Pretty Printing in HTML

In this project you will implement Pretty, a simple pretty printer written in C++ that takes C source code and outputs the source code in reformatted HTML ("pretty printed"). The HTML-formatted text highlights C keywords and directives `define` and `include`, operators, constants, and comments. It adds line numbers to the output and properly indents statements and statement blocks. The objective of this assignment is to use a lex specification to build pretty using the RE/flex scanner generator. We will use RE/flex as an improvement of Flex for C++ and other features that Flex lacks, such as lazy repeats.


Lex Specification

To create Pretty, you should first write a lex specification that contains definitions of the patterns for the tokens that make up a C source program to tokenize. We will use simpler patterns for C tokens than the official C99 standard. The patterns defined by the regular definitions below define simple patterns for C tokens that are used by the translation rules (shown on the next page):

\[
\begin{align*}
quote & \rightarrow \ \text{'}' \\
ditto & \rightarrow \ \text{""} \\
digit & \rightarrow \ 0 \ | \ ... \ | \ 9 \\
oct & \rightarrow \ 0 \ | \ ... \ | \ 7 \\
exp & \rightarrow \ (e \ | \ E) \ (+ \ | \ - \ | \ \varepsilon) \ digit^+ \\
hex & \rightarrow \ digit \ | \ a \ | \ ... \ | \ f \ | \ A \ | \ ... \ | \ F \\
alpha & \rightarrow \ a \ | \ ... \ | \ z \ | \ A \ | \ ... \ | \ Z \ | \ 
\end{align*}
\]
The following lexical translation rules convert C into HTML by actions that are triggered by specific lexical patterns. The action functions pretty-print the tokens in HTML using the write\_X methods:

<table>
<thead>
<tr>
<th>pattern</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>quote (notquote</td>
<td>\ \ quote)* quote</td>
</tr>
<tr>
<td>ditto (notditto</td>
<td>\ \ ditto)* ditto</td>
</tr>
<tr>
<td>0 oct+</td>
<td>write_oct()</td>
</tr>
<tr>
<td>0 (x</td>
<td>X) hex+</td>
</tr>
<tr>
<td>digit+</td>
<td>write_int()</td>
</tr>
<tr>
<td>digit* \ . digits* (exp</td>
<td>$\varepsilon$)</td>
</tr>
<tr>
<td>alpha (alpha</td>
<td>digit)*</td>
</tr>
<tr>
<td>{</td>
<td>write_begin()</td>
</tr>
<tr>
<td>}</td>
<td>write_end()</td>
</tr>
<tr>
<td>(</td>
<td>write_open()</td>
</tr>
<tr>
<td>)</td>
<td>write_close()</td>
</tr>
<tr>
<td>[</td>
<td>write_bopen()</td>
</tr>
<tr>
<td>]</td>
<td>write_bclose()</td>
</tr>
<tr>
<td>;</td>
<td>write_sep()</td>
</tr>
<tr>
<td>operator, see note(^a) below</td>
<td>write_op()</td>
</tr>
<tr>
<td>in-line comment, see note(^b) below</td>
<td>write_inline()</td>
</tr>
<tr>
<td>multi-line comment, see note(^b) below</td>
<td>write_comment()</td>
</tr>
<tr>
<td>directive, see note(^c) below</td>
<td>write_directive()</td>
</tr>
<tr>
<td>white space, see note(^d) below</td>
<td>(none)</td>
</tr>
<tr>
<td>any remaining character</td>
<td>display an error message</td>
</tr>
</tbody>
</table>

where notditto is any character but a ” “ and notquote is any character but a ’.

Note\(^a\): Obtain an ANSI C manual and list all ANSI C operators (arithmetic, logic, relational, assignments, etc.) that need to go into your lex specification.

Note\(^b\): Multi-line comments are enclosed in /* and */ and in-line comments start with // and end at a newline (i.e. you need to consume any characters except newline up to the first newline). You may use lazy repeats \(^*\)? (RE/flex supports lazy repeats, Flex does not).

Note\(^c\): To scan directives such as \#include and \#define, look for a \#. Then consume any characters (except newline) up to the first newline.

Note\(^d\): White space consisting of blanks, horizontal tabs \t, vertical tabs \v, newlines \n, carriage returns \r, and form feeds \f must be ignored.

The rules above are written in a generic RE notation. You should rewrite them according to the Lex specification requirements for your regular definitions and translation patterns in your Lex specification. Make sure you use the full expressive power of the regular expressions in Lex (see textbook page 148 and Chapter 3 PPT notes page 19). Searching the Web can help, but be aware that our regular definitions and expressions we use are simpler and different than those for a full-fledged ANSI C scanner. Remember that yytext contains the lexeme as a string, so use this in your action functions.

Your first task for the programming assignment is to define the translation rules in a lex specification pretty.l.
The pretty-l specification has the following structure:

```c
#include <stdio.h>
#include <stdlib.h>
#define INDENT (4)

int line;
int column;
int paren;
... // symbol table to store identifier names with line numbers
... // your yyFlexLexer::write_X methods (declarations, method code can go elsewhere)
```

Your regular definitions

```c
%
```

Your translation rules

```c
%
```

Your program code

The translation rules should invoke the `write_X` functions defined in the program code part of the lex specification and discussed in the next section.

Translation to HTML

The main program writes the opening and closing HTML tags, formats the input C source code in a PRE(formatted) HTML block, starts the output with a new indented line, and invokes `yylex` to translate the input as follows:

```c
int main()
{
    printf("<html><pre>\n");
    yyFlexLexer().yylex();
    printf("\n</pre></html>\n");
    return 0;
}
```
We will only use the basic HTML formatting markup. You may want to consult an HTML manual or HTML tutorial for beginners if you are unfamiliar with HTML.

