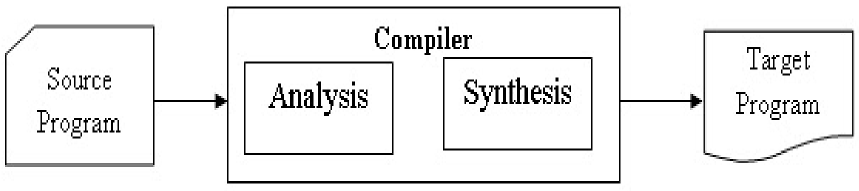
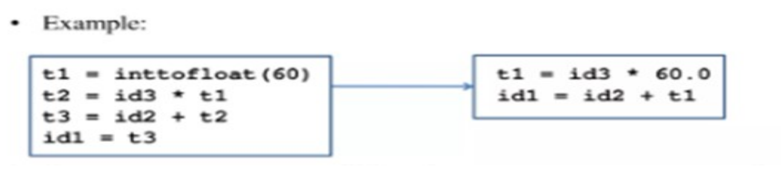
- **Language Processors:** Overview of language processing system: – preprocessors – compiler – assembler – Linkers & loaders, difference between compiler and interpreter- structure of a compiler:–phases of a compiler.
- **Lexical Analysis:** - Role of Lexical Analysis: Lexical analysis Versus Parsing – Tokens, Patterns, and Lexemes – Attributes for Tokens – Lexical errors - Input Buffering: Buffer Pairs – Sentinels
- **Specification of Tokens:** Strings and Languages – Operations on Languages – Regular Expressions – Regular Definitions
- **Recognition of Tokens:** Transition Diagrams – Recognition of Reserved Words and Identifiers - Completion of the Running Example – Architecture of a Transition–Diagram-Based Lexical Analyzer
- **The Lexical Analyzer Generator (LEX):** Use of Lex – Structure of Lex Programs



Analysis and Synthesis model

Code optimization

- Code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result.
- Faster/shorter/Less power consumable target code.
- Compiler spent significant amount of time on this phase.
- Optimized Three address code after Code Optimization phase for the example statement is



- Example:

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

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

Code Generation

- It takes intermediate representation of the source program as input 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.
- Intermediate instructions are translated into sequences of machine instructions

- Crucial part is assignment of registers to hold variables.
- First operand of each instruction specifies destination.
- F-> floating point number
- #-> 60.0 consider as immediate constant

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

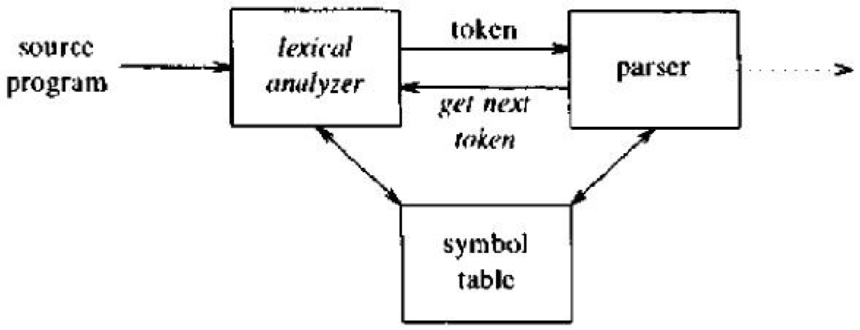
Error handler

- Each phase encounters errors.
- 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.
- Lexical analysis phase can detect errors that do not form any token of the language.
- Syntax analysis phase can detect the token stream that violates the (structure (or) syntax rules of the language.
- Semantic analysis phase detects the constructs that have no meaning to the operation involved.

Phase	Pass
The process of compilation is carried in various steps. Each step is called a phase	One complete scan of the source language is called pass It includes reading an input file and writing to an output file
Different phases include: LA,SA,SeA,ICG,CO,CG	Many phases can be grouped as one pass The task of compilation may be carried out in single pass or multiple passes

Role of Lexical Analysis

- The lexical analyzer is the first phase of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
- Another task of lexical analyzer is stripping out from the source program comments and white space in the form of blank and tab and newline characters.
- Correlating error messages from the compiler with the source program.



- The lexical analyzer may keep track of the number of newline characters seen, so that line number can be associated with an error message.
- In some compilers, the lexical analyzer is in charge of making a copy of the source program with the error messages marked in it.
- If the lexical analyzer finds a token invalid, it generates an error.
- The lexical analyzer works closely with the syntax analyzer.
- It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.
- The lexical analyzer collects information about tokens into their associated attributes.
- After identifying the tokens, the strings are entered into database called a symbol table.
- It works in two phases:
    1. Scan
    2. Separation of tokens

Lexical analysis Vs Parsing

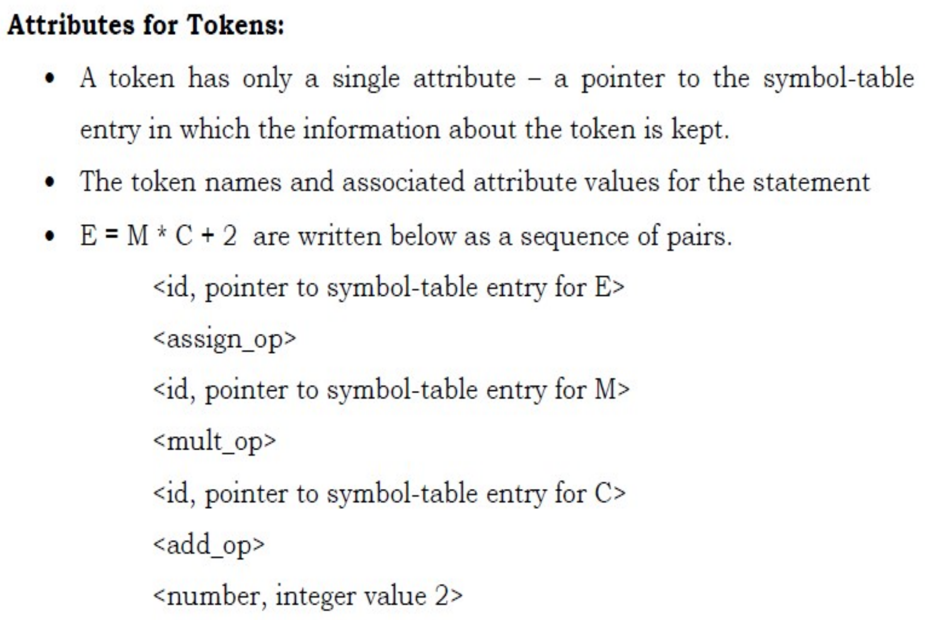
- All compilers separate the task of analyzing syntax into two different parts.
- Lexical and syntax
- Lexical-> small scale language constructs
  - Names and literals
- Syntax-> large scale language constructs
  - expressions, statements and program units

Why lexical analysis is separated from syntax analysis?

1. Simplicity
   - lexical analysis is simplified because it is less complex than syntax analyser
   - Syntax analyser can be smaller and cleaner by removing low level details of lexical analysis
2. Efficiency
   - lexical analysis should be optimized (requires significant portion of total compile time)
   - Syntax analysis should not be optimized
3. Portability
   - Lexical analysis may not be portable because input device-specific peculiarities can be restricted to scanner
   - Syntax analysis is always portable

Token

- Token is a sequence of characters that can be treated as a single logical entity. Sequence of characters having the collective meaning in the source program
- Typical tokens are identifiers, keywords, operators, special symbols, constants.
- **Pattern:** Set of rules that describe tokens
- **Lexeme:** Sequence of characters in the source program that are matched with a pattern of the token
- Ex: keyword if if condition
- relational op <,>,<= < or <= or >



Attributes for Tokens:

- A token has only a single attribute – a pointer to the symbol-table entry in which the information about the token is kept.
- The token names and associated attribute values for the statement
- E = M * C + 2 are written below as a sequence of pairs.

