COSC 461/561

cscan - scanner for a C-like language

Your assignment is to implement the *cscan* library, which conducts lexical analysis for programs written in a C-style language. The *cscan* library includes two procedures that you will implement: *init_scanner* and *get_token*. *init_scanner* completes any initial setup that may be necessary for conducting lexical analysis on the input program. *get_token* simply reads and returns the next token from standard input.

You will test your scanner implementation using the *scan_print* program we have provided. This program invokes the *cscan* routines to read an input program from standard input and print tokens in the input program to standard output. It prints one token per line and continues printing tokens until it encounters EOF or there is an error. When the *get_token* routine encounters an invalid token, it should print an appropriate error message and exit the process.

cscan should support the following tokens for identifiers and literals:

Token Name	**Description**	**Example Lexemes**	**Attribute Value**
IDENTIFIER	letter or '_' followed by letters or digits or '_"s	`foo`, `x1`	Pointer to lexeme as character array
INT_LITERAL	constant integer number	`58`, `0`	value of constant as long integer
REAL_LITERAL	constant real number	`3.14`, `0.6`	value of constant as double
STRING_LITERAL	anything but `"`, surrounded by `"`'s	`"core dumped"`	Pointer to string as character array

Additionally, it should support the following tokens for keywords, operators, and delimiters. These tokens only need to match a single lexeme and do not have other attribute values.

Token Name	**Lexeme**	**Token Name**	**Lexeme**
Keywords		**Assignment Operators**	
CHAR	`char`	ASSIGN	`=`
INT	`int`	ASSIGN_OR	`|=`
FLOAT	`float`	ASSIGN_XOR	`^=`
DOUBLE	`double`	ASSIGN_AND	`&=`
IF	`if`	ASSIGN_LSHIFT	`<<=`
ELSE	`else`	ASSIGN_RSHIFT	`>>=`
WHILE	`while`	ASSIGN_ADD	`+=`
DO	`do`	ASSIGN_SUB	`-=`
FOR	`for`	ASSIGN_MUL	`*=`
RETURN	`return`	ASSIGN_DIV	`/=`
BREAK	`break`	ASSIGN_MOD	`%=`
CONTINUE	`continue`	**Delimiters**	
GOTO	`goto`	SEMICOLON	`;`
Logical & Bitwise Operators		COMMA	`,`
OR	`||`	COLON	`:`
AND	`&&`	LPAREN	`(`
NOT	`!`	RPAREN	`)`
BIT_OR	`|`	LBRACE	`[`
BIT_AND	`&`	RBRACE	`]`
BIT_XOR	`&`	LBRACKET	{
INVERSE	`~`	RBRACKET	}
Relational Operators		**Arithmetic Operators**	
GT	`>`	ADD	`+`
GTE	`>=`	SUB	`-`
LT	`<`	MUL	`*`
LTE	`<=`	DIV	`/`
EQUAL	`==`	MOD	`%`
NOT_EQUAL	`!=`	**Other**	
Shift Operators		EOF_TOKEN	`EOF`
LSHIFT	`<<`		
RSHIFT	`>>`		

In addition to scanning and printing out the tokens listed above, *cscan* should identify and appropriately discard text in single-line (i.e., `// ...`) and multi-line (i.e., `/* ... */`) comments. You cannot make any assumptions about the length of lexemes (including literals and identifiers) or single- or multi-line comments. For full credit, the output of your solution must match the output of our solution exactly with all of the provided test cases. We will test your submission with additional private test cases as well so you should design and conduct additional testing to ensure your solution is robust.

You will complete your *cscan* implementation in the provided *scan.c* file. You must use and are not allowed to modify the *scan.h*, *scan_print.c*, and *makefile* files provided in the starter directory. Additionally, the starter directory includes some sample inputs as well as a reference executable (`ref_scan`) that you can use to test alternative inputs.

For submission, you should upload a copy of your modified scan.c file to the Canvas course website by 11:59pm on the assignment due date. Partial credits will be given for incomplete efforts. However, a program that does not compile or run will get 0 points. Point breakdown is below:

- program starts and exits properly (10)
- identifiers (10)
- keywords (10)
- numeric literals (10)
- string literals (10)
- other operators and delimiters (10)
- whitespace identified and discarded (10)
- comments identified and discarded (10)
- error cases handled appropriately (10)
- efficient / elegant design (e.g., no unnecessary computation or restrictions) (10)

```
Example Input:

/*
 * author: Michael Jantz
 *
 * this simple program initializes and prints a couple integers and a couple
 * double values
 */

int main()
{
  int a, c;
  double b, d;

  // initialize
  a = 3;
  b = 4;
  c = 5;
  d = 6;

  // print
  print("%d %3.2f %d %3.2f\n", a, b, c, d);
  return 0;
}
```

```
Example Output:
INT                int
IDENTIFIER         main
LPAREN             (
RPAREN             )
LBRACE             {
INT                int
IDENTIFIER         a
COMMA              ,
IDENTIFIER         c
SEMICOLON          ;
DOUBLE             double
IDENTIFIER         b
COMMA              ,
IDENTIFIER         d
SEMICOLON          ;
IDENTIFIER         a
ASSIGN             =
INT_LITERAL        3
SEMICOLON          ;
IDENTIFIER         b
ASSIGN             =
INT_LITERAL        4
SEMICOLON          ;
IDENTIFIER         c
ASSIGN             =
INT_LITERAL        5
SEMICOLON          ;
IDENTIFIER         d
ASSIGN             =
INT_LITERAL        6
SEMICOLON          ;
IDENTIFIER         print
LPAREN            (
STRING_LITERAL    "%d %3.2f %d %3.2f\n" (length=18)
COMMA             ,
IDENTIFIER        a
COMMA             ,
IDENTIFIER        b
COMMA             ,
IDENTIFIER        c
COMMA             ,
IDENTIFIER        d
RPAREN            )
SEMICOLON         ;
RETURN            return
INT_LITERAL       0
SEMICOLON         ;
RBRACE            }
```