UE19CS351 : Compiler Design

Chapter 3: Lexical Analysis

1. **The Role of the Lexical Analyzer**
2. **Input buffering**
3. **Specification of Tokens.**
4. **Recognition of Tokens**
5. **Design of a Lexical Analyzer Generator.**

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU,
coprakasha@pes.edu

PES UNIVERSITY

The Role of the Lexical Analyzer

- **The main task of the lexical analyzer** is to
  - **Read the input characters of the source program, group them into meaningful units called 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.
- **When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.**

The Role of the Lexical Analyzer

- The interactions between Lexical analyzer, symbol table and parser are suggested in Fig. 3.1.

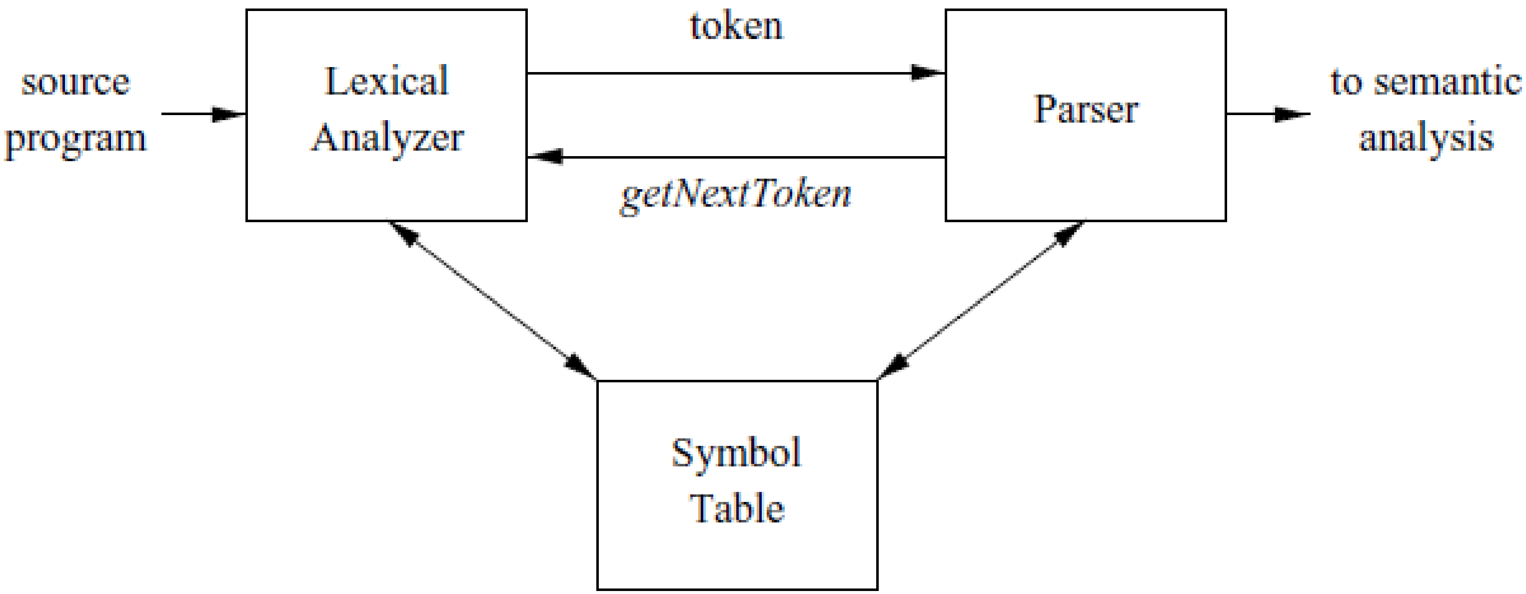


Figure 3 1 Interactions between the lexical analyzer and the parser

The Role of the Lexical Analyzer

The lexical analyzer 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 separate tokens in the input).
- 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**.

The Role of the Lexical Analyzer

The lexical analyzer may perform certain other tasks besides identification of lexemes. Cont..

- In some compilers,
  - 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, **the expansion of macros may also be performed by the lexical analyzer.**

The Role of the Lexical Analyzer

- Sometimes, **lexical analyzers are divided into a cascade of two processes**:

  a) **Scanning** consists of the simple processes such as

     - **deletion of comments** and
     - **compaction of consecutive whitespace characters into one**.

  b) **Lexical analysis** is the more complex portion, where the scanner **produces the sequence of tokens as output.**

The Role of the Lexical Analyzer

- **Assignments:**

1. Write a Lex program to remove comment lines and also count the number of comments in a program.

   Take as input the lex.yy.c file and eliminate all the comments.

2. Write a program in a preferred programming language to develop a simple lexical analyzer.
   - Input : C/C++ program
   - Output : Remove comments ; Generate Tokens; Store generated tokens in table(Symbol table).

The Role of the Lexical Analyzer

Lexical Analysis Versus Parsing

- There are a number of reasons why the analysis portion of a compiler is normally separated into
  1. Lexical Analysis and
  2. Syntax Analysis (parsing) phases.

The Role of the Lexical Analyzer

Lexical Analysis Versus Parsing

1. **Simplicity of design is the most important consideration.**

   - The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.

     For example, **a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex** than one that can assume comments and whitespace have already been removed by the lexical analyzer.

   - **If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.**

The Role of the Lexical Analyzer

Lexical Analysis Versus Parsing

2. **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**.

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

The Role of the Lexical Analyzer

Tokens, Patterns, and Lexemes

- **Token**
  - A **token** is a pair consisting of a **token name** and **an optional attribute value.** *< token-name, attribute-value>*
  - **Token name:**
    - It is an abstract symbol representing a kind of lexical unit.
    - Examples: A particular keyword(E.g. <IF> , <WHILE>), or a sequence of input characters denoting an identifier.
  - The token names are the input symbols that the parser processes.

we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

The Role of the Lexical Analyzer

Tokens, Patterns, and Lexemes

- A **pattern** is a description of the form that the lexemes of a token may take.

  For keywords, the pattern is just the sequence of characters that form the keyword.

```
if    {printf("\nkeyword\n"); return IF;}

while {printf("\nkeyword\n"); return WHILE;}
```

  For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.   <ID, while>

```
[a-zA-Z][a-zA-Z0-9_]*   {printf("\nValid input\n"); yylval = yytext; return ID;}
```

- 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.

The Role of the Lexical Analyzer

- Table below gives some **typical tokens**, their informally described patterns, and some sample lexemes.

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"

Figure 3.2: Examples of tokens

The Role of the Lexical Analyzer

- Example 3.1: To see how these concepts are used in practice, in the C statement

```
printf("Total = %d\n", score);
```

The Role of the Lexical Analyzer

- E x a m p l e 3 . 1 : To see how these concepts are used in practice, in the C statement

```
printf("Total=%d\n", score);
```

  - both **printf** and **score** are lexemes matching the pattern for token **id**, and
  - **"Total = °/,d\n"** is a lexeme matching the pattern for token **literal**.

The Role of the Lexical Analyzer

In many programming languages, the **following classes cover most or all of the tokens**:

1. One token for each **keyword**. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the **operators**, either individually or in classes such as the token comparison mentioned in Fig. 3.2.
3. One token representing all **identifiers**.
4. One or more tokens representing **constants**, such as numbers and literal
5. Tokens for each **punctuation symbol**, such as left and right parentheses, comma, and semicolon.

The Role of the Lexical Analyzer

Attributes for Tokens

- When **more than one lexeme can match a pattern**, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.
  - For example, the pattern for token **number** matches both 10 and 11, but it is extremely important for the code generator to know which lexeme was found in the source program.

The Role of the Lexical Analyzer

Attributes for Tokens

- The lexical analyzer returns to the parser a token; **the token name influences parsing decisions**, while **the attribute value influences translation of tokens after the parse**.

The Role of the Lexical Analyzer

Attributes for Tokens

- **Assume that tokens have at most one associated attribute**, although **this attribute may have a structure that combines 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 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.

The Role of the Lexical Analyzer

- **Example 3.2** : The token names and associated attribute values for the Fortran statement **E = M * C ** 2** are written below as a sequence of pairs.

```
E        <id, pointer to symbol-table entry for E>
=        <assign_op> or <=>
M        <id, pointer to symbol-table entry for M>
*        <mult_op> or <*>
C        <id, pointer to symbol-table entry for C>
**       <exp_op> or <**>
2        <number, integer value 2>
```

- Note that in certain pairs, **especially operators, punctuation, and keywords, there is no need for an attribute value**. In this example, the token ***number*** has been given an integer-valued attribute.

Lexical Analyzer

Lexer implementation is language dependent

Lexical Analyzer

Syntactic Sugar:

- **Syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express.**
- Example:
  - In C : a[i] is a syntactic sugar for *(a+i)
  - In C : a+=b is equivalent to a = a + b
  - In C# : var x = expr (compiler deduces type of x from expr)
- Compilers expand sugared constructs into fundamental constructs (Desugaring).

Lexical Analyzer

- **Scanning is Hard in C++**
  - **C++ template syntax:**
    - `template <class T>`
  - **C++ stream syntax:**
    - `cin >> var;`
  - **C++ binary right shift syntax:**
    - `a >> 4;`
  - **Nested templates:**
    - `A<B<C>>D;`

Lexical Analyzer

Scanning is Hard in PL/I

- Identifiers can be PL/I keywords or programmer-defined names. Because **PL/I can determine from the context if an identifier is a keyword**, you can use any identifier as a programmer-defined name.

- PL/1: Keywords can be used as identifiers.

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

- Can be difficult to determine how to label lexemes.

Lexical Analyzer

Scanning is Hard in Python

Python Blocks

- Scoping handled by whitespace:

```
if w == z:
    a = b
    c = d
else:
    e = f
g = h
```

- What does that mean for the scanner?

In Python,

- **Indentation is not ignored.**
- **Leading whitespace is used to compute a line's indentation level, which in turn is used to determine the block of statements.**

Lexical Analyzer

- **`Scanning is Hard in Python`**
  - **In Python lexical analysis, most of the white space magic is in the lexer, the lexer emits three special tokens: NEWLINE, INDENT,** and **DEDENT**.
  - The parser reads
    - **`NEWLINE`** as the end of a logical line,
    - **`INDENT`** as the beginning of a block and
    - **`DEDENT`** as the end of a block.
  - Note that INDENT and DEDENT encode change in indentation, not the total amount of indentation.

