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

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

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

We can use the regular definition we introduced last time:

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

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:

\[
\begin{align*}
\text{delim} & \rightarrow \text{blank} \mid \text{tab} \mid \text{newline} \\
\text{ws} & \rightarrow \text{delim}^*
\end{align*}
\]

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

\((\text{token}, \text{attribute})\).

We can do this according to the following table:

<table>
<thead>
<tr>
<th>regular expression</th>
<th>token</th>
<th>attribute value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>if</td>
<td>if</td>
<td>none</td>
</tr>
<tr>
<td>then</td>
<td>then</td>
<td>none</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
<td>none</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>lexeme</td>
</tr>
<tr>
<td>num</td>
<td>num</td>
<td>lexeme</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>

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 \texttt{relop}.)
if { return(IF); } then { return(THEN); } else { return(ELSE); }
{id} { return(ID); }
{num} { return(NUM); }
"<" { attribute = LT; return(RELOP); }
"<=" { attribute = LE; 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 \n", 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.)

The translation rules are statements of the form

```
p_1  \hspace{1em} action_1
p_2  \hspace{1em} action_2
\vdots
p_n  \hspace{1em} action_n
```

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.
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.

```plaintext
% i
C declarations
%
regular definitions
%
translation rules
%
C functions, incl. yywrap()

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:

```plaintext
%{
  int lines = 0, words = 0, characters = 0;
%
%
[^ \t\n\]+ { words++; characters += yyleng; }
[ \t\]+ { characters += yyleng; }
{ lines++; characters++; }
%
}

yywrap()
{
  printf("%d lines, %d words, %d characters\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.

```plaintext
%{
  int lines = 0, words = 0, characters = 0;
%
%
[^ \t\n\]+ { words++; characters += yyleng; ECHO; }
[ \t\]+ { characters += yyleng; putchar(' '); }
\n { lines++; characters++; }
%
}

yywrap()
{
  printf("%d lines, %d words, %d characters\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:

<table>
<thead>
<tr>
<th>Lex regular expression</th>
<th>matches</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>any non-operator character</td>
<td>a</td>
</tr>
<tr>
<td>\c</td>
<td>operator character c literally</td>
<td>*</td>
</tr>
<tr>
<td>&quot;s&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 line</td>
<td>^abc</td>
</tr>
<tr>
<td>$</td>
<td>end of line</td>
<td>abc$</td>
</tr>
<tr>
<td>[s]</td>
<td>any character in s</td>
<td>[abc]</td>
</tr>
<tr>
<td>[^s]</td>
<td>any character not in s</td>
<td>[^abc]</td>
</tr>
<tr>
<td>r*</td>
<td>zero or more r's</td>
<td>a*</td>
</tr>
<tr>
<td>r+</td>
<td>one or more r's</td>
<td>a+</td>
</tr>
<tr>
<td>r?</td>
<td>zero or one r's</td>
<td>a?</td>
</tr>
<tr>
<td>r{m,n}</td>
<td>m to n occurrences of r</td>
<td>a{1,3}</td>
</tr>
<tr>
<td>pr</td>
<td>p then r</td>
<td>ab</td>
</tr>
<tr>
<td>p</td>
<td>r</td>
<td>p or r</td>
</tr>
<tr>
<td>r</td>
<td></td>
<td>r</td>
</tr>
<tr>
<td>p/r</td>
<td>p when followed by r</td>
<td>ab/c</td>
</tr>
</tbody>
</table>

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.
Lex allows regular definitions, in addition to regular expressions.

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

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

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

```c
%Start name1 name2 ...
```

in the definitions section.

For each `namei`, a pattern prefixed with `<namei>` is active only when the scanner is in start condition `namei`. 

```c
delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}{(letter)|{digit}}*
um {digit}+{\.(digit)}+?(E[+\-]{digit})?
%
{ws} {} if { return(IF); }
then { return(THEN); }
else { return(ELSE); }
{id} { return(ID); }
{num} { return(NUM); }
```
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(comment1); }

% %

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