CS F 363 Compiler Construction Assignments

Prof. Santonu Sarkar

February 3, 2018

1 Create a Beautifier for a language

In this assignment implement a class Beautify, and then a subclass called CBeautify which formats a C code in HTML. The instructions below should be implemented in CBeautify class (design the class appropriatly). The main function will open a C program file, invoke an appropriate function of CBeautify object which will eventually generate an HTML file. 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 build the beautifier using GNU Flex.

1.1 Lex Specification

To create the beautifier, you first need to write a Lex 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, define some common basic character patterns for the translation rules:

$quote \rightarrow$	’
$ditto \rightarrow$	"
$back \rightarrow$	\
$digit \rightarrow$	$0\|...\|9$
$exp \rightarrow$	$(e\|E)(+\|-\|\epsilon)digit+$
$hex \rightarrow$	$digit\|a\|...\|f\|A\|...\|F$
$alpha \rightarrow$	$a\|...\|z\|A\|...\|Z\|\_$
ch $\rightarrow$	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 print the tokens in HTML:

Pattern	function
ditto (back ch | back back | back quote | back ditto | ch |quote) ditto	`write_string()`
quote (back ch | back back | back quote | back ditto | ch | ditto) quote	`write_char()`
0 (0 | ...| 7)+	`write_oct()`
0 (x | X) hex+	`write_hex()`
digit+	`write_int()`
digit*. digit* (exp | ϵ)	`write_fp()`
alpha (alpha | digit)*	`write_id()`
{	`write_begin()`
}	`write_end()`
(	`write_open()`
)	`write_close()`
`[`	`write_bopen()`
`]`	`write_bclose()`
;	`write_sep()`
operator, see **Notea** below	`write_op()`
in-line comment, see **Noteb** below	`write_inline()`
multi-line comment, see **Noteb** below	`write_comment()`
directive, see **Notec** below	`write_directive()`
white space, see **Noted** below	no action
any remaining character	`error()`

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

Noteb 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 here.

Notec To scan directives such as `#include, #define`, look for a #. Then consume any characters (except newline) up to the first newline.

Noted White space consisting of blanks, `\t, \v, \n, \r, \f` must be ignored.

Rewrite the above rules 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. Remember that yytext contains the lexeme as a C string, so use this in your action functions as needed. Your first task is to define the translation rules in a Lex file beautify.l. The beautify.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.

1.2 Beautification

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;
}
```

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()) != '/' && c != 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);
}}
```

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();
}
```

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	Cyan, underlined	'\n'
hex	Cyan, italicized	*0x..*
oct	Cyan, italicized	*0..*
int	Cyan, italicized	*123*
fp	Cyan	-3.14e6
keywords	Blue, boldface	**while**
id	plain blue	main()
operators/punctuation	Black, boldface	&&
comments	Green	// TODO: Do it later
directives	Magenta, boldface	**#define**

1.3 Makefile

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

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

1.4 Use of Symbol Table

Create a symbol table with a suitable data structure, simplest being array of struct. Use the symbol table to store keywords and identifiers.

```
#define MAXSYMBOLS 200
struct entry
{   char *lexptr;
   int token;
};
struct entry symtable[MAXSYMBOLS];
int lookup(char *lexeme); // returns index of the symbol table where the lexeme appears, -
int insert(char *a, int token); // inserts a new entry into a symbol table
void init(); // initializes the symbol table with preloaded keywords
```

Now modify your previous program as follows:

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 ¡a name="idname"/¿ to anchor the first occurrence of the identifier in your HTML output, and use `<a href="#idname">idname</a>` to link it from all other sites where the identifier occurs in the HTML file.