Lexical Analyzer

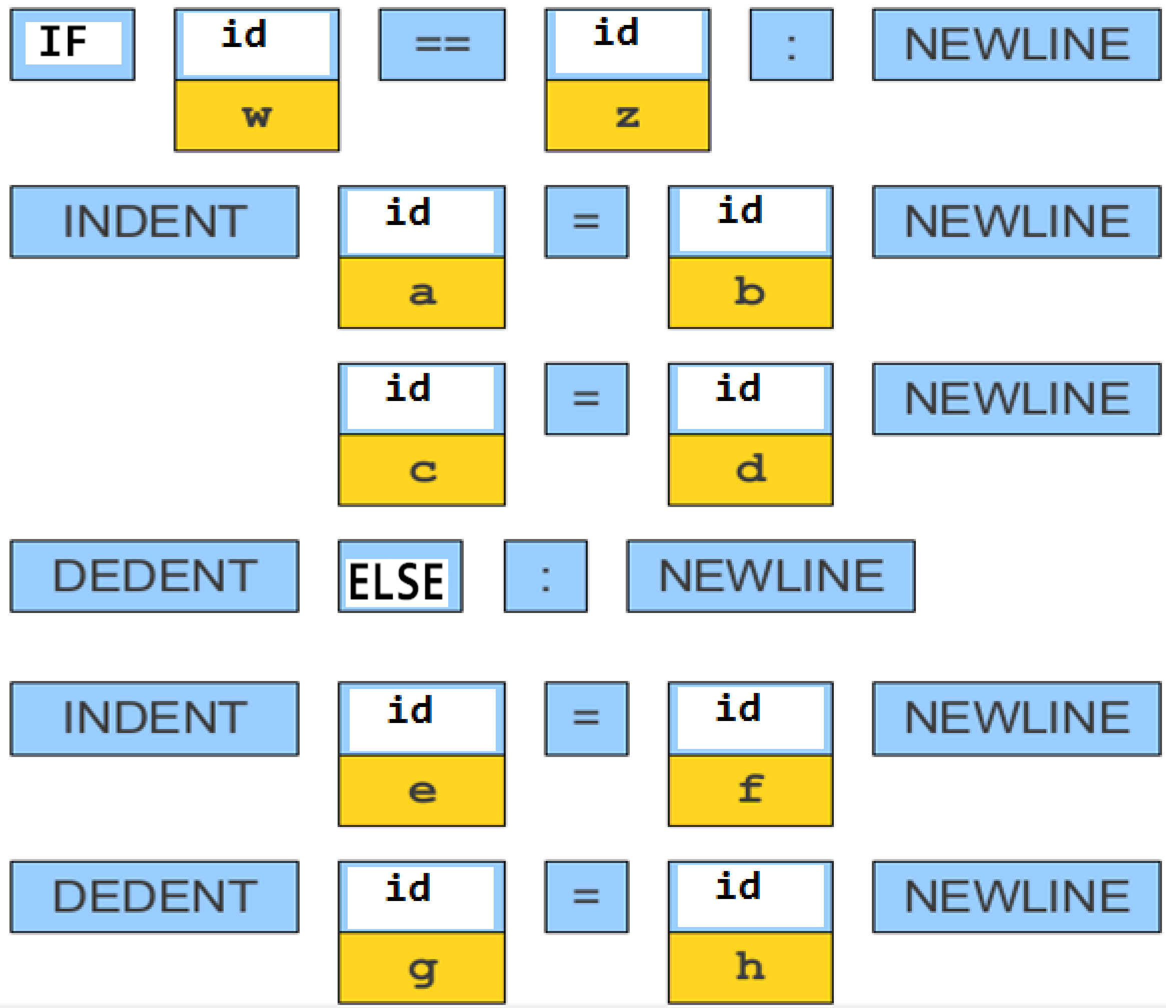
Scanning Python

```
if w == z:
    a = b
    c = d
else:
    e = f
g = h
```

Lexical Analyzer

Scanning Python

```
if w == z:
    a = b
    c = d
else:
    e = f
g = h
```



Lexical Analyzer

- **How INDENT and DEDENT tokens are generated in Python Lexical Analyzer?**
  - **Before the first line of the file is read, a single zero is pushed on to stack**; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top.
  - **At the beginning of each logical line, the line's indentation level is compared to the top of the stack.**
    - **If it is equal, nothing happens.**
    - **If it is larger, it is pushed on to the stack, and one INDENT token is generated.**
    - **If it is smaller, it *must* be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated.**
  - **At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.**

Lexical Analyzer

- Example of a correctly (though confusingly) indented piece of Python code:

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

Lexical Analyzer

- **How INDENT and DEDENT tokens are generated in Python Lexical Analyzer?**
  - **Before the first line of the file is read, a single zero is pushed on the stack; At the beginning of each logical line, the line's indentation level is compared to the top of the stack.**
    - **If it is equal, nothing happens.**
    - **If it is larger, it is pushed on to the stack, and one INDENT token is generated.**
    - **If it is smaller, it *must* be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated.**
  - **At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.**

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

Lexical Analyzer

- The following example shows various indentation errors:

```
 def perm(l):
for i in range(len(l)):
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])
        for x in p:
                r.append(l[i:i+1] + x)
            return r
```

Lexical Analyzer

- The following example shows various indentation errors:

```
 def perm(l):                       # error: first line indented
for i in range(len(l)):             # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # error: inconsistent dedent
```

Lexical Analyzer

- The following example shows various indentation errors:

```
 def perm(l):                       # error: first line indented
for i in range(len(l)):             # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # error: inconsistent dedent
```

Actually, **the first three errors are detected by the parser**; only **the last error is found by the lexical analyzer** — the indentation of statement *`return r`* does not match a level popped off the stack.

Lexical Analyzer

Lexer Feedback

- **C and C++ lexers require lexical feedback to differentiate between *typedef names and identifiers***.

  That is, **the context-sensitive lexer needs help from the "context-free" parser to distinguish between an identifier "foo" and a typedef name "foo".** In this snippet,

  ```
  int foo;
  typedef int foo;
  foo x;
  ```

  the first "foo" is an identifier while the second and third are typedef names.

  You need to parse typedef declarations to figure this out.

- **This means that the parser and lexer are mutually recursive**, so it doesn't make sense to say that the parser is context free while the lexer is context sensitive.

Lexical Analyzer

- **How C/C++ parser/lexer makes the difference between '*' of pointer and '*' of multiplication?**

```
... {
     ...
     obj *var1;     // * used to make var1 as pointer to obj
     var3 = var1 * var2;  // * used to multiply var1 and var2
}
```

- The **tokeniser doesn't make a distinction between the two**. It just treats it as the token *.
- **The parser knows how to look up names**. It knows that obj is a type, so can parse `<type> * <identifier>` differently from `<non-type> * <non-type>`.
- When `typedef int obj;` is parsed, it is interpreted and taken to mean obj now names a type. When parsing continues and `obj *var1;` is seen, the results of the earlier semantic analysis are available for use.

Lexical Analyzer

Contextual keywords

- Sometimes, **we want to add a new keyword to a language, but can't break backward compatibility**.

  For example, when C# added ***async/await***, they needed to retain *async* as an identifier, so that old programs that used variables or classes named *async* would still compile.

  That is to say, **the token *async* should be treated as an identifier under some circumstances**, and **as a keyword under others.**

  A keyword with this property is called a "**contextual keyword**" in C#.

- Then the question is: **when should we lex the string "async" appearing in the source code as an identifier, and when should we lex it as the async token**?

  We need information about the surrounding tokens to decide that.

Tricky Problems When Recognizing Tokens

- **In some languages it is not immediately apparent when we have seen an instance of a lexeme corresponding to a token.**

- **The following example is taken from Fortran**, in the fixed-format still allowed in Fortran 90. In the statement

DO 5 I = 1.25

```
        do label var = expr1, expr2, expr3
        statements
label   continue
```

it is not apparent that the first lexeme is **DO5I**, an instance of the identifier token, until we see the dot following the 1.

Note that **blanks in fixed-format Fortran are ignored** (an archaic convention).

Tricky Problems When Recognizing Tokens

- Had we seen a comma instead of the dot, we would have had a do-statement

DO 5 I = 1,25

```
        do label var = expr1, expr2, expr3
        statements
label   continue
```

in which the first lexeme is the keyword DO.

Standard Fixed Format

The standard fixed format source lines are defined as follows:

The first 72 columns of each line are scanned.

The first five columns must be blank or contain a numeric label.

Continuation lines are identified by a nonblank, nonzero in column 6.

Short lines are padded to 72 characters.

Long lines are truncated.

Tricky Problems When Recognizing Tokens

- Lexical analysis is complicated in some languages. FORTRAN, for example, allows white space inside of lexemes. The FORTRAN statement:

  ```
  DO 5 I = 1.25
  ```

  - is an assignment statement with three tokens:

ID	ASSIGNOP	NUM
DO5I	=	1.25

- but the FORTRAN statement:

  ```
  DO 5 I = 1,25
  ```

  - is a DO-statement with seven tokens:

DOTOK	NUM	ID	ASSIGNOP	NUM	COMMA	NUM
DO	5	I	=	1	,	25

- Before the lexical analyzer can produce the first token it must look ahead to see if there is a dot or a comma in this statement.

The Role of the Lexical Analyzer

Lexical Errors

- **It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.**

- For instance, if the string **fi** is encountered for the first time in a C program in the context:

  fi (a == f(x)) ...

  a lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier.

  Since fi is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.

The Role of the Lexical Analyzer

Lexical Errors

- **A lexical error is any input that can be rejected by the lexer.** This generally results from token recognition falling off the end of the rules you've defined.

- For example:
  - [0-9]+ ===> NUMBER token
  - [a-zA-Z] ===> LETTERS token
  - anything else ===> **error**

- **If lexer is a finite state machine that accepts valid input strings**, then **errors are going to be any input strings that do not result in that finite state machine reaching an accepting state.**

The Role of the Lexical Analyzer

Lexical Errors

- Examples:
  - A lexical error could be an invalid or unacceptable character by the language, like '@' which is rejected as a lexical error for identifiers in Java (it's reserved).
  - Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognize a *lexeme* as a valid *token* for your lexer.

Input Buffering

Input Buffering

- let us examine some ways that the simple but **important task of reading the source program can be speeded.**

  This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.

Input Buffering

- **There are many situations where we need to look at least one additional character ahead.**

Input Buffering

- **There are many situations where we need to look at least one additional character ahead.**
  - For instance, we cannot be sure we've seen the **end of an identifier until we see a character that is not a letter or digit**, and therefore is not part of the lexeme for **id**.
  - In C, **single-character operators** like -, =, or < **could also be the beginning of a two-character operator** like ->, ==, or <=.
- Thus, a **two-buffer scheme** that **handles large lookaheads safely is introduced.**
- We then consider an improvement involving "**sentinels**" that **saves time checking for the ends of buffers.**

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.

- **An important scheme involves two buffers that are alternately reloaded**, as suggested in Fig. 3.3.

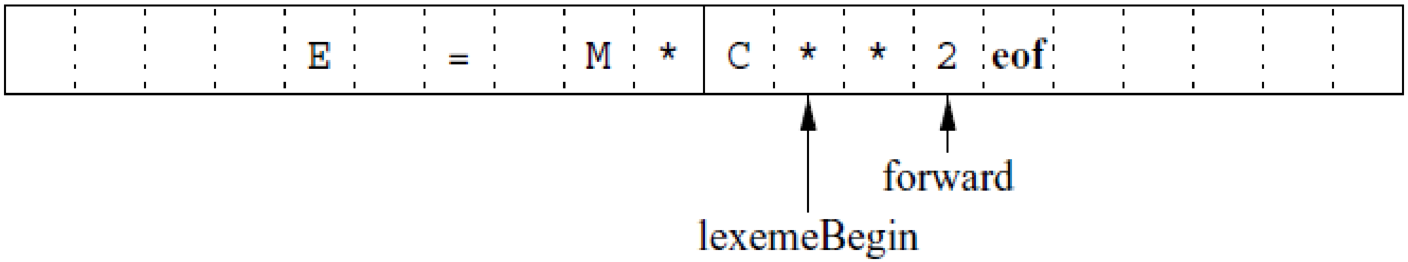


Figure 3.3: Using a pair of input buffers

Input Buffering

Buffer Pairs

- 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.
- 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.

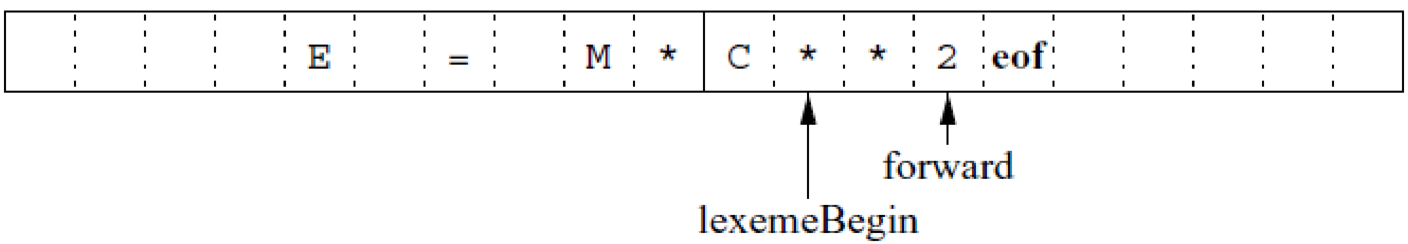
Input Buffering

Buffer Pairs

- 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**;



Input Buffering

Buffer Pairs

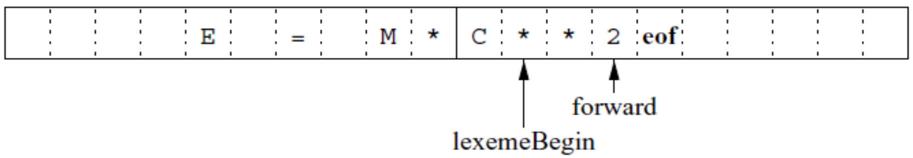
- Once the next lexeme is determined,
  - **forward** is set to
  - **lexemeBegin** is set to
  -

Input Buffering

Buffer Pairs

- **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.
  - In Fig. 3.3 below, we see **forward** has passed the end of the next lexeme, ** (the Fortran exponentiation operator), and **must be retracted one position to its left**.



