Lexical Analysis and Lexical Analyzer Generators

Chapter 3
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. UTF8, trigraphs)
Interaction of the Lexical Analyzer with the Parser

Source Program → Lexical Analyzer → Parser

- Error
- Token, tokenval → Get next token
- Symbol Table
- Error
Attributes of Tokens

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

Lexical analyzer

Parser

\text{token} (lookahead)

\text{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 *exponentiation* of a string $s$ is defined by
  \[
  s^0 = \varepsilon \\
  s^i = s^{i-1}s \quad \text{for} \quad i > 0
  \]

  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
  - $r | s$ 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 with name-to-regular-expression bindings:
  
  \[
  d_1 \rightarrow r_1 \\
  d_2 \rightarrow r_2 \\
  \vdots \\
  d_n \rightarrow r_n 
  \]

  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 cannot be recursive:**

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

- The following shorthands are often used:

\[ r^+ = rr^* \]
\[ r? = r \mid \varepsilon \]
\[ [a-z] = a \mid b \mid c \mid \ldots \mid z \]

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

Grammar

\[
stmt \rightarrow \text{if } expr \text{ then } stmt \\
| \text{if } expr \text{ then } stmt \text{ else } stmt \\
| \epsilon
\]

\[
expr \rightarrow \text{term } \text{relop } \text{term} \\
| \text{term}
\]

\[
term \rightarrow \text{id} \\
| \text{num}
\]

Regular definitions

\[
if \rightarrow \text{if} \\
then \rightarrow \text{then} \\
else \rightarrow \text{else}
\]

\[
relop \rightarrow < | <= | <> | > | >= | =
\]

\[
id \rightarrow \text{letter ( letter } | \text{ digit })^* \\
num \rightarrow \text{digit}^+ (\text{. digit})^? (\text{ E (} + | -)? \text{ digit}^+ )?
\]
Coding Regular Definitions in Transition Diagrams

relop → < | <= | <> | > | >= | =

id → letter ( letter | digit )*
**Coding Regular Definitions in Transition Diagrams: Code**

```c
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;
}
```

Decides the next start state to check
The Lex and Flex Scanner Generators

• *Lex* and its newer cousin *flex* are *scanner generators*

• 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 (or flex) source program lex.l -> lex.yy.c

lex.yy.c -> a.out

C compiler

input stream a.out -> sequence of tokens
Lex Specification

• A lex specification consists of three parts:
  regular definitions, C declarations in % { % }
%%
  translation rules
%%
  user-defined auxiliary procedures

• The translation rules are of the form:
  \[ p_1 \{ action_1 \} \]
  \[ p_2 \{ action_2 \} \]
  \[ \ldots \]
  \[ p_n \{ action_n \} \]
Regular Expressions in Lex

- \x  match the character \x
- \.  match the character .
- "string"  match contents of string of characters
- .  match any character except newline
- ^  match beginning of a line
- $  match the end of a line
- [xyz]  match one character \x, \y, or \z (use \ to escape -)
- [^xyz] match any character except \x, \y, and \z
- [a–z] match one of \a to \z
- r*  closure (match zero or more occurrences)
- r+ positive closure (match one or more occurrences)
- r? optional (match zero or one occurrence)
- r_1r_2  match r_1 then r_2 (concatenation)
- r_1|r_2  match r_1 or r_2 (union)
- ( r )  grouping
- r_1/r_2  match r_1 when followed by r_2
- {d}  match the regular expression defined by d
Example Lex Specification 1

Translation rules

```c
{%
#include <stdio.h>
%
%
[0-9]+   { printf("%s\n", yytext); } 
.|\n   { } 
%
main()
{ yylex(); } 
%
}
```
Example Lex Specification 2

```c
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%
delim     [ \t]+  
%^{delim}  { ch+=yyleng; }
{delim}   { ch+=yyleng; wd++; }
.         { ch++; }
%
main()
{ yylex();
   printf("%8d%8d%8d\n", nl, wd, ch);
}
```
Example Lex Specification 3

```c
#include <stdio.h>

digit     [0-9]
letter    [A-Za-z]
id        {letter}({letter}|{digit})*

{digit}+ { printf("number: %s\n", yytext); }  
{id}     { printf("ident: %s\n", yytext); }   
.        { printf("other: %s\n", yytext); }   

main()
{    yylex(); }
```
Example Lex Specification 4

{% /* definitions of manifest constants */
#define LT (256)
...
%
%
delim     \[ \t\n] 
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?)?{digit}+)?
%
{ws}      { } 
if        {return IF;}
then      {return THEN;}
else      {return ELSE;}
{id}      {yylval = install_id(); return ID;}
{number}  {yylval = install_num(); return NUMBER;}
"<"      {yylval = LT; return RELOP;}
"<="     {yylval = LE; return RELOP;}
"="      {yylval = EQ; return RELOP;}
"<>"     {yylval = NE; return RELOP;}
">"      {yylval = GT; return RELOP;}
">="     {yylval = GE; return RELOP;}
%
int install_id()
...
Design of a Lexical Analyzer Generator

