Lex: A Lexical Analyser Generator
Compiler-Construction Tools

The compiler writer uses specialised tools (in addition to those normally used for software development) that produce components that can easily be integrated in the compiler and help implement various phases of a compiler.

- **Scanner generators**
- **Parser generators**
- **Syntax-directed translation**
- **Code-generator generators**
- **Data-flow analysis**: key part of code optimisation
Problem:

Write a piece of code that examines the input string and find a prefix that is a lexeme matching one of the patterns for all the needed tokens.
A Simple Example

Example

Consider the following grammar:

\[
\begin{align*}
stmt & \rightarrow \text{if expr then stmt} \\
& \quad | \text{if expr then stmt else stmt} \\
& \quad | \epsilon \\
expr & \rightarrow \text{term relop term} \\
& \quad | \text{term} \\
term & \rightarrow \text{id} \\
& \quad | \text{number}
\end{align*}
\]

Figure 3.10: A grammar for branching statements
<table>
<thead>
<tr>
<th><strong>Expression</strong></th>
<th><strong>Matches</strong></th>
<th><strong>Example</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>the one non-operator character $c$</td>
<td>a</td>
</tr>
<tr>
<td>$\backslash c$</td>
<td>character $c$ literally</td>
<td>$\backslash*$</td>
</tr>
<tr>
<td>&quot;$s&quot;&quot;</td>
<td>string $s$ literally</td>
<td>&quot;$**&quot;</td>
</tr>
<tr>
<td>.</td>
<td>any character but newline</td>
<td>a.*b</td>
</tr>
<tr>
<td>^</td>
<td>beginning of a line</td>
<td>^abc</td>
</tr>
<tr>
<td>$$</td>
<td>end of a line</td>
<td>abc$</td>
</tr>
<tr>
<td>[s]</td>
<td>any one of the characters in string $s$</td>
<td>[abc]</td>
</tr>
<tr>
<td>[^s]</td>
<td>any one character not in string $s$</td>
<td>[^abc]</td>
</tr>
<tr>
<td>r*</td>
<td>zero or more strings matching $r$</td>
<td>a*</td>
</tr>
<tr>
<td>r+</td>
<td>one or more strings matching $r$</td>
<td>a+</td>
</tr>
<tr>
<td>r?</td>
<td>zero or one $r$</td>
<td>a?</td>
</tr>
<tr>
<td>r{m,n}</td>
<td>between $m$ and $n$ occurrences of $r$</td>
<td>a{1,5}</td>
</tr>
<tr>
<td>$r_1 r_2$</td>
<td>an $r_1$ followed by an $r_2$</td>
<td>ab</td>
</tr>
<tr>
<td>$r_1</td>
<td>r_2$</td>
<td>an $r_1$ or an $r_2$</td>
</tr>
<tr>
<td>(r)</td>
<td>same as $r$</td>
<td>(a</td>
</tr>
<tr>
<td>$r_1/r_2$</td>
<td>$r_1$ when followed by $r_2$</td>
<td>abc/123</td>
</tr>
</tbody>
</table>

Figure 3.8: Lex regular expressions
Regular Definitions for the Language Tokens

\[
\begin{align*}
    \text{digit} & \rightarrow [0-9] \\
    \text{digits} & \rightarrow \text{digit}^+ \\
    \text{number} & \rightarrow \text{digits} (\ . \ \text{digits})? \ (E \ [+-]? \ \text{digits})? \\
    \text{letter} & \rightarrow [A-Za-z] \\
    \text{id} & \rightarrow \text{letter} (\ \text{letter} \ | \ \text{digit} )^* \\
    \text{if} & \rightarrow \text{if} \\
    \text{then} & \rightarrow \text{then} \\
    \text{else} & \rightarrow \text{else} \\
    \text{relop} & \rightarrow < \ | \ > \ | \ <= \ | \ >= \ | \ = \ | \ <> \\
\end{align*}
\]

Figure 3.11: Patterns for tokens of Example 3.8

Note that keywords \textbf{if}, \textbf{then}, \textbf{else}, also match the patterns for \textbf{relop}, \textbf{id} and \textbf{number}.
Assumption: consider keywords as ‘reserved words’.
### Tokens Table