  <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>

  <add_op>

  <number, integer value 2>

Lexical errors

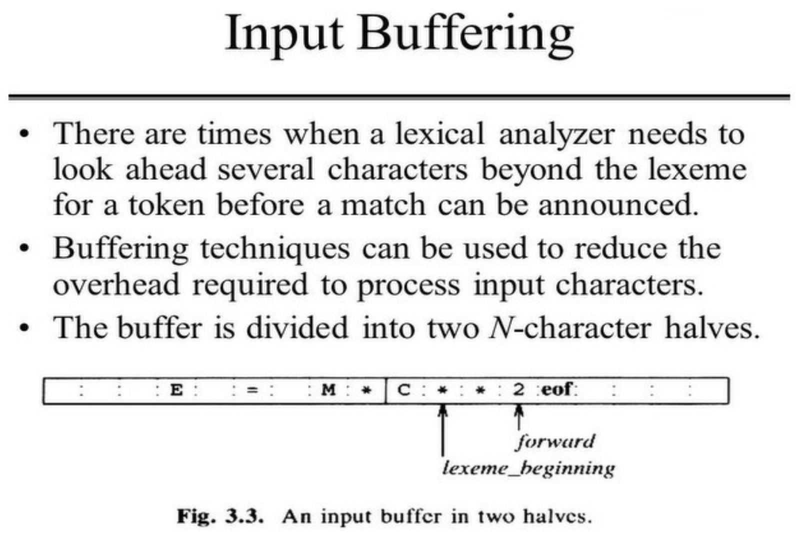
- It is hard for lexical analyzer to tell without aid of other computers, that there is a source code error.
- Some errors are out of power of lexical analyzer to recognize: – fi (a == f(x)) …
- Lexical analyzer can not tell whether fi is a misspelling keyword if or an undeclared function identifier. Since fi is valid lexeme.
- Such errors are recognized when no pattern for tokens matches a character sequence.

- Other phase of the compiler probably parser handle this type of error.
- If lexical analyser unable to proceed because of none of the patterns for tokens matches any prefix of the remaining input ,
- The simplest recovery strategy is **panic mode recovery**

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
  - 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: Buffer Pairs – Sentinels



Input Buffering

- There are times when a lexical analyzer needs to look ahead several characters beyond the lexeme for a token before a match can be announced.
- Buffering techniques can be used to reduce the overhead required to process input characters.
- The buffer is divided into two *N*-character halves.

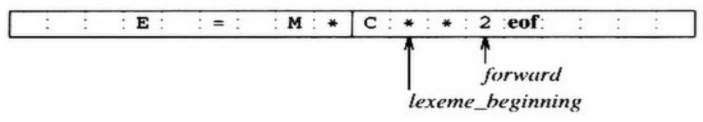


Fig. 3.3. An input buffer in two halves.

Buffer pairs

- Because of the amount of time taken to process characters and number of characters must be processed during the compilation of large source program, specialized buffering techniques have been introduced.
- We need to introduce a two buffer scheme to handle large look-aheads safely

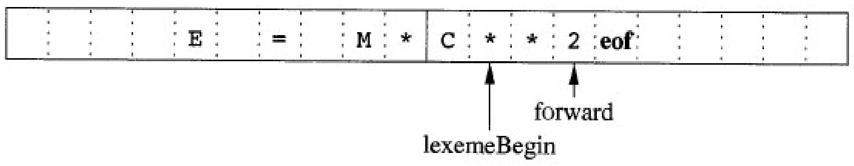
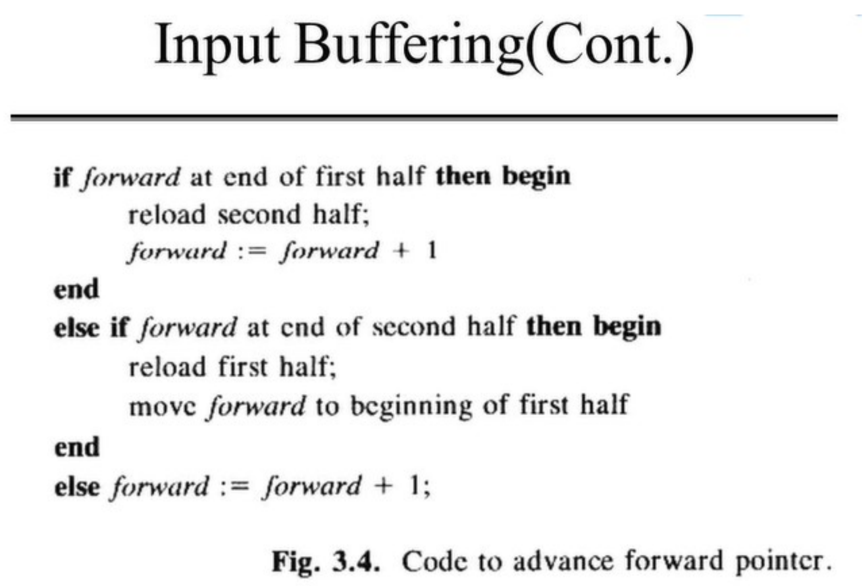


Figure 3.3: Using a pair of input buffers

- Each buffer is of same size N
- N is usually size of disk block
- 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.

- Two pointers to the input are maintained:
- 1. **Pointer lexeme begin** :marks the beginning of the current lexeme
- 2. **Pointer forward:** scans until a pattern match is found
- Once the next lexeme is determined, forward is set to the character at its right end.
- Lexeme begin is set to the character immediately after the lexeme just found.



Input Buffering(Cont.)

```
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;
```

Fig. 3.4. Code to advance forward pointer.

Sentinels

- For each character read we make two tests:
  - one for the end of the buffer
  - 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 sentinel is a special character that can not be part of the source program -eof

- Eof is marked for the end of the entire input.
- Any eof that appears other than at the end of a buffer means that the input is at an end.

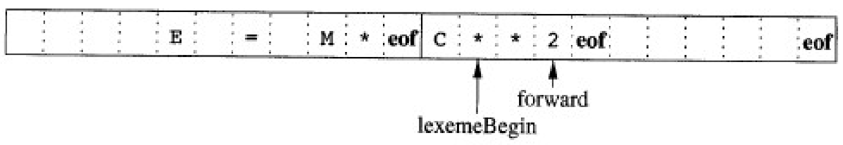
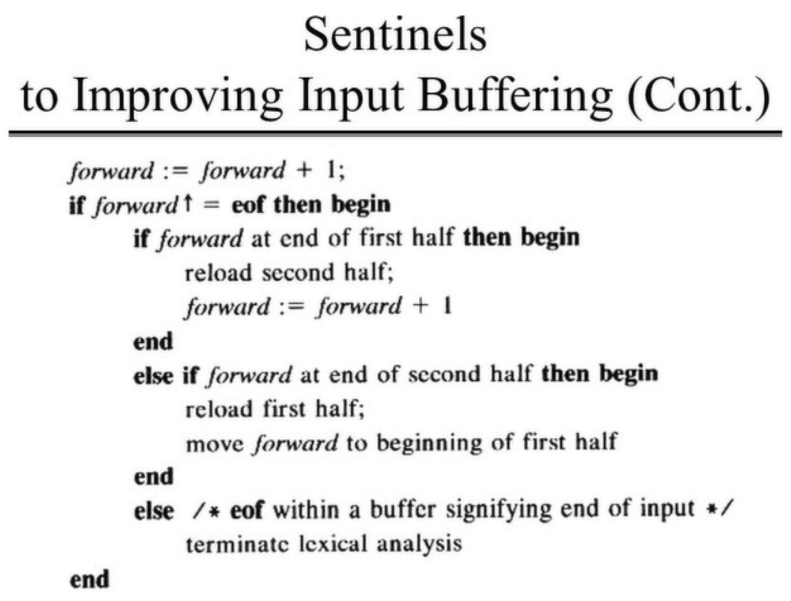


Figure 3.4: Sentinels at the end of each buffer



Sentinels to Improving Input Buffering (Cont.)

```
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
```

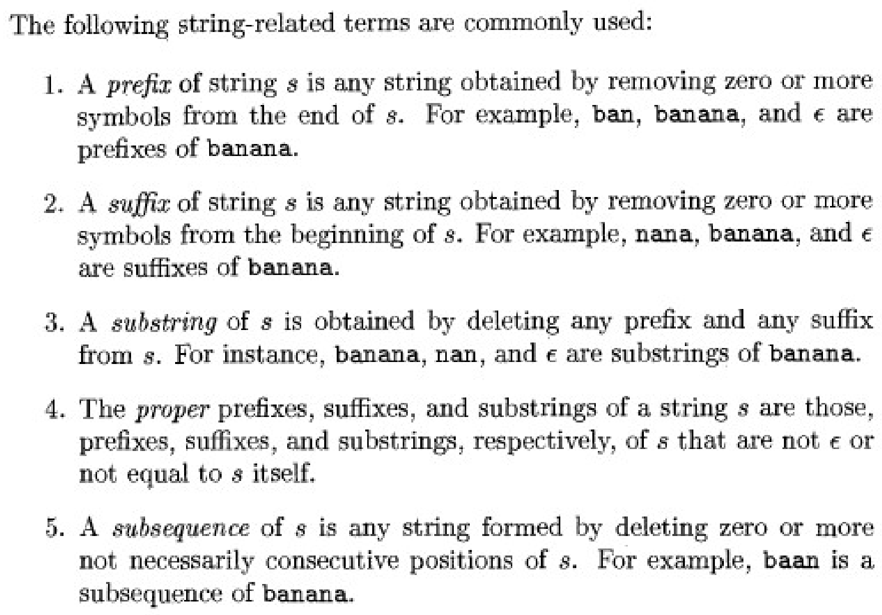
Specification of Tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
  - Strings and Languages
  - Operations on Languages
  - Regular Expressions
  - Regular Definitions

