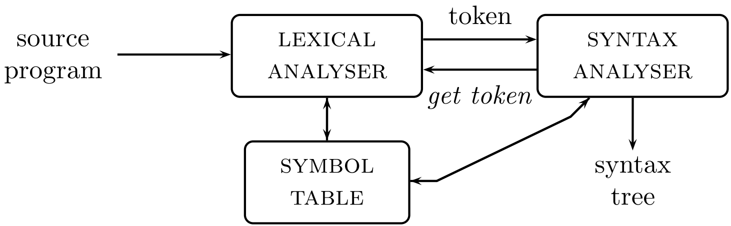
Lexer

The lexical analyser is the first phase of a compiler. Its main task is to read the input characters and produce a sequence of tokens that the syntax analyser uses.



Upon receiving a request for a token (*get token*) from the parser, the lexical analyser reads input characters until a lexeme is identified and returned to the parser together with the corresponding token.

Lexer (cont)

Usually, a lexical analyser is in charge of

- stripping out from the source program **comments** and **white spaces**, in the form of blank, tabulation and newline characters;
- keeping trace of the position of the lexemes in the source program, so the error handler can refer to exact positions in error messages.

Lexer/Tokens, lexemes, patterns

A **token** is a set of strings which are interpreted in the same way, for a given source language. For instance, **id** is a token denoting the set of all possible identifiers.

A **lexeme** is a string belonging to a token. For example, `5.4` is a lexeme of the token **num**.

The tokens are defined by means of **patterns**. A pattern is a kind of compact rule describing all the lexemes of a token. A pattern is said to *match* each lexeme in the token.

For example, in the Pascal statement

```
const pi = 3.14159;
```

the substring `pi` is a lexeme for the token **id** (*identifier*).

Lexer/Tokens, lexemes, patterns (cont)

Token	Sample lexemes	Informal pattern
id	`pi count D2 ...`	letter followed by letters and digits
relop	`< <= = >= >`	< or <= or < or = or >= or >
const	`const`	`const`
if	`if`	`if`
num	`3.14 4 .2E2 ...`	any numeric constant
literal	`"message" "" ...`	any characters between " and " except "

Lexer/Tokens, lexemes, patterns (cont)

Most recent programming languages distinguish a finite set of strings that match the identifiers but are not part of the identifier token: the **keywords**.

For example, in Ada, `function` is a keyword and, as such, is not a valid identifier.

In C, `int` is a keyword and, as such, cannot be used an identifier (e.g. to declare a variable).

Nevertheless, it is common **not** to create explicitly a **keyword** token and let each keyword lexeme be the only one of its own token, as displayed in the table page 44.

Specification of tokens

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for sets of strings.

Strings and formal languages

The term **alphabet** denotes any finite set of symbols. Typical examples of symbols are letters and digits. The set $\{0, 1\}$ is the *binary alphabet*. ASCII is an example of computer alphabet.

Specification of tokens (cont)

A **string** over some alphabet is a finite sequence of symbols drawn from that alphabet. The terms **sentence** and **word** are often used as synonyms.

The length of string s, usually noted $|s|$, is the number of occurrences of symbols in s. For example, `banana` is a string of length six. The **empty string**, denoted ε, is a special string of length zero.

Specification of tokens/Strings and formal languages (cont)

Term	Informal definition
prefix of s	A string obtained by removing zero or more trailing symbols of string s; e.g. `ban` is a prefix of `banana`.
suffix of s	A string formed by deleting zero or more of the leading symbols of s; e.g. `nana` is a suffix of `banana`.
substring of s	A string obtained by deleting a prefix and a suffix from s; e.g. `nan` is a substring a `banana`. Every prefix and every suffix of s is a substring s, but not every substring of s is a prefix or a suffix of s. For every string s, both s and ε are prefixes, suffixes and substrings of s.

Specification of tokens/Strings and formal languages (cont)

Term	Informal definition
proper prefix, suffix or substring of s	Any non-empty string x that is, respectively, a prefix, suffix, substring of s such that $s \neq x$; e.g. ε and `banana` are **not** proper prefixes of `banana`.
subsequence of s	Any string formed by deleting zero or more not necessarily contiguous symbols from s; e.g. `baaa` is a subsequence of `banana`.

Specification of tokens/Strings and formal languages (cont)

The term **language** denotes any set of strings over some fixed alphabet.

The **empty set**, noted $\varnothing$, or $\{\varepsilon\}$, the set containing only the empty word are languages. The set of valid C programs is an infinite language.

If x and y are strings, then the **concatenation** of x and y, written xy or $x \cdot y$, is the string formed by appending y to x.

For example, if $x = \texttt{dog}$ and $y = \texttt{house}$, then $xy = \texttt{doghouse}$.

The empty string is the identity element under concatenation: $s\varepsilon = \varepsilon s = s$.

Specification of tokens/Strings and formal languages (cont)

If we think of concatenation as a product, we can define string exponentiation as follows.

- $s^0 = \varepsilon$
- $s^n = s^{n-1}s$, if $n > 0$.

Since $\varepsilon s = s$, $s^1 = s$, then $s^2 = ss$, $s^3 = sss$ etc.

Specification of tokens/Strings and formal languages (cont)

We can now revisit the definitions we gave in table page 48 and 49 using a formal notation. Let L be the language under consideration.

Term	Formal definition
x is a *prefix* of s	$\exists y \in L.s = xy$
x is a *suffix* of s	$\exists y \in L.s = yx$
x is a *substring* of s	$\exists u, v \in L.s = uxv$
x is a *proper prefix* of s	$\exists y \in L.y \neq \varepsilon$ and $s = xy$
x is a *proper suffix* of s	$\exists y \in L.y \neq \varepsilon$ and $s = yx$
x is a *proper substring* of s	$\exists u, v \in L.uv \neq \varepsilon$ and $s = uxv$

Specification of tokens/Operations on languages

It is possible to define operation in languages. For lexical analysis, we are interested mainly in **union**, **concatenation** and **closure**. Let L and M be two languages.

Operation	Formal definition
union of L and M	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
concatenation of L and M	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$ where $L^0 = \{\varepsilon\}$
positive closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

In other words, L^* means "zero or more concatenations of L", and L^+ means "one or more concatenations of L."

Specification of tokens/Operations on languages/Examples

Let $L = \{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$ and $D = \{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}\}$.

1. L is the alphabet consisting of the set of upper and lower case letters and D is the alphabet of the decimal digits.
2. Since a symbol is a string of length one, the sets L and D are finite languages too.

These two ways of considering L and D and the operations on languages allow us to create new languages from other languages defined by their alphabet.

Here are some examples of new languages created from L and D:

- $L \cup D$ is the language of letters and digits.
- LD is the language whose words consist of a letter followed by a digit.

Specification of tokens/Operations on languages/Examples (cont)

- L^4 is the language whose words are four-letter strings.
- L^* is the language made up on the alphabet L, i.e. the set of all strings of letters, including the empty string ε.
- $L(L \cup D)^*$ is the language whose words consist of letters and digits and beginning with a letter.
- D^+ is the language whose words are made of one or more digits, i.e. the set of all decimal integers.