Input Buffering

Buffer Pairs

- **Advancing forward pointer** 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 from the input, and
  - move forward to the beginning of the newly loaded buffer.

Input Buffering

Sentinels

- **If we use Buffer Pairs, 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 ***sentinel*** **is a special character that cannot be part of the source program**, and a natural choice is the character **eof.**

Input Buffering

Sentinels

- Figure 3.4 shows the same arrangement as Fig. 3.3, but with the sentinels added. Note that **eof** retains its use as a marker 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.**

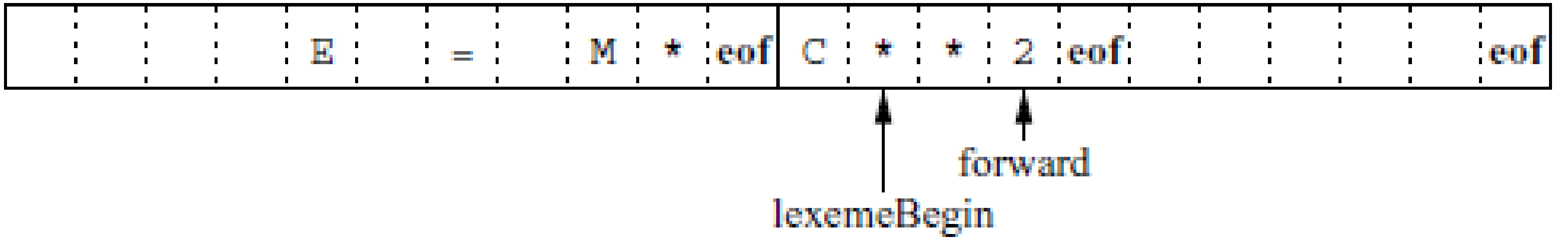


Figure **3.4: Sentinels at the end of each buffer**

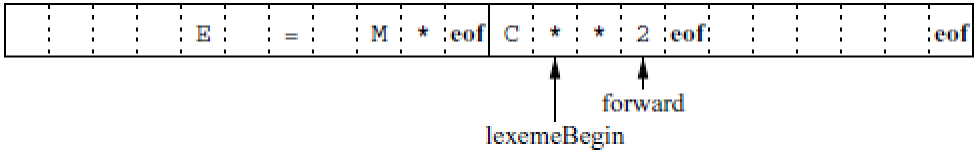
Input Buffering

Sentinels

- Figure 3.5 summarizes the **algorithm for advancing forward**.
  - Notice how the first test, which can be part of a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

Input Buffering

Sentinels




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

Figure 3.5: Lookahead code with sentinels

Specification of Tokens

Specification of Tokens

- **Regular expressions** are an **important notation for specifying lexeme patterns.**

- we shall study **the formal notation for regular expressions**, and we shall see **how these expressions are used in a lexical-analyzer generator**.

Specification of 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.

- **Unicode**, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet.

Specification of Tokens

Strings and Languages

- A **string** over an alphabet is a **finite sequence of symbols** drawn from that 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.

Specification of Tokens

Strings and Languages

- A **language** is any **countable set of strings over some fixed alphabet**. This definition is very broad.
- Abstract languages like $\emptyset$, the empty set, or $\{\epsilon\}$, the set containing only the empty string, are languages under this definition.

Specification of Tokens

Terms for Parts of Strings

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.

Specification of Tokens

Terms for Parts of Strings

The following string-related terms are commonly used:

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.

Specification of Tokens

- If x and y are strings, then the **concatenation of x and y**, denoted **xy**, is the string formed by appending y to x.

- For example, if x = *dog* and *y* = *house*, then *xy* — *doghouse*. The **empty string is the identity under concatenation**; that is, for any string s, **$\epsilon s = s\epsilon = s$**.

- If we think of **concatenation as a product**, we can define the "**exponentiation" of strings** as follows.

  Define **s^0 to be ϵ**, and for all i > 0, define s^i to be $s^{i-1}s$. Since $\epsilon s = s$, it follows that $s^1 = s$. Then $s^2 = ss$, $s^3 = sss$, and so on.

Specification of Tokens

Operations on Languages

- In lexical analysis, the most **important operations on languages are**
  - **union**,
  - **concatenation**, and
  - **closure**
- **Union** is the familiar operation on sets.
- The **concatenation of languages** is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them.

Specification of Tokens

Operations on Languages

- The **(Kleene) closure of a language L**, denoted **L^***, is the set of strings you get by concatenating L zero or more times.
  - Note that **L°, the "concatenation of L zero times**," is defined to be {ϵ}, and inductively, **L^i** is **$L^{i-1}L$.**

- The **positive closure of a language L**, denoted **L^+**, is the same as the **Kleene closure, but without the term L°.** That is, ϵ will not be in L^+ unless it is in L itself.

Specification of Tokens

Fig 3.6 Definitions of Operations on Languages

OPERATION	DEFINITION AND NOTATION
Union of L and M	L U M = {s | s is in L or s is in M}
Concatenation of L and M	LM = {st | s is in L and t is in M}
Kleene closure of L	$L^* = U^{\infty}_{i=0} L^i$
Positive closure of	$L^+ = U^{\infty}_{i=1} L^i$

Figure 3.6: Definitions of operations on languages

Specification of Tokens

Operations on Languages

- Example: Let L={A,B,….,Z,a,b,….,z} and D={0,1,2,….,9}
  - L U D is
  - LD is
  - L^4 is
  - L* is
  - L(LUD)* is
  - D^+ is

Specification of Tokens

Operations on Languages

- Example: Let L={A,B,….,Z,a,b,….,z} and D={0,1,2,….,9}

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.

Specification of Tokens

Regular Expressions

- A **regular expression** is **a sequence of characters that define a search pattern.**

- If **letter_** is established to **stand for any letter** or **the underscore**, and **digit** is established to **stand for any digit**,
  then we could describe the language of C identifiers by:

letter_ (letter_ | digit)*

Specification of Tokens

Regular Expressions

- The **regular expressions are built recursively out of smaller regular expressions**. Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r's subexpressions.

- Here are the rules that define the regular expressions over some **alphabet** ∑ and the languages that those expressions denote.

- **BASIS: There are two rules that form the basis:**

1. ϵ is a regular expression, and $L(\epsilon)$ is $\{\epsilon\}$, that is, the language whose sole member is the empty string.

2. If a is a symbol in ∑, then **a** is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with a in its one position.

Specification of Tokens

Regular Expressions

- **INDUCTION:** There are four parts to the induction whereby larger regular expressions are built from smaller ones.

  Suppose **r** and **s** are regular expressions denoting languages L(r) and L(s), respectively.

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).

   This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

Specification of Tokens

Regular Expressions

a) The unary operator * has **highest precedence** and is left associative.

b) Concatenation has **second highest precedence** and is left associative.

c) | has **lowest precedence** and is left associative.

- Under these conventions, for example, we may replace the regular expression **(a)|((b)*(c))** by **a|b*c.**

  Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.

Specification of Tokens

Regular Expressions

- Figure 3.7 shows some of the **algebraic laws** that hold for arbitrary regular expressions r, s, and t.

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

Specification of Tokens

Regular Definitions

- **For notational convenience**, we may wish to **give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols.**

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

$$\mathbf{d_1 \rightarrow r_1}$$
$$\mathbf{d_2 \rightarrow r_2}$$
$$\ldots\ldots$$
$$\mathbf{d_n \rightarrow r_n}$$

where:

- Each d_i is a new symbol, not in ∑ and not the same as any other of the d's and
- Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Specification of Tokens

Regular Definitions

- **Example 3.5:** A regular definition for the language of C identifiers.

Specification of Tokens

Regular Definitions

- **Example 3.5:** A regular definition for the language of C identifiers.

$$\text{letter\_} \rightarrow A \mid B \mid \ldots. \mid Z \mid a \mid b \mid \ldots. \mid z \mid \_$$

$$\text{digit} \rightarrow 0 \mid 1 \mid \ldots. \mid 9$$

$$\text{id} \rightarrow \text{letter\_} \ (\ \text{letter\_} \mid \text{digit}\ )^*$$

Specification of Tokens

Regular Definitions

- **Example 3.6: Unsigned numbers (integer or floating point)** are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

Specification of Tokens

Regular Definitions

- **Example 3.6: Unsigned numbers (integer or floating point)** are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

digit	$\rightarrow$
digits	$\rightarrow$
optionalFraction	$\rightarrow$
optionalExponent	$\rightarrow$
number	$\rightarrow$

Specification of Tokens

Regular Definitions

- **Example 3.6: Unsigned numbers (integer or floating point)** are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

$$
\begin{array}{rcl}
\textbf{digit} & \rightarrow & \mathbf{0 \mid 1 \mid \ldots. \mid 9} \\
\textbf{digits} & \rightarrow & \textbf{digit digit}^* \\
\textbf{optionalFraction} & \rightarrow & \textbf{\textbackslash. digits} \mid \epsilon \\
\textbf{optionalExponent} & \rightarrow & \mathbf{(\,E\,(\,+ \mid - \mid \epsilon\,)\ \textbf{digits}\,)} \mid \epsilon \\
\textbf{number} & \rightarrow & \textbf{digits optionalFraction optionalExponent}
\end{array}
$$

Specification of Tokens

Extensions of Regular Definitions

- Many extensions have been added to regular expressions to enhance their ability to specify string patterns.

1. ***One or more instances***. *The unary, postfix operator* **+** *represents the positive closure of a regular expression and its language*.

   If **r** is a regular expression, then $(r)^+$ denotes the language $(L(r))^+$.

   The operator + has the same precedence and associativity as the operator *.

   Two useful algebraic laws, $r^* = r^+ | \epsilon$ *and* $r^+ = r\,r^* = r^*\,r$ *relate the* ***Kleene closure*** and **positive closure**.

Specification **of Tokens**

Extensions of Regular Definitions

2. **Zero or one instance.** The unary postfix operator **?** means **"zero or one occurrence."** That is, **r? is equivalent to r|ϵ,** or put another way,

   L(r?) = L(r) U {ϵ}.

   The ? operator has the same precedence and associativity as * and +.

3. **Character classes.** A regular expression $\mathbf{a_1 | a_2 | \cdots | a_n}$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $\mathbf{[a_1 a_2 \ldots a_n]}$.