Strings and Languages

Some Concepts:

- *symbol*: letters, digits, and punctuation
- *alphabet:* any finite set of symbols
  e.g. {0,1}, ASCII, Unicode
- *string*: a finite sequence of symbols
  |s|: length of a string *s*
  ∈: empty string
- *language*: any countable set of strings
  e.g. Φ, {∈}, C programs, English sentences



The following string-related terms are commonly used:

1. A *prefix* of string s is any string obtained by removing zero or more symbols from the end of s. For example, **ban**, **banana**, and ϵ are prefixes of **banana**.
2. A *suffix* of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, **nana**, **banana**, and ϵ are suffixes of **banana**.
3. A *substring* of s is obtained by deleting any prefix and any suffix from s. For instance, **banana**, **nan**, and ϵ are substrings of **banana**.
4. The *proper* prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not ϵ or not equal to s itself.
5. A *subsequence* of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, **baan** is a subsequence of **banana**.

Operations on strings:

- *concatenation*: xy

  e.g. 1) x = dog ,y = house ,xy = doghouse.

  2) ∈s=s∈=s

- *exponentiation:*

$$S^0 = \in \qquad S^i = S^{i-1}S$$

$$S^1 = S$$

$$S^2 = SS$$

$$S^3 = SSS$$

Operations on Languages

- *union*: *L U M = {s |s is in L or s is in M}*
- *concatenation: LM = {st |s is in L and t is in M}*
- *closure:*

a) Kleene closure:

b) Positive closure: $L^* = U_{i=0}^{\infty} L^i$

$$L^+ = U_{i=1}^{\infty} L^i$$

1. $L \cup D$ is the set of letters and digits — strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3. L^4 is the set of all 4-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.

Regular Expressions

- **Describing languages**

  e.g. C identifiers: *letter_(letter_|digit)**

  notice:

  a) The regular expressions are built recursively out of smaller regular expressions

  b) Each regular expression *r* denotes a language *L(r)*
- **BASIS: (two rules)**

  1. ∈ is a regular expression, and *L(∈)* is *{∈}*

  2. If a is a symbol in ∑ ,then a is a regular expression, and L(a) = {a}

- **INDUCTION:**

  1. (r)|(s) is a regular expression denoting the language L(r) U L(s)
  2. (r)(s) is a regular expression denoting the language L(r)L(s)
  3. (r)* is a regular expression denoting (L(r))*
  4. (r) is a regular expression denoting L(r)

- **Some conventions:**

  1. * has highest precedence and is left associative
  2. Concatenation has second highest precedence and is left associative

  3. | has lowest precedence and is left associative

  e.g. (a)|((b)*(c)) = a|b*c

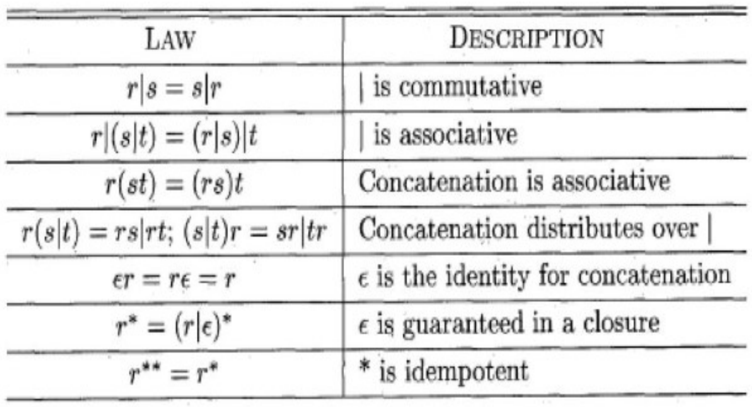
- *regular* set:

  A language that can be defined by a regular expression

- *equivalent*

  Two regular expressions r and s denote the same regular set, write r=s

- 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

- Regular Definition

  A sequence of definitions of the form:

  d1->r1

  d2->r2

  …

  dn->rn

  where:

