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) with 1 lookahead would not be possible
- Provides efficient implementation
  - Systematic techniques to implement lexical analyzers by hand or automatically
  - Stream buffering methods to scan input
- Improves portability
  - Non-standard symbols and alternate character encodings can be more easily translated
Interaction of the Lexical Analyzer with the Parser

Source Program → Lexical Analyzer

Token, tokenval

Parser

error

Symbol Table

Error

Attributes of Tokens

y := 31 + 28*x

Lexical analyzer

<id, “y”>, <assign, >, <num, 31>, <+, >, <num, 28>, <*, >, <id, “x”>

token

tokenval (token attribute)

Parser
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: Terminology

• 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

- **Concatenation** of two strings $x$ and $y$ is denoted by $xy$
- **Exponentiation** of a string $s$ is defined by
  \[
  s^0 = \varepsilon \\
  s^i = s^{i-1}s \text{ for } i > 0
  \]
  (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
  - $r \cup s$ is a regular expression denoting $L(r) \cup M(s)$
  - $rs$ is a regular expression denoting $L(r)M(s)$
  - $r^n$ is a regular expression denoting $L(r)^n$
  - $(r)$ 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

- Naming convention for regular expressions:
  
  $d_1 \rightarrow r_1$
  
  $d_2 \rightarrow r_2$
  
  $\ldots$
  
  $d_n \rightarrow r_n$

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

- Each $d_j$ in $r_i$ is textually substituted in $r_i$
Specification of Patterns for Tokens: Regular Definitions

- Example:

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

- Cannot use recursion, this is illegal:

  \[
  \text{digits} \rightarrow \text{digit digits} \mid \text{digit}
  \]

Specification of Patterns for Tokens: Notational Shorthands

- We frequently use the following shorthands:

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

- For example:

  \[
  \text{digit} \rightarrow [0-9] \\
  \text{num} \rightarrow \text{digit}^+ (, \text{digit}^+) \? ( E (\pm) )? \text{digit}^+ )?
  \]


Regular Definitions and Grammars

Grammar

\[
stmt \rightarrow \text{if} \ expr \ \text{then} \ stmt \\
\quad \mid \text{if} \ expr \ \text{then} \ stmt \ \text{else} \ stmt \\
\quad \mid \varepsilon
\]

\[
expr \rightarrow \ term \ \text{relop} \ term \\
\quad \mid \ term
\]

\[
\text{term} \rightarrow \ id \\
\quad \mid \num
\]

Regular definitions

\[
\text{id} \rightarrow \text{letter} \ (\text{letter} \mid \text{digit}^+)^*
\]

\[
\text{num} \rightarrow \text{digit}^+ (\text{. digit}^+)? (\text{E} (\pm)\text{digit}^+)?
\]

Implementing a Scanner Using Transition Diagrams

\[
\text{relop} \rightarrow < | \leq | \lt | \geq | =
\]

\[\begin{align*}
\text{start} & \quad \rightarrow \quad 1 \\
1 & \quad \rightarrow \quad 2 \quad \text{return(relop, LE)} \\
1 & \quad \rightarrow \quad 3 \quad \text{return(relop, NE)} \\
1 & \quad \rightarrow \quad 4 \quad \text{return(relop, LT)} \\
1 & \quad \rightarrow \quad 5 \quad \text{return(relop, EQ)} \\
2 & \quad \rightarrow \quad 6 \quad \text{return(relop, GE)} \\
2 & \quad \rightarrow \quad 7 \quad \text{return(relop, GT)} \\
1 & \quad \rightarrow \quad 8 \quad \text{return(relop, GE)} \\
1 & \quad \rightarrow \quad 9 \quad \text{return(relop, GT)} \\
\end{align*}\]

\[
\text{id} \rightarrow \text{letter} \ (\text{letter} \mid \text{digit}^+)^*
\]

\[\begin{align*}
\text{letter or digit} & \quad \rightarrow \quad 10 \\
10 & \quad \rightarrow \quad 11 \quad \text{return(gettoken(), install_id())}
\end{align*}\]
Implementing a Scanner Using 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 what other start state is applicable

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 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:
  
  \[
  p_1 \{ \text{action}_1 \} \\
  p_2 \{ \text{action}_2 \} \\
  \ldots \\
  p_n \{ \text{action}_n \}
  \]
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 \( \\backslash \) 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
- \( r_1\backslash r_2 \) match \( r_1 \) when followed by \( r_2 \)
- \( \{d\} \) match the regular expression defined by \( d \)

Example Lex Specification 1

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

Contains the matching lexeme

Invokes the lexical analyzer

Translation rules

```
lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```
Example Lex Specification 2

```c
{%
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%
} delim [ \t]+ %

\n { ch++; wd++; nl++; }
^{delim} { ch+=yyleng; }
{delim} { ch+=yyleng; wd++; }
. { ch++; }
%
main()
{ yylex();
 printf("%8d%8d%8d\n", nl, wd, ch);
}%
```

Translation rules:
- `delim`: \s+ [\t]
- Regular definition:
  - `\n`: \n  - `^{delim}`: ch+=yyleng;
  - `{delim}`: ch+=yyleng; 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();
}%
```

Translation rules:
- `digit`: [0-9]
- `letter`: [A-Za-z]
- `id`: {letter}({letter}|{digit})*
- Regular definitions:
  - `{digit}+`: printf("number: %s\n", yytext);
  - `{id}`: printf("ident: %s\n", yytext);
  - `.`: printf("other: %s\n", yytext);
Example Lex Specification 4

```c
/* definitions of manifest constants */
#define LT (256)
...
%
#define LT (256)

delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
number {digit}+(\.{digit}+)?:${[+\-]?{digit}+}?

{ws} { }
if {return IF;}
then {return THEN;}
else {return ELSE;}
{id} {yyval = install_id(); return ID;}
{number} {yyval = install_num(); return NUMBER;}
"<" {yyval = LT; return RELOP;}
"<=" {yyval = LE; return RELOP;}
"=" {yyval = EQ; return RELOP;}
"<>" {yyval = NE; return RELOP;}
">" {yyval = GT; return RELOP;}
">=" {yyval = GE; return RELOP;}

int install_id()
...
```

Design of a Lexical Analyzer Generator

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

```
regular expressions → NFA → DFA
```

Optional

Simulate NFA to recognize tokens
Simulate DFA to recognize tokens
Nondeterministic Finite Automata

- Definition: 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 input symbol 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*

<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>

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

The Language Defined by an NFA

- An NFA *accepts* an input string $x$ *iff* 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

\[
\begin{align*}
p_1 & \{ \text{action}_1 \} \\
p_2 & \{ \text{action}_2 \} \\
\vdots \\
p_n & \{ \text{action}_n \}
\end{align*}
\]

From Regular Expression to NFA (Thompson’s Construction)

\[
\begin{align*}
\varepsilon & \quad \text{start} \xrightarrow{\varepsilon} \text{start} \\
a & \quad \text{start} \xrightarrow{a} \text{start} \\
\varepsilon \mid r_2 & \quad \text{start} \xrightarrow{\varepsilon} \text{start} \\
r_1 r_2 & \quad \text{start} \xrightarrow{\varepsilon} \text{start} \\
r_1^* & \quad \text{start} \xrightarrow{\varepsilon} \text{start}
\end{align*}
\]
Combining the NFAs of a Set of Regular Expressions

Example 1

Simulating the Combined NFA

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 ε-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:
  \[ \epsilon\text{-closure}(s) = \{s\} \cup \{t \mid s \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} t\} \]
  \[ \epsilon\text{-closure}(T) = \bigcup_{s \in T} \epsilon\text{-closure}(s) \]
  \[ move(T,a) = \{t \mid s \xrightarrow{a} t \text{ and } s \in T\} \]

- 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
**ε-closure and move Examples**

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

Also used to simulate NFAs

---

**Simulating an NFA using ε-closure and move**

\[
\begin{align*}
S &:= \varepsilon\text{-closure}\{s_0\} \\
S_{prev} &:= \emptyset \\
a &:= \text{nextchar()} \\
\text{while } S \neq \emptyset \text{ do} \\
\quad S_{prev} &:= S \\
\quad S &:= \varepsilon\text{-closure}(\text{move}(S,a)) \\
\quad a &:= \text{nextchar()} \\
\text{end do} \\
\text{if } S_{prev} \cap F \neq \emptyset \text{ then} \\
\quad \text{execute action in } S_{prev} \\
\quad \text{return} \text{ “yes”} \\
\text{else} \\
\quad \text{return} \text{ “no”}
\end{align*}
\]
The Subset Construction Algorithm

Initially, $\varepsilon$-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

$U := \varepsilon$-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 2

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 \( \text{nullable}, \text{firstpos}, \text{lastpos}, \) and \( \text{followpos} \)
From Regular Expression to DFA Directly: Syntax Tree of \((a|b)^*abb\#

From Regular Expression to DFA Directly: Annotating the Tree

- **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 by 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 $\varepsilon$</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>$/\backslash$ $c_1 , c_2$</td>
<td>$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$</td>
<td>$\text{firstpos}(c_1)$ $\cup$ $\text{firstpos}(c_2)$</td>
<td>$\text{lastpos}(c_1)$ $\cup$ $\text{lastpos}(c_2)$</td>
</tr>
<tr>
<td>$\ast$ $/\backslash$ $c_1 , c_2$</td>
<td>$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$</td>
<td>if $\text{nullable}(c_1)$ then $\text{firstpos}(c_1)$ $\cup$ $\text{firstpos}(c_2)$ else $\text{firstpos}(c_1)$</td>
<td>if $\text{nullable}(c_2)$ then $\text{lastpos}(c_1)$ $\cup$ $\text{lastpos}(c_2)$ else $\text{lastpos}(c_2)$</td>
</tr>
<tr>
<td>$/\backslash$ $c_1$</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 $(a+b)^*abb#$

[Diagram of the syntax tree]
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 lastpos\((c_1)\) do 
      followpos\((i)\) := followpos\((i)\) \( \cup \) firstpos\((c_2)\) 
    end do 
  else if \( n \) is a star-node 
    for each \( i \) in lastpos\((n)\) do 
      followpos\((i)\) := followpos\((i)\) \( \cup \) firstpos\((n)\) 
    end do 
  end if 
end do
\end{verbatim}

From Regular Expression to DFA Directly: Algorithm

\begin{verbatim}
s_0 := firstpos(root) where root is the root of the syntax tree 
Dstates := \{s_0\} and is unmarked 
while there is an unmarked state \( T \) in Dstates do 
  mark \( T \) 
  for each input symbol \( a \in \Sigma \) do 
    let \( U \) be the set of positions that are in followpos\((p)\) for some position \( p \) in \( T \), 
    such that the symbol at position \( p \) is \( a \) 
    if \( U \) is not empty and not in Dstates then 
      add \( U \) as an unmarked state to Dstates 
    end if 
    Dtran\([T,a]\) := U 
  end do 
end do
\end{verbatim}
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>