   When $\mathbf{a_1, a_2, \ldots, a_n}$ **form a logical sequence, we can replace them by** $\mathbf{a_1\text{-}a_n}$. Thus, **[abc] is shorthand for a|b|c**, and **[a-z] is shorthand for a|b|....|z .**

Specification of Tokens

Extensions of Regular Definitions

- Example 3.7 Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$
\begin{aligned}
letter\_ &\rightarrow A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z \mid \_ \\
digit &\rightarrow 0 \mid 1 \mid \ldots \mid 9 \\
id &\rightarrow letter\_ \, (\, letter\_ \mid digit \,)^*
\end{aligned}
$$

- The regular definition of Example 3.6 can also be simplified:

$$
\begin{aligned}
digit &\rightarrow 0 \mid 1 \mid \ldots \mid 9 \\
digits &\rightarrow digit\ digit^* \\
optionalFraction &\rightarrow \backslash.\ digits \mid \epsilon \\
optionalExponent &\rightarrow (\, E\, (\, + \mid - \mid \epsilon \,)\ digits \,) \mid \epsilon \\
number &\rightarrow digits\ \ optionalFraction\ \ optionalExponent
\end{aligned}
$$

Specification of Tokens

Extensions of Regular Definitions

- Example 3.7 Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

  letter_ → [A-Za-z_]

  digit → [0-9]

  id → letter_ (letter_ | digit)*

- The regular definition of Example 3.6 can also be simplified:

  digit → [0-9]

  digits → digit⁺

  number → digits (\. digits)? | digit \. digits E [+-]? digits

Recognition of Tokens

Recognition of Tokens

- Now, **we must study**
  - **how to take the patterns for all the needed tokens** and
  - **build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns**.

- Our discussion will make use of the following running example

$$\begin{aligned} \text{stmt} &\rightarrow \textbf{if } \text{expr } \textbf{then } \text{stmt} \mid \textbf{if } \text{expr } \textbf{then } \text{stmt } \textbf{else } \text{stmt} \mid \epsilon \\ \text{expr} &\rightarrow \text{term } \textbf{relop } \text{term} \mid \text{term} \\ \text{term} &\rightarrow \textbf{id} \mid \textbf{number} \end{aligned}$$

Figure 3.10: A grammar for branching statements in Pascal language

Recognition of Tokens

$$\text{stmt} \rightarrow \textbf{if} \ \text{expr} \ \textbf{then} \ \text{stmt} \mid \textbf{if} \ \text{expr} \ \textbf{then} \ \text{stmt} \ \textbf{else} \ \text{stmt} \mid \epsilon$$

$$\text{expr} \rightarrow \text{term} \ \textbf{relop} \ \text{term} \mid \text{term}$$

$$\text{term} \rightarrow \textbf{id} \mid \textbf{number}$$

Figure 3.10: A grammar for branching statements

- **Example 3.8 :** The grammar fragment of Fig. 3.10 describes a simple form of branching statements and conditional expressions.

- The **terminals of the grammar**, which are **if, then, else, relop, id,** and **number**, **are the names of tokens.**
  The patterns for these tokens are described using regular definitions, as in Fig. 3.11.

Recognition of Tokens

- **Figure 3.11:** Patterns for tokens of Example 3.8

digit → [0-9]

digits → digit$^+$

letter_ → [A-Za-z_]

number → digits (\. digits)? | digit \. digits E [+-]? digits

id → letter_ (letter_ | digit)*

if → if

then → then

else → else

relop → < | > | <= | >= | = | <>

- **Note: In Pascal,**
  = is "equals" and
  <> is "not equals".

- For this language, **the lexical analyzer will recognize the keywords if, then,** and **else , as well as lexemes that match the patterns for *relop, id,*** *and* ***number*.**

Recognition of Tokens

- The lexical analyzer has an additional job of stripping out whitespace, by recognizing **the "token" ws defined by**:

  ws $\rightarrow$ (blank | tab | newline)$^+$

  - Here, **blank, tab, and newline are abstract symbols** that we use to express the ASCII characters of the same names.
  - **Token ws is different from the other tokens** in that, **when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace.** It is the following token that gets returned to the parser.

Recognition of Tokens

- Our goal for the lexical analyzer is summarized in Fig. 3.12.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Figure 3.12: Tokens, their patterns, and attribute values

- This table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value is returned.

- Note that for the six relational operators, symbolic constants **LT, LE,** and so on are used as the attribute value, in order to indicate which instance of the token **relop** we have found.

Recognition of Tokens

Transition Diagrams

- **As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "transition diagrams."**

- Transition diagrams have a collection of nodes or circles, called **states**.

  Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

Recognition of Tokens

Transition Diagrams

- **Edge**s are directed from one state of the transition diagram to another.

  Each edge is labeled by a symbol or set of symbols.

- We shall assume that all our **transition diagrams are deterministic.**

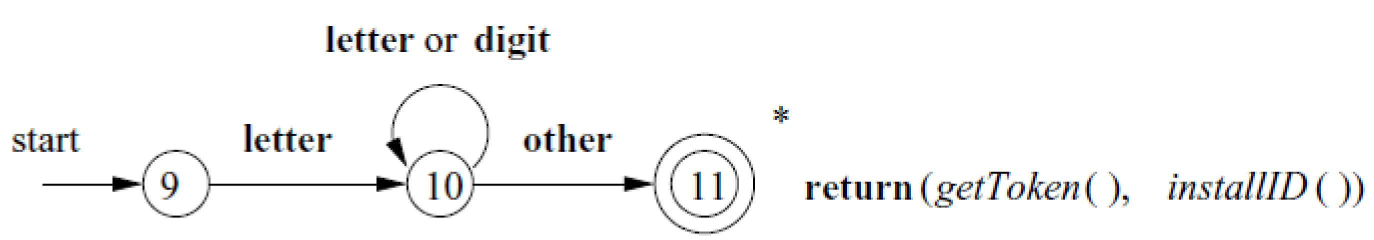
Recognition of Tokens

Transition Diagrams

➢ Some important conventions about transition diagrams are:

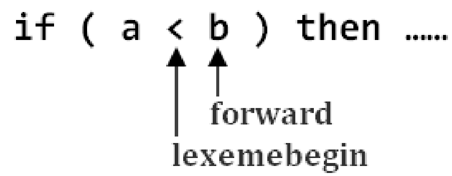
1. **Accepting, or final states indicate that a lexeme has been found**, although the actual lexeme may not consist of all positions between the lexemeBegin and forward pointers.

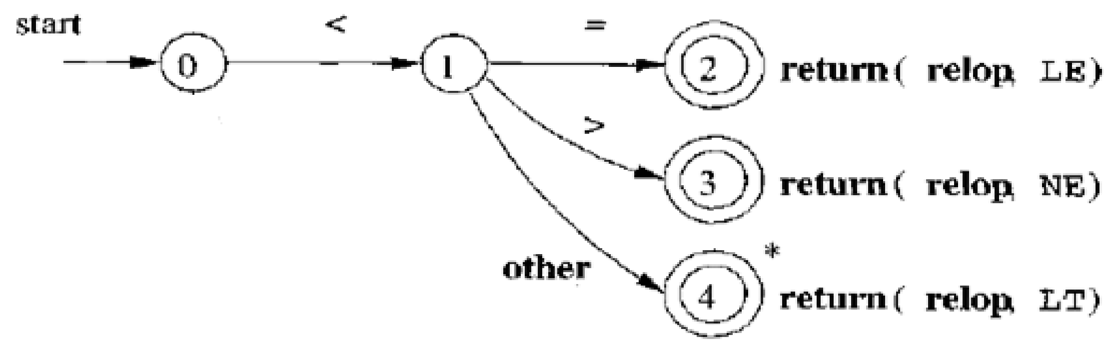
   In accepting state, if there is an action to be taken — typically returning a token and an attribute value to the parser — we shall attach that action to the accepting state.



Recognition of Tokens

Transition Diagrams





- Some important conventions about transition diagrams are:

2. In addition, **if it is necessary to retract the *forward pointer* one position** (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally **place a * near that accepting state**.

   In our example, it is never necessary to retract forward by more than one position, but if it were, we could attach any number of *'s to the accepting state.

3. One state is designated the **start state**, or **initial state**; it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Recognition of Tokens

Transition Diagrams

- **Example 3.9 :** Figure 3.13 is a transition diagram that recognizes the lexemes matching the token **relop.**

Recognition of Tokens

Transition Diagrams

- **Example 3.9 :** Figure 3.13 is a transition diagram that recognizes the lexemes matching the token **relop.**

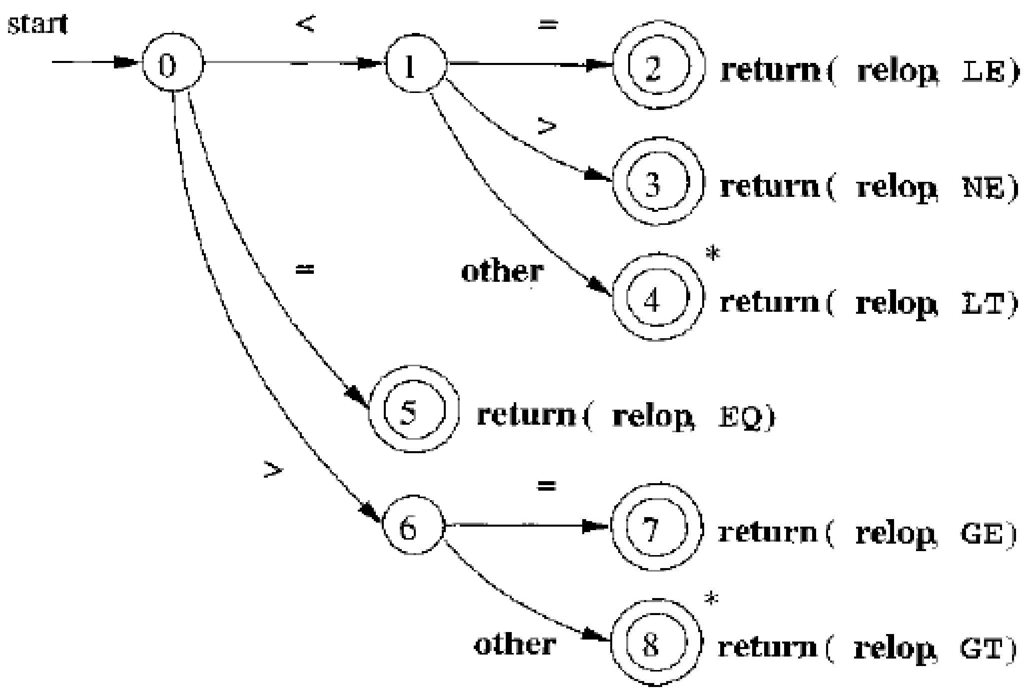


Figure 3.13: Transition diagram for **relop**

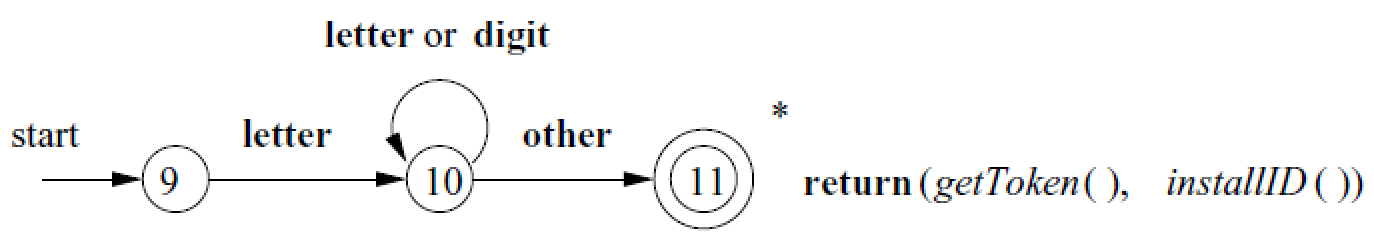
Recognition of Tokens

Recognition of Reserved Words and Identifiers

- **Recognizing keywords and identifiers presents a problem**. Usually, **keywords** like **if** or **then** are reserved, so they **are not identifiers even though they look like identifiers**.

- A transition diagram like that of Fig. 3.14 is used to search for identifier lexemes, will also recognize the keywords if, then, and else of our running example.

- **Figure 3.14:** A transition diagram for **id**'s and **keywords**



Recognition of Tokens

Recognition of Reserved Words and Identifiers

- There are two ways that we can handle reserved words that look like identifiers:

1. **Install the reserved words in the symbol table initially.**

A field of the symbol-table entry indicates that

- **these strings are never ordinary identifiers, and**
- **tells which token they represent.**

1	if	IF
2	then	THEN
3	else	ELSE
4	…	…

This method is in use in Fig. 3.14. **When we find an identifier**, **a call to *installID***

- **places it in the symbol table if it is not already there** and
- **returns a pointer to the symbol-table entry for the lexeme found.**

Any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id.