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

Optional
Nondeterministic Finite Automata

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

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

\[
\begin{align*}
\delta(0, a) &= \{0, 1\} \\
\delta(0, b) &= \{0\} \\
\delta(1, b) &= \{2\} \\
\delta(2, b) &= \{3\}
\end{align*}
\]

<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 $(a \mid b)^*abb$ for the example NFA.
Design of a Lexical Analyzer Generator: RE to NFA to DFA

Lex specification with regular expressions

\[ p_1 \{ \text{action}_1 \} \]
\[ p_2 \{ \text{action}_2 \} \]
\[ \ldots \]
\[ p_n \{ \text{action}_n \} \]

NFA

\[ \begin{array}{c}
\text{start} \\
\varepsilon \\
\varepsilon \\
\varepsilon
\end{array} \]

\[ \begin{array}{c}
N(p_1) \\
\text{action}_1 \\
N(p_2) \\
\text{action}_2 \\
\ldots \\
N(p_n) \\
\text{action}_n
\end{array} \]

Subset construction

DFA
From Regular Expression to NFA
(Thompson’s Construction)

\[
\begin{align*}
\varepsilon & \quad \xrightarrow{\text{start}} \quad \varepsilon & \quad \rightarrow & \quad f \\
\text{a} & \quad \xrightarrow{\text{start}} \quad i & \quad \rightarrow & \quad a & \quad \rightarrow & \quad f \\
\mid \quad & \quad \text{start} \rightarrow \quad i & \quad \rightarrow & \quad N(r_1) & \quad \rightarrow & \quad f \\
\mid \quad & \quad \text{start} \rightarrow \quad i & \quad \rightarrow & \quad N(r_2) & \quad \rightarrow & \quad f \\
\cdot \quad & \quad \text{start} \rightarrow \quad i \quad \rightarrow & \quad N(r_1) \quad \rightarrow & \quad N(r_2) & \quad \rightarrow & \quad f \\
\cdot \quad & \quad \text{start} \rightarrow \quad i & \quad \rightarrow & \quad N(r) & \quad \rightarrow & \quad f
\end{align*}
\]
Combining the NFAs of a Set of Regular Expressions

\[ \begin{align*}
\text{a} & \quad \{ \text{action}_1 \} \\
\text{abb} & \quad \{ \text{action}_2 \} \\
\text{a}^*\text{b}^+ & \quad \{ \text{action}_3 \}
\end{align*} \]
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 \( \varepsilon \)-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 \((a | b)^*abb\)
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:

*Dstates* is the set of states of the new DFA consisting of sets of states of the NFA

*Dtran* is the transition table of the new DFA
\textit{\varepsilon}\text{-}closure\ and\ move\ Examples

\begin{align*}
\varepsilon\text{-}closure\{\{0\}\} &= \{0,1,3,7\} \\
move(\{0,1,3,7\},a) &= \{2,4,7\} \\
\varepsilon\text{-}closure\{\{2,4,7\}\} &= \{2,4,7\} \\
move(\{2,4,7\},a) &= \{7\} \\
\varepsilon\text{-}closure\{\{7\}\} &= \{7\} \\
move(\{7\},b) &= \{8\} \\
\varepsilon\text{-}closure\{\{8\}\} &= \{8\} \\
move(\{8\},a) &= \emptyset
\end{align*}

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

\[
S := \varepsilon\text{-closure}({s_0})
\]
\[
S_{\text{prev}} := \emptyset
\]
\[
a := \text{nextchar}()
\]

\textbf{while} \( S \neq \emptyset \) \textbf{do}
\[
S_{\text{prev}} := S
\]
\[
S := \varepsilon\text{-closure}(\text{move}(S,a))
\]
\[
a := \text{nextchar}()
\]
\textbf{end do}

\textbf{if} \( S_{\text{prev}} \cap F \neq \emptyset \) \textbf{then}
\[
\text{execute action in } S_{\text{prev}}
\]
\[
\text{return “yes”}
\]
\textbf{else}
\[
\text{return “no”}
\]
The Subset Construction Algorithm

Initially, \( \varepsilon\text{-}closure(s_0) \) is the only state in \( Dstates \) and it is unmarked

while there is an unmarked state \( T \) in \( Dstates \) do

mark \( T \)

for each input symbol \( a \in \Sigma \) do

\( U := \varepsilon\text{-}closure(move(T,a)) \)

if \( U \) is not in \( Dstates \) then

add \( U \) as an unmarked state to \( Dstates \)

end if

\( Dtran[T,a] := U \)

end do

end do
Subset Construction Example 1

Dstates
A = \{0,1,2,4,7\}
B = \{1,2,3,4,6,7,8\}
C = \{1,2,4,5,6,7\}
D = \{1,2,4,5,6,7,9\}
E = \{1,2,4,5,6,7,10\}
Subset Construction Example 2

\[ D\text{states} \]
\[ A = \{0,1,3,7\} \]
\[ B = \{2,4,7\} \]
\[ C = \{8\} \]
\[ D = \{7\} \]
\[ E = \{5,8\} \]
\[ F = \{6,8\} \]
Minimizing the Number of States of a DFA
From Regular Expression to DFA Directly

