### Lexical Analysis

**Roles**
- **Primary role:** Scan a source program (a string) and break it up into small, meaningful units, called *tokens*.
  
  Example:
  ```
  position := initial + rate * 60;
  ```
  
  Transform into meaningful units: identifiers, constants, operators, and punctuation.
- **Other roles:**
  - Removal of comments
  - Case conversion
  - Removal of white spaces
  - Interpretation of compiler directives or pragmas: For instance, in Turbo Pascal `{ $R+$ } means range checking is enabled.
  - Communication with symbol table: Store information regarding an identifier in the symbol table. Not advisable in cases where scopes can be nested.
  - Preparation of output listing: Keep track of source program, line numbers, and correspondences between error messages and line numbers.

**Why separate LA from parser?**
- Simpler design of both LA and parser
- More efficient compiler
- More portable compiler

**How are patterns specified?**
- Using a meta-language, called *regular expressions*.
- Alphabet: finite set of symbols. Use term $\Sigma$ for specifying an alphabet.
- Sentence or term: string.
- Empty string: denoted $\epsilon$, string of length 0.
- Language: Any set of strings defined over an alphabet. From the lexical analyzer point of view, this language denotes the set of all tokens in programming language.

** Define following operators over sets of strings:**
1. **Union:** $L \cup U$
   
   $S = L \cup U = \{ s | (s \in L) \lor (s \in U) \}$

2. **Concatenation:** $LU$ or $LU$
   
   $S = L \cup U = \{ s \in L, t \in U \}$

3. **Kleene closure:** $L^*$, set of all strings of letters, including $\epsilon$,
   
   $S = L^* = \bigcup_{i=0}^{\infty} L^i$

4. **Positive closure:** $L^+$.
   
   $S = LL^+^

**Regular expression:** a notation for defining the set of tokens that normally occur in programming languages.

For each regular expression $r$, there is a corresponding set of strings, say $L(r)$ that is said to be derived from regular expressions. Also called *regular set*.

### Tokens, Lexemes, and Patterns

- **Token:** a certain classification of entities of a program.
  
  Four kinds of tokens in previous example: *identifiers, operators, constraints*, and **punctuation**.

- **Lexeme:** A specific instance of a token. Used to differentiate tokens. For instance, both *position* and *initial* belong to the identifier class, however each a different lexeme.

**Lexical analyzer may return a token type to the Parser, but must also keep track of “attributes” that distinguish one lexeme from another.**

#### Examples of attributes:
- **Identifiers:** string
- **Numbers:** value
- **Removal of comments**
- **Case conversion**

Attributes are used during semantic checking and code generation. They are not needed during parsing.

- **Patterns:** Rule describing how tokens are specified in a program. Needed because a language can contain infinite possible strings. They all cannot be enumerated.

**Formal mechanisms used to represent these patterns. Formalism helps in describing precisely (i) which strings belong to the language, and (ii) which do not.**

Also, form basis for developing tools that can automatically determine if a string belongs to a language.

### Regular expressions

- $\epsilon$ is a regular expression.
  
  $L(\epsilon) = \{ \epsilon \}$
  
  Note that this set is different from an empty set.

- If $a \in \Sigma$, then $a$ is a regular expression.
  
  $L(a) = \{ a \}$

- **Operators:** Assume that $r, s$ regular expressions. The following operators construct regular expressions from $r$ and $s$:
  
  1. **Choice:** $r | s$
     
    $L(r|s) = L(r) \cup L(s)$

  2. **Concatenation:** $rs$
     
    $L(rs) = L(r)L(s)$

  3. **Kleene closure:** $r^*$
     
    $L(r^*) = (L(r))^*$

Any finite set of strings can be represented by a regular expression of the form $s_1|s_2|\cdots|s_k$.

- Some additional operations for notational convenience:
  
  1. $r^+$ denoting all strings consisting of one or more $r$.
     
     $r^+ = (r^+ | \epsilon)$

  2. **Not(r):** denoting strings in set $(\Sigma^* - L(r))$

  3. $r^k$: denotes all strings formed by concatenating $k$ strings from $L(r)$.
Examples

- \(a(b|c)\) represents \(\{ab, ac\}\)
- \(a'\ b\) represents \(\{b, ab, aab, aaab, \ldots\}\)
- \(\{a|b\}\) represents any combination of \(a\) or \(b\).
- \(\{ab\}\) represents \(\{a, ab, abab, ababab, \ldots\}\)
- \(\{a|b\}|c(d)\) represents \(\{ac, ad, bc, bd\}\)
- \(0^*10^*\) represents the set of all strings over \(\{0, 1\}\) containing exactly one 1.
- Let \(d = \{0, \ldots, 9\}\), \(l = (A, \ldots, Z)\)

1. A comment that begins with -- and ends with Eol:
   ```
   Comment = -- Not(Eol)* Eol
   ```
2. A fixed decimal literal:
   ```
   Lit = d^*.d
   ```
3. An identifier, composed of letters, digits, and underscores, that begins with a letter, ends with a letter or digit, and contains no consecutive underscores:
   ```
   Id = t(l(id)|((id)d))*
   ```
4. Comment delimited by ## markers, but allow single # within the comment body:
   ```
   Comment2 = ##((#|\epsilon)Not(#))*##
   ```

Regular expressions are limited in description power. They cannot represent many of the programming language constructs. Cannot describe languages that contain strings of the form: \(\{a^n b^n\}\). This language describes balanced parentheses.

### How to recognize tokens?

- A recognizer for a language \(L\) is a program that takes a string \(x\) and answers "yes" if \(x \in L\) else "no".
- A recognizer, called finite automaton, for regular expressions can be constructed from regular expressions.
- Two classes of finite automaton: i) **Nondeterministic** and ii) **Deterministic**.
- Nondeterministic Finite Automaton (NFA): A mathematical model that consists of i) a set of states \(S\), ii) a set of input symbols \(\Sigma\), iii) a transition function that maps state-symbol pairs to sets of states, iv) a state \(s_0\) that is distinguished as the start state, and v) a set of states \(F\) distinguished as accepting or final states.
- An NFA can be represented as a labeled directed graph, called transition graph: nodes are states, and labeled edges represent transition function. Also through Transition table.
- An NFA accepts an input string \(x\) if there is some path in transition graph from start state to some accepting state.
- Language defined by NFA: set of strings it accepts.

#### Example NFA for RE \((a|b)^* abb:\)

![NFA Diagram](image)

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Deterministic Finite Automaton (DFA)

- DFA: special case of NFA where i) no \(\epsilon\)-transition, and ii) for each state \(s\), input \(a\), only one transition possible. In other words, at every state \(s\), the transition for an input \(a\) is known.
- Easy to determine if a DFA accepts a string as there is only one path.
- A nondeterministic finite automaton can be converted into a deterministic finite automaton.

DFA for \((a|b)^* abb:\)

![DFA Diagram](image)

#### Approaches to building Scanner

- Many ways in which a scanner can be built.
- Approach 1: Construct an NFA from the regular expression specification, and use an NFA simulator (ASU Alg.3.4) for simulating the generated NFA. Number of states: \(O(|r|)\), where \(|r|\) is the length of regular expression.
  - Time Cost: \(O(|r| \times |x|)\) where \(|x|\) is string length.
  - Space Requirements: \(O(|r|)\)
- Approach 2: Construct DFA from the NFA and then simulate the DFA. In this approach, there is a possibility of state explosion.
  - Time: \(O(|x|)\)
  - Space: \(O(2^{|r|})\)
- Approach 3: Lazy transition evaluation. Construct a transition as and when needed. The computed transition table is stored in a cache. Each time a transition is about to be made, the cache is consulted. If it does not exist, compute the new transition.
  - Combines the space requirements of the NFA with method with time requirement of DFA.
  - Space requirements: size of cache + \(|r|\)

Observed running time is almost as fast as that of DFA recognizer. Sometimes faster because does not compute transition tables that need not be computed.
Tools for constructing Scanners

▶ Several tools for building lexical analyzers from special purpose notation based on regular expressions.

Regular expression Scanner Generator Lexical Analyzer
String Lexical Analyzer yes/no

▶ Lex produces an entire scanner module that can be compiled and linked with other compiler modules.

Regular expression Scanner Generator Lexical Analyzer
String Lexical Analyzer yes/no

▶ Three components of a lex program:
  - declarations
  - transition rules
  - auxiliary procedures

Any part may be omitted, but separators must appear.

▶ lex generate a function yylex. Every time yylex is called, it returns a token.

Lex - cont’d.

▶ Part 1: Define pattern for a token and give it a symbolic name, so that the name can be used when referring to the token in description and action part.

```c
digs [0-9]*
integer {digs}
```

It can also contain variables and other declarations to be included in the C code generated by Lex. Usually defined in the beginning and included in `{ % and %}:

```c
int linecount = 1;
%
```

▶ Part 3: In certain cases, actions associated with tokens may be complex enough to warrant function and procedure definitions. Such procedures can be defined in the third section.

