Recognition of tokens with Lex

Having described a way to characterize the patterns associated with tokens, we begin to consider how to recognize tokens — i.e. recognize instances of patterns — i.e. recognize the strings of a regular language.

We'll use Lex: it generates an efficient scanner automatically, based on regular expressions.

Consider the grammar

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

We need to specify patterns for the tokens: **if**, **then**, **else**, **relop**, **id**, **num**.

We can use the regular definition we introduced last time:

$$
\begin{array}{rcl}
\textbf{if} & \rightarrow & \text{if} \\
\textbf{then} & \rightarrow & \text{then} \\
\textbf{else} & \rightarrow & \text{else} \\
\textbf{relop} & \rightarrow & \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{>} \mid \texttt{>=} \\
\textbf{digit} & \rightarrow & [0\text{–}9] \\
\textbf{letter} & \rightarrow & [\text{a–zA–Z}] \\
\textbf{id} & \rightarrow & \textbf{letter}\ (\ \textbf{letter} \mid \textbf{digit}\ )^* \\
\textbf{num} & \rightarrow & \textbf{digit}^+\ (\ .\ \textbf{digit}^+\ )?\ (\ \text{E}\ (\ \texttt{+} \mid \texttt{-}\ )?\ \textbf{digit}^+\ )?
\end{array}
$$

We'll assume in addition that keywords are reserved. So although the string `if`, for instance, belongs to the language denoted by **id** as well as the language denoted by **if**, our lexical analyzer should return token **if** given lexeme `if`.

We will also assume that lexemes may be separated by whitespace — a nonempty string of blanks, tabs and newlines. Our scanner will strip out white space, using the regular definition below:

$$
\begin{array}{rcl}
\textbf{delim} & \rightarrow & \textbf{blank} \mid \textbf{tab} \mid \textbf{newline} \\
\textbf{ws} & \rightarrow & \textbf{delim}^+
\end{array}
$$

If a match for **ws** is found, no token will be returned; instead we return the token after **ws**.

As before, we imagine that the scanner returns pairs

$$\langle \texttt{token}, \texttt{attribute} \rangle .$$

We can do this according to the following table:

regular expression	*token*	*attribute value*
ws	*none*	*none*
if	**if**	*none*
then	**then**	*none*
else	**else**	*none*
id	**id**	*lexeme*
num	**num**	*lexeme*
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

As before, in practice we will return the token and place the attribute value in a global variable.

We'll build our Lex scanner in accordance with this table. (For instance, we won't "directly" define a pattern for **relop**.)

Scanner in lex

```
%{
 /* C declarations */
#define LT 256
#define LE 257
#define EQ 258
#define NE 259
#define GT 260
#define GE 261

#define RELOP 262
#define ID 263
#define NUM 264
#define IF 265
#define THEN 266
#define ELSE 267

int attribute;
%}

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

%%
```

```
{ws}      {}
if        { return(IF); }
then      { return(THEN); }
else      { return(ELSE); }
{id}      { return(ID); }
{num}     { return(NUM); }
"<"       { attribute = LT; return(RELOP); }
"<="      { attribute = LE; return(RELOP); }
"<>"      { attribute = NE; return(RELOP); }
"="       { attribute = EQ; return(RELOP); }
">"       { attribute = GT; return(RELOP); }
">="      { attribute = GE; return(RELOP); }

%%

int yywrap()      /* lex expects this function -- it is */
{                 /*   called whenever EOF is read      */
  return 1;
}

int main()        /*  main function for the scanner     */
{
  int token;
  while(token = yylex()) {
    printf("( %d, ", token);
    switch(token) {
      case ID: case NUM:
        printf("%s )\n", yytext);
        break;
```



```
      case RELOP:
        printf("%d )\n", attribute);
        break;
      default:
        printf(")\n");
        break;
    }
  }
  return 0;
}
```

```
> lex s3.18.l

> gcc -o s3.18 lex.yy.c
```

The central function in `lex.yy.c` is `yylex()`.

Typically, each call to `yylex()` returns a token.

Lex specifications

A Lex program consists of three (four?) parts:

```
%{
C declarations
%}
regular definitions
%%
translation rules
%%
C functions, incl. yywrap()
```

Anything included between the funny braces `%{` and `%}` is copied verbatim from the lex file to `lex.yy.c`.

The Lex regular definitions are similar to the regular definitions we have studied already.

(We'll look more closely at the syntax of these in a moment.)


```
%{
C declarations
%}
regular definitions
%%
translation rules
%%
C functions, incl. yywrap()
```

The translation rules are statements of the form

$$\begin{array}{ll} p_1 & action_1 \\ p_2 & action_2 \\ \vdots & \\ p_n & action_n \end{array}$$

where each p_i is a regular expression and each $action_i$ is a C program fragment.

When `yylex()` is called, it **finds the longest prefix of the input that matches one of the regular expressions** p_i, places the lexeme in `yytext`, and executes the corresponding action $action_i$. (**If two expressions match longest lexeme, prefer the first!**)

Typically, the action ends by returning the appropriate token. But if the action does not end with a return of control, then the parser proceeds to find the next lexeme and execute the corresponding action.

```
%{
C declarations
%}
regular definitions
%%
translation rules
%%
C functions, incl. yywrap()
```

Note: unmatched characters are simply written to `stdout`.

The function `yywrap()` is called whenever `EOF` is encountered in the input. (It seems there is no way to write a regular expression to match `EOF`.)

`yywrap()` can be used to continue processing on additional files. (Arrange for new input file and return 0.) Otherwise, `yywrap()` should return a 1.

