Compiler Design

Lexical Analysis

Dr. Mousumi Dutt

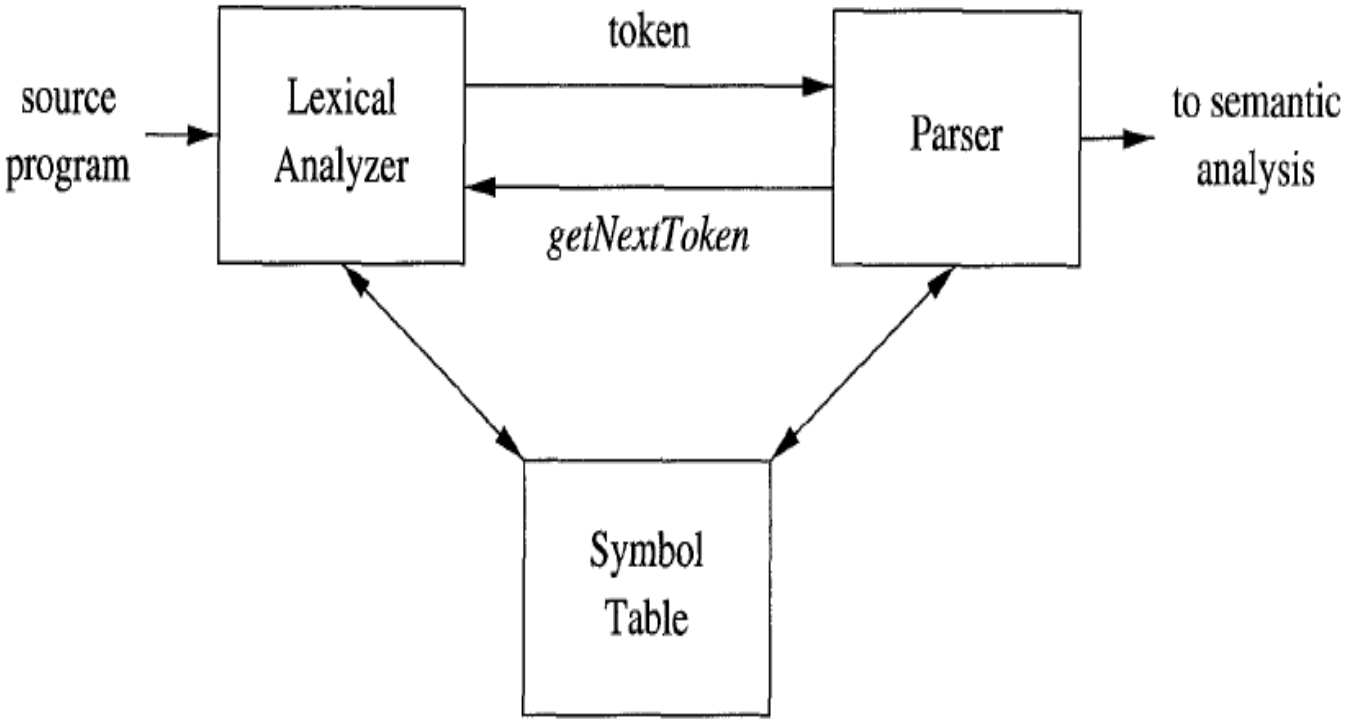
Lexical Analysis

- How to construct lexical analyser
- A lexical-analyzer generator called Lex (or Flex in a more recent embodiment)
- Regular expression -> lexeme
- How this notation can be transformed, first into nondeterministic automata and then into deterministic automata
- The latter two notations can be used as input to a "driver," that is, code which simulates these automata and uses them as a guide to determining the next token
- This driver and the specification of the automaton form the nucleus of the lexical analyzer.

Lexical Analyzer - TASKS

- First phase of compiler
- To read the input characters of the source program
- group them into lexemes
- 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
- Interacts with the symbol table
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table
- information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser

Interaction between Lexical Analyzer and Parser



The call, suggested by the *getNextToken* command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

Other tasks by lexical analyzer:

- stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- correlating error messages generated by the compiler with the source program
- keep track of the number of newline characters seen, so it can associate a line number with each error message

Lexical Analyzer - TASKS

- 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-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

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

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

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

Reason of Separating Lexical Analysis and Parsing

- 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. Better for designing new language
- 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
- Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer

Tokens, Patterns, and Lexemes

- A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.
- A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.
- 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.

