COP4020 Programming languages
Programming assignment 1: Lexical Analyzer
Due Feb. 6, 11:59pm

Lexical Analyzer

In this assignment, you will write a lexical analyzer for a pascal liked language called PASC. The analyzer will be written in Lex.

Token Specification

The table in Page 3 defines the tokens that must be recognized, with their associated symbolic names. All multi-symbol tokens are separated by blanks, tabs, newlines, comments or delimiters.

Comments are enclosed in (* ... *), and cannot be nested. An identifier is a sequence of (at least one) upper or lower case letters followed by zero, one or more digits. Upper and lower cases are not distinguished in PASC. There is no limit on the length of identifiers. However, you may impose limits on the total number of distinct identifiers and string lexemes and on the total number of characters in all distinct identifiers and string taken together. There should be no other limitation on the number of lexemes that the lexical analyzer will process.

An integer constant is an unsigned sequence of digits representing a 10-based number. A float constant can either be a simple float constant or a scientific notation float constant. A simple float constant consists of an unsigned sequence of (at least one) digits, followed by a '.', followed by another unsigned sequence of (at least one) digits representing a 10-based floating point number. A scientific notation float constant is a simple float constant followed by a letter 'e' or 'E', followed by a signed integer or an unsigned integer. A signed integer consists of a sign ('+' or '-' character) followed by an integer constant. Examples of float constants in PASC include 2.0, 1000.00001, $10000.0001e-10$, $10000.0001e20$, and $10000.0001e+10$. A character constant is a character enclosed in single quotes (e.g., 'a' is a character constant a). A string constant is a sequence of characters surrounded by single quotes (e.g. 'Hello, world'). The internal representations of character constant and string constant are different in PASC. Hard-to-type or invisible characters can be represented in character and string constants by *escape sequences*; these sequences look like two characters, but represent only one. The escape sequences supported by the PASC language are \n for newline, \t for tab, \' for the single quote and \\ for the backslash itself. Any other character that follows a backslash is not treated as an escape sequence.

Token attributes

A unique identification of each token (integer aliased with the symbolic token name) must be returned by the lexical analyzer. In addition, the lexical analyzer must pass extra information for identifier, integer constant, floating point constant, string constant, and character constant tokens to the parser. For identifier, integer constant, string constant, and character constant, the extra information is passed to the parser as a single value of the integer type, through a global variable as described below (*yylval.semantic_value*). For integer constants, the numeric value of the constant is passed. For character constants, the numeric value of the character in the local set is passed. In order to allow other phases of the compiler to access the original identifier lexeme, the lexical analyzer passes an integer uniquely identifying the identifier. String constants are treated in the same way, with a unique identifying number being passed. The unique identifying number, for both identifiers and string constants, should be an index into a string table created by the lexical analyzer to record the lexemes. For floating point constant, the extra information is passed to the parser as a single value of the float type, through a global variable (*yylval.fvalue*), that is the numeric value of the floating point constant.

Implementation

The central routine of the scanner is *yylex*, an integer function that returns a *token number*, indicating the type (identifier, integer constant, semicolon, etc.), of the next token in the input stream. In addition to the token type, *yylex* must set the global variables *yyline* and *yycolumn* to the line and column number at which that token appears, and – in the case of identifiers and constants –, put the extra information needed, as described above, into a global integer variable *yylval.semantic_value*, and a global float variable *yylval.fvalue*. Lex will write yylex for you, using the patterns and rules defined in your lex input file (which should be called lexer.l). Your rules must include the code to maintain *yyline*, *yycolumn* and *yylval*.

You are to write a routine Lex_error that takes a message and line and column numbers and reports an error, printing the message and indicating the position of the error. You need only print the line and column number to indicate the position.

The #define mechanism should be use to allow the lexical analyzer to return token numbers symbolically. In order to avoid using token names that are reserved or significant in C/C++ or in the parser, the token names have been specified for you in the subsequent table. The parser and the lexical analyzer must agree on the token numbers to ensure correct communication between them. The token number can be chosen by you, as the compiler writer, or, by default, by *Yacc*. Regardless of how token numbers are chosen, the end-marker must have token number 0 or negative, and thus, your lexical analyzer must return a 0 (or a negative) as token number upon reaching the end of input.

Error Handling

Your lexical analyzer should recover from all malformed lexemes, as well as such things as string constants that extended across a line boundary or comments that are never terminated.

Due date and materials to be handed in

The assignment is due on Thursday, Feb 6. Test cases named hello.pasc, sieve.pasc, hist.pasc, error.pasc, try.pasc, and lexerror.pasc can be found the class web page. A sample driver file, driver.c, that calls yylex and prints each token with its value as the input is scanned, can also be found in the class web page. This file is for your reference. You should either use driver.c to test your lex program or make our output EXACTLY the same (except the column number in error messages) as the sample lexical analyzer provided (5 points in the grade).

Note: Do not ask questions about how to make driver.c work with your lex code. It is YOUR choice and YOUR job if you decide to use the code. The reason that driver.c is provided is to show how the output is produced so that you can do exactly the same. Nonetheless, for the ones who understand lex and this project, using driver.c will save some coding and guarantee output matching if all other logics are correct.

Grading policy:

- Recognizing all tokens (in the right order) (60). Identifiers, integer constants, float constants, char constants, and string constants will account for 40 points.
- Conversion of the special characters (5)
- Report errors with line and column numbers (10).
- Non-recognizable characters error (5)
- Non-closed comment error (5)
- String constant over the line boundary error (5)

- Output match with the sample output (5)
- Makefile, README file, and other documentary (5)

Hints

- Make sure that you understand lex in the first week of the project and start coding for the project in the first week. If you start learning lex in the second week, you will feel the deadline pressure and get frustrated constantly. The deadline will NOT be postponed.

Table 1: Token numbers to be returned by yylex

Token	Symbolic name	Token	Symbolic name
end of file	EOFnumber	else	ELSEnumber
;	SEMInumber	elsif	ELSIFnumber
:	COLONnumber	end	ENDnumber
,	COMMMAnumber	endif	ENDIFnumber
.	DOTnumber	endloop	ENDLOOPnumber
(	LPARENnumber	endrec	ENDRECnumber
)	RPARENnumber	exit	EXITnumber
$<$	LTnumber	for	FORnumber
$>$	GTnumber	forward	FORWARDnumber
$=$	EQnumber	function	FUNCTIONnumber
$-$	MINUSnumber	if	IFnumber
$+$	PLUSnumber	is	ISnumber
$*$	TIMESnumber	loop	LOOPnumber
..	DOTDOTnumber	not	NOTnumber
:=	COLEQnumber	of	OFnumber
<=	LEnumber	or	ORnumber
>=	GEnumber	procedure	PROCEDUREnumber
<>	NEnumber	program	PROGRAMnumber
identifier	IDnumber	record	RECORDnumber
integer constant	ICONSTnumber	repeat	REPEATnumber
float constant	FCONSTnumber	float	FLOATnumber
char constant	CCONSTnumber	return	RETURNnumber
string constant	SCONSTnumber	then	THENnumber
and	ANDnumber	to	TOnumber
array	ARRAYnumber	type	TYPEnumber
begin	BEGINnumber	until	UNTILnumber
constant	CONSTnumber	var	VARnumber
div	DIVnumber	while	WHILEnumber
downto	DOWNTOnumber	print	PRINTnumber
integer	INTnumber		