CS 2210 Programming Project (Part I)

January 9, 2019

Lexical Analyzer

In this phase of the project, you will write a lexical analyzer for the CS 2210 programming language, MINI-JAVA. The analyzer will consist of a scanner, written in LEX, and routines to manage a lexical table, written in C. The rest of the compiler project will communicate with these program modules.

Due date

The assignment is due **January 30, 2019** at the beginning of the class.

Token specification

Figure 1 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 (upper or lower