Tokens, Patterns, and Lexemes

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except "

Tokens Codes

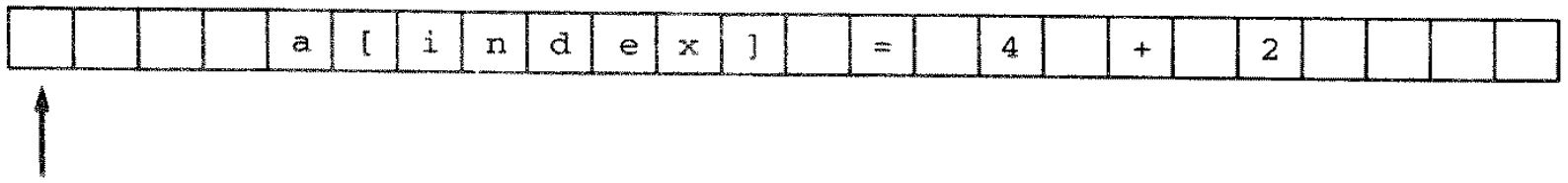
```
typedef struct
   { TokenType tokenval;
     char * stringval;
     int numval;
   } TokenRecord;
```

or possibly as a union

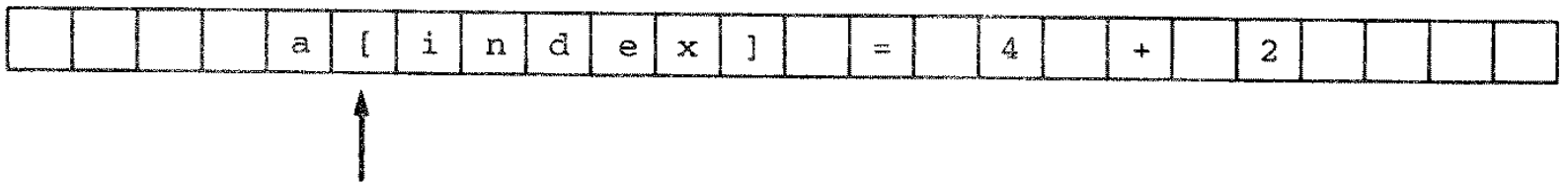
```
typedef struct
   { TokenType tokenval;
     union
     { char * stringval;
       int numval;
     } attribute;
   } TokenRecord;
```

Tokens

a[index] = 4 + 2



A call to **getToken** will now need to skip the next four blanks, recognize the string "a" consisting of the single character *a* as the next token, and return the token value **ID** as the next token, leaving the input buffer as follows:



E = M * C ** 2

<**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>
<**exp_op**>
<**number**, integer value 2>

Attributes

- 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 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program
- In many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token
- **the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse**

Attributes

- We shall 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 (in case an error message about that identifier must be issued) - 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**