- The “important states” of an NFA are those without an ε-transition, that is if \( \text{move}(\{s\},a) \neq \emptyset \) for some \( a \) then \( s \) is an important state
- The subset construction algorithm uses only the important states when it determines \( \varepsilon\text{-closure}(\text{move}(T,a)) \)
From Regular Expression to DFA Directly (Algorithm)

- Augment the regular expression $r$ with a special end symbol $#$ to make accepting states important: the new expression is $r#$
- Construct a syntax tree for $r#$
- Traverse the tree to construct functions `nullable`, `firstpos`, `lastpos`, and `followpos`
From Regular Expression to DFA Directly: Syntax Tree of \((a|b)^*abb#\)
nullable(n): the subtree at node n generates languages including the empty string

firstpos(n): set of positions that can match the first symbol of a string generated by the subtree at node n

lastpos(n): the set of positions that can match the last symbol of a string generated be the subtree at node n

followpos(i): the set of positions that can follow position i in the tree
# From Regular Expression to DFA Directly: Annotating the Tree

<table>
<thead>
<tr>
<th>Node $n$</th>
<th>$\text{nullable}(n)$</th>
<th>$\text{firstpos}(n)$</th>
<th>$\text{lastpos}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf $\epsilon$</td>
<td>true</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Leaf $i$</td>
<td>false</td>
<td>${i}$</td>
<td>${i}$</td>
</tr>
<tr>
<td>$</td>
<td>$ $\backslash$ $c_1$ $c_2$</td>
<td>$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$</td>
<td>$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$</td>
</tr>
<tr>
<td>$\cdot$ $\backslash$ $c_1$ $c_2$</td>
<td>$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$</td>
<td>$\text{if } \text{nullable}(c_1) \text{ then } \text{firstpos}(c_1) \cup \text{firstpos}(c_2) \text{ else } \text{firstpos}(c_1)$</td>
<td>$\text{if } \text{nullable}(c_2) \text{ then } \text{lastpos}(c_1) \cup \text{lastpos}(c_2) \text{ else } \text{lastpos}(c_2)$</td>
</tr>
<tr>
<td>$*$</td>
<td>true</td>
<td>$\text{firstpos}(c_1)$</td>
<td>$\text{lastpos}(c_1)$</td>
</tr>
</tbody>
</table>
From Regular Expression to DFA Directly: Syntax Tree of \((a|b)^*abb#\)
From Regular Expression to DFA
Directly: \textit{followpos}

\begin{verbatim}
for each node \( n \) in the tree do
    if \( n \) is a cat-node with left child \( c_1 \) and right child \( c_2 \) then
        for each \( i \) in \textit{lastpos}(\( c_1 \)) do
            \textit{followpos}(i) := \textit{followpos}(i) \cup \textit{firstpos}(c_2)
        end do
    end if
else if \( n \) is a star-node
    for each \( i \) in \textit{lastpos}(\( n \)) do
        \textit{followpos}(i) := \textit{followpos}(i) \cup \textit{firstpos}(n)
    end do
end if
end do
\end{verbatim}
From Regular Expression to DFA Directly: Algorithm

\[ s_0 := \text{firstpos}(\text{root}) \] where root is the root of the syntax tree

\[ Dstates := \{s_0\} \] and is unmarked

\textbf{while} there is an unmarked state \( T \) in \( Dstates \) \textbf{do}

\hspace{1em} mark \( T \)

\hspace{1em} \textbf{for} each input symbol \( a \in \Sigma \) \textbf{do}

\hspace{2em} let \( U \) be the set of positions that are in \( \text{followpos}(p) \) for some position \( p \) in \( T \), such that the symbol at position \( p \) is \( a \)

\hspace{2em} \textbf{if} \( U \) is not empty and not in \( Dstates \) \textbf{then}

\hspace{3em} add \( U \) as an unmarked state to \( Dstates \)

\hspace{2em} \textbf{end if}

\hspace{2em} \( Dtran[T,a] := U \)

\hspace{1em} \textbf{end do}

\hspace{1em} \textbf{end do}
From Regular Expression to DFA Directly: Example

<table>
<thead>
<tr>
<th>Node</th>
<th>followpos</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>2</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>3</td>
<td>{4}</td>
</tr>
<tr>
<td>4</td>
<td>{5}</td>
</tr>
<tr>
<td>5</td>
<td>{6}</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
</tr>
</tbody>
</table>
# Time-Space Tradeoffs

<table>
<thead>
<tr>
<th>Automaton</th>
<th>Space (worst case)</th>
<th>Time (worst case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFA</td>
<td>$O(</td>
<td>r</td>
</tr>
<tr>
<td>DFA</td>
<td>$O(2^{</td>
<td>r</td>
</tr>
</tbody>
</table>