<table>
<thead>
<tr>
<th>Lexemes</th>
<th>Token Name</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any ws</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>if</td>
<td>if</td>
<td>–</td>
</tr>
<tr>
<td>then</td>
<td>then</td>
<td>–</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
<td>–</td>
</tr>
<tr>
<td>Any id</td>
<td>id</td>
<td>Pointer to table entry</td>
</tr>
<tr>
<td>Any number</td>
<td>number</td>
<td>Pointer to table entry</td>
</tr>
<tr>
<td>&lt;</td>
<td>relop</td>
<td>LT</td>
</tr>
<tr>
<td>&lt;=</td>
<td>relop</td>
<td>LE</td>
</tr>
<tr>
<td>=</td>
<td>relop</td>
<td>EQ</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>relop</td>
<td>NE</td>
</tr>
<tr>
<td>&gt;</td>
<td>relop</td>
<td>GT</td>
</tr>
<tr>
<td>&gt;=</td>
<td>relop</td>
<td>GE</td>
</tr>
</tbody>
</table>

Figure 3.12: Tokens, their patterns, and attribute values
Whitespace

The LA also recognises the ‘token’ $ws$ defined by:

$$ws \rightarrow (\text{blank}|\text{tab}|\text{newline})$$

This token will not be returned to the parser; the LA will restart from the next character.
Recogniser for `relop`

Figure 3.13: Transition diagram for `relop`
An Implementation

```java
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
        or failure occurs */
        switch(state) {
        case 0: c = nextChar();
            if ( c == '<' ) state = 1;
            else if ( c == '=' ) state = 5;
            else if ( c == '>' ) state = 6;
            else fail(); /* lexeme is not a relop */
            break;
        case 1: ...
            ...
        case 8: retract();
            retToken.attribute = GT;
            return(retToken);
        }
    }
}
```

Figure 3.18: Sketch of implementation of relop transition diagram
Recogniser for \textbf{id}

Figure 3.14: A transition diagram for \textbf{id}'s and keywords
Recogniser for number

Figure 3.16: A transition diagram for unsigned numbers
Figure 3.17: A transition diagram for whitespace
The **Lex compiler** is a tool that allows one to specify a lexical analyser from regular expressions. Inputs are specified in the **Lex language**.

A **Lex program** consists of declarations %% translation rules %% auxiliary functions.
Example

{%
    /* definitions of manifest constants */
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+({digit}+)?(E[+-]?{digit}+)?

%

{ws}    { /* no action and no return */}
if      { return(IF); }
then    { return(THEN); }
else    { return(ELSE); }
{id}    { yylval = (int) installID(); return(ID); }
{number} { yylval = (int) installNum(); 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); }
%}
Example (ctd.)

```c

int installID() { /* function to install the lexeme, whose 
    first character is pointed to by yytext, 
    and whose length is yyleng, into the 
    symbol table and return a pointer 
    thereto */
}

int installNum() { /* similar to installID, but puts numer- 
    ical constants into a separate table */
}

```

Figure 3.23: Lex program for the tokens of Fig. 3.12
Design of a LA Generator

Two approaches:
- NFA-based
- DFA-based

The *Lex compiler* is implemented using the second approach.
Figure 3.49: A Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator.
We show how to convert regular expressions to NFA’s and a conversion of NFA’s into DFA’s. The NFA produced can then be transformed into a DFA by means of the latter construction if desired. It can also be used directly via a simulation following the subset construction translation algorithm.
Constructing the Automaton

For each regular expression in the *Lex program* construct a NFA.

Figure 3.50: An NFA constructed from a *Lex* program
Simulating NFA’s

Example:

Figure 3.51: NFA’s for a, abb, and a*b+
Pattern Matching

Figure 3.52: Combined NFA

Figure 3.53: Sequence of sets of states entered when processing input aaba
LA Based on DFA’s

Figure 3.54: Transition graph for DFA handling the patterns $a$, $abb$, and $a^*b^+$