Lexical Errors

1. Delete one character from the remaining input.

2. Insert a missing character into the remaining input.

3. Replace a character by another character.

4. Transpose two adjacent characters.

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

Input Buffering

How source program reading can be speeded up

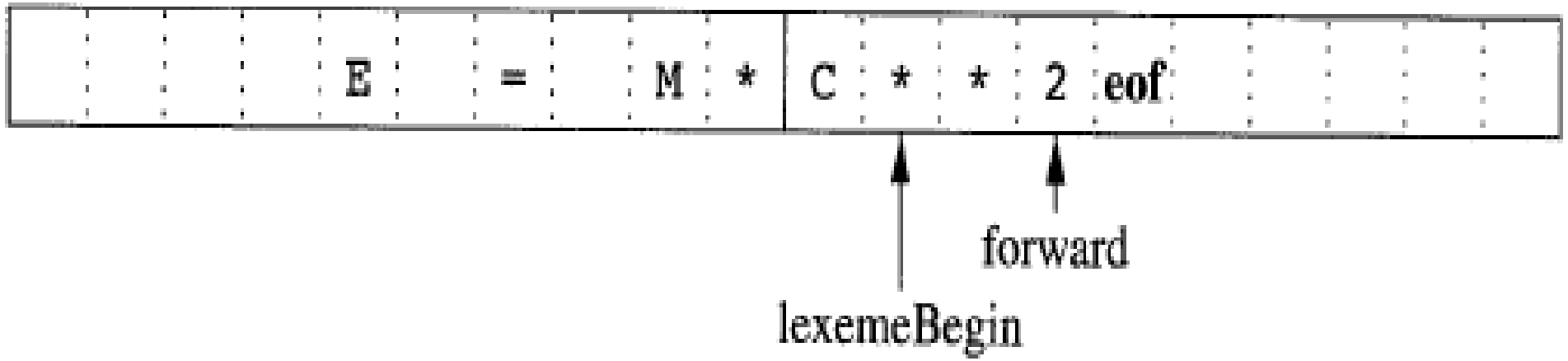
- **Difficult task:** we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme

- In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=

- Two buffer scheme to handle large lexeme safely
- Inclusion of sentinels

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



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

Two pointers to the input are maintained:

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

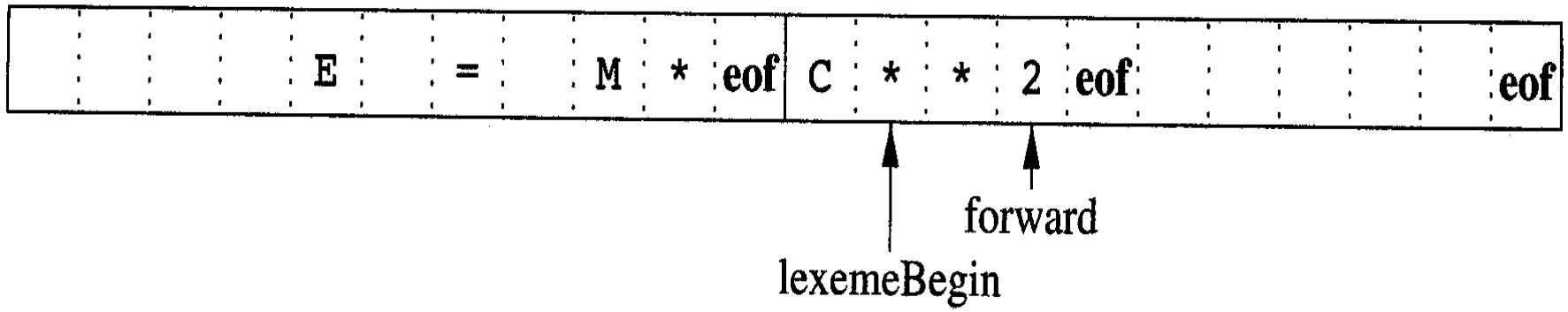
2. Pointer forward scans ahead until a pattern match is found

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.
- Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer
- As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

Sentinels

- For each character read, two tests are there:
- one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch)
- 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.**



Sentinels Codes

```
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
}
```

Specification of Tokens

- Regular expression to express lexeme patterns
- Specify only those that are needed
- 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.
- 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 Isl, 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

- A language is any countable set of strings over some fixed alphabet.
- Abstract languages like **Ø,** the empty set, or {**ε**}, the set containing only the empty string, are languages under this definition.
- What about C Programming Language r English Language?
- If x and y are strings, then the concatenation of x and y, denoted xy, is the string formed by appending y to x.
- Exponentiation:
- Define so to be **ε**, and for all $i > 0$, define s^i to be $s^{i-1}s$.
- Since **εs=s**, it follows that $s^1 = s$. Then $s^2 = ss$, $s^3 = sss$, and so on.

Specification of Tokens

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**.

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**.

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**.

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.

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 Languages

Operation	Definition and Notation
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Example

Let L be the set of letters $\{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$ and let D be the set of digits $\{\texttt{0}, \texttt{1}, \ldots \texttt{9}\}$. We may think of L and D in two, essentially equivalent, ways. One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages L and D, using the operators of Fig. 3.6:

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 Expression

letter (letter | digit) * Regular Expression for Identifiers

The regular expressions are built recursively out of smaller regular expressions, using the rules described below. 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 ***C*** 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 Σ, then $\mathbf{a}$ is a regular expression, and $L(\mathbf{a}) = \{a\}$, that is, the language with one string, of length one, with a in its one position. Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.[1]

Regular Expression

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 $L(r) \cup L(s)$.
2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.
3. $(r)^*$ is a regular expression denoting $\left(L(r)\right)^*$.
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.

Regular Expression

As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

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 $(\mathbf{a})|((\mathbf{b})^*(\mathbf{c}))$ by $\mathbf{a}|\mathbf{b}^*\mathbf{c}$. Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.

Regular Expression

Let $\Sigma = \{a, b\}$.

1. The regular expression **a**|**b** denotes the language $\{a, b\}$.

2. (**a**|**b**)(**a**|**b**) denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is **aa**|**ab**|**ba**|**bb**.

3. **a*** denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \ldots\}$.

4. (**a**|**b**)* denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. Another regular expression for the same language is (**a*****b***)*.

5. **a**|**a*****b** denotes the language $\{a, b, ab, aab, aaab, \ldots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

Regular Expression

A language that can be defined by a regular expression is called a *regular set*. If two regular expressions r and s denote the same regular set, we say they are *equivalent* and write $r = s$. For instance, $(\mathbf{a}|\mathbf{b}) = (\mathbf{b}|\mathbf{a})$. There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent. Figure 3.7 shows some of the algebraic laws that hold for arbitrary regular expressions r, s, and t.

Regular Expression

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 Definition

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

$$\begin{aligned} d_1 &\rightarrow r_1 \\ d_2 &\rightarrow r_2 \\ &\cdots \\ d_n &\rightarrow r_n \end{aligned}$$

where:

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

Regular Definition

By restricting r_i to Σ and the previously defined d's, we avoid recursive definitions, and we can construct a regular expression over Σ alone, for each r_i. We do so by first replacing uses of d_1 in r_2 (which cannot use any of the d's except for d_1), then replacing uses of d_1 and d_2 in r_3 by r_1 and (the substituted) r_2, and so on. Finally, in r_n we replace each d_i, for $i = 1,2,\ldots,n-1$, by the substituted version of r_i, each of which has only symbols of Σ.

letter → A | B | · · · | Z | a | b | · · · | z

digit → 0 | 1 | · · · | 9

id → letter (letter | digit)*

digit → 0 | 1 | · · · | 9

digits → digit digit*

optional_fraction → . digits | ϵ

optional_exponent → (E (+ | - | ϵ) digits) | ϵ

num → digits optional_fraction optional_exponent

Extension of Regular Expression

1. *One or more instances.* The unary, postfix operator $^+$ represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then $(r)^+$ denotes the language $\big(L(r)\big)^+$. The operator $^+$ has the same precedence and associativity as the operator $*$. Two useful algebraic laws, $r^* = r^+|\epsilon$ and $r^+ = rr^* = r^*r$ relate the Kleene closure and positive closure.

2. *Zero or one instance.* The unary postfix operator ? means "zero or one occurrence." That is, $r?$ is equivalent to $r|\epsilon$, or put another way, $L(r?) = L(r) \cup \{\epsilon\}$. The ? operator has the same precedence and associativity as $*$ and $+$.

3. *Character classes.* A regular expression $a_1|a_2|\cdots|a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2\cdots a_n]$. More importantly, when $a_1, a_2, \ldots, a_n$ form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a_1-a_n, that is, just the first and last separated by a hyphen. Thus, [**abc**] is shorthand for **a**|**b**|**c**, and [**a-z**] is shorthand for **a**|**b**|$\cdots$|**z**.

