`Lex`: A Lexical Analyser Generator

Constructing a Lexical Analyser

- Problem:
- Write a piece of code that examines the input string and find the a prefix that is a *lexeme* matching one of the *patterns* for all the needed *tokens*.

A Simple Example

Example

Consider the following grammar:

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
 & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\
 & | & \epsilon \\
\mathit{expr} & \rightarrow & \mathit{term} \textbf{ relop } \mathit{term} \\
 & | & \mathit{term} \\
\mathit{term} & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

Figure 3.10: A grammar for branching statements

Regular Definitions for the Language Tokens

$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & [\texttt{0-9}] \\
\mathit{digits} & \rightarrow & \mathit{digit}^{+} \\
\mathit{number} & \rightarrow & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ [\texttt{+-}]?\ \mathit{digits}\ )? \\
\mathit{letter} & \rightarrow & [\texttt{A-Za-z}] \\
\mathit{id} & \rightarrow & \mathit{letter}\ (\ \mathit{letter} \mid \mathit{digit}\ )^{*} \\
\mathit{if} & \rightarrow & \texttt{if} \\
\mathit{then} & \rightarrow & \texttt{then} \\
\mathit{else} & \rightarrow & \texttt{else} \\
\mathit{relop} & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

Figure 3.11: Patterns for tokens of Example 3.8

Note that keywords **if**, **then**, **else**, also match the patterns for *relop*, *id* and *number*.
Assumption: consider keywords as ‘reserved words’.

Tokens Table

Lexemes	Token Name	Attribute Value
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Figure 3.12: Tokens, their patterns, and attribute values

Whitespace

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

$$\mathit{ws} \to (\mathbf{blank}|\mathbf{tab}|\mathbf{newline})$$

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

Recogniser for **relop**

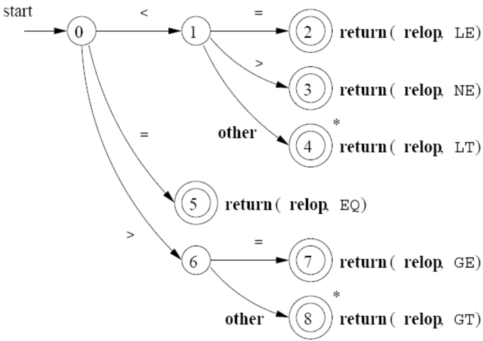

Figure 3.13: Transition diagram for **relop**

An Implementation

```
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 **id**

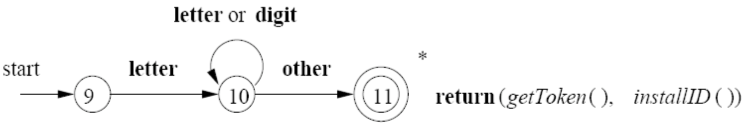

Figure 3.14: A transition diagram for **id**'s and keywords

Recogniser for **number**

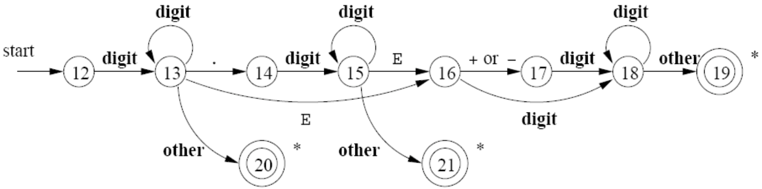

Figure 3.16: A transition diagram for unsigned numbers

Recogniser for whitespace

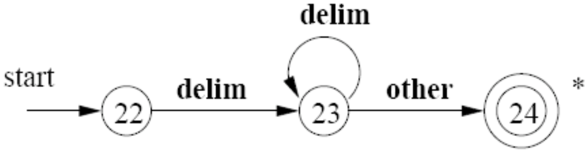

Figure 3.17: A transition diagram for whitespace

Lex

The *Lex compiler* is a tool that allows one to specify a lexical analyser from regular expressions.
Inputs are specified in the *Lex language*.

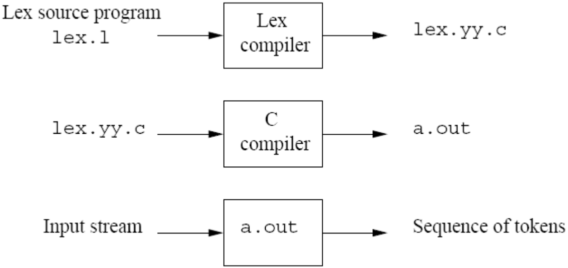

Figure 3.22: Creating a lexical analyzer with Lex

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.)

