Dispensa LEX

Lex: architettura software

Il tool Lex è uno strumento software per la generazione automatica di analizzatori lessicali a partire da una specifica input scritta in un'appropriata sintassi e basata su espressioni regolari. Grazie alla presenza di "azioni" (cioè frammenti di programmi scritti in linguaggio C) accanto alle espressioni regolari, quello che Lex genera è un traduttore input/output che oltre a fare l'analisi lessicale di costrutti input ne fa anche la traslazione in un determinato output come specificato dalle azioni.

L'architettura software che mostra il contesto d'uso di Lex è la seguente. Nell'architettura i nomi dei files generati (*lex.yy.c* e *a.out*) sono dipendenti da sistema, per esempio in questo caso si è considerato il sistema Unix.

<nome_file>.l ⇒ Lex ⇒ lex.yy.c

lex.yy.c ⇒ Compilatore C ⇒ a.out

input ⇒ a.out ⇒ output (sequenza di token)

Il formato generale di una specifica sorgente Lex è costituito da tre sezioni ed ha la seguente struttura:

```
DICHIARAZIONI
%%
REGOLE DI TRASLAZIONE
%%
PROCEDURE AUSILIARIE
```

La sezione "REGOLE DI TRASLAZIONE" è il cuore della specifica Lex in quanto è in essa che sono scritte le espressioni regolari. I simboli speciali %% ne delimitano l'inizio e la fine (il primo è obbligatorio, il secondo è opzionale e si omette nel caso la terza sezione sia vuota). Le sezioni "DICHIARAZIONI" e "PROCEDURE AUSILIARIE" contengono rispettivamente tutte le dichiarazioni (delimitate dai simboli speciali %{ e %}) e le routines utili alle Regole. La sezione Regole di Traslazione è strutturata come una tabella:

p_1	{action_1}
p_2	{action_2}
	……
p_n	{action_n}

dove ogni p_i è un'espressione regolare scritta in sintassi Lex che descrive un pattern per un token del linguaggio sorgente, e ogni action_i è un'azione, ossia un frammento di programma in codice C. Se Lex è usato nel contesto analisi lessicale – parsing, le azioni conterranno sostanzialmente delle istruzioni di `return` con le quali l'analizzatore lessicale generato passerà al parser i token individuati. Più in generale, il ruolo del programma generato da Lex, ossia del traduttore input/output finale, può essere così formalizzato:

"il programma finale generato da Lex legge il suo rimanente input un carattere alla volta da sinistra a destra finchè trova il più lungo prefisso di input (quello sarà il corrente lessema) che fa match con uno dei pattern p_i, allora esegue $action_i$"

In questo processo si possono presentare due tipologie di conflitti che Lex risolve grazie a due regole (euristiche) di cui è fornito:

1) Se c'è più di un match, viene preferito il più lungo lessema che fa match con qualche pattern
2) Se ci sono due o più pattern che fanno match col più lungo lessema, viene preferito il pattern listato più in alto nella sezione delle Regole di Traslazione.

Lex: sintassi

http://dinosaur.compilertools.net/lex/index.html (Appendice A)

Lex: esempi di specifiche

Esercizio1:
Si consideri il token PIPPO definito dal seguente pattern: "PIPPO è una sequenza non vuota di vocali che può iniziare oppure no con una cifra". Scrivere un programma Lex per generare un traduttore input/output che riconosca i lessemi di PIPPO e ne stampi la prima vocale della sequenza.

Possibile soluzione:

```
%{#include <stdio.h> %}
%%
[0-9]?[AEIOUaeiou]+              { if isdigit(yytext[0]) printf ("%c", yytext[1]);
                                  else printf ("%c", yytext[0]); }
```

Esercizio2:
Scrivere un programma Lex contenente la coppia *p action* dove:
- *p* è un pattern che ha come lessemi sequenze (possibilmente vuote) di cifre seguite da sequenze (non vuote) di lettere con la condizione che queste sequenze sono separate dal carattere '?'
- *action* è un'azione che stampa in output la prima lettera che compare nei lessemi input

Possibile soluzione:

```
%{#include <stdio.h>
    int i;  %}
%%
[0-9]*"?"[A-Za-z]+  { i=0;
                      while(yytext[i]!='?')
                     i=i+1;
                       printf ("%c", yytext[i+1]); }
```

Appendice A

9.2. `lex` Regular Expressions

The definitions of regular expressions are very similar to those in the editors *ex*(1) and *vi*(1). A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string `integer` wherever it appears and the expression

```
a57D
```

looks for the string `a57D`.

Operators

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape must be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string `xyz++` when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the programmer can avoid remembering the list above of current operator characters, and is safe should further extensions to `lex` lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character Classes

Classes of characters can be specified using the operator pair [ ]. The construction `[abc]` matches a single character, which may be `a`, `b`, or `c`. Within square brackets, most operator meanings are ignored. Only three characters are special: \, –, and ^. The – character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using – between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation-dependent and generates a warning message. For example, `[0-z]` in ASCII is many more characters than it is in EBCDIC. If it is desired to include the character – in a character class, it should be first or last, thus:

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the system's character set. Thus

```
[^abc]
```

matches all characters except a, b, or c, including all special or control characters; and

```
[^a-zA-Z]
```

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary Character

To match almost any character, the operator character

.

(period) is the class of all characters except newline. Escaping into octal is possible although non-portable:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional Expressions

The operator ? indicates an optional element of an expression. Thus

```
ab?c
```

matches either ac or abc.

Repeated Expressions

Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive a characters, including zero; while

```
a+
```

is one or more instances of a. For example,

```
[a-z]+
```

is all strings of lower case letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping

The operator | indicates alternation:

```
(ab | cd)
```

matches either ab or cd. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab | cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab | cd+)?(ef)*
```

matches such strings as abefef, efefef, cdef, or cddd; but not abc, abcd, or abcdef.

Context Sensitivity

lex recognizes a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression is only be matched at the beginning of a line This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [ ] operators. If the very last character is $, the expression is only be matched at the end of a line (when immediately followed by newline).

The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string `ab`, but only if it is followed by `cd`. Thus

```
ab$
```

is the same as

```
ab/\n.
```

Left context is handled in `lex` by *start conditions* as explained in section 9.9 — *Left Context-Sensitivity*. If a rule is only to be executed when the `lex` automaton interpreter is in start condition `x`, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered 'being at the beginning of a line' to be start condition `ONE`, then the ^ operator would be equivalent to

```
<ONE>.
```

Start conditions are explained more fully below.

Repetitions and Definitions

The operators { } specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named `digit` and inserts it at that point in the expression. The definitions are given in the first part of the `lex` input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of `a`.

Finally, initial `%` is special, being the separator for `lex` source segments.

9.3. `lex` Actions

When an expression written as above is matched, `lex` executes the corresponding action. This section describes some features of `lex` which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the `lex` programmer who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When `lex` is being used with `yacc`, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ; as an action does this. A frequent rule is

```
[ \t\n] ;
```

which ignores the three spacing characters (blank, tab, and newline).

Another easy way to avoid writing actions is the action character |, which indicates that the action to be used for this rule is the action given for the next rule. The previous example could also have been written

```
" "             |
"\t"            |
"\n"            ;
```

with the same result. The quotes around \n and \t are not required.

Actual Text that Matched

In more complex actions, the programmer often wants to know the actual text that matched some expression like [a-z]+. lex leaves this text in an external character array named yytext.
Thus, to print the name found, a rule like

```
[a-z]+ printf("%s", yytext);
```

prints the string in yytext. The C function printf accepts a format argument and data to be printed; in this case, the format is 'print string' (% indicating data conversion, and s indicating string type), and the data are the characters in yytext. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+ ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches read() it normally matches the instances of read contained in bread or readjust; to avoid this, a rule of the form [a-z]+ is needed. This is explained further below.

Length of Matched Text

Sometimes it is more convenient to know the end of what has been found; hence lex also provides a count yyleng of the number of characters matched. To count both the number of words and the number of characters in words in the input, the programmer might write

```
[a-zA-Z]+      {words++; chars += yyleng;}
```

which accumulates in chars the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1].
```

Table 9-2 *Regular Expression Operators in lex*

Operator	*Meaning*
`x`	the character "x"
`"x"`	an "x", even if x is an operator
`\x`	an "x", even if x is an operator
`[xy]`	the character x or y
`[x-z]`	the characters x, y or z
`[^x]`	any character but x
`.`	any character but newline
`^x`	an x at the beginning of a line
~~`<y>x`~~	~~an x when Lex is in start condition y~~
`x$`	an x at the end of a line
`x?`	an optional x
`x*`	0,1,2, ... instances of x
`x+`	1,2,3, ... instances of x
`x|y`	an x or a y
`(x)`	an x
`x/y`	an x but only if followed by y
~~`{xx}`~~	~~the translation of xx from the definitions section~~
~~`x{m, n}`~~	~~*m* through *n* occurrences of x~~