Lexical Analysis and Design of Lexical Analyzer
Lexical Analysis

- Input is scanned completely to identify the tokens
- Tokens (Logical unit)
  - Identifier, Keywords, operators etc.
Specification of Tokens

– Strings and Languages
  • Finite sequence of Symbols is called Strings
  • Set of strings over some alphabet is called Language

– Operation on Languages
  • Concatenation:
    – \( L_1 L_2 = \{ s_1 s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \} \)
  • Union
    – \( L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \} \)
  • Kleene Closure
    – \( L^* = \bigcup \{ L_i \} \)
  • Positive Closure
    – \( L^+ = \bigcup \{ L_i \} \)

– Regular Expressions
Regular Expression

• Notation for representing Tokens
• Ex: Identifiers in Pascal
  letter → A | B | ... | Z | a | b | ... | z
  digit → 0 | 1 | ... | 9
  id → letter (letter | digit ) *
The Reason Why Lexical Analysis is a Separate Phase

• Simplifies the design of the compiler
  – LL(1) or LR(1) parsing with 1 token lookahead would not be possible (multiple characters/tokens to match)

• Provides efficient implementation
  – Systematic techniques to implement lexical analyzers by hand or automatically from specifications
  – Stream buffering methods to scan input

• Improves portability
  – Non-standard symbols and alternate character encodings can be normalized (e.g. trigraphs)
Interaction of the Lexical Analyzer with the Parser

Source Program

Lexical Analyzer

Token, tokenval

Get next token

Parser

error

Symbol Table

error
Attributes of Tokens

\[
y := 31 + 28 \times x
\]

Lexical analyzer

Parser

token

tokenval
(token attribute)
Tokens, Patterns, and Lexemes

• A token is a classification of lexical units
  – For example: id and num

• Lexemes are the specific character strings that make up a token
  – For example: abc and 123

• Patterns are rules describing the set of lexemes belonging to a token
  – For example: “letter followed by letters and digits” and “non-empty sequence of digits”
Specification of Patterns for Tokens: *Definitions*

• An *alphabet* $\Sigma$ is a finite set of symbols (characters)

• A *string* $s$ is a finite sequence of symbols from $\Sigma$
  – $|s|$ denotes the length of string $s$
  – $\varepsilon$ denotes the empty string, thus $|\varepsilon| = 0$

• A *language* is a specific set of strings over some fixed alphabet $\Sigma$
Specification of Patterns for Tokens: *String Operations*

- The *concatenation* of two strings $x$ and $y$ is denoted by $xy$.
- The *exponentation* of a string $s$ is defined by:
  
  \[
  s^0 = \varepsilon \\
  s^i = s^{i-1}s \quad \text{for } i > 0
  \]

  note that $s\varepsilon = \varepsilon s = s$
Specification of Patterns for Tokens: Language Operations

• **Union**
  \[ L \cup M = \{ s \mid s \in L \text{ or } s \in M \} \]

• **Concatenation**
  \[ LM = \{ xy \mid x \in L \text{ and } y \in M \} \]

• **Exponentiation**
  \[ L^0 = \{ \varepsilon \}; \quad L^i = L^{i-1}L \]

• **Kleene closure**
  \[ L^* = \bigcup_{i=0,\ldots,\infty} L^i \]

• **Positive closure**
  \[ L^+ = \bigcup_{i=1,\ldots,\infty} L^i \]
Specification of Patterns for Tokens: *Regular Expressions*

- **Basis symbols:**
  - $\varepsilon$ is a regular expression denoting language $\{\varepsilon\}$
  - $a \in \Sigma$ is a regular expression denoting $\{a\}$

- **If** $r$ and $s$ are regular expressions denoting languages $L(r)$ and $M(s)$ respectively, then
  - $r \mid s$ is a regular expression denoting $L(r) \cup M(s)$
  - $rs$ is a regular expression denoting $L(r)M(s)$
  - $r^*$ is a regular expression denoting $L(r)^*$
  - $(r)$ is a regular expression denoting $L(r)$

- A language defined by a regular expression is called a *regular set*
Specification of Patterns for Tokens: *Regular Definitions*

- Regular definitions introduce a naming convention:
  
  \[
  \begin{align*}
  d_1 & \rightarrow r_1 \\
  d_2 & \rightarrow r_2 \\
  \vdots \\
  d_n & \rightarrow r_n
  \end{align*}
  \]

  where each \( r_i \) is a regular expression over \( \Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\} \)