```
%%

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.

Generated LA

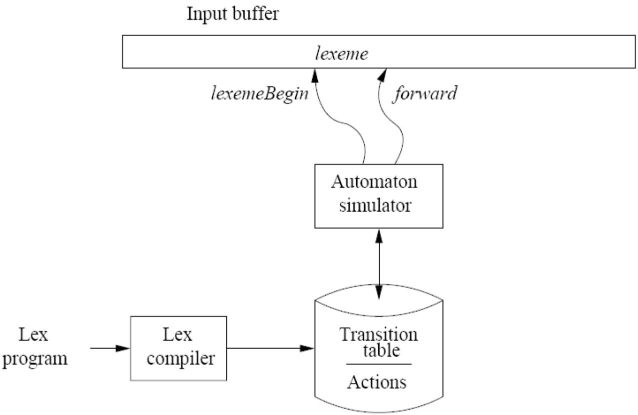

Figure 3.49: A Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator

Constructing the Automaton

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

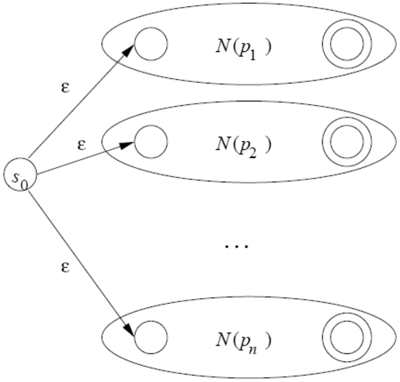

Figure 3.50: An NFA constructed from a Lex program

Simulating NFA's

Example:

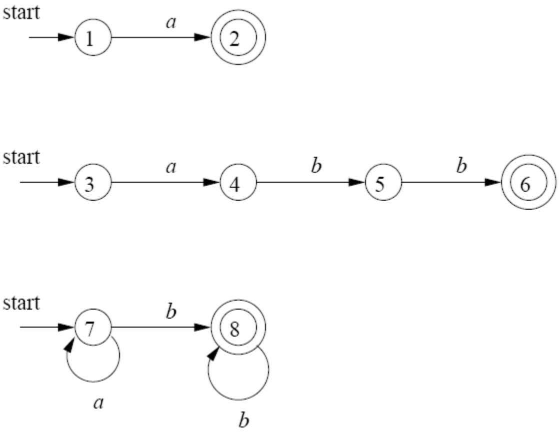

Figure 3.51: NFA's for $\mathbf{a}$, $\mathbf{abb}$, and $\mathbf{a}^*\mathbf{b}^+$

Pattern Matching

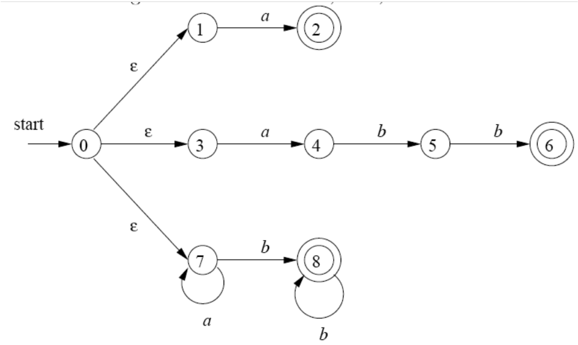

Figure 3.52: Combined NFA

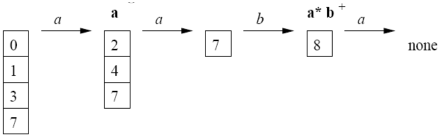

Figure 3.53: Sequence of sets of states entered when processing input *aaba*

LA Based on DFA's

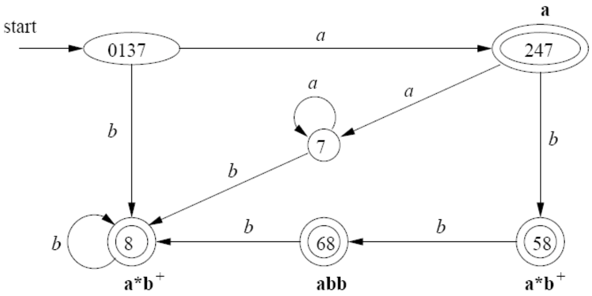

Figure 3.54: Transition graph for DFA handling the patterns **a**, **abb**, and $\mathbf{a}^*\mathbf{b}^+$