Token classes correspond to sets of strings.

Identifier:
- strings of letters or digits, starting with a letter.

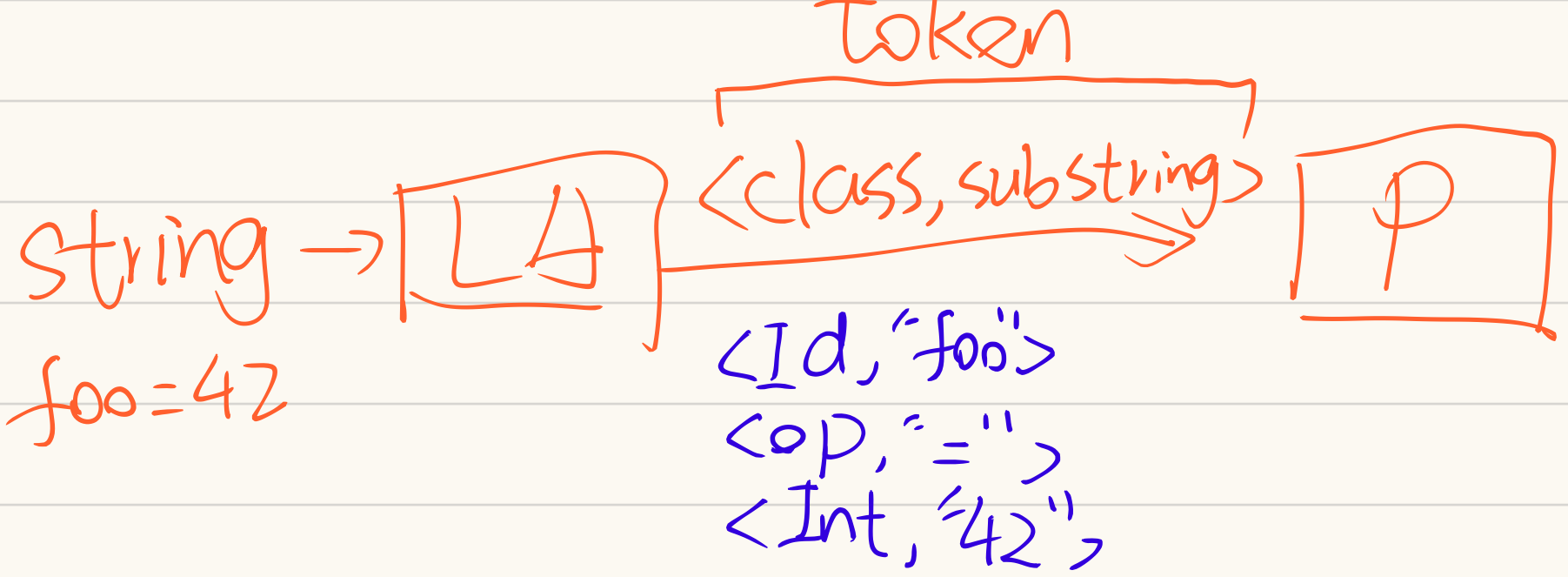
Integer:
- a non-empty string of digits.

Keyword:
- "else" or "if" or "begin" or ...

Whitespace:
- a non-empty sequence of blanks, newlines, and tabs.

Classify program substrings according to role.
token class

Communicate tokens to the parser.



An Implementation must do two things:

1. Recognize substrings corresponding to tokens
   - the lexemes 词素
2. Identify the token class of each lexeme.
   - <token class, lexeme> — token

FORTRAN rule: Whitespace is insignificant

VAR1 is the same as VA R1

1. The goal is to partition the string. This is 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.

The goal of LA is to

- Partition the input string into lexemes
- Identify the token of each lexeme

Left-to-right scan ⇒ lookahead sometimes required

Lexical structure = token classes

We must say what set of strings is in a token class.

– Use regular languages.

Regular Languages

- single character

$'c' = \{"c"\}$

- Epsilon

$\varepsilon = \{""\}$

two base cases

three compound expressions

- Union

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

- Concatenation

$AB = \{ab \mid a \in A \wedge b \in B\}$

- Iteration

$A^* = \bigcup_{i \geq 0} A^i$

$A^i = A \ldots A$ (i times)

$A^0 = \varepsilon$

Def. The regular expression over Σ are the smallest set of expressions including

Def. Let Σ be a set of characters (an alphabet) A language over Σ is a set of strings of characters drawn from Σ

Alphabet = English characters
Language = English sentences

Alphabet = ASCII
Language = C program

Meaning function L maps syntax to semantics.

$L(e) = M$
↓ ↓
reg exp set of strings

Why use a meaning function?

- Make clear what is syntax, what is semantics
- Allows us to consider notation as a separate issue.
- Because expressions and meanings are not 1-1

Meaning is many to one, and never one to many!