- Any \( d_j \) in \( r_i \) can be textually substituted in \( r_i \) to obtain an equivalent set of definitions
Specification of Patterns for Tokens: *Regular Definitions*

- **Example:**

  
  \[
  \begin{align*}
  \text{letter} & \rightarrow A | B | \ldots | Z | a | b | \ldots | z \\
  \text{digit} & \rightarrow 0 | 1 | \ldots | 9 \\
  \text{id} & \rightarrow \text{letter} ( \text{letter} | \text{digit} )^* \\
  
  \end{align*}
  \]

- **Regular definitions are not recursive:**

  \[
  \begin{align*}
  \text{digits} & \rightarrow \text{digit digits} | \text{digit} \quad \text{wrong!}
  
  \end{align*}
  \]
Specification of Patterns for Tokens: *Notational Shorthand*

- The following shorthands are often used:

\[
\begin{align*}
  r^+ &= rr^* \\
  r? &= r | \varepsilon \\
  [a-z] &= a | b | c | \ldots | z
\end{align*}
\]

- Examples:
  - digit → [0-9]
  - num → digit+ (. digit+)\(^{+}\)? ( E (+ | -)? digit\(^{+}\) )?
Regular Definitions and Grammars

Grammar

\( stmt \rightarrow if \ expr \ then \ stmt \)
\[ \mid \ if \ expr \ then \ stmt \ else \ stmt \]
\[ \mid \ \varepsilon \]

\( expr \rightarrow term \ relop \ term \)
\[ \mid \ term \]

\( term \rightarrow id \)
\[ \mid \ num \]

\( if \rightarrow if \)
\( then \rightarrow then \)
\( else \rightarrow else \)

\( relop \rightarrow < \mid <= \mid <> \mid > \mid >= \mid = \)

\( id \rightarrow letter \ ( \ letter \mid digit \ )^* \)

\( num \rightarrow digit^+ \ ( . \ digit^+ )^? \ ( E \ (+ \mid -)^? \ digit^+ )? \)
Coding Regular Definitions in Transition Diagrams

\[
\begin{align*}
\text{relop} & \rightarrow < \mid \leq \mid <> \mid > \mid \geq \mid = \\
id & \rightarrow \text{letter} ( \text{letter} \mid \text{digit} )^* \\
\end{align*}
\]
Coding Regular Definitions in Transition Diagrams: Code

token nexttoken()
{ while (1) {
    switch (state) {
    case 0: c = nextchar();
        if (c==blank || c==tab || c==newline) {
            state = 0;
            lexeme_beginning++;
        }
        else if (c=='<') state = 1;
        else if (c=='=') state = 5;
        else if (c=='>') state = 6;
        else state = fail();
        break;
    case 1:
        ...
    case 9: c = nextchar();
        if (isletter(c)) state = 10;
        else state = fail();
        break;
    case 10: c = nextchar();
        if (isletter(c)) state = 10;
        else if (isdigit(c)) state = 10;
        else state = 11;
        break;
    ...
    }
}
Decides the next start state to check

int fail()
{ forward = token_beginning;
    switch (start) {
    case 0: start = 9; break;
    case 9: start = 12; break;
    case 12: start = 20; break;
    case 20: start = 25; break;
    case 25: recover(); break;
    default: /* error */
} 
return start;
The Lex and Flex Scanner Generators

- *Lex* and its newer cousin *flex* are scanner generators
- Systematically translate regular definitions into C source code for efficient scanning
- Generated code is easy to integrate in C applications
Creating a Lexical Analyzer with Lex and Flex

lex source program lex.l

lex or flex compiler

lex.yy.c

C compiler

lex.yy.c

a.out

input stream

a.out

sequence of tokens
Design of a Lexical Analyzer Generator

• Translate regular expressions to NFA
• Translate NFA to an efficient DFA

regular expressions → NFA → DFA

Simulate NFA to recognize tokens
Simulate DFA to recognize tokens

Optional
Nondeterministic Finite Automata

- An NFA is a 5-tuple \((S, \Sigma, \delta, s_0, F)\) where

  \(S\) is a finite set of states
  \(\Sigma\) is a finite set of symbols, the alphabet
  \(\delta\) is a mapping from \(S \times \Sigma\) to a set of states
  \(s_0 \in S\) is the start state
  \(F \subseteq S\) is the set of accepting (or final) states
Transition Graph

- An NFA can be diagrammatically represented by a labeled directed graph called a *transition graph*

\[ S = \{0,1,2,3\} \]
\[ \Sigma = \{a,b\} \]
\[ s_0 = 0 \]
\[ F = \{3\} \]
Transition Table

- The mapping $\delta$ of an NFA can be represented in a transition table.