  1. Each d_i is a new symbol
  2. Each r_i is a regular expression

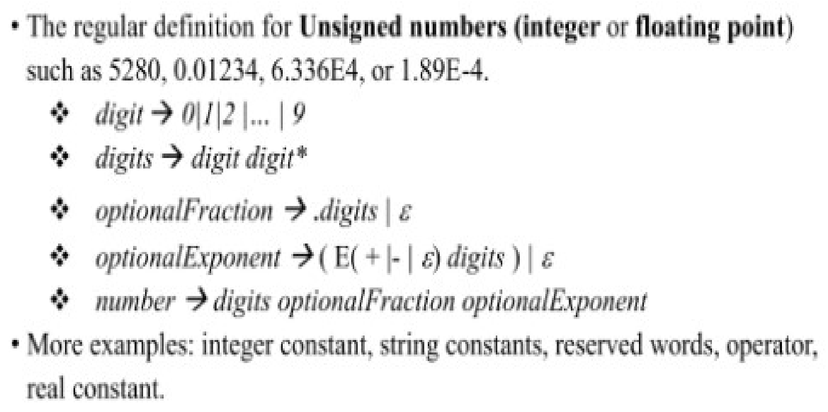
- Example:

  C identifiers

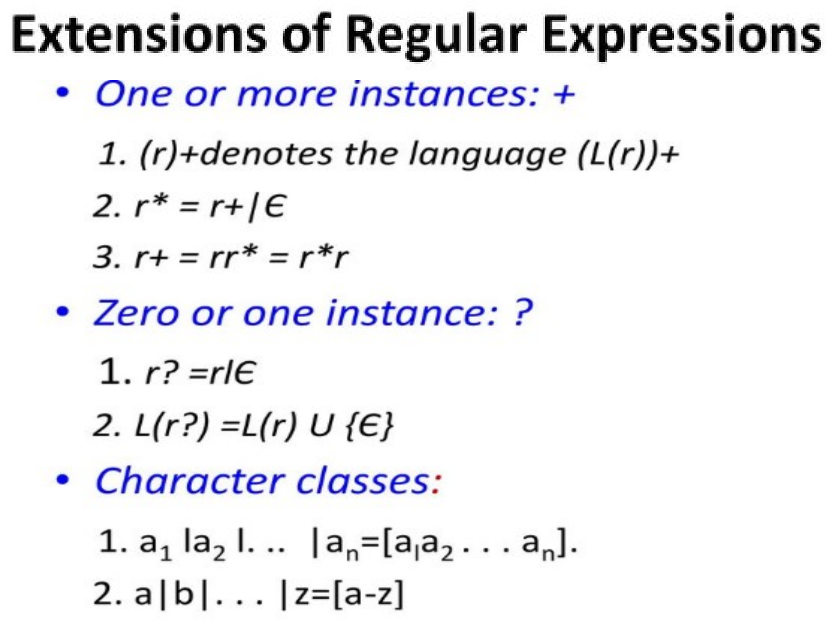
  *letter_ -> A|B|…|Z|a|b| …|z|_*

  digit->0|1|…|9

  *id ->letter_(letter_ | digit)**



- The regular definition for **Unsigned numbers (integer** or **floating point**) such as 5280, 0.01234, 6.336E4, or 1.89E-4.
  - ❖ *digit* → *0|1|2 |… | 9*
  - ❖ *digits* → *digit digit**
  - ❖ *optionalFraction* → *.digits* | ε
  - ❖ *optionalExponent* → (E(+ |- | ε) *digits*) | ε
  - ❖ *number* → *digits optionalFraction optionalExponent*
- More examples: integer constant, string constants, reserved words, operator, real constant.



Extensions of Regular Expressions

- *One or more instances: +*
  1. *(r)+denotes the language (L(r))+*
  2. *r* = r+/∈*
  3. *r+ = rr* = r*r*
- *Zero or one instance: ?*
  1. *r? =rl∈*
  2. *L(r?) =L(r) U {∈}*
- *Character classes:*
  1. a_1 la_2 l. .. |a_n=[$a_l a_2$ . . . a_n].
  2. a|b|. . . |z=[a-z]

Recognition of Tokens

- Transition Diagrams
- Recognition of Reserved Words and Identifiers
- Completion of the Running Example
- Architecture of a Transition–Diagram-Based Lexical Analyzer

Recognition of Tokens

How to recognize tokens?

- *Reserved words: if, else, then…*
- *Id: letter*
- *Number: digit*
- *Relop: <, >, =, <=, >=, <>…*
- *Ws: blank, tab, newline…*

Transition Diagrams

- States: represents a condition
- Edges: directed from one state to another
- Some Conventions:

  1. Accepting or final states
  2. *: retract the forward pointer one position
  3. Start or initial state

Transition Diagrams for >=

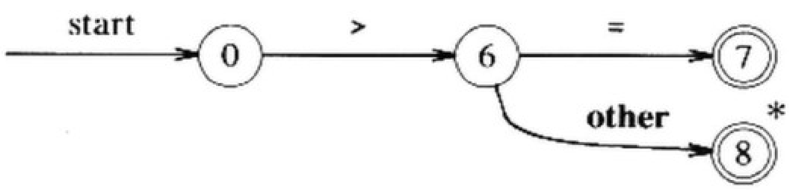


Fig. 3.11. Transition diagram for >=.

