

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

Constructing a Lexical Analyser

- 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:

$stmt$	\rightarrow	if $expr$ then $stmt$
		if $expr$ then $stmt$ else $stmt$
		ϵ
$expr$	\rightarrow	$term$ relop $term$
		$term$
$term$	\rightarrow	id
		number

Figure 3.10: A grammar for branching statements

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	<code>a</code>
$\backslash c$	character c literally	<code>*</code>
<code>"s"</code>	string s literally	<code>***</code>
$.$	any character but newline	<code>a.*b</code>
\wedge	beginning of a line	<code>^abc</code>
$\$$	end of a line	<code>abc\$</code>
$[s]$	any one of the characters in string s	<code>[abc]</code>
$[\wedge s]$	any one character not in string s	<code>[^abc]</code>
r^*	zero or more strings matching r	<code>a*</code>
r^+	one or more strings matching r	<code>a+</code>
$r^?$	zero or one r	<code>a?</code>
$r\{m, n\}$	between m and n occurrences of r	<code>a\{1,5\}</code>
$r_1 r_2$	an r_1 followed by an r_2	<code>ab</code>
$r_1 \mid r_2$	an r_1 or an r_2	<code>a b</code>
(r)	same as r	<code>(a b)</code>
r_1/r_2	r_1 when followed by r_2	<code>abc/123</code>

Figure 3.8: Lex regular expressions

Regular Definitions for the Language Tokens

<i>digit</i>	→	[0-9]
<i>digits</i>	→	<i>digit</i> ⁺
<i>number</i>	→	<i>digits</i> (. <i>digits</i>)? (E [+-]? <i>digits</i>)?
<i>letter</i>	→	[A-Za-z]
<i>id</i>	→	<i>letter</i> (<i>letter</i> <i>digit</i>)*
<i>if</i>	→	if
<i>then</i>	→	then
<i>else</i>	→	else
<i>relop</i>	→	< > <= >= = <>

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 <i>ws</i>	–	–
if	if	–
then	then	–
else	else	–
Any <i>id</i>	id	Pointer to table entry
Any <i>number</i>	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:

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

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

Recogniser for **relop**

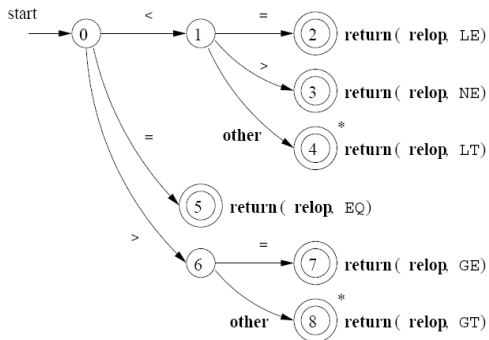


Figure 3.13: Transition diagram for **relop**

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**

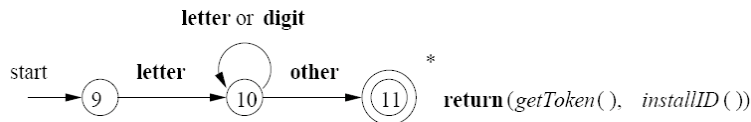


Figure 3.14: A transition diagram for **id**'s and keywords

Recogniser for **number**

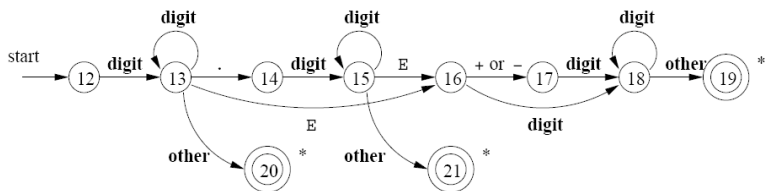


Figure 3.16: A transition diagram for unsigned numbers

Recogniser for whitespace

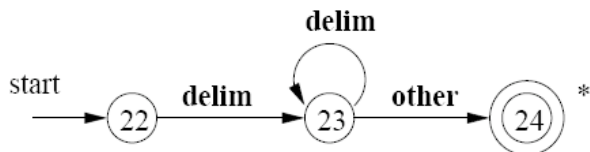


Figure 3.17: A transition diagram for whitespace

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*.

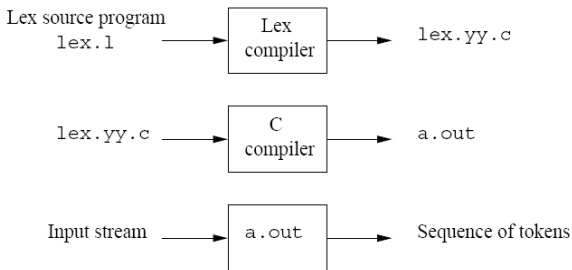


Figure 3.22: Creating a lexical analyzer with Lex

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}       {yyval = (int) installID(); return(ID);}
{number}   {yyval = (int) installNum(); 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);}
```

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

Esercizi

- Modifica il programma Lex di Fig. 3.23 in modo da aggiungere la parola chiave **while** e permettere l'uso del simbolo “_” (underscore) come lettera aggiuntiva.
- Scrivere un programma Lex che copia un file sostituendo ogni sequenza non vuota di spazi bianchi con un singolo carattere vuoto (blank).
- Scrivere un programma Lex che copia un programma C sostituendo ogni istanza della parola chiave `float` con `double`.
- Scrivere un programma Lex che stampa tutti i tag HTML presenti in un file (per default Lex legge dallo standard input).
- Scrivere un programma Lex che converte un file trasformando ogni parola nel seguente modo:
 - se la prima lettera è una consonante, allora questa viene spostata alla fine e viene aggiunto 'ay';
 - se la prima lettera è una vocale, allora si riscrive la stessa parola aggiungendo alla fine 'ay' .