$\delta(0,a) = \{0,1\}$
$\delta(0,b) = \{0\}$
$\delta(1,b) = \{2\}$
$\delta(2,b) = \{3\}$

<table>
<thead>
<tr>
<th>State</th>
<th>Input a</th>
<th>Input b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0, 1}</td>
<td>{0}</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>{2}</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>{3}</td>
</tr>
</tbody>
</table>
The Language Defined by an NFA

- An NFA accepts an input string $x$ if and only if there is some path with edges labeled with symbols from $x$ in sequence from the start state to some accepting state in the transition graph.
- A state transition from one state to another on the path is called a move.
- The language defined by an NFA is the set of input strings it accepts, such as $(a | b)^*abb$ for the example NFA.
Design of a Lexical Analyzer Generator: RE to NFA to DFA

Lex specification with regular expressions

\[ p_1 \mapsto \{ \text{action}_1 \} \]
\[ p_2 \mapsto \{ \text{action}_2 \} \]
\[ \ldots \]
\[ p_n \mapsto \{ \text{action}_n \} \]
From Regular Expression to NFA  
(Thompson’s Construction)
Combining the NFAs of a Set of Regular Expressions

\[ a \{ action_1 \} \]
\[ abb \{ action_2 \} \]
\[ a^*b^+ \{ action_3 \} \]
Simulating the Combined NFA

Example 1

Must find the longest match:
Continue until no further moves are possible
When last state is accepting: execute action
Simulating the Combined NFA

Example 2

When two or more accepting states are reached, the first action given in the Lex specification is executed.
Deterministic Finite Automata

- A deterministic finite automaton is a special case of an NFA
  - No state has an \( \epsilon \)-transition
  - For each state \( s \) and input symbol \( a \) there is at most one edge labeled \( a \) leaving \( s \)

- Each entry in the transition table is a single state
  - At most one path exists to accept a string
  - Simulation algorithm is simple
Example DFA

A DFA that accepts \((a \mid b)^*abb\)
Conversion of an NFA into a DFA

• The subset construction algorithm converts an NFA into a DFA using:

\[ \varepsilon\text{-closure}(s) = \{s\} \cup \{t \mid s \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} t\} \]

\[ \varepsilon\text{-closure}(T) = \bigcup_{s \in T} \varepsilon\text{-closure}(s) \]

\[ \text{move}(T,a) = \{t \mid s \xrightarrow{a} t \text{ and } s \in T\} \]

• The algorithm produces:

\( D\text{states} \) is the set of states of the new DFA consisting of sets of states of the NFA

\( D\text{tran} \) is the transition table of the new DFA
\( \varepsilon \)-closure and move Examples

\[ \varepsilon \text{-closure}(\{0\}) = \{0,1,3,7\} \]
\[ \text{move}(\{0,1,3,7\},a) = \{2,4,7\} \]
\[ \varepsilon \text{-closure}(\{2,4,7\}) = \{2,4,7\} \]
\[ \text{move}(\{2,4,7\},a) = \{7\} \]
\[ \varepsilon \text{-closure}(\{7\}) = \{7\} \]
\[ \text{move}(\{7\},b) = \{8\} \]
\[ \varepsilon \text{-closure}(\{8\}) = \{8\} \]
\[ \text{move}(\{8\},a) = \emptyset \]

Also used to simulate NFAs
Simulating an NFA using \( \varepsilon \)-closure and move

\[
S := \varepsilon\text{-closure}(\{s_0\}) \\
S_{prev} := \emptyset \\
a := \text{nextchar}() \\
\text{while } S \neq \emptyset \text{ do} \\
\quad S_{prev} := S \\
\quad S := \varepsilon\text{-closure}(\text{move}(S,a)) \\
\quad a := \text{nextchar}() \\
\text{end do} \\
\text{if } S_{prev} \cap F \neq \emptyset \text{ then} \\
\quad \text{execute action in } S_{prev} \\
\quad \text{return “yes”} \\
\text{else} \\
\quad \text{return “no”}
\]
Minimizing the Number of States of a DFA