

## Lexical analysis

```
if (x == y)
    z = 1;
else
    z = 2;
```

will look like this to the lexical analyzer:

```
\tif (x == y)\n\t z = 1;\n\telse\n\t z = 2;\n
```

The lexical analyzer has to put dividers between different units. e.g., a divider between `\t` and `if` and `<whitespace>` and `(` and so on ...

## Token Class (or Class)

- In English: Noun, Verb, Adjective, ...
- In programming language: Identifier, Keyword, `(`, `)`, ...

Token classes correspond to sets of strings

- Identifier: a string of letters or digits, starting with a letter.
- Integer: a non-empty string of digits
- Keyword: 'if', 'then', 'else', ';;', etc.
- Whitespace: a non-empty sequence of blanks, newlines, and tabs

Lexical analyzer classifies strings according to role (token class). Communicates tokens to parser.

```
lexer :: string --> [ token ]
token = <class, string>
```

Example input:

```
foo = 42
```

Example output:

```
<Id, "foo"> <Op, "="> <Int, "42">
```

In our original example

```
\tif (x == y)\n\t z = 1;\n\telse\n\t z = 2;\n
```

the relevant token classes are:

- Operator
- Whitespace
- Keywords
- Identifiers
- Numbers
- '('
- ')'
- ';
- '=='

Work through the example to show the resulting list of tokens

A lexical analyzer needs to

1. recognize substrings corresponding to tokens (called *lexemes*)
2. recognize token class of each lexeme to generate `<token-class, lexeme>`

## Lexical analysis examples: Fortran

In Fortran, whitespace is insignificant:

```
VAR1
```

is exactly the same as

```
VA R1
```

Fortran idea: removing all the whitespace should not change the meaning of the program.

Loop example

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

Here DO is a keyword representing a loop (similar to FOR in C), I = 1,25 represents that the iteration variable I ranges from 1..25. The number 5 represents that the loop extends from the DO statement till the statement with label 5, including all statements that are in between.

Another example:

```
DO 5 I = 1.25
```

The only difference in this code from the previous code is the replacement of , with .. This simply means that the variable DO5I = 1.25, i.e., a variable has been assigned an integer (there is no loop). The problem is that just by looking at the first three characters, I cannot tell whether DO is a keyword or a prefix of a variable name. Hence we need a *lookahead*. In this example, a large lookahead is required. Ideally, the language design should ensure that the lookahead is small.

1. Goal: partition the string. Implemented by reading left-to-right, recognizing one token at a time
2. *Lookahead* may be required to decide where one token ends and the next token begins. We would like to minimize lookahead.

## Lookahead is always required

For example, when we read the first character of the keyword else, we need to lookahead to decide whether it is an identifier or a keyword. Similarly, we need lookahead to disambiguate between == and =.

PL/1: keywords are not reserved.

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

This makes lexical analysis a bit more difficult -- need to decide what is a variable name and what is a keyword, and so need to look at what's going on in the rest of the expression.

In fact, there are examples where PL/1 may require unbounded lookahead!

Some of these problems exist in languages today: e.g., C++

C++ template syntax

```
Foo<Bar>
```

C++ stream syntax

```
cin >> var
```

For a long time, you had to put a blank between two ">" signs in the template syntax. Most C++ compilers have fixed this now.

## Regular Languages

Typically, language designers would like to keep lexical analysis as simple as possible. Perhaps the most simple class of languages is *Regular Languages*. Hence, lexical analysis of most programming languages can be implemented by implementing a classifier for regular languages. Regular languages are the weakest formal languages widely used, and have many applications.

## Regular Expressions

'c' = {"c"}

\epsilon = {""}

Note that \epsilon (a language that has a single element, namely the empty string) is not the same as \phi (empty language).

**Union:**  $A + B = \{ a \mid a \in A \} \cup \{ b \mid b \in B \}$

**Concatenation:**  $AB = \{ ab \mid a \in A \text{ and } b \in B \}$

**Iteration (Kleene closure):**  $A^* = A^0 + A^1 + A^2 + \dots + A^\infty$

$A^0 = \epsilon$

$A^i = A$  concatenated with itself  $i$  times

**Definition:** The regular expressions over \Sigma are the smallest set of expressions that includes the following:

R =

- | \epsilon
- | 'c'
- | R + R
- | RR
- | R\*

This is called the grammar of the regular expressions. We will learn more about grammars when we talk about parsing (not relevant at this stage).

\Sigma = {0, 1}

$1^* = \text{all strings of 1s}$   $(1 + 0)^1 = \{11, 01\}$

$0^* + 1^* = \text{all strings of 0s UNION all strings of 1s}$   $(0 + 1)^* = \text{all strings of 0s and 1s (or all strings over the entire alphabet)} = \Sigma^*$  (special name for this language)

Many ways of writing any regular language

Regular expressions specify regular languages

Five constructs: Two base cases (\epsilon, single-character strings). Three compound expressions (union, concatenation, iteration)