The indent method starts a new line with the line number stored in variable line and an indent by variable column:

```c
void yyFlexLexer::write_indent()
{
    printf("\n%-*d", column, line++);
}
```

The write_html_char function outputs a character in HTML by translating reserved characters to HTML entities:

```c
void yyFlexLexer::write_html_char(int c)
{
    switch (c)
    { case '<': printf("&lt;"); break;
      case '>': printf("&gt;"); break;
      case '"': printf("&quot;"); break;
      case '&': printf("&"); break;
      case '
': break; // we don't want newlines to show up in HTML PRE blocks
      default: putchar(c);
    }
}
```

We convert the yytext lexeme to HTML using:

```c
void yyFlexLexer::write_html()
{
    char *s = yytext;
    while (*s)
        write_html_char(*s++);
}
```

We use this function extensively to copy the content of yytext to our HTML output. For example, the following functions are responsible for formatting statements terminated with a ; and statement blocks enclosed in { and }:

```c
// output ';', i.e. statement terminator or for()-expression separator
void yyFlexLexer::write_sep()
{
    write_html();
    if (!paren)
        write_indent();
    else
        putchar(' ');
}
```

```c
// begin {}-block
```
void yyFlexLexer::write_begin()
{
    write_indent();
    write_html();
    column += INDENT;
    write_indent();
}
// end {}-block
void yyFlexLexer::write_end()
{
    column -= INDENT;
    write_indent();
    write_html();
    write_indent();
}

As you can see, the yyFlexLexer::write_sep method checks if the ; does not occur in a paren pair, e.g. in a for construct. So the paren global variable keeps track of the depth of the parenthesis as shown by the following functions:

// start opening paren
void yyFlexLexer::write_open()
{
    write_html();
    putchar(' ');    
    paren++;          
}
// close paren
void yyFlexLexer::write_close()
{
    write_html();
    putchar(' ') ;
    paren--;           
}

For this assignment you need to implement the remaining yyFlexLexer::write_* methods to produce the HTML output. The following table shows the required HTML output:

<table>
<thead>
<tr>
<th>token</th>
<th>HTML</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Red</td>
<td>&quot;Hello world!&quot;</td>
</tr>
<tr>
<td>char</td>
<td>Brown, underlined</td>
<td>\n</td>
</tr>
<tr>
<td>hex</td>
<td>Brown, italicized</td>
<td>0xaf</td>
</tr>
<tr>
<td>oct</td>
<td>Brown, italicized</td>
<td>077</td>
</tr>
<tr>
<td>int</td>
<td>Brown</td>
<td>123</td>
</tr>
<tr>
<td>fp</td>
<td>Brown</td>
<td>3.14e6</td>
</tr>
<tr>
<td>id</td>
<td>Blue, boldface for keywords</td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>and plain blue hyper-linked</td>
<td>main()</td>
</tr>
<tr>
<td></td>
<td>(see below) for identifiers</td>
<td></td>
</tr>
<tr>
<td>operators/punctuation</td>
<td>Black, boldface</td>
<td>&amp; &amp;</td>
</tr>
<tr>
<td>comments</td>
<td>Green</td>
<td>// TODO: needs work</td>
</tr>
<tr>
<td>directives</td>
<td>Magenta, boldface</td>
<td>#define</td>
</tr>
</tbody>
</table>
To store identifier names you should add a symbol table to the lexer class. You may also store keywords in this table (or use patterns to recognize keywords, but that may slow down scanning). Initialize the symbol table with keywords and markers to distinguish keywords from identifiers. You may use the symbol table of programming assignment 1 or use an STL map.

Use this symbol table to make identifiers hyper-linked in the HTML output. Add a `yyFlexLexer::write_id()` method that produces the HTML anchor if this identifier is first encountered and produces the HTML link to the anchor if the identifier is encountered again. That is, we produce `<a name="idname"/>` to anchor the first occurrence of the identifier in the HTML output, and we produce `<a href="#idname">idname</a>` to link it from all other sites where the identifier occurs in the HTML file.

This means that when you click on an identifier in the HTML displayed in a browser, the browser *should jump to the first line of code in the same file where the identifier first occurred*. This assumes that identifiers are mostly globally declared in the input C source code, such as functions and other globals. Note that this does not necessarily work for local variables. Since a lexical analyzer isn’t aware of the syntax, we are not making any attempts to make the linkage more intelligent.

**Makefile**

To compile your C++ application, create a Makefile for make:

```makefile
CC=c++
LEX=reflex/src/reflex
LIBS=reflex/lib/libreflex.a
COFLAGS=-O2
CWFLAGS=-Wall -Wextra
CFLAGS=-I. -Ireflex/include
CMFLAGS=
CFLAGS=$(CWFLAGS) $(COFLAGS) $(CFLAGS) $(CMFLAGS)
pretty: pretty.o
  $(CC) $(CFLAGS) -opretty $(LIBS) $<
.cpp.o:
  $(CC) $(CFLAGS) -c $<
pretty.cpp: pretty.l
  $(LEX) -- -opretty.cpp pretty.l
```

Note that your Makefile may differ from the above example in order to compile additional C++ files in addition to `pretty.cpp` (the generated scanner). It is assumed that reflex is located in a local directory in the current project directory, but you are free to adapt this to reflect your installation of RE/flex.

**Conclusions**

The pretty printer does a decent job formatting your code. For future projects, you may be interested in using Doxygen (www.doxygen.org) developed by Dimitri van Heesch, High Tech Campus Eindhoven, the Netherlands. It produces HTML with cross-references, class inheritance diagrams, and file dependencies.

*End*