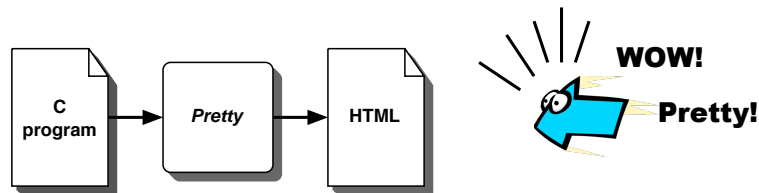


Compiler Construction Assignment 2 – Fall 2011

Robert van Engelen

What's Pretty?

In this project you will implement *Pretty*, a simple pretty printer for C code. *Pretty* formats your C code in HTML. The HTML-formatted text highlights C keywords, 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 build *pretty* using GNU Flex.



Lex Specification

To create *Pretty*, you first need to write a Lex (GNU Flex) specification that contains definitions of the patterns for the tokens that make up a C source program (including C header files).

The regular definitions below, written in a generic form, define some common basic character patterns for the translation rules (shown on the next page):

<i>quote</i>	→	'
<i>ditto</i>	→	”
<i>back</i>	→	\
<i>digit</i>	→	0 ... 9
<i>exp</i>	→	(e E) (+ - ϵ) <i>digit</i> ⁺
<i>hex</i>	→	<i>digit</i> a ... f A ... F
<i>alpha</i>	→	a ... z A ... Z -
<i>ch</i>	→	any ASCII character except newline, \ (back), ' (quote), and ” (ditto)

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:

pattern	action
<i>ditto</i> (<i>back ch</i> <i>back back</i> <i>back quote</i> <i>back ditto</i> <i>ch</i> <i>quote</i>)* <i>ditto</i>	write_string()
<i>quote</i> (<i>back ch</i> <i>back back</i> <i>back quote</i> <i>back ditto</i> <i>ch</i> <i>ditto</i>)* <i>quote</i>	write_char()
0 (0 ... 7)+	write_oct()
0 (x X) <i>hex</i> +	write_hex()
<i>digit</i> +	write_int()
<i>digit</i> * . <i>digit</i> * (<i>exp</i> ϵ)	write_fp()
<i>alpha</i> (<i>alpha</i> <i>digit</i>)*	write_id()
{	write_begin()
}	write_end()
(write_open()
)	write_close()
[write_bopen()
]	write_bclose()
;	write_sep()
<i>operator</i> , see note ^a below	write_op()
<i>in-line comment</i> , see note ^b below	write_inline()
<i>multi-line comment</i> , see note ^b below	write_comment()
<i>directive</i> , see note ^c below	write_directive()
<i>white space</i> , see note ^d below	(none)
<i>any remaining character</i>	error()

Note^a: Obtain an ANSI C manual and list all ANSI C operators that need to go into your lex specification. C++ operators are not needed.

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). A simple RE definition to match `/* ... */` suffices, but it may cause the lexical analyzer to fail on large comment blocks when its internal buffer overflows. A fix for this is described in this document.

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 C string, so use this in your action functions as needed.

Your first task for the programming assignment is to define the translation rules in a Lex file `pretty.1`.

The `pretty.l` Lex file has the following structure:

```
%option noyywrap
%{
#include <stdio.h>
#include <stdlib.h>
#define INDENT (4)
int line = 1;
int column = 4;
int paren = 0;
%}
```

Your regular definitions

```
%%
```

Your translation rules

```
%%
```

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 code in a PRE(formatted) HTML block, starts the output with a new indented line, and invokes `yylex` to translate the input as follows:

```
int main()
{ printf("<html><pre>\n");
  indent();
  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` function starts a new line with the line number stored in global variable `line` and an indent spaced by global variable `column`:

```
indent()
{ printf("\n%-*d", column, line++);
}
```

Special care has to be taken to scan multi-line comments. The reason is that the Lex buffer is too small to hold a larger multi-line comment when we use a regular expression to define the pattern of a multi-line comment. Instead, we use parts of code borrowed from an open source C compiler and insert instructions to write HTML:

```
write_comment()
{ char c, c1;
  printf("<font color='#00FF00'>/*");
loop:
  while ((c = input()) != '*' && c != 0)
    write_html_char(c);
  write_html_char('*');
  if ((c1 = input()) != '/' && c1 != 0)
  { unput(c1);
    goto loop;
  }
  if (c != 0)
    write_html_char(c1);
  printf("</font>");
}
```

Because this function reads from the input directly up to and including the terminating `*/`, the translation rule in the Lex specification is simplified to:

```
"/*"          { write_comment(); }
```

This `write_comment` code also illustrates the formatting of the comment in HTML FONT tags, where the color attribute value contains the 24-bit RGB (Red, Green, and Blue) color value in hexadecimal. Thus, multi-line comments are shown in bright green.

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

```
write_html_char(int c)
{ switch (c)
  { case '<': printf("&lt;"); break;
    case '>': printf("&gt;"); break;
    case '"': printf("&quot;"); break;
    case '&': printf("&amp;"); break;
    default: putchar(c);
  }
}
```

We convert the `yytext` lexeme to HTML using:

```
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 `}`:

```
// output ';', i.e. statement terminator or for()-expression separator
write_sep()
{ write_html();
  if (!paren)
    indent();
  else
    putchar(' ');
}
// begin {}-block
write_begin()
{ indent();
  write_html();
  column += INDENT;
  indent();
}
// end {}-block
write_end()
{ column -= INDENT;
  indent();
  write_html();
  indent();
}
```

As you can see, the `write_sep` function 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
write_open()
{ write_html();
  putchar(' ');
  paren++;
}
// close paren
write_close()
{ write_html();
  putchar(' ');
  paren--;
}
```

For this assignment you need to implement the remaining `write_X` functions to produce the HTML output. The following table shows the required HTML output:

token	HTML	Example
string	Red	" Hello world! "
char	Brown, underlined	' <u>n</u> '
hex	Brown, italicized	<i>0x..</i>
oct	Brown, italicized	<i>0..</i>
int	Brown, italicized	<i>123</i>
fp	Brown	-3.14e6
id	Blue, boldface for keywords and plain blue hyper-linked (see below) for identifiers	while main()
operators/punctuation	Black, boldface	&&
comments	Green	// TODO: needs work
directives	Magenta, boldface	#define

You need a symbol table to store keywords and identifiers. Use the symbol table of programming assignment 1. Make identifiers hyper-linked in the HTML output you produce with `write_id()`. Thus, when you click on an identifier, 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 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. Use `` to anchor the first occurrence of the identifier in your HTML output, and use `idname` to link it from all other sites where the identifier occurs in the HTML file.

Makefile

To compile your application, use `make` and a `Makefile` with:

```
CC=gcc
COFLAGS=
CWFLAGS=
CIFLAGS=
CMFLAGS=
CFLAGS= $(CWFLAGS) $(COFLAGS) $(CIFLAGS) $(CMFLAGS)
pretty:      pretty.o symbol.o init.o
             $(CC) $(CFLAGS) -opretty $<
.c.o:
             $(CC) $(CFLAGS) -c $<
pretty.c:    pretty.l
             flex -opretty.c pretty.l
```

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.