▶ Second part (transition rules): specifies how to define tokens and what actions to take when tokens are identified. For instance, return the identifier (say an integer value) for the token.

```c
Part 1 and Part 3 are mostly to enable define second part.

%{real} {return FLOAT;}
begin {return BEGIN;}
whitespace ;
```

▶ Lex maintains a set of variables to define attributes of lexemes.
  - yytext: Actual contents of the lexeme identified.
  - yyleng: Length of the lexeme.
  - yylval: used to store lexical value of token if any.
  - yylineno: number of the current input line.

How are tokens specified in Lex?

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[xz]</td>
<td>x or z</td>
</tr>
<tr>
<td>[x - z]</td>
<td>x through z</td>
</tr>
<tr>
<td>[^x]</td>
<td>Any character except x</td>
</tr>
<tr>
<td>.</td>
<td>Any character except end of line</td>
</tr>
<tr>
<td>x</td>
<td>An x at the beginning of a line</td>
</tr>
<tr>
<td>x$</td>
<td>An x at the end of a line</td>
</tr>
<tr>
<td>x?</td>
<td>Optional x</td>
</tr>
<tr>
<td>x*</td>
<td>0 or more instances of x</td>
</tr>
<tr>
<td>x+</td>
<td>1 or more instances of x</td>
</tr>
<tr>
<td>x{m,n}</td>
<td>m through n occurrences of x</td>
</tr>
<tr>
<td>x/y</td>
<td>x or y</td>
</tr>
<tr>
<td>(x)</td>
<td>same as x</td>
</tr>
<tr>
<td>x/y</td>
<td>x only if followed by y</td>
</tr>
</tbody>
</table>

Examples:

▶ Single character such as b matches b in source program.
  Special characters such as ., %, (, ) etc. have special meaning. Use "\".
  - [a-dw-z] = ab|bc|d|w|y|z
  - a(b|c) means ab or ac.
  - ab? is equivalent to a|ab and (ab)? is equivalent to ab|c.
  - [a-z]$ match a=z if at end of line
  - [a-z] match a=z if at beginning of line

Lex: Example 2

A scanner that adds line numbers to text:

```c
%(#include <stdio.h>
int lineno = 1;
%
line .*\n%
{line} {printf("%5d %s", lineno++, yytext);}%
main() {
yylex(); return 0;
}]
```

A scanner that selects only lines that end or begin with letter 'a':

```c
%(#include <stdio.h>
%
ends_with_a .*\n
begins_with_a a.*\n%
{ends_with_a} ECHO;
{begins_with_a} ECHO;
.*\n%
main() {
yylex(); return 0;
}]
```
Practical Considerations

- How to handle words such as if and while? Note that these words match lexical syntax of ordinary identifiers.
- Most languages make such words reserved. Facilitates parsing and ease of programming. If not, what does the following string really mean?
  - if if then else = then;
- Reserving such words makes regular expression complicated because reserved words are similar to identifiers.
  - Use nots
  - Directly write regular expression (very hard though)
- Simple solution: Treat reserved words as ordinary identifiers and use a separate table look up to detect them.
- Another solution: Define distinct regular expression for each reserved word. A string may match more than one regular expression. Define some mechanism for choosing one of them. In Lex, order of listing of token specifications makes a difference.

```plaintext
if {return(IF);} then {return(THEN);} [id] {return(ID);}
```

Problem: Underlying finite automaton and its transition table will be significantly larger.

---

Practical Considerations - cont’d.

Lexical Errors

- Scanner may come across certain errors: invalid character, invalid token etc. Usually detected by reaching a state that is not final and there are no transitions for the current input symbol.
- Cannot uncover syntactical, semantic or logical errors: view of the lexical analyzer is localized.
- What to do when lexical errors occur?
  - Delete the characters read so far and restart scanning at the next unread character.
  - Delete the first character read by the scanner and resume scanning at the character following it.
  - Local transformations: replace a char by another, transpose adjacent chars etc.
- Note that error recovery at this stage may create errors in the parsing stage: for instance, replace beg#in by beg in which will cause error during the parsing phase.
- Other approach could be for the scanner to provide a certain warning token to the parser. Parser can use this information to do syntactic error-repair.
- Error recovery of a common problem: runaway comments and strings. Possible solutions: Introduce error token that represent a runaway string or comment. Once the runaway error token is recognized, a special error message may be issued.