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. trigraphs)

Interaction of the Lexical Analyzer with the Parser
Attributes of Tokens

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

Lexical analyzer

\[ \langle \text{id}, \text{y} \rangle \langle \text{assign,} \rangle \langle \text{num,} \rangle \langle 31 \rangle \langle + \rangle \langle \text{num,} \rangle \langle 28 \rangle \langle \times \rangle \langle \text{id,} \rangle \langle x \rangle \]

token

\[ \text{tokenval} \]
(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: 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 \( s \)
  - \( \epsilon \) denotes the empty string, thus \( |\epsilon| = 0 \)
- 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
  
  \[
  s^0 = \epsilon \\
  s^i = s^{i-1}s \quad \text{for } i > 0
  \]
  
  note that $s\epsilon = \epsilon s = s$

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
  
  \[L^0 = \{ \epsilon \}; L^i = L^{i-1}L \]

- Kleene closure
  
  \[L^* = \bigcup_{i=0}^{\infty} L^i\]

- Positive closure
  
  \[L^+ = \bigcup_{i=1}^{\infty} L^i\]

Specification of Patterns for Tokens: Regular Expressions

- Basis symbols:
  
  - $\epsilon$ is a regular expression denoting language $\{ \epsilon \}$
  - $a \in \Sigma$ 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 $L(r)^*$
  - $r^\ast$ 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:
  
  \[
  d_1 \rightarrow r_1 \\
  d_2 \rightarrow r_2 \\
  \vdots \\
  d_n \rightarrow r_n
  \]

  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 are not 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:

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

- Examples:

  \[
  \begin{align*}
  \text{digit} & \rightarrow [0-9] \\
  \text{num} & \rightarrow \text{digit}^* (\text{. digit}^*)? (E (+ | -)? \text{digit}^*)?
  \end{align*}
  \]
Regular Definitions and Grammars

Grammar

\[ \text{stmt} \rightarrow \text{if expr then stmt} \]

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

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

\[ \text{id} \rightarrow \text{letter (letter | digit) *} \]

\[ \text{num} \rightarrow \text{digit (+ (digit) + (E (+ -) digit) +?)}} \]

\[ \text{relop} \rightarrow < | <= | <> | > | >= | = \]

\[ \text{stmt} \rightarrow \text{if expr then stmt} \]

\[ \text{else stmt} \]

\[ \epsilon \]

expr → term relop term

num → digit (. digit + (E (+ -) digit) +?)

id → letter ( letter | digit ) *

relop → < | <= | <> | > | >= | =

Grammar

Regular definitions

if → if
then → then
else → else
relop → < | <= | <> | > | >= | =

id → letter ( letter | digit ) *

num → digit (. digit + (E (+ -) digit) +?)

Coding Regular Definitions in Transition Diagrams

rellop → < | <= | <> | > | >= | =

start

\[
\begin{array}{cc}
\rightarrow & \text{return rellop, LE) return rellop, NE) return rellop, LT) return rellop, EQ) return rellop, GE) return rellop, GT)} \\
\rightarrow & \text{other return (relop, LE) return (relop, NE) return (relop, LT) return (relop, EQ) return (relop, GE) return (relop, GT) return gettoken(), install_id())}
\end{array}
\]

Coding Regular Definitions in Transition Diagrams: Code

Decides the next start state to check

\[
\text{前进 = token_ beginnings;}
\text{while (1) { switch (state) { case 0: if (c == blank || c == tab || c == newline) { state = 0; lexeme_beginning++; } else if (c == '<') state = 1; else if (c == '=') state = 5; else if (c == '>') state = 6; else state = fail(); break; case 1: c = nextchar(); if (isletter(c)) state = 10; else state = fail(); break; case 9: c = nextchar(); if (isletter(c)) state = 10; else state = 9; break; case 12: c = nextchar(); if (isletter(c)) state = 10; else state = fail(); break; case 11: state = 11; break; default: case 0: state = 0; break; case 9: state = 9; break; case 12: state = 12; break; case 11: state = 11; break; default: state = fail(); break; }} return state; return;} return state;}
\]
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 \ 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)
r1 | r2 match r1 or r2 (union)
(r) grouping
r*;r2 match r1 when followed by r2
{d} match the regular expression defined by d

Example Lex Specification 1

```
#include <stdio.h>

main()
{
    yylex();
}
```

Example Lex Specification 2

```
#include <stdio.h>

main()
{
    yylex();
}
```
Example Lex Specification 3

```c
#include <stdio.h>

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

{digit}+    {printf("number: %s", yytext);}
{id}        {printf("ident: %s", yytext);}
%         
main()    { yylex(); }
```

Example Lex Specification 4

```c
/* 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 = LT; return RELOP;}
"<"      {yylval = LT; return RELOP;}
"<"      {yylval = LT; return RELOP;}
"<"      {yylval = LT; return RELOP;}
```

Design of a Lexical Analyzer Generator

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

\[
\begin{array}{c}
\text{start} \\
\downarrow b \\
\Rightarrow a \\
\Rightarrow \quad \Rightarrow b \\
\Rightarrow \quad \Rightarrow \quad \Rightarrow 0 \\
\end{array}
\]

$S = \{0,1,2,3\}$

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

$s_0 = 0$

$F = \{3\}$

Transition Table

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

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

$p_1 \quad \{\text{action}_1\}$
$p_2 \quad \{\text{action}_2\}$
$\ldots$
$p_n \quad \{\text{action}_n\}$

NFA

DFA

From Regular Expression to NFA (Thompson’s Construction)

$E$
$a$
$r_1 | r_2$
$r_1 r_2$
$r^*$
Combining the NFAs of a Set of Regular Expressions

Simulating the Combined NFA Example 1

Simulating the Combined NFA Example 2

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

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:
  \[ e\text{-closure}(s) = \{s\} \cup \{t \mid s \xrightarrow{e} \ldots \xrightarrow{e} t\} \]
  \[ e\text{-closure}(T) = \bigcup_{s \in T} e\text{-closure}(s) \]
  \[ \text{move}(T,a) = \{t \mid s \xrightarrow{a} t \text{ and } s \in T\} \]
- 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
The Subset Construction Algorithm

Initially, ε-closure(s₀) 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 ∈ Σ do
U := ε-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

ε-closure and move Examples

ε-closure(∅) = {0,1,3,7}
move({0,1,3,7},a) = {2,4,7}
ε-closure(2,4,7)) = {2,4,7}
move(2,4,7),a) = {7}
ε-closure(7)) = {7}
move(7),b) = {8}
ε-closure(8)) = {8}
move(8),a) = ∅

Simulating an NFA using ε-closure and move

S := ε-closure(s₀)
Sprev := ∅
a := nextchar()
while S ≠ ∅ do
Sprev := S
S := ε-closure(move(S,a))
a := nextchar()
end do
if Sprev ∩ F ≠ ∅ then
execute action in Sprev
return "yes"
else
return "no"
end if

Also used to simulate NFAs
Subset Construction Example 1

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 \( \text{nullable}, \text{firstpos}, \text{lastpos}, \text{and followpos} \)

From Regular Expression to DFA Directly: Syntax Tree of \( (ab)^*abb# \)

concatenation

\( a \)

\( b \)

\( # \)

\( \varepsilon \text{-closure} \)

\( \text{firstpos} \)

\( \text{lastpos} \)

\( \text{followpos} \)

\( \text{position number} \) (for leafs #)
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

<table>
<thead>
<tr>
<th>Node n</th>
<th>nullable(n)</th>
<th>firstpos(n)</th>
<th>lastpos(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf ε</td>
<td>true</td>
<td>{0}</td>
<td>{0}</td>
</tr>
<tr>
<td>Leaf a</td>
<td>false</td>
<td>{1}</td>
<td>{1}</td>
</tr>
<tr>
<td>e₁ / e₂</td>
<td>nullable(e₁) or nullable(e₂)</td>
<td>firstpos(e₁) ∪ firstpos(e₂)</td>
<td>lastpos(e₁) ∪ lastpos(e₂)</td>
</tr>
<tr>
<td>e₁ / * e₂</td>
<td>nullable(e₁) and nullable(e₂)</td>
<td>if nullable(e₁) then firstpos(e₁) ∪ lastpos(e₁) else firstpos(e₂)</td>
<td>if nullable(e₂) then firstpos(e₂) ∪ lastpos(e₂) else lastpos(e₁)</td>
</tr>
<tr>
<td>e₁ / e₂</td>
<td>true</td>
<td>firstpos(e₁)</td>
<td>lastpos(e₂)</td>
</tr>
</tbody>
</table>

From Regular Expression to DFA Directly: Syntax Tree of (ab)*abb#
From Regular Expression to DFA

Directly: **followpos**

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

From Regular Expression to DFA

Directly: **Algorithm**

\( s_0 := \text{firstpos(root)} \) where \( \text{root} \) 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

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>

[Diagram of DFA]
## 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^n)$</td>
<td>$O(</td>
</tr>
</tbody>
</table>