If you want a stand-alone scanner, you must supply a main function.

(Actually, *assuming the library is available*, you can compile with `-ll` to get a default `main` and `yywrap`.)

Here is a short Lex program that simply copies its input.

```
%%

%%

yywrap()
{
  return 1;
}

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

It has no translation rules, so each input character is simply written to output.

Here's one that eliminates all whitespace:

```
%%

[ \n\t]  {}

%%

yywrap()
{
  return 1;
}

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

The only translation rule matches each whitespace character, and since the action is empty, nothing happens.

Again, non-whitespace characters go unmatched, and so are echoed to output.

Here's a minor variation. Each nonempty sequence of whitespace characters is replaced by a single blank.

We also put a newline on output upon encountering EOF.

```
%%

[ \n\t]+  { putchar(' '); }

%%

yywrap()
{
  putchar('\n');
  return 1;
}

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

The only translation rule matches each nonempty sequence of whitespace characters, taking the longest match possible, and puts a single blank on output.

Again, non-whitespace characters go unmatched, and so are echoed to output.

Here's a Lex program for counting lines, words and characters:

```
%{
int lines = 0, words = 0, characters = 0;
%}

%%

[^ \t\n]+         { words++; characters += yyleng; }
[ \t]+            { characters += yyleng; }
\n                { lines++; characters++; }

%%

yywrap()
{
  printf("\n %d lines,  %d words,  %d characters\n\n",
                                         lines, words, characters);
  return 1;
}

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

Notice the use of ^ to denote the complement of a character class.

Here's another Lex program for counting lines, words and characters.

This one echoes all words, and replaces strings of whitespace with a single blank, unless the whitespace ends a line, in which case it is "replaced" by a newline.

```
%{
int lines = 0, words = 0, characters = 0;
%}

%%

[^ \t\n]+         { words++; characters += yyleng; ECHO; }
[ \t]+            { characters += yyleng; putchar(' '); }
[ \t]*\n          { lines++; characters += yyleng; putchar('\n'); }

%%

yywrap()
{
  printf("\n %d lines,  %d words,  %d characters\n\n",
              lines, words, characters);
  return 1;
}

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

Regular expressions in Lex

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

are operators in Lex, and so must be handled carefully in Lex regular expressions.

Below are some Lex regular expression constructs:

Lex regular expression	*matches*	*example*
`c`	any non-operator character `c`	`a`
`\c`	operator character `c` literally	`\*`
`"s"`	string *s* literally	`"**"`
`.`	any character but newline	`a.*b`
`^`	beginning of line	`^abc`
`$`	end of line	`abc$`
`[s]`	any character in *s*	`[abc]`
`[^s]`	any character not in *s*	`[^abc]`
`r*`	zero or more *r*'s	`a*`
`r+`	one or more *r*'s	`a+`
`r?`	zero or one *r*'s	`a?`
`r{m,n}`	*m* to *n* occurrences of *r*	`a{1,3}`
`pr`	*p* then *r*	`ab`
`p|r`	*p* or *r*	`a|b`
`(r)`	*r*	`(a|b)`
`p/r`	*p* when followed by *r*	`ab/c`

As suggested by the entries in the table, the special meaning of the operator symbols

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

must be "turned off" if they are to be matched literally. This can be done by quoting or by using the backslash.

For instance, `**` is matched by both `"**"` and `\*\*`.

What is a Lex expression to match `"\` ?

In Lex, we can also use the character class notation. For instance, `[a-zA-Z0-9]` matches any alphanumeric character (string of length 1).

In Lex, a complemented character class is one the begins with `^` — for example `[^a]` matches any symbol different from `a`, and `[^a-zA-Z0-9]` matches any non-alphanumeric character.

```
%{
C declarations
%}
regular definitions
%%
translation rules
%%
C functions, incl. yywrap()
```

Lex allows regular definitions, in addition to regular expressions.

Here is a fragment of our first Lex example, starting with the regular definitions section:

```
delim  [ \t\n]
ws     {delim}+
letter [A-Za-z]
digit  [0-9]
id     {letter}({letter}|{digit})*
num    {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
{ws}     {}
if       { return(IF); }
then     { return(THEN); }
else     { return(ELSE); }
{id}     { return(ID); }
{num}    { return(NUM); }
```


Here's an interesting difficulty.

What's a regular expression for comments in C?

How about `"/*"(.|\n)*"*/"` ?   (Looks like a hard problem.)

One approach is to use Lex "start conditions"...

Idea: conditionally activate patterns. Use for...

- conceptually different components of input
- situations where Lex defaults such as "longest possible match" don't work well. For example, comments and quoted strings.

Declare a set of start condition names using

`%Start` $name_1$ $name_2$ ...

in the definitions section.

For each $name_i$, a pattern prefixed with $\langle name_i \rangle$ is active only when the scanner is in start condition $name_i$.

The scanner starts out in start condition INITIAL. (Start condition INITIAL is "built-in": you may obtain confusing behavior if you *declare* a start condition with name INITIAL.)

All rules not prefixed with some $\langle name_i \rangle$ are active in all start conditions, including INITIAL.

```
%Start comment0 comment1

%%

<INITIAL>"/*"      { BEGIN(comment0); }
<comment0>\*       { BEGIN(comment1); }
<comment0>[^\*]    {}
<comment1>\*       {}
<comment1>\/       { BEGIN(INITIAL); }
<comment1>[^\*\/]  { BEGIN(comment0); }

%%

yywrap()
{
  return 1;
}

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


For next time. . .

We'll begin the study of finite automata, to get a solid understanding of the general problem Lex solves.

Read Section 3.6