Extension of Regular Expression

$$
\begin{aligned}
\textbf{digit} &\rightarrow 0 \mid 1 \mid \cdots \mid 9 \\
\textbf{digits} &\rightarrow \textbf{digit}^{+} \\
\textbf{optional\_fraction} &\rightarrow (\ . \ \textbf{digits} \ )? \\
\textbf{optional\_exponent} &\rightarrow (\ \mathbf{E} \ (\ + \mid - \ )? \ \textbf{digits} \ )? \\
\textbf{num} &\rightarrow \textbf{digits} \ \ \textbf{optional\_fraction} \ \ \textbf{optional\_exponent}
\end{aligned}
$$

Nonregular Sets

Some languages cannot be described by regular expression

Cannot describe

- Balanced or nested constructs
- Repeating strings

Examples

Consider the simple alphabet consisting of just three alphabetic characters: $\Sigma = \{a, b, c\}$. Consider the set of all strings over this alphabet that contain exactly one b. This set is generated by the regular expression

```
(a|c)*b(a|c)*
```

Note that, even though **b** appears in the center of the regular expression, the letter b need not be in the center of the string being matched. Indeed, the repetition of a or c before and after the b may occur different numbers of times. Thus, all the following strings are matched by the above regular expression: *b*, *abc*, *abaca*, *baaaac*, *ccbaca*, *cccccb*. §

Examples

With the same alphabet as before, consider the set of all strings that contain at most one *b*. A regular expression for this set can be obtained by using the solution to the previous example as one alternative (matching those strings with exactly one *b*) and the regular expression `(a|c)*` as the other alternative (matching no *b*'s at all). Thus, we have the following solution:

```
(a|c)*|(a|c)*b(a|c)*
```

An alternative solution would allow either *b* or the empty string to appear between the two repetitions of *a* or *c*:

```
(a|c)*(b|ε)(a|c)*
```

This example brings up an important point about regular expressions: the same language may be generated by many different regular expressions. Usually, we try to find as simple a regular expression as possible to describe a set of strings, though we will never attempt to prove that we have in fact found the "simplest"—for example, the shortest. There are two reasons for this. First, it rarely comes up in practical situations, where there is usually one standard "simplest" solution. Second, when we study methods for recognizing regular expressions, the algorithms there will be able to simplify the recognition process without bothering to simplify the regular expression first. §

Examples

Consider the set of strings S over the alphabet $\Sigma = \{a, b\}$ consisting of a single b surrounded by the same number of a's:

$$S = \{b, aba, aabaa, aaabaaa, \ldots\} = \{a^n b a^n | n \neq 0\}$$

This set cannot be described by a regular expression. The reason is that the only repetition operation we have is the closure operation *, which allows any number of repetitions. So if we write the expression **a*ba*** (about as close as we can get to a regular expression for S), then there is no guarantee that the number of a's before and after the b will be the same. We express this by saying that "regular expressions can't count." To give a mathematical proof of this fact, however, would require the use of a famous theorem about regular expressions called the **pumping lemma**, which is studied in automata theory, but which we will not mention further here.

Clearly, not all sets of strings that we can describe in simple terms can be generated by regular expressions. A set of strings that *is* the language for a regular expression is, therefore, distinguished from other sets by calling it a **regular set**. Occasionally, nonregular sets appear as strings in programming languages that need to be recognized by a scanner. These are usually dealt with when they arise, and we will return to this matter again briefly in the section on practical scanner considerations. §

Examples

Consider the strings over the alphabet $\Sigma = \{a, b, c\}$ that contain no two consecutive b's. Thus, between any two b's there must be at least one a or c. We build up a regular expression for this set in several stages. First, we can force an a or c to come *after* every b by writing

```
(b(a|c))*
```

We can combine this with the expression `(a|c)*`, which matches strings that have no b's at all, and write

```
((a|c)*|(b(a|c))*)*
```

or, noting that $(r^*|s^*)^*$ matches the same strings as $(r|s)^*$

```
((a|c)|(b(a|c)))*
```

or

```
(a|c|ba|bc)*
```

(Warning! This is not yet the correct answer.)

The language generated by this regular expression does, indeed, have the property we seek, namely, that there are no two consecutive b's (but isn't quite correct yet). Occasionally, we should prove such assertions, so we sketch a proof that all strings in $L($`(a|c|ba|bc)*`$)$ contain no two consecutive b's. The proof is by induction on the length of the string (i.e., the number of characters in the string). Clearly, it is true for all strings of length 0, 1, or 2: these strings are precisely the strings ε, a, c, aa, ac, ca, cc, ba, bc. Now, assume it is true for all strings in the language of length $i < n$, and let s be a string in the language of length $n > 2$. Then, s contains more than one of the non-ε strings just listed, so $s = s_1 s_2$, where s_1 and s_2 are also in the language and are not ε. Hence, by the induction assumption, both s_1 and s_2 have no two consecutive b's. Thus, the only way s itself could have two consecutive b's would be for s_1 to end with a b and for s_2 to begin with a b. But this is impossible, since no string in the language can end with a b.

Examples

This last fact that we used in the proof sketch—that no string generated by the preceding regular expression can end with a *b*—also shows why our solution is not yet quite correct: it does not generate the strings *b*, *ab*, and *cb*, which contain no two consecutive *b*'s. We fix this by adding an optional trailing *b*, as follows:

```
(a|c|ba|bc)*(b|ε)
```

Note that the mirror image of this regular expression also generates the given language

```
(b|ε)(a|c|ab|cb)*
```

We could also generate this same language by writing

```
(notb|b notb)*(b|ε)
```

where **notb** = a|c. This is an example of the use of a name for a subexpression. This solution is in fact preferable in cases where the alphabet is large, since the definition of **notb** can be adjusted to include all characters except *b*, without complicating the original expression. §

Examples

This example is one where we are given the regular expression and are asked to determine a concise English description of the language it generates. Consider the alphabet $\Sigma = \{a, b, c\}$ and the regular expression

```
((b|c)*a(b|c)*a)*(b|c)*
```

This generates the language of all strings containing an even number of *a*'s. To see this, consider the expression inside the outer left repetition:

```
(b|c)*a(b|c)*a
```

This generates those strings ending in *a* that contain exactly two *a*'s (any number of *b*'s and *c*'s can appear before or between the two *a*'s). Repeating these strings gives all strings ending in *a* whose number of *a*'s is a multiple of 2 (i.e., even). Tacking on the repetition `(b|c)*` at the end (as in the previous example) gives the desired result.

We note that this regular expression could also be written as

```
(not a* a not a* a)* not a*
```

§

Recognition of Tokens

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

ws → (**blank** | **tab** | **newline**)$^+$

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

Recognition of Tokens

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

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

A state: as summarizing all between the lexemeBegin pointer and the forward pointer

Edges are directed from one state of the transition diagram to another

Each edge is labeled by a symbol or set of symbols

If we are in some state, s, and the next input symbol is, a, we look for an edge out of state s labelled by a

If we find such an edge, we advance the forward pointer arid enter the state of the transition diagram to which that edge leads

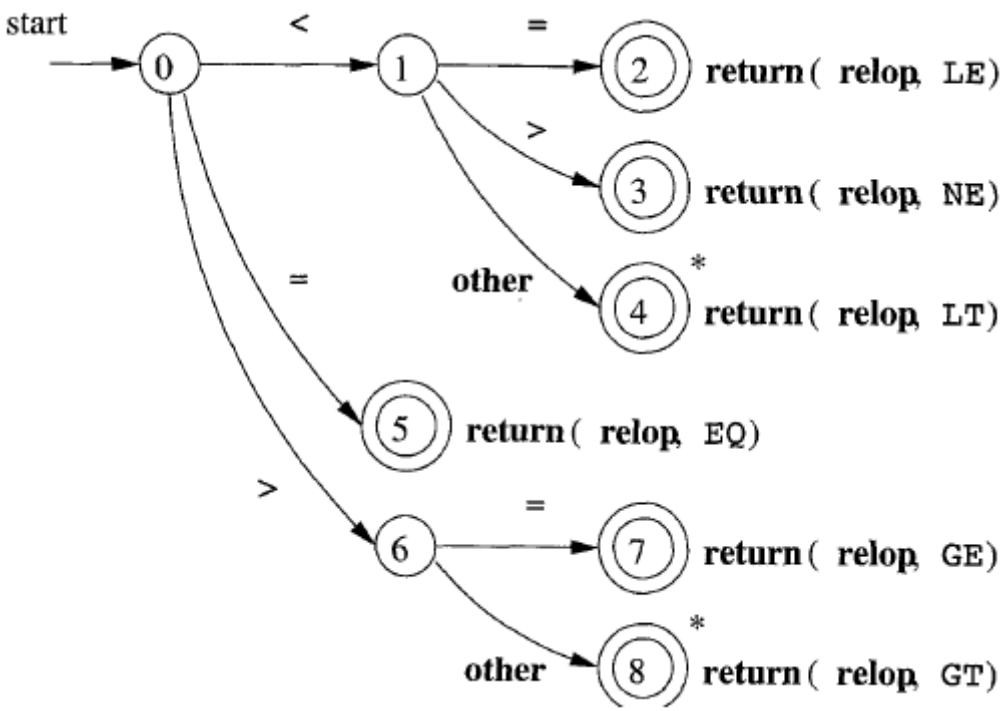
Transition Diagrams

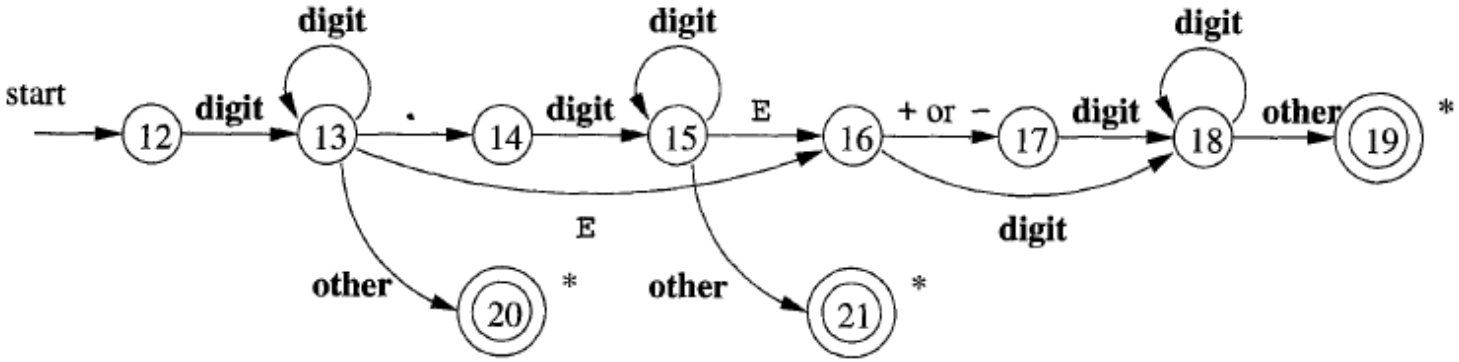
1. Certain states are said to be *accepting*, or *final*. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the *lexemeBegin* and *forward* pointers. We always indicate an accepting state by a double circle, and if there is an action to be taken — typically returning a token and an attribute value to the parser — we shall attach that action to the accepting state.

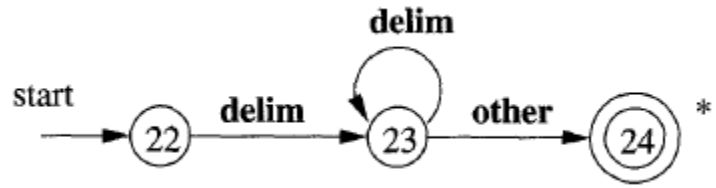
2. In addition, if it is necessary to retract the *forward* pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract *forward* by more than one position, but if it were, we could attach any number of *'s to the accepting state.

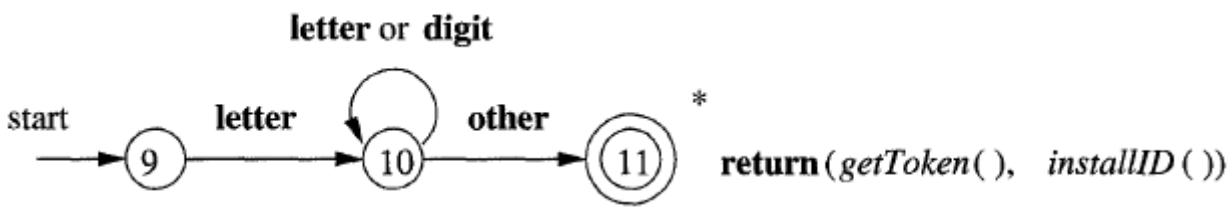
3. One state is designated the *start state*, or *initial state*; it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Transition Diagrams





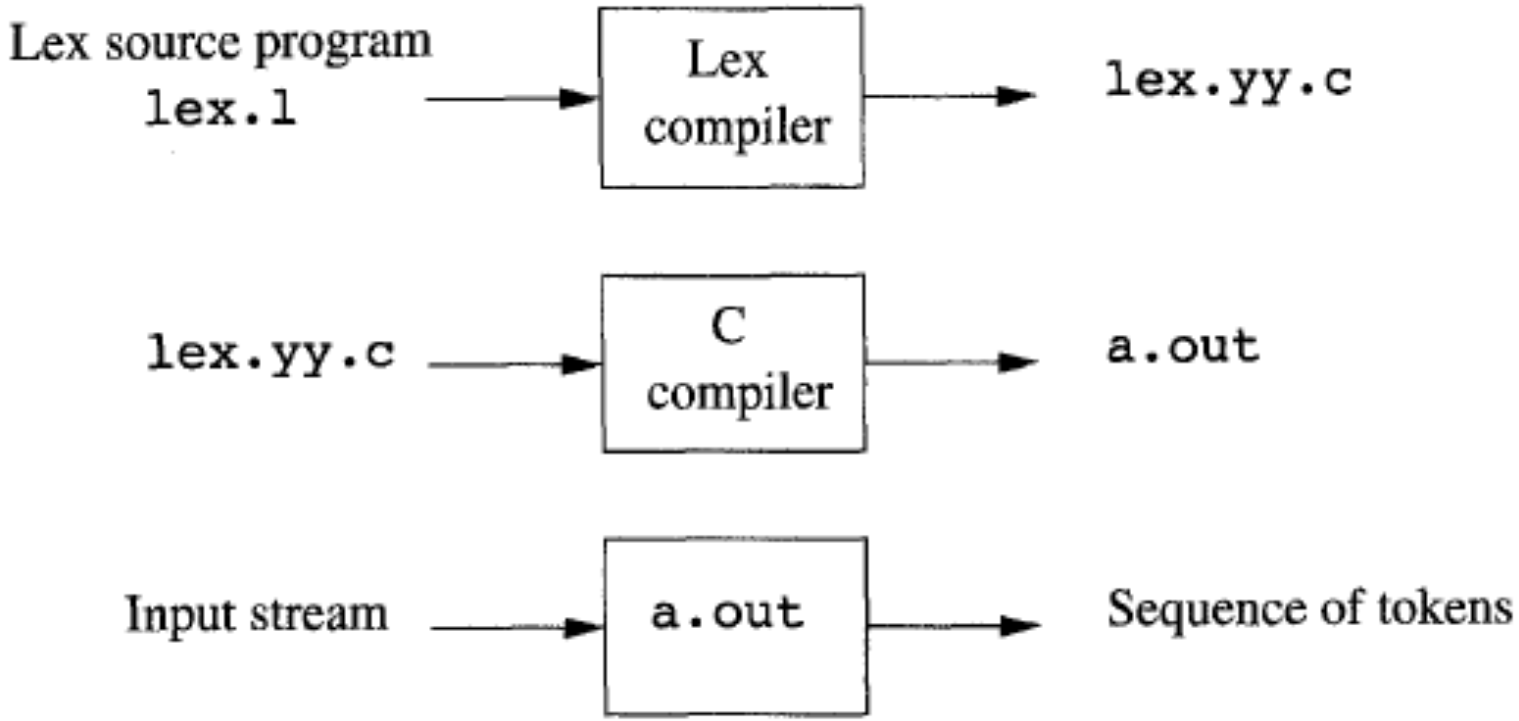




LEX

A Lex program has the following form:

declarations
%%
translation rules
%%
auxiliary functions



More on next class