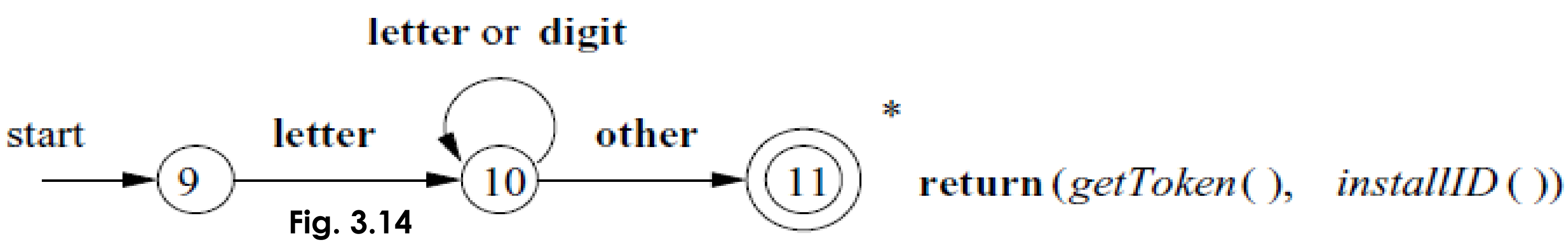


Fig. 3.14

Recognition of Tokens

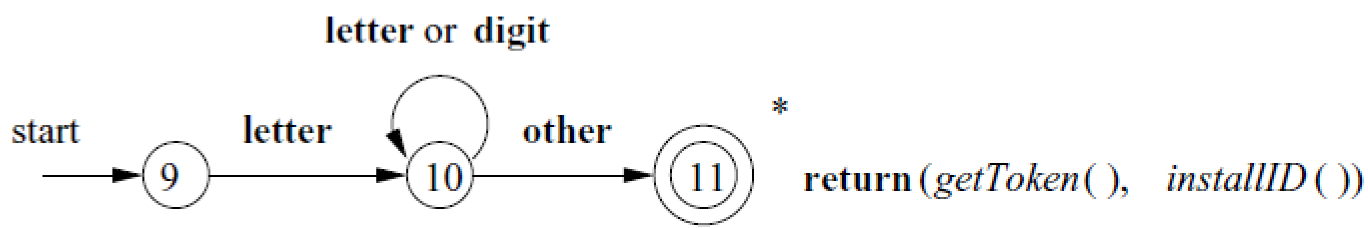
Recognition of Reserved Words and Identifiers

- There are two ways that we can handle reserved words that look like identifiers:

1. **Install the reserved words in the symbol table initially** **Cont...**

   The function *getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either **id or one of the keyword tokens** that was initially installed in the table.

1	if	IF
2	then	THEN
3	else	ELSE
4	…	…
53	sum	ID
54	avg	ID



Recognition of Tokens

Recognition of Reserved Words and Identifiers

2. **Create separate transition diagrams for each keyword.**

- An example for the keyword then is shown in Fig. 3.15.
- **A transition diagram of Fig. 3.15 consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an keyword/identifier.**
- **It is necessary to check that the identifier has ended**, or else we would return token **then** (as keyword) in situations where the correct token was id, with a lexeme like **thennextvalue** that has **then** as a proper prefix.

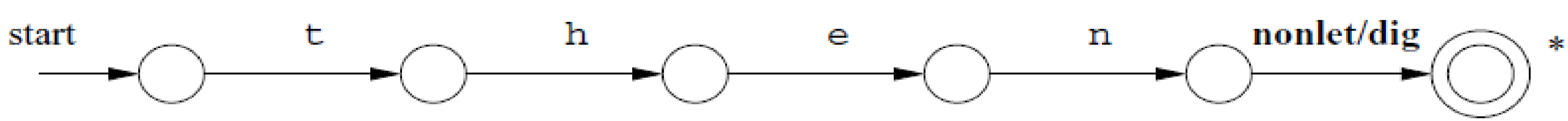


Figure 3 15 Hypothetical transition diagram for the keyword then

Recognition of Tokens

Recognition of Reserved Words and Identifiers

2.

If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme matches both patterns. ***We do not use this approach in our example****, which is why the* states in Fig. 3.15 are unnumbered.

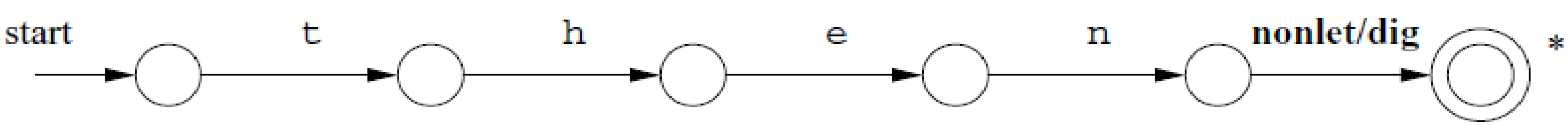
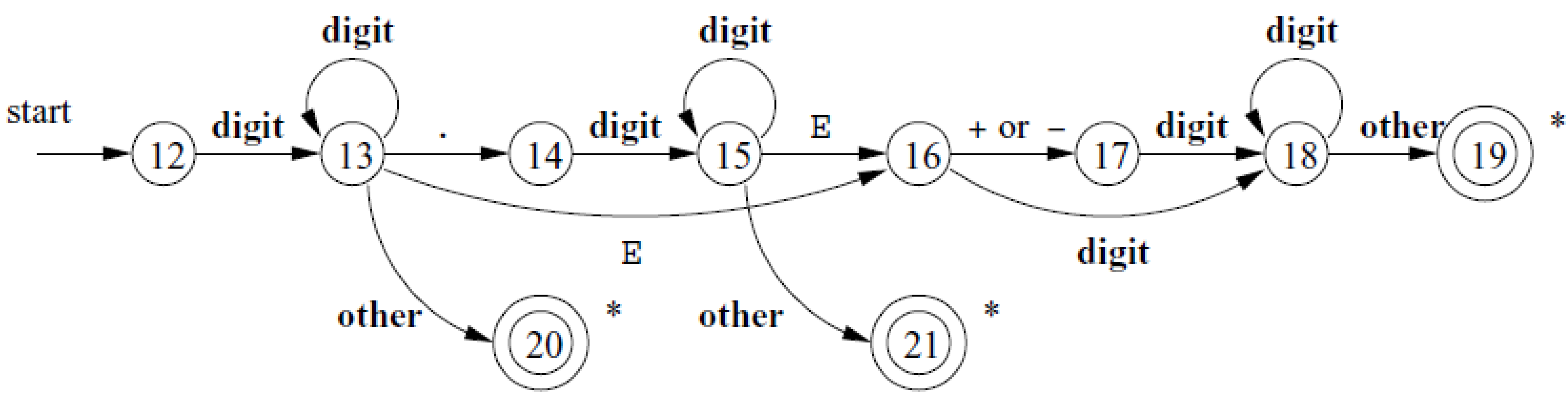


Figure 3.15 Hypothetical transition diagram for the keyword `then`

Recognition of Tokens

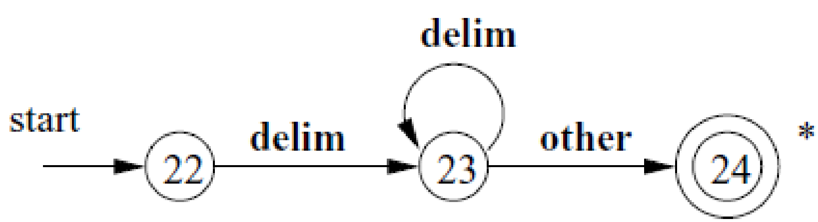
Transition Diagrams

- **Figure 3.16:** A transition diagram for unsigned numbers



Check for examples 15567, 55.99, 5.105E255, 6.22E-88

- **Figure 3.17:** A transition diagram for whitespace



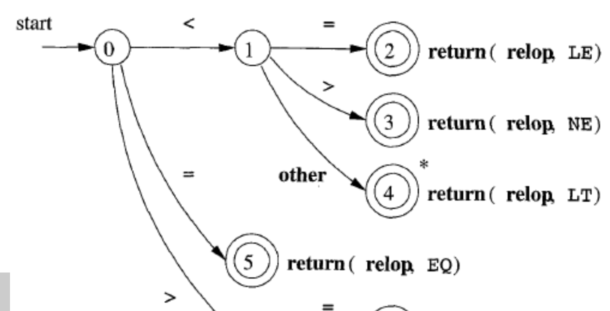
Recognition of Tokens

Architecture of a Transition-Diagram-Based Lexical Analyzer

- There are **several ways that a collection of transition diagrams can be used to build a lexical analyzer**.
  - **Each state is represented by a piece of code**. (fig. 3.18)
  - We may imagine a **variable state** holding the **number** of the current state for a transition diagram.
  - A **switch based** on the value of **state** takes us to code for each of the possible states, where we find the action of that state.

    Often, the code for a state is itself a switch statement or multiway branch that determines the next state by reading and examining the next input character.

```
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;
```



Recognition of Tokens

Architecture of a Transition-Diagram-Based Lexical Analyzer

```
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);
        }
    }
}
```

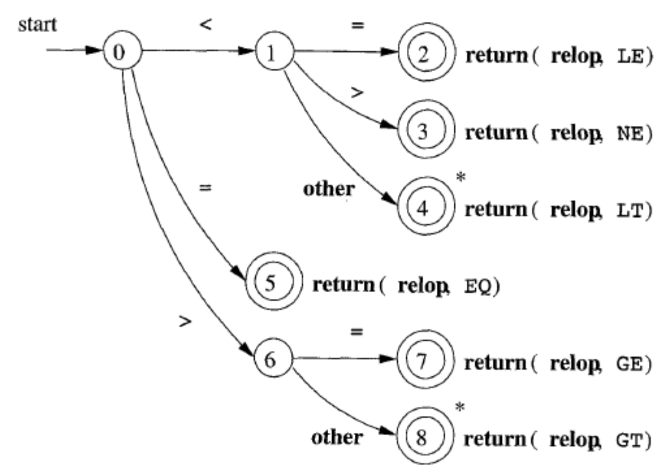


Figure 3.13: Transition diagram for **relop**

Figure 3.18: Sketch of implementation of **relop** transition diagram

Example 3 .10 : In Fig. 3.18 we see a sketch of getRelop(), a C++ function whose job is to simulate the transition diagram of Fig. 3.13 and return an object type TOKEN, that is, a pair consisting of the token name (which must be **relop** in this case) and an attribute value (the code for one of the six comparison operators in this case).

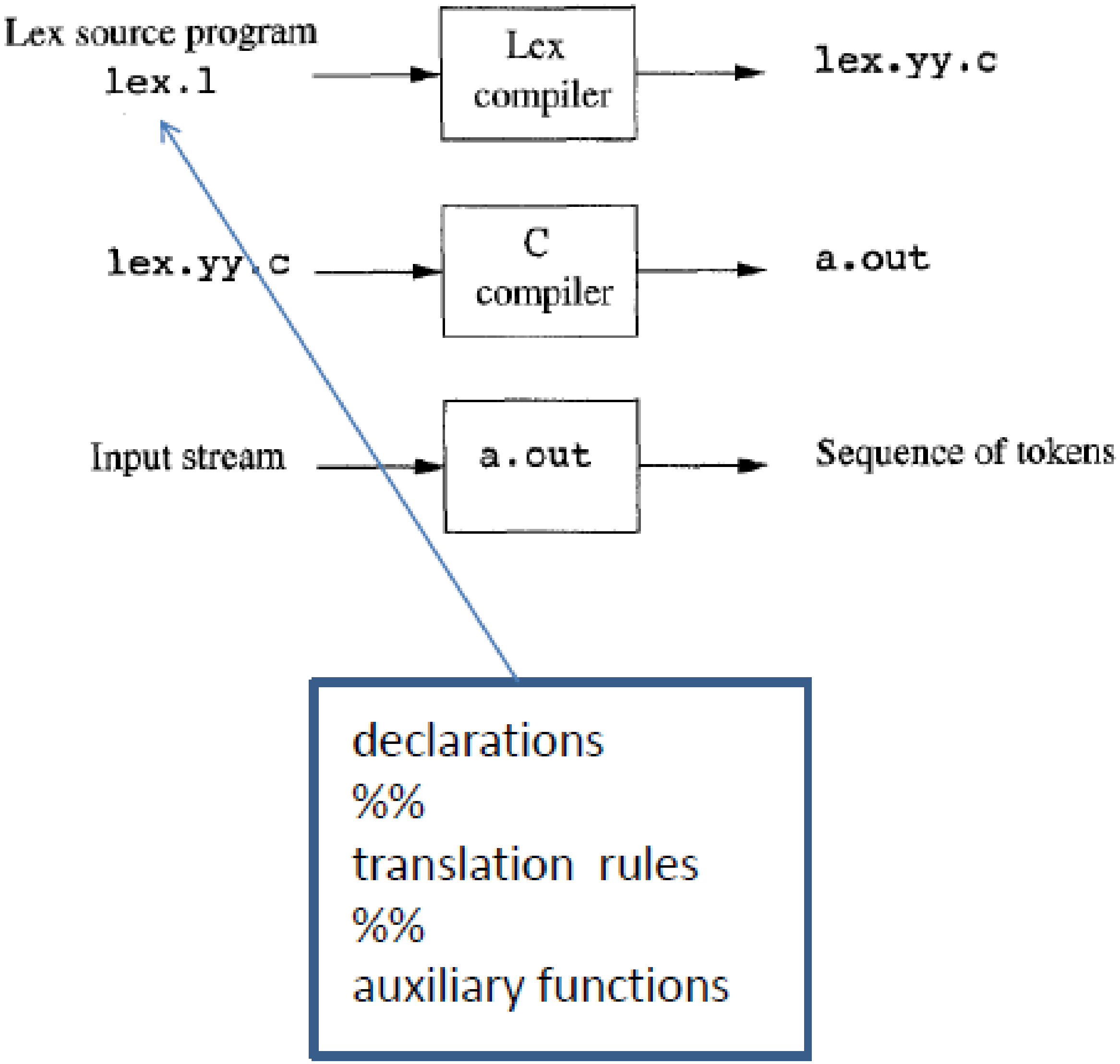
The Lexical-Analyzer Generator

Lex

The Lexical-Analyzer Generator Lex

- **Lex:**
  - It is a tool, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.
- **Lex:**
  - **Lex reads a specification file containing regular expressions and generates a C routine that performs lexical analysis**. Matches sequences that identify tokens.
- **Lex:**
  - Lex generates a lexical analyzer, which is called through a call to a function yylex.

The Lexical-Analyzer Generator Lex



The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler.

Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy.c, that simulates this transition diagram.

The Lexical-Analyzer Generator Lex

- A **Lex program has the following form:**

declarations (C includes and definitions)

%%

translation rules

%%

auxiliary functions(user subroutines)

The Lexical-Analyzer Generator Lex

- **The declarations section** includes
  1. declarations of variables,
  2. manifest constants (identifiers declared to stand for a constant, e.g., the name of a token),
  3. regular definitions, and
  4. a pair of special brackets, **%{ and %}** . Anything within these brackets is copied directly to the file lex.yy.c , and is not treated as a regular definition.

- **Regular Definitions example**

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

- Regular definitions that are used in later definitions or in the patterns of the translation rules are surrounded by curly braces.

The Lexical-Analyzer Generator Lex

- The **translation rules** 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.

The Lexical-Analyzer Generator Lex

- **Translation rules : line oriented:**

  <pattern> <whitespace> <action>

  - **<pattern>** : starts at beginning of line,

    continues upto first unescaped whitespace
  - **<action>** : a single C statement

    (multiple statements: enclose in braces {}).
  - **unmatched input characters** : copied to stdout.

The Lexical-Analyzer Generator Lex

Anything between these 2 marks is copied as it is in lex.yy.c

braces means the pattern is defined somewhere

pattern

Actions

```
%{
    /* 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 Lexical-Analyzer Generator Lex

- The **final section** is the **user subroutines section**, which can consist of any legal C code. Lex copies it to the C file after the end of the lex generated code. We have included a main() program.

```
%%
main( )
{
  yylex();
}
```

- **The lexer produced by lex is a C routine called yylex(), so we call it.** Unless the actions contain explicit return statements, yylex() won't return until it has processed the entire input.

The Lexical-Analyzer Generator Lex

- The lexical analyzer created by **Lex** behaves in concert with the parser as follows.
  - 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_i. It then executes the associated action A_i.
  - Typically, A_i will return to the parser, but if it does not (e.g., because P_i 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.

The Lexical-Analyzer Generator Lex

Table 1: Special Characters

Pattern	Matches
`.`	any character except newline
`\.`	literal `.`
`\n`	newline
`\t`	tab
`^`	beginning of line
`$`	end of line

The Lexical-Analyzer Generator Lex

Table 2: Operators

Pattern	Matches
?	zero or one copy of the preceding expression
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
a|b	a or b (alternating)
(ab)+	one or more copies of ab (grouping)
abc	abc
abc*	ab abc abcc abccc ...
"abc*"	literal abc*
abc+	abc abcc abccc abcccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc

The Lexical-Analyzer Generator Lex

Table 3: Character Class

<table>
<tr><th>Pattern</th><th>Matches</th></tr>
<tr><td>[abc]</td><td>one of: a b c</td></tr>
<tr><td>[a-z]</td><td>any letter a through z</td></tr>
<tr><td>[a\-z]</td><td>one of: a - z</td></tr>
<tr><td>[-az]</td><td>one of: - a z</td></tr>
<tr><td>[A-Za-z0-9]+</td><td>one or more alphanumeric characters</td></tr>
<tr><td>[\t\n]+</td><td>whitespace</td></tr>
<tr><td>[^ab]</td><td>anything except: a b</td></tr>
<tr><td>[a^b]</td><td>one of: a ^ b</td></tr>
<tr><td>[a|b]</td><td>one of: a | b</td></tr>
</table>

The Lexical-Analyzer Generator Lex

- operators:
  - ***** : **match zero or more times**, eg: ab*c → ac, abc, abbc...
  - **+** : **match one or more times**, eg: ab+c → abc, abbc...
  - ? : **zero or one occurrence**, eg: ab?c → abc, ac
  - **( )** : **grouping**, eg: (ab)+ → ab, abab...
  - | : **alternatives**, eg ab|cd → ab, cd
  - ***{n,m}*** : ***repitition***, *eg a{1,3} → a, aa, aaa*
  - ***{defn}*** : **substitute *defn*** (from first section).

The Lexical-Analyzer Generator Lex

- operators (cont):
  - brackets **[ ]** enclose a sequence of characters, termed a *character class*.
    1) [ ] matches any character in the sequence
    2) a '**-**' in a character class denotes an inclusive range,
       e.g: [0-9] matches any digit.
    3) a **^** at the beginning denotes *negation*:
       e.g: [^0-9] matches any character that is not a digit.

The Lexical-Analyzer Generator Lex

- **Actions**
  - ; → Null action.
  - ECHO; → printf("%s", yytext);
  - {...} → Multi-statement action.
  - return yytext; → send contents of yytext to the parser.
- **yytext** : C-String of matched characters (Make a copy if neccessary!)
- **yyleng** : Length of the matched characters.

The Lexical-Analyzer Generator Lex

- Example:

  Figure 3.23 is a Lex program that recognizes the tokens of Fig. 3.12 and returns the token found.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	-	-
if	if	-
then	then	-
else	else	-
Any *id*	id	Pointer to table entry
Any *number*	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	

Figure 3.12: Tokens, their patterns, and attribute values

declarations

```
%{
    /* 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}+)?
```

```
%%
```

translation rules

```
{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);}
```

```
%%
```

auxiliary functions

```
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 */
}
```

Figure 3.23: **Lex program for the tokens of Fig. 3.12**

The Lexical-Analyzer Generator Lex

Anything between these 2 marks is copied as it is in lex.yy.c

braces means the pattern is defined somewhere

pattern

Actions

```
%{
    /* 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 Lexical-Analyzer Generator Lex

- **The action taken when id is matched is threefold:**
  1. Function **installID( )** is called to place the lexeme found into 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.
     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().**

The Lexical-Analyzer Generator Lex

- **Choosing between different possible matches:**

  When more than one pattern can match the input, lex chooses as follows:

  1. The *longest match is preferred*.
  2. Among rules that match the same number of characters, the rule that occurs earliest in the list is preferred.

Transition Rules (cont'd)

- Four special options for actions:
  |, ECHO;, BEGIN, and REJECT;

- | indicates that the action for this rule is from the action for the next rule
  - ```
    [ \t\n]         ;
    " "             |

     "\t"           |

     "\n"           ;
    ```

- The unmatched token is using a default action that ECHO from the input to the output
    ```

Lex Predefined Variables

- yytext -- a string containing the lexeme
- yyleng -- the length of the lexeme
- yyin -- the input stream pointer
  - the default input of default main() is **stdin**
- yyout -- the output stream pointer
  - the default output of default main() is **stdout**.
- **`%./a.out < inputfile > outfile`**
- E.g.
  - `[a-z]+        printf("%s", yytext);`
  - `[a-z]+        ECHO;`
  - `[a-zA-Z]+     {words++; chars += yyleng;}`

Lex Library Routines

- yylex()
  - The default main() contains a call of yylex()
- yymore()
  - return the next token
- yyless(n)
  - retain the first n characters in yytext
- yywarp()
  - is called by lex, when input is exhausted.
  - The default yywarp() always returns 1

The Lexical-Analyzer Generator Lex

- Write a lex specification to echo all strings of capital letters, followed by a space tab (\t)or newline (\n) dot (\.) or comma (\,) to stdout, and all other characters will be ignored.

The Lexical-Analyzer Generator Lex

- Write a lex specification to echo all strings of capital letters, followed by a space or tab (\t)or newline (\n) dot (\.) or comma (\,) to stdout, and all other characters will be ignored.

