بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

Lexical analysis

This is the initial part of reading and analyzing the program text: The text is read and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number. Lexical analysis is often abbreviated to lexing.

A lexical analyser, also called a lexer or scanner, will as input take a string of individual letters and divide this string into a sequence of classified tokens. Additionally, it will filter out whatever separates the tokens (the so-called white-space), i.e., lay-out characters (spaces, newlines etc.) and comments.

The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase.

In theory, the work that is done during lexical analysis can be made an integral part of syntax analysis, and in simple systems this is indeed often done. However, there are reasons for keeping the phases separate:

☞ *Efficiency: A specialised lexer may do the simple parts of the work faster than the parser, which uses more general methods, can. Furthermore, the size of a system that is split in two phases may be smaller than a combined system. This may seem paradoxical but, as we shall see, there is a non-linear factor involved which may make a separated system smaller than a combined system.*

☞ *Modularity: The syntactical description of the language need not be cluttered with small lexical details such as white-space and comments.*

☞ *Tradition: Languages are often designed with separate lexical and syntactical phases in mind, and the standard documents of such languages typically separate lexical and syntactical elements of the languages.*

Lexical Analysis Versus Parsing

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

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.

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.

lexer generators

It is usually not terribly difficult to write a lexer by hand: You first read past initial white-space, then you, in sequence, test to see if the next token is a keyword, a number, a variable or whatnot. However, this is not a very good way of handling the problem: You may read the same part of the input repeatedly while testing each possible token, and in some cases it may not be clear where one token ends and the next begins. Furthermore, a handwritten lexer may be complex and difficult to maintain. Hence, lexers are normally constructed by lexer generators, that transform (somewhat) human-readable specifications of tokens and white-space into efficient programs.

`flex`: *A Fast Lexical Analyzer Generator*

`https://web.mit.edu/gnu/doc/html/flex\_toc.html\#SEC1`

The scanner's task is to transform a stream of characters into a stream of words in the input language. Each word must be classified into a syntactic category, or "part of speech." The scanner is the only pass in the compiler to touch every character in the input program. Compiler writers place a premium on speed in scanning, in part, because the scanner's input is larger, in some measure, than that of any other pass, and, in part, because highly efficient techniques for scanning are easy to understand and to implement.

The scanner aggregates characters into words. For each word, it determines if the word is valid in the source language. For each valid word, it assigns the word a syntactic category, or part of speech. The scanner is the only pass in the compiler that manipulates every character of the input program. Because scanners perform a relatively simple task, grouping characters together to form words and punctuation in the source language, they lend themselves to fast implementations.

The lexical analyzer gathers the characters of the source program into lexical units. The lexical units of a program are identifiers, special words, operators, and punctuation symbols. The lexical analyzer ignores comments in the source program because the compiler has no use for them.

The lexical analyzer takes a stream of characters and produces a stream of names, keywords, and punctuation marks; it discards white space and comments between the tokens. It would unduly complicate the parser to have to account for possible white space and comments at every possible point; this is the main reason for separating lexical analysis from parsing.

Lexical analysis is not very complicated, but we will attack it with high-powered formalisms and tools, because similar formalisms will be useful in the study of parsing and similar tools have many applications in areas other than compilation.

Lexical Analysis

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

that it passes on to the subsequent phase, syntax analysis.

In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

$$\texttt{position} = \texttt{initial} + \texttt{rate} * 60$$

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

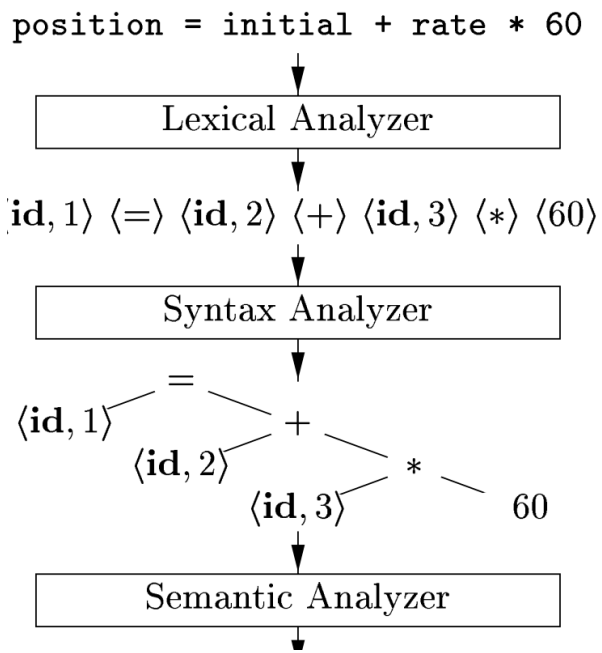
1. $\texttt{position}$ *is a lexeme that would be mapped into a token* $\langle \mathrm{id}, 1 \rangle$, *where* id *is an abstract symbol standing for identifier and* 1 *points to the symbol-table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.*

2. *The assignment symbol* $=$ *is a lexeme that is mapped into the token* $\langle = \rangle$. *Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as* assign *for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.*
3. `initial` *is a lexeme that is mapped into the token* $\langle \mathrm{id}, 2 \rangle$, *where* 2 *points to the symbol-table entry for initial.*
4. $+$ *is a lexeme that is mapped into the token* $\langle + \rangle$.
5. `rate` *is a lexeme that is mapped into the token* $\langle \mathrm{id}, 3 \rangle$, *where* 3 *points to the symbol-table entry for rate.*
6. $*$ *is a lexeme that is mapped into the token* $\langle * \rangle$.
7. 60 *is a lexeme that is mapped into the token* $\langle 60 \rangle$.

Blanks separating the lexemes would be discarded by the lexical analyzer.

$$\langle \mathbf{id}, 1\rangle\langle =\rangle\langle \mathbf{id}, 2\rangle\langle +\rangle\langle \mathbf{id}, 3\rangle\langle *\rangle\langle 60\rangle$$

In this representation, the token names $=$, $+$, and $*$ are abstract symbols for the assignment, addition, and multiplication operators, respectively.



1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE

The Role of the Lexical Analyzer

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.

The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

These interactions are suggested in Fig. 3.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. 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.

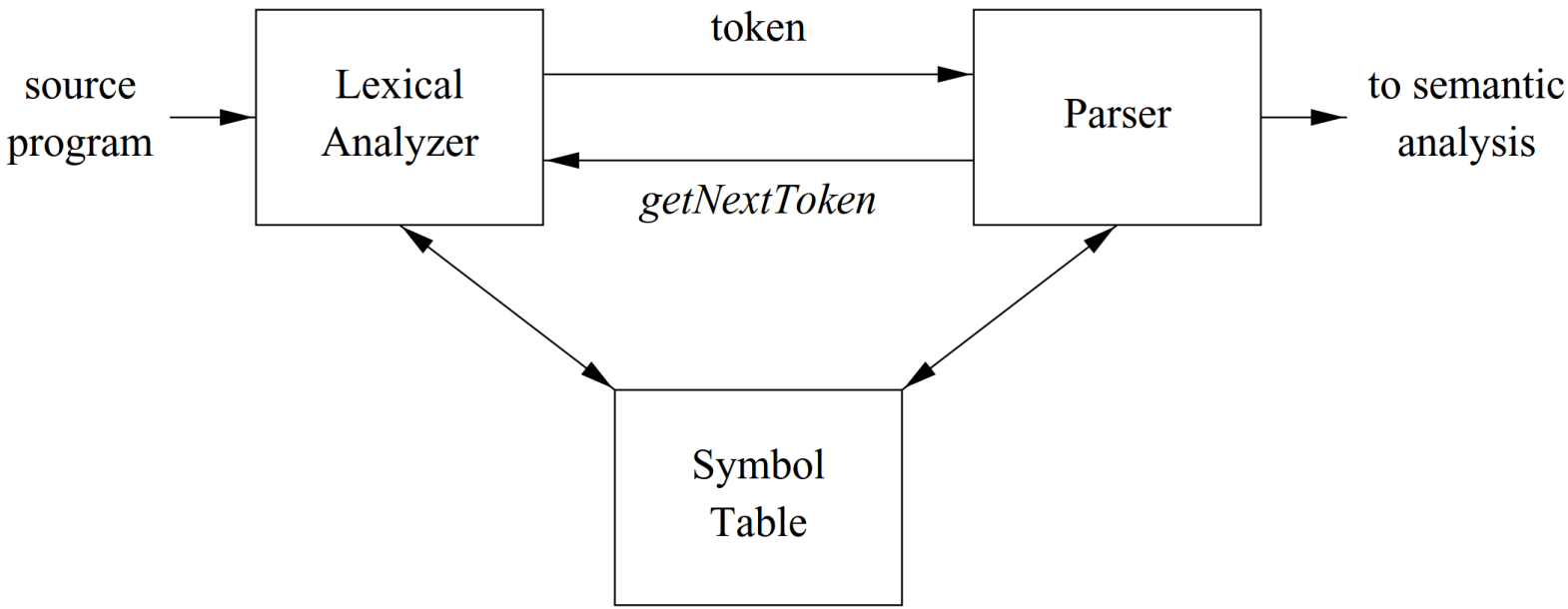


Figure 3.1: Interactions between the lexical analyzer and the parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Sometimes, lexical analyzers are divided into a cascade of two processes:

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

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

Token, Pattern, and Lexeme

☞ *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.*

☞ *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.*

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

Example 3.1 : Figure 3.2 gives some typical tokens, their informally described patterns, and some sample lexemes. 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 **literal**. □

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

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 strings.*
5. *Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.*

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 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, 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.

The token id

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.

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.

<**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>

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. In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for **number** a pointer to that string. □

بیان کتاب ببر و یک مثال از این کتاب

A lexical token is a sequence of characters that can be treated as a unit in the grammar of a programming language. A programming language classifies lexical tokens into a finite set of token types. For example, some of the token types of a typical programming language are Type Examples

Type	Examples
ID	`foo n14 last`
NUM	`73 0 00 515 082`
REAL	`66.1 .5 10. 1e67 5.5e-10`
IF	`if`
COMMA	`,`
NOTEQ	`!=`
LPAREN	`(`
RPAREN	`)`

Punctuation tokens such as IF, VOID, RETURN constructed from alphabetic characters are called reserved words and, in most languages, cannot be used as identifiers.

Given a program such as

```
float match0(char *s) /* find a zero */
{if (!strncmp(s, "0.0", 3))
  return 0.;
}
```

the lexical analyzer will return the stream

```
FLOAT   ID(match0)   LPAREN   CHAR   STAR   ID(s)   RPAREN
LBRACE   IF   LPAREN   BANG   ID(strncmp)   LPAREN   ID(s)
COMMA   STRING(0.0)   COMMA   NUM(3)   RPAREN   RPAREN
RETURN   REAL(0.0)   SEMI   RBRACE   EOF
```

where the token-type of each token is reported; some of the tokens, such as identifiers and literals, have semantic values attached to them, giving auxiliary information in addition to the token-type.

Examples of nontokens are

comment	`/* try again */`
preprocessor directive	`#include<stdio.h>`
preprocessor directive	`#define NUMS 5 , 6`
macro	`NUMS`
blanks, tabs, and newlines	

In languages weak enough to require a macro preprocessor, the preprocessor operates on the source character stream, producing another character stream that is then fed to the lexical analyzer. It is also possible to integrate macro processing with lexical analysis.

اهمیت عبارات منظم در بحث طراحی lexer

For lexical analysis, specifications are traditionally written using regular expressions: An algebraic notation for describing sets of strings. The generated lexers are in a class of extremely simple programs called finite automata.

Any reasonable programming language serves to implement an ad hoc lexer. But we will specify lexical tokens using the formal language of regular expressions, implement lexers using deterministic finite automata, and use mathematics to connect the two. This will lead to simpler and more readable lexical analyzers.

اهمیت عبارات منظم در بحث طراحی lexer

Regular expressions are an important notation for specifying lexeme patterns. Using this language, we can specify the lexical tokens of a programming language. While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens.

We shall study the formal notation for regular expressions, we shall see how these expressions are used in a lexical-analyzer generator (e.g., `flex`), we see how to build the lexical analyzer by converting regular expressions to automata that perform the recognition of the specified tokens.

Regular expressions are convenient for specifying lexical tokens, but we need a formalism that can be implemented as a computer program. For this we can use finite automata (N.B. the singular of automata is automaton).

تعریف رسمی یک عبارت منظم

Let Σ be a given alphabet. Then

1. $\varnothing$, λ, and $a \in \Sigma$ are all regular expressions. These are called primitive regular expressions.

2. If r_1 and r_2 are regular expressions, so are $r_1 + r_2$, $r_1 \cdot r_2$, r_1^*, and (r_1).

3. A string is a regular expression if and only if it can be derived from the primitive regular expressions by a finite number of applications of the rules in (2).

Languages Associated with REs

If r is an RE, we will let $L(r)$ denote the language associated with r. The language $L(r)$ denoted by any regular expression r is defined by the following rules.

1. *$\varnothing$ is a regular expression denoting the empty set,*
2. *λ is a regular expression denoting $\{\lambda\}$,*
3. *For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.*

If r_1 and r_2 are regular expressions, then

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 r_2) = L(r_1)L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

The last four rules of this definition are used to reduce $L(r)$ to simpler components recursively; the first three are the termination conditions for this recursion. To see what language a given expression denotes, we apply these rules repeatedly.

از کتاب ببر (∅ به‌عنوان یک عبارت منظم ***primitive*** در نظر گرفته نشده است)

Symbol: For each symbol **a** in the alphabet of the language, the regular expression **a** denotes the language containing just the string `a`.

Alternation: Given two regular expressions M and N, the alternation operator written as a vertical bar | makes a new regular expression $M \mid N$. A string is in the language of $M \mid N$ if it is in the language of M or in the language of N. Thus, the language of **a** | **b** contains the two strings `a` and `b`.

Concatenation: Given two regular expressions M and N, the concatenation operator $\cdot$ makes a new regular expression $M \cdot N$. A string is in the language of $M \cdot N$ if it is the concatenation of any two strings α and β such that α is in the language of M and β is in the language of N. Thus, the regular expression (**a** | **b**) · **a** defines the language containing the two strings `aa` and `ba`.

Epsilon: The regular expression ϵ represents a language whose only string is the empty string. Thus, $(a \cdot b) \mid \epsilon$ represents the language {"","ab"}.

Repetition: Given a regular expression M, its Kleene closure is M^*. A string is in M^* if it is the concatenation of zero or more strings, all of which are in M. Thus, ((**a** | **b**) · **a**)* represents the infinite set { "" , "aa", "ba", "aaaa", "baaa", "aaba", "baba", "aaaaaa", . . . }.

Using symbols, alternation, concatenation, epsilon, and Kleene closure we can specify the set of ASCII characters corresponding to the lexical tokens of a programming language. First, consider some examples:

$(\mathbf{0} \mid \mathbf{1})^* \cdot \mathbf{0}$	Binary numbers that are multiples of two.
$\mathbf{b}^*(\mathbf{abb}^*)^*(\mathbf{a}\mid\epsilon)$	Strings of `a`'s and `b`'s with no consecutive `a`'s.
$(\mathbf{a}\mid\mathbf{b})^*\mathbf{aa}(\mathbf{a}\mid\mathbf{b})^*$	Strings of `a`'s and `b`'s containing consecutive `a`'s.

In writing regular expressions, we will sometimes omit the concatenation symbol or the epsilon, and we will assume that Kleene closure "binds tighter" than concatenation, and concatenation binds tighter than alternation; so that $\mathbf{ab} \mid \mathbf{c}$ means $(\mathbf{a} \cdot \mathbf{b}) \mid \mathbf{c}$, and $(\mathbf{a} \mid)$ means $(\mathbf{a} \mid \epsilon)$.

Extensions of REs (Shorthands)

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns. Here we mention a few notational extensions that were first incorporated into Unix utilities such as `Lex` *that are particularly useful in the specification lexical analyzers.*

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**.

یک نکتهٔ مهم: این توسیع‌ها چیزی به قدرت عبارات منظم اضافه نمی‌کند

We must stress that these shorthands are just that. They do not allow more languages to be described, they just make it possible to describe some languages more compactly. In the case of s^+, it can even make an exponential difference: If $+$ is nested n deep, recursive expansion of s^+ to ss^* yields $2^n - 1$ occurrences of $*$ in the expanded regular expression. For example, $((a^+b)^+c)^+$ expands to $aa^*b(aa^*b)^*c(aa^*b(aa^*b)^*c)^*$.

Example: $[abcd]$ ***means*** $(a|b|c|d)$, `[b-g]` ***means*** `[bcdefg]`, `[b-gM-Qkr]` ***means*** `[bcdefgMNOPQkr]`, $M?$ ***means*** $(M|\varepsilon)$, ***and*** M^+ ***means*** $(M \cdot M^*)$.

`if`	IF
`[a-z][a-z0-9]*`	ID
`[0-9]+`	NUM
`([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)`	REAL
`("--"[a-z]*"\n")|(" "|"\n"|"\t")+`	*no token, just white space*
`.`	*error*

FIGURE 2.2. Regular expressions for some tokens.

The shorthand symbols $+$ ***and*** $?$ ***bind with the same precedence as*** $*$.

Some algebraic properties of regular expressions

$(r\|s)\|t = r\|s\|t = r\|(s\|t)$	$\|$ is associative.
$s\|t = t\|s$	$\|$ is commutative.
$s\|s = s$	$\|$ is idempotent.
$s? = s\|\varepsilon$	by definition.
$(rs)t = rst = r(st)$	concatenation is associative.
$s\varepsilon = s = \varepsilon s$	ε is a neutral element for concatenation.
$r(s\|t) = rs\|rt$	concatenation distributes over $\|$.
$(r\|s)t = rt\|st$	concatenation distributes over $\|$.
$(s^*)^* = s^*$	* is idempotent.
$s^*s^* = s^*$	0 or more twice is still 0 or more.
$ss^* = s^+ = s^*s$	by definition.
$(s^+)^+ = s^+$	$^+$ is idempotent.
$s^+s^* = s^+ = s^*s^+$	still just at least 1

Shorthands

For example, if we want to describe non-negative integer constants, we can do so by saying that a number constant is a sequence of one or more digits, which is expressed by the regular expression

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

The large number of different digits makes this expression rather verbose. It gets even worse when we get to variable names, where we must enumerate all alphabetic letters (in both upper and lower case). Hence, we introduce a shorthand for sets of letters. We can now write this much shorter as `[0-9]`$^+$.

☞ *A sequence of letters enclosed in square brackets represents the set of these letters. For example, we use* $[ab01]$ *as a shorthand for* $a|b|0|1$.

☞ *Additionally, we can use interval notation to abbreviate*

$$[0123456789]$$

to `[0-9]`.

☞ *We can combine several intervals within one bracket and for example write* `[a-zA-Z]` *to denote all alphabetic letters in both lower and upper case.*

Remark: When using intervals, we must be aware of the ordering for the symbols involved. For the digits and letters used above, there is usually no confusion. However, if we write, e.g., `[0-z]` *it is not immediately clear what is meant. When using such notation in lexer generators, a character set encoding such as ASCII, ISO 8859-1, or UTF-8 is usually implied, so the symbols are ordered as defined by these encodings.*

Example (Floats): *In C, a floating-point constant can have an optional sign. After this, the **mantissa** part is described as a sequence of digits followed by a decimal point and then another sequence of digits. Either one (but not both) of the digit sequences can be empty. Finally, there is an optional **exponent** part, which is the letter e(in upper or lower case) followed by an (optionally signed) integer constant. If there is an exponent part to the constant, the mantissa part can be written as an integer constant (i.e., without the decimal point). Some examples:* `3.14`, `-3.`, `.23`, `3e+4`, `11.22e-3`.

This rather involved format can be described by the following regular expression:

$$[+-]?(([0-9]^{+}.[0-9]^{*} \mid .[0-9]^{+})([eE][+-]?[0-9]^{+})?) \mid [0-9]^{+}[eE][+-]?[0-9]^{+})$$

This regular expression is complicated by the fact that the exponent is optional if the mantissa contains a decimal point, but not if it does not (as that would make the number an integer constant). We can make the description simpler if we make the regular expression for floats also include integers, and instead use other means of distinguishing integers from floats. If we do this, the regular expression can be simplified to

$$[+-]?(([0-9]^{+}(.[0-9]^{*})?|.[0-9]^{+})([eE][+-]?[0-9]^{+})?)$$

Some languages require digits on both sides of the decimal point (if there is a decimal point). This simplifies the description considerably, as there are fewer special cases:

$$[+-]?(([0-9]^{+}(.[0-9]^{+})?([eE][+-]?[0-9]^{+})?)$$