- *start* state : stare 0 in the above example
- If input character is >, go to state 6.
- **other** refers to any character that is not indicated by any of the other edges leaving *s*.

Transition Diagrams for Relational Operators

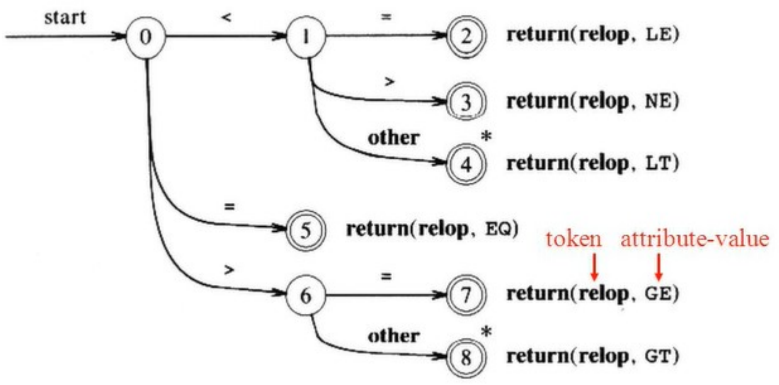
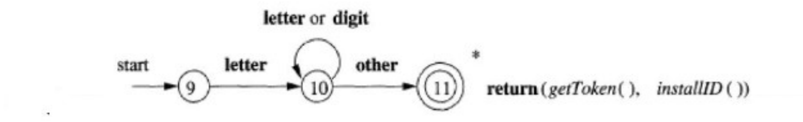


Fig. 3.12. Transition diagram for relational operators.

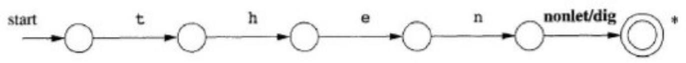
Recognition of Reserved Words and Identifiers

Two ways to handle reserved words:

- Install the reserved words in the symbol table initially

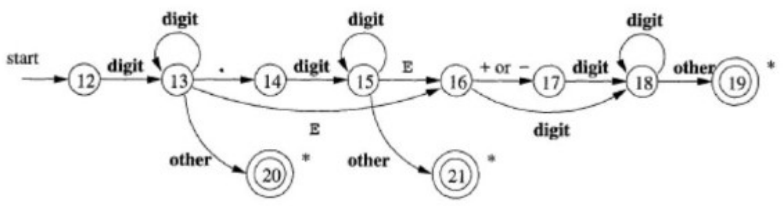


Create separate transition diagrams for each keyword

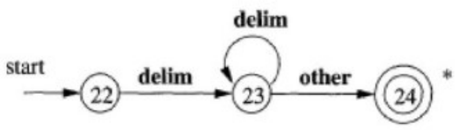


- gettoken( ): return token (**id, if, then**,…) if it looks the symbol table
- install_id( ): return 0 if keyword or a pointer to the symbol table entry if **id**

- Transition diagram for token number



- Transition diagram for whitespace



Implement a Transition Diagrams

- A sequence of transition diagrams can be converted into a program to look for tokens.
- Each state gets a segment of code.

- state and start record the current state and the start state of current transition diagram.
- lexical_value is assigned the *pointer* returned by install_id( ) and install_num( ) when an identifier or number is found.
- When a diagram fails, the function fail( ) is used to *retract* the *forward pointer* to the position of the *lexeme beginning pointer* and to return the start state of the next diagram. If all diagrams fail the function fail( ) calls an error-recovery routine.

Architecture of a Transition-Diagram-Based Lexical Analyzer

- A sketch of getRelop() to simulate the transition diagram for **relop**

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

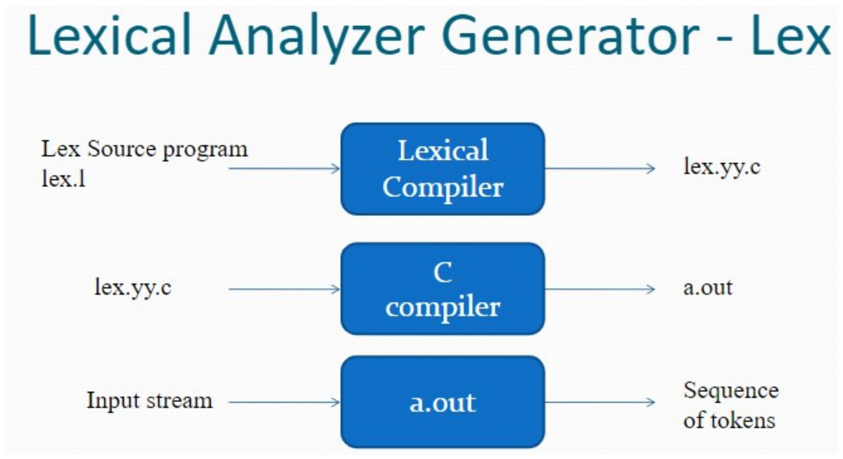
```
int state = 0, start = 0;
int lexical_value;
    /* to "return" second component of token */

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;
}
```

- Ways code fit into the entire lexical analyzer

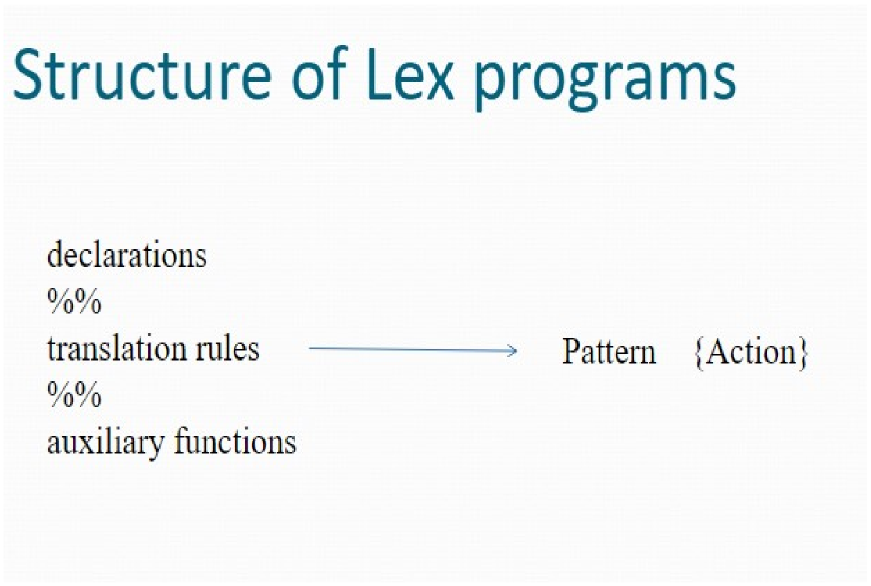
  1. Arrange for the transition diagrams for each token to be tried sequentially

  2. Run the various transition diagrams "in parallel"

  3. Combine all the transition diagrams into one (preferred)



- An input file, which we call lex.1, is written in the Lex language and describes the lexical analyzer to be generated.
- The Lex compiler transforms lex.1 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.
- The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable yylval, which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.



Structure of Lex programs

```
declarations
%%
translation rules          Pattern    {Action}
%%
auxiliary functions
```

$$\text{translation rules} \longrightarrow \text{Pattern} \quad \{\text{Action}\}$$

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

- When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns P.
- It then executes the associated action A.
- Typically, A, will return to the parser, but if it does not (e.g., because P describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser.
- The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable yylval to pass additional information about the lexeme found, if needed.

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}       {/* no action and no return */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = (int) installID(); return(ID);}
{number}   {yylval = (int) installNum(); return(NUMBER);}
"<"        {yylval = LT; return(RELOP);}
"<="       {yylval = LE; return(RELOP);}
```

```
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

The action taken when *id* is matched is

1. Function `installID()` is called to place the lexeme found in the symbol table.
2. This function returns a pointer to the symbol table, which is placed in global variable `yylval`, where it can be used by the parser or a later component of the compiler. Note that `installID()` has available to it two variables that are set automatically by the lexical analyzer that `Lex` generates:
   (a) `yytext` is a pointer to the beginning of the lexeme, analogous to `lexemeBegin` in Fig. 3.3.
   (b) `yyleng` is the length of the lexeme found.
3. The token name `ID` is returned to the parser.

The action taken when a lexeme matching the pattern *number* is similar, using the auxiliary function `installNum()`. □