```
/* echo-upcase-words.l */
%option main
%%
[A-Z]+[  \t\n\.\,]       printf("%s",yytext);
.   ;   /* no action specified */
```

The Lexical-Analyzer Generator Lex

Conflict Resolution in Lex

- We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, **when several prefixes of the input match one or more patterns:**
  1. **Always prefer a longer prefix to a shorter prefix.**
  2. **If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.**

The Lexical-Analyzer Generator Lex

➢ Example:

Figure 3.23 is a Lex program that recognizes the tokens of Fig. 3.12 and returns the token found.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	-	-
if	**if**	-
then	**then**	-
else	**else**	-
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	

Figure 3.12: Tokens, their patterns, and attribute values

declarations

```
%{
    /* 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}+)?
```

```
%%
```

translation rules

```
{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);}
```

```
%%
```

auxiliary functions

```
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 */
}
```

Figure 3.23: **Lex program for the tokens of Fig. 3.12**

The Lexical-Analyzer Generator Lex

Conflict Resolution in Lex

1. Always prefer a longer prefix to a shorter prefix.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

Example 3.12:

- The first rule tells us to continue reading letters and digits to find the longest prefix of these characters to group as an identifier. It also tells us to treat <= as a single lexeme, rather than selecting < as one lexeme and = as the next lexeme.

- The second rule makes keywords reserved, if **we list the keywords before id in the program**. For instance, if ***then*** is determined to be the longest prefix of the input that matches any pattern, and the pattern ***then*** precedes { id } , as it does in Fig. 3.23, then the token THEN is returned, rather than ID.

The Lexical-Analyzer Generator Lex

- Consider the following lexical specification:

```
%%
a*b      { printf(" 1 "); }
ca       { printf(" 2 "); }
a*ca*    { printf(" 3 "); }
```

- Given the following input string

abcaa cacaaabbaaabcaaca

- What does the lexer print? Clearly indicate all the tokens.

The Lexical-Analyzer Generator Lex

- Consider the following lexical specification:

```
%%
aa      { printf(" 1 "); }
a       { printf(" 2 "); }
ab      { printf(" 3 "); }
```

- Given the following input string: **aab**
- What does the lexer print? Clearly indicate all the tokens.

The Lexical-Analyzer Generator Lex

- Consider the following lex script:

```
%%
a*b        { printf(" 1 "); }
(a|b)*b    { printf(" 2 "); }
c*         { printf(" 3 "); }
```

- Give an example of an input to this scanner that will produce **132** as an output, or explain why one does not exist.

- Explain how lex would tokenize the following input string and the output produced.

  (i) cbbbbac (ii) cbabc

The Lexical-Analyzer Generator Lex

- Consider the following lex script:

```
%%
(01|10)              { printf(" course "); }
0(01)*1              { printf(" compiler "); }
(1010*1|0101*0)      { printf(" design "); }
```

- Give an input to this scanner such that the output string is:

$$(\textbf{compiler}^{11}\ \textbf{design}^{2})^{4}\ \textbf{course}^{3}$$

- Where, A^i denotes A repeated i times. (And, of course, the parentheses are not part of the output.) You may use similar shorthand notation in your answer.

The Lexical-Analyzer Generator Lex

The Lookahead Operator

- **Lex reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input.**

- **Sometimes, we need a certain pattern to be matched to the input only when it is followed by a certain other characters. (e.g. ab/cd)**
  - If so, we may use the slash (/) in a pattern to indicate the end of the part of the pattern that matches the lexeme.
  - What follows / is additional pattern that must be matched before we can decide that the token in question was seen, but what matches this second pattern is not part of the lexeme.

The Lexical-Analyzer Generator Lex

The Lookahead Operator

- The regular expression

  ab/cd

  matches the string ab, but only if followed by cd.

The Lexical-Analyzer Generator Lex

The Lookahead Operator

- **Example 3.13 : In Fortran** and some other languages, **keywords are not reserved**. That situation creates problems, such as a statement

  IF(I,J) = 3

  where IF is the name of an array, not a keyword. This statement contrasts with statements of the form

  IF(condition) THEN ...

  where IF is a keyword. we can be sure that the keyword IF is always followed by a left parenthesis, some text — the condition —- that may contain parentheses, a right parenthesis and a letter. Thus, **we could write a Lex rule for the keyword IF like:**

  This rule says that the pattern the lexeme matches is just the two letters **IF.** The slash says that additional pattern follows but does not match the lexeme.

The Lexical-Analyzer Generator Lex

The Lookahead Operator

- **Example 3.13 : In Fortran** and some other languages, **keywords are not reserved**. That situation creates problems, such as a statement

  IF(I,J) = 3

  where IF is the name of an array, not a keyword. This statement contrasts with statements of the form

  IF(condition) THEN ...

  where IF is a keyword. we can be sure that the keyword IF is always followed by a left parenthesis, some text — the condition —- that may contain parentheses, a right parenthesis and a letter. Thus, **we could write a Lex rule for the keyword IF like**:

  IF / \(.* \) {letter}

  This rule says that the pattern the lexeme matches is just the two letters **IF.** The slash says that additional pattern follows but does not match the lexeme.

Design of a Lexical-Analyzer Generator

Design of a Lexical-Analyzer Generator

➢ The Structure of the Generated Analyzer

- ➢ Figure 3.49 Overviews the **architecture of a lexical analyzer generated by Lex.**
- ➢ The program that serves as the **lexical analyzer** includes **a fixed program that simulates an automaton.**
- ➢ The rest of the lexical analyzer consists of components that are created from the Lex program by Lex itself.

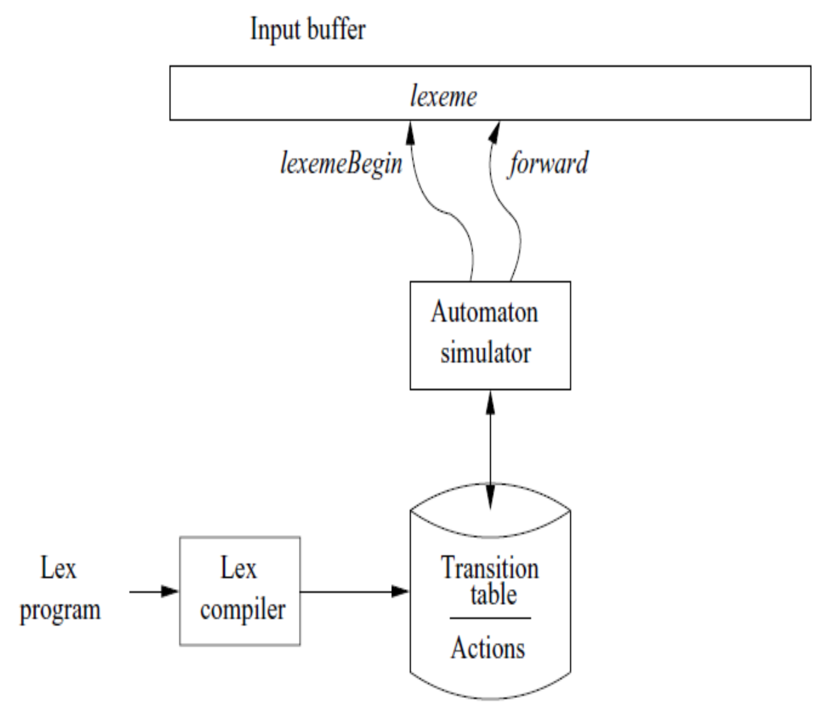


Figure 3.49: A Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator

Design of a Lexical-Analyzer Generator

- **To construct the automaton**,

  we begin by taking each regular-expression pattern in the Lex program and converting it, using Algorithm to convert a RE to an NFA.

- **We need a single automaton that will recognize lexemes matching any of the patterns in the program**,

  so we combine all the NFA's into one by introducing a new start state with **ε**-transitions to each of the start states of the NFA's N_i for pattern p_i

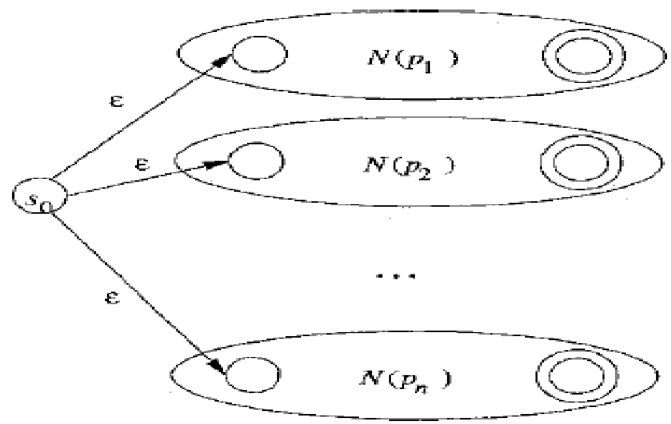


Figure 3.50: An NFA constructed from a **Lex** program

Design of a Lexical-Analyzer Generator

- **Example 3.26:** We shall illustrate the ideas of this section with the following simple, abstract example:

a	**{ action *A1 for pattern p1* }**
abb	**{ action *A2 for pattern p2* }**
a*b+	**{ action *A3 for pattern p3* }**

- Input string: **abb**
- Input string: **aabbb...**

Design of a Lexical-Analyzer Generator

a **{ action *A1 for pattern p1* }**

abb **{ action *A2 for pattern p2* }**

a*b+ **{ action *A3 for pattern p3* }**

- In particular, string ***abb*** matches both the second and third patterns, but we shall consider it a lexeme for pattern *p2, since that pattern* is listed first in the above Lex program.

- Input strings such as ***aabbb...*** have many prefixes that match the third pattern. The Lex rule is to take the longest, so we continue reading b's, until another *a is met, whereupon we report* the lexeme to be the initial a's followed by as many b's as there are.

Design of a Lexical-Analyzer Generator

a	{ action *A1 for pattern p1* }
abb	{ action *A2 for pattern p2* }
a*b+	{ action *A3 for pattern p3* }

- Convert each regular expression to NFA

Figure 3.51: NFA's for **a**, **abb**, and $\mathbf{a}^*\mathbf{b}^+$

- Combine all NFAs as

Figure 3.52: Combined NFA

Design of a Lexical-Analyzer Generator

a	{ action *A1 for pattern p1* }
abb	{ action *A2 for pattern p2* }
a*b+	{ action *A3 for pattern p3* }

- Convert each regular expression to NFA

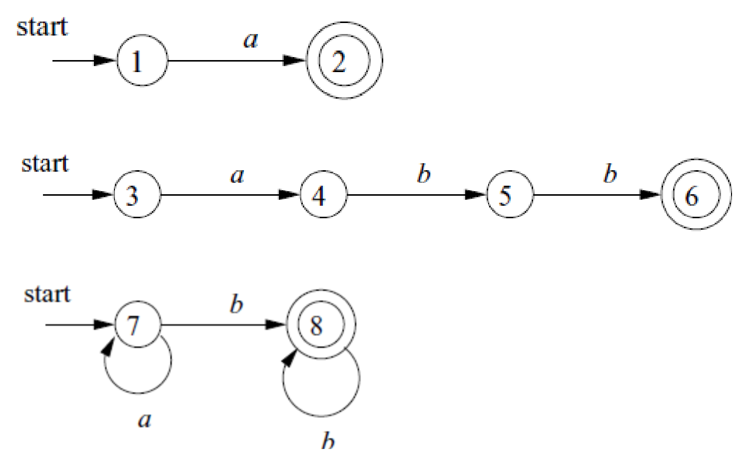


Figure 3.51: NFA's for **a**, **abb**, and $\mathbf{a}^*\mathbf{b}^+$

- Combine all NFAs as

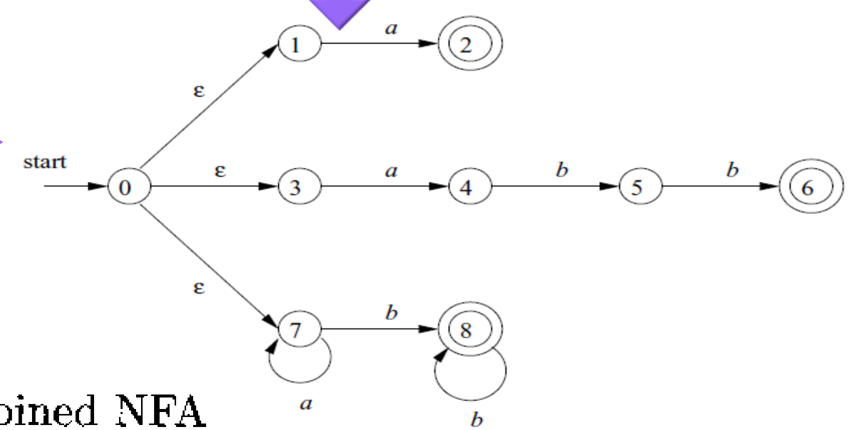


Figure 3.52: Combined NFA

Design of a Lexical-Analyzer Generator

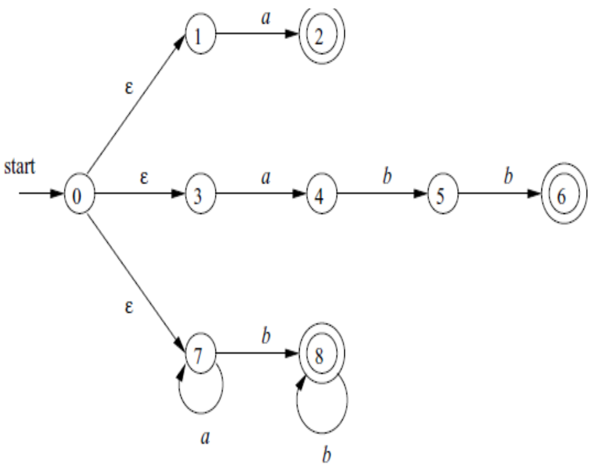


Figure 3.52 Combined NFA

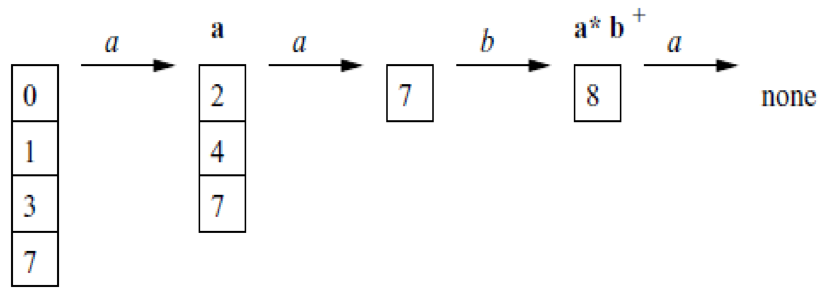


Figure 3.53 Sequence of sets of states entered when processing input *aaba*

- **Example 3.27:** Suppose the **input begins aaba**.
  - Figure 3.53 shows the sets of states of the NFA of Fig. 3.52 that **we enter, starting with ε -closure of the initial state 0, which is {0,1,3,7}**, and proceeding from there.
  - After reading the fourth input symbol, we are in an empty set of states, since in Fig. 3.52, there are no transitions out of state 8 on input *a*.
  - Thus, we need to back up, looking for a set of states that includes an accepting state.

Design of a Lexical-Analyzer Generator

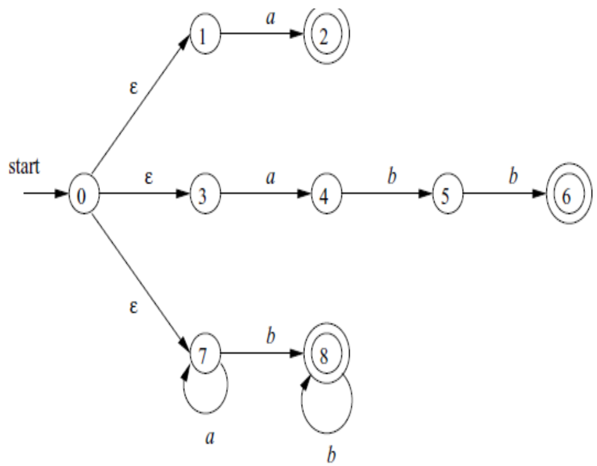


Figure 3 52 Combined NFA

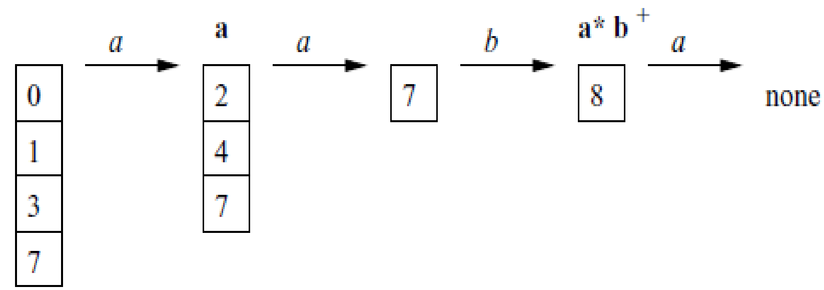


Figure 3 53 Sequence of sets of states entered when processing input *aaba*

- Notice that, as indicated in Fig. 3.53, after reading **a** we are in a set that includes state 2 and therefore indicates that the pattern **a** has been matched.
- However, after reading **aab,** we are in state 8, which indicates that **a*b+** has been matched; prefix aab is the longest prefix that gets us to an accepting state.
- We therefore select aab as the lexeme, and execute action A3, which should include a return to the parser indicating that the token whose pattern is p_3 = a*b+ has been found.

Design of a Lexical-Analyzer Generator

- Using the following patterns explain the design of a lexical analyzer. Show the working of your lexical analyzer that simulates an NFA on the input string: **abaab**

ab	{// some action ....}
aab	{// some action ....}
aba	{// some action ....}

Design of a Lexical-Analyzer Generator

DFA's for Lexical Analyzers

- Figure 3.54 shows a transition diagram based on the DFA that is constructed by the subset construction from the NFA in Fig. 3.52.

- **The accepting states are labelled by the pattern that is identified by that state**. For instance, the state {6,8} has two accepting states, corresponding to patterns **abb** and **a*b+ .** Since the former is listed first, that is the pattern associated with state {6,8}.

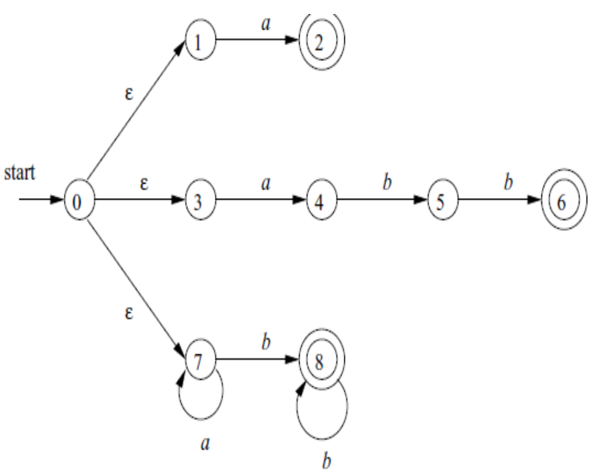


Figure 3.52 Combined NFA

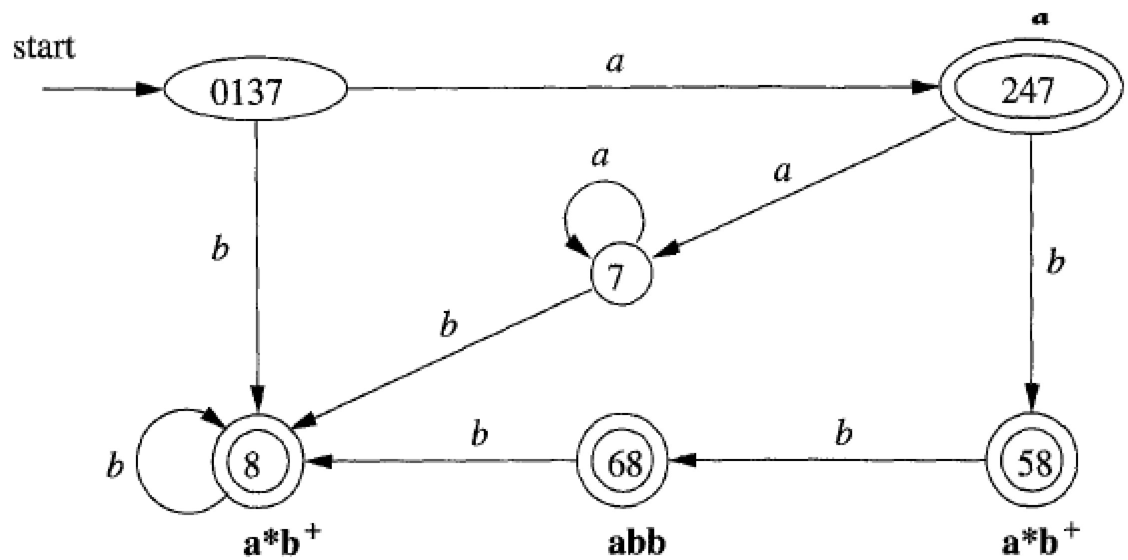


Figure 3.54: Transition graph for DFA handling the patterns a, abb, and a*b⁺

Design of a Lexical-Analyzer Generator

Implementing the Lookahead Operator

- **When converting the pattern r_1/r_2 to an NFA**, **we treat the / as if it were ε, so we do not actually look for a / on the input.**
  However, if the NFA recognizes a prefix **xy** of the input buffer as matching this regular expression, the **end of the lexeme is not where the NFA entered its accepting state**. Rather the end occurs when the NFA enters a state s such that

  1. s has an ε-transition on the (imaginary) /,
  2. There is a path from the start state of the NFA to state s that spells out **x**
  3. There is a path from state s to the accepting state that spells out **y**.
  4. **x** is as long as possible for any **xy** satisfying conditions 1-3.

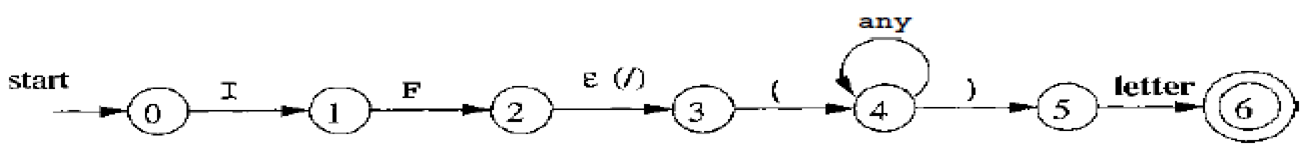


Figure 3.55: NFA recognizing the keyword IF

Design of a Lexical-Analyzer Generator

Implementing the Lookahead Operator

- An **NFA for the pattern for the Fortran IF (branching statement) with lookahead** is shown in Fig. 3.55.
  - Notice that the **ε**-transition from state 2 to state 3 represents the lookahead operator.
  - State 6 indicates the presence of the keyword IF.
  - However, we find the lexeme IF by scanning backwards to the last occurrence of state 2, whenever state 6 is entered.

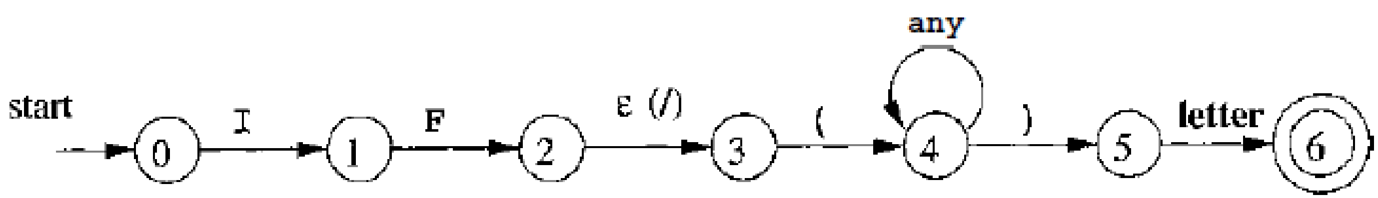


Figure 3.55: NFA recognizing the keyword IF

Lexical-Analyzer - Regex

- <u>Macros, Quantifiers, characters, Character Classes, Anchors and Boundaries.</u>

- Mastering Quantifiers
  - **Greedy:** As Many As Possible (longest match)
  - **Docile:** Give Back When Needed
  - **Lazy:** As Few As Possible (shortest match)
  - **Helpful:** Expand When Needed
  - **Possessive:** Don't Give Up Characters

- The *Longest Match* and *Shortest Match* Traps
  - What does "longest match" mean?
  - What does "shortest match" mean?

References

- **Compilers–Principles, Techniques and Tools**, Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffery D. Ullman, 2nd Edition
- https://docs.python.org/3/reference/lexical_analysis.html
- https://www.epaperpress.com/lexandyacc/prl.html

Thank You

The Lexical-Analyzer Generator Lex

- Figure: **Lex program structure**

```
%{

/* C includes */

}%


/* Definitions */


%%
/* Rules */


%%
/* user subroutines */
```

The Lexical-Analyzer Generator Lex

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

Table 3: Lex Predefined Variables