, they are very effective in spacifying those types of patterns that we actually need for tokens.

strings and languages.

An alphabet is any finite set of symbols. typical examples of symbols are letters, digits, and punctuation.

The set {0, 1} is the binary alphabet.

ASCII is an important example of an alphabet; it is used in many software systems unit

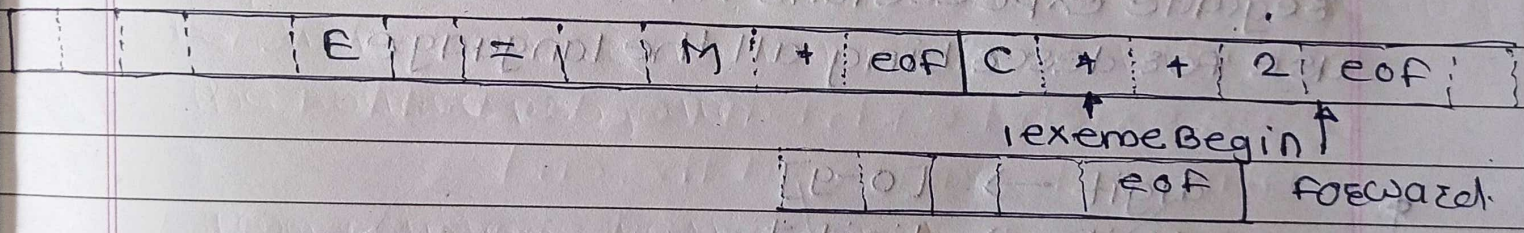


fig : sentinels at the end of each buffer.

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

```
else /* eof within a buffer marks the end
        input */
        terminate lexical analysis;
    break;
    cases for the other characters
}
```

fig lookahead code with sentinels

* Regular Expressions.

suppose we wanted to describe the set of valid c identifiers. It is almost exactly the language described in item the only difference is that the underscore is included among the letters.

This process is so useful that a notation called regular expressions has come into common use for describing all the language

* digit → [0-9]
  digits → digit+
  number → digits (. digits)? (E [+-]? digits)?
  letter → [A-Za-z]
  id → letter (letter | digit)*
  if → if
  then → then
  else → else
  relop → < | > | <= | >= | = | <>

patterns for tokens of Example.

```
else /* eof within a buffer marks the end
        input */
        terminate lexical analysis;
    break;
    cases for the other characters
}
```

fig lookahead code with sentinels

* Regular Expressions.

Suppose we wanted to describe the set of valid c identifiers. It is almost exactly the language described in item 3 the only difference is that the underscore is included among the letters.

This process is so useful that a notation called regular expressions has come into common use for describing all the language :

* digit → [0-9]
  digits → digit+
  number → digits (. digits)? (E [+-]? digits)?
  letter → [A-Za-z]
  id → letter (letter | digit)*
  if → if
  then → then
  else → else
  relop → < | > | <= | >= | = | <>

patterns for tokens of Example.

Recognition of TOKENS:-

expeess patteens using eegulae expeesions, Now, we must study how to take the patteens foe all the needed tokens and build a piece of code that examines the input string and finds a peefix that is a lexeme matching one of the patteens.
Oue discussion will make use of the following eunning example.

```
stmt → if expe then stmt
     | if expe then stmt else stmt
     | e
expe → teem eelop teem
     | teem
teem → id
     | numbee
```

fig: A geammae foe branching statements.

foe eelop, we use the compaeison opeeatoes of languages like pascal oe SQL, wheee = is "equals" and <> is "not equals" because it peesents an inteesting steuctuee of lexemes.

The teeminals of the geammae, which aee if, then, else, eelop, id, and numbee, aee the names of tokens as fae as the lexical analyzee is conceened.
The patteens foe these tokens: aee descaibed using eegulae definitions,
The patteens foe id and numbee aee similae to what we saw in.

digit → [0-9]
digits → digit+
number → digits (. digits)? (E [+-]? digits)?
letter → [A-Za-z]
id → letter (letter | digit)*
if → if
then → then
else → else
relop → < | > | <= | >= | = | <>

fig Patterns for tokens

The lexical analyzer will recognize the keywords if, 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 are not identifiers, even though their lexemes match the pattern for identifiers.

In addition, we assign the lexical analyzer the job of stripping out whitespace, by recognizing the "token" we defined by:

ws → (blank | tab | newline)+

* Finite automata implications

Lex turns its input program into a lexical analyzer. At the heart of the transition is the formalism known as finite automata. These are essentially graphs, like transition diagrams, with a few differences:

1. Finite automata are recognizers; they simply say "yes" or "no" about each possible input string

2) Finite automata come in two flavors:

a) Nondeterministic finite automata (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ε, the empty string, is a possible label.

b) Deterministic finite automata (DFA) have, for each state, and for each symbol of it's input alphabet, exactly one edge with that symbol leaving that state.

Nondeterministic Finite Automata

A nondeterministic finite automaton (NFA) consists of:

1) A finite set of states S.
2) A set of input symbols Σ, the input alphabet. We assume that ε, which stands for the empty string, is never a member of Σ

3) A transition function that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of next states.

4) A state s_0 from S that is distinguished as the start state (or initial state)

5) A set of states F, a subset of S, that is distinguished as the accepting states (or final states).

We can represent either an NFA or DFA by a transition graph, where the nodes are states and the labeled edges represent the transition function.

There is an edge labeled a from state s to state t if and only if t is one of the next states for state s and input a, This graph is very much like a transition diagram except:

a) The same symbol can label edges from one state to several different states, and.

b) An edge may be labeled by ϵ, the empty string, instead of, or in addition to, symbols from the input alphabet.

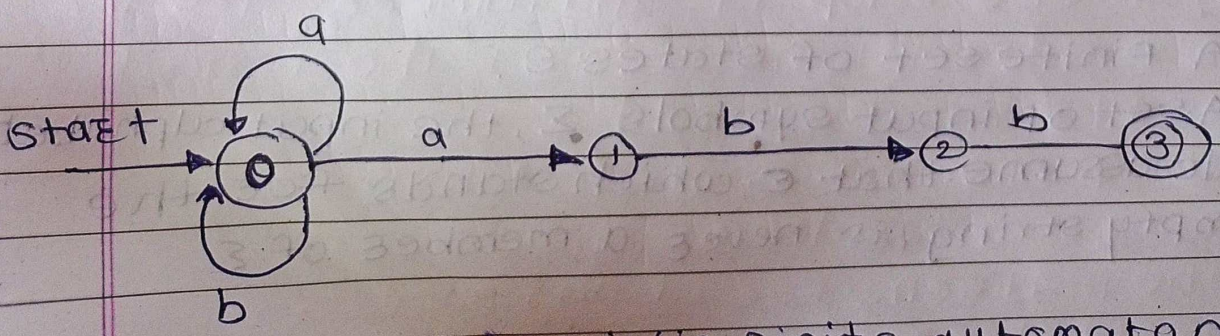
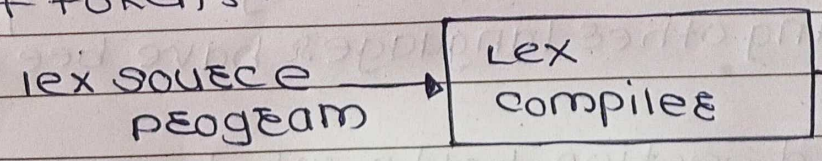


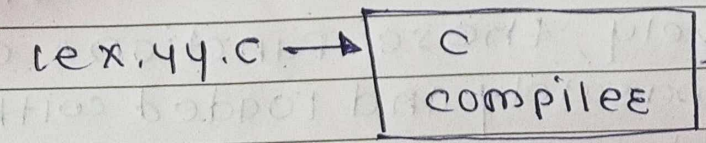
fig:- A nondeterministic finite automaton.

* The lexical-Analyzer Generator Lex.

lexical analyzer by specifying regular expressions to describe patterns for tokens.
the input notation for the lex tool is referred to as the lex language and the tool itself is the lex compiler.
Behind diagram and generates code, in a file called lex.yy.c, that simulates this transition diagram, the mechanics of how this translation from regular expressions to transition diagrams occurs is the subject of the next sections: here we only learn the lex language.

use of lex
lex is used, An input file, which we call lex.l, is written in the lex language, and describes the lexical analyzer to be generated
The Lex compiler transforms lex.l to a c program in a file that is always named lex.yy.c, the latter file is compiled by the C compiler into a file called a.out, as always. the c-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.





Structure of lex programs.

A lex program has the following form.

```
declarations
%%
translation rules
%%
auxiliary functions
```

The declaration section includes declaration of variables, manifest constants (identifiers declared to stand for a constant e.g... the name of a token), and regular definitions, in the style of

The translation rules each have the form

```
pattern {Action}
```

Each pattern is a regular expression, which may use the regular definitions of the declaration section.
The actions are fragments of code, typically written in C, although many variants of lex using other languages have been created.

The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer