

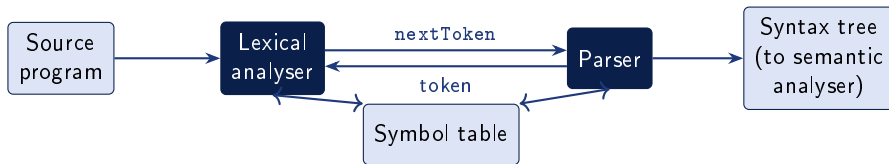
Lexical Analysis

Compiler Construction, Lecture 2

K.R. Chowdhary, Professor of CS

July 12, 2019 | Fall 2019

Role of the Lexical Analyser



Tasks

- Read characters, group into lexemes, output tokens
- Enter identifiers into the symbol table
- Strip whitespace and comments

Why separate it from parsing?

- **Simplicity:** parser need not handle whitespace/comments
- **Efficiency:** specialised techniques and input buffering
- **Portability:** device-specific details stay in the lexer

- Token name guides parsing; attribute value guides translation
- At most one attribute per token; for `id` it points to the symbol-table entry (lexeme, type, location)
- **Tokens:** keywords, identifiers, operators, punctuation, brackets
- **Non-tokens:** comments, preprocessor directives, macros, blanks, tabs, newlines

Fortran `D05I = 1,20`: until the comma is seen, `D05I` could be a variable or a `DO` statement.

Example: `e = m * c ** 2`

<code><id, 1></code>	<code>e</code>
<code><assignop></code>	<code>=</code>
<code><id, 2></code>	<code>m</code>
<code><multop></code>	<code>*</code>
<code><id, 3></code>	<code>c</code>
<code><expop></code>	<code>**</code>
<code><number, 2></code>	<code>2</code>

Operators, punctuation and keywords need no attribute.

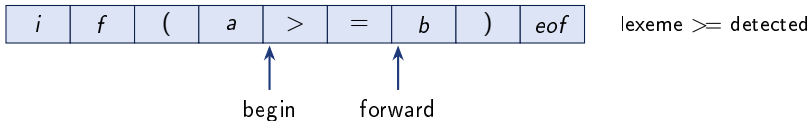
Lexical Errors and Input Buffering

Lexical errors

- `fi(a == b)`: `fi` is a valid `id`, so the lexer cannot tell it is a misspelt `if`; the parser catches it
- **Panic mode**: delete characters until a well-formed token starts
- Other repairs: delete, insert, replace, or transpose a character

Input buffering

- Source is held in a RAM buffer (1, 2 or 4 KB block) for fast reads and look-ahead
- Look-ahead is needed: `-` vs. `—`, `>` vs. `>=`
- Two pointers: *begin* and *forward*
- `eof` acts as a sentinel; buffer reloads at its end



Operators, by precedence

1. $*$ (closure): highest, left assoc.
2. concatenation
3. $|$ (union): lowest

Examples over $\Sigma = \{a, b\}$

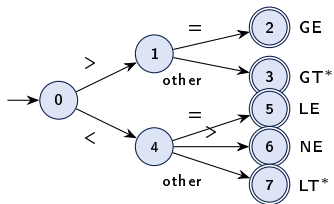
- $a|b = \{a, b\}$
- $(a|b)(c|d) = \{ac, ad, bc, bd\}$
- $a^* = \{\epsilon, a, aa, aaa, \dots\}$

Token definitions

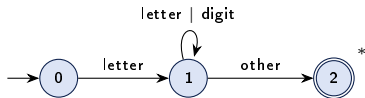
```
letter_  $\rightarrow A | \dots | Z | a | \dots | z | \_$   
digit  $\rightarrow 0 | 1 | \dots | 9$   
id  $\rightarrow letter\_ (letter\_ | digit)^*$   
digits  $\rightarrow digit digit^*$   
fraction  $\rightarrow . digits | \epsilon$   
exponent  $\rightarrow (E (+ | - | \epsilon) digits) | \epsilon$   
number  $\rightarrow digits fraction exponent$   
relop  $\rightarrow < | > | <= | >= | = | <>$   
ws  $\rightarrow (\text{blank} | \text{tab} | \text{newline})^+$ 
```

Recognition: Transition Diagrams

relop (>, >=, <, <=, <>)



Identifier



- Diagrams are deterministic; double circle = accepting state
- * means retract the forward pointer one position
- Unlike an FA, a transition diagram has no explicit “other” symbol

Reserved words: two approaches

1. Pre-install keywords in the symbol table; `installID()` finds the tag
2. One diagram per keyword, with keywords prioritised over `id`

Key takeaways

- The lexer is called by the parser via `nextToken` and returns tokens plus attributes
- It also strips whitespace and comments
- Buffering with `begin`/`forward` pointers enables look-ahead
- Token patterns are written as regular expressions and implemented as transition diagrams

Review and exercises

1. Why can't the lexer detect `fi(a == b)` as an error?
2. Why is there a `retract` operation?
3. Does the lexer return whitespace or comments?
4. Draw a transition diagram for signed and unsigned integers
5. Write regular expressions for integer, float, identifier and whitespace

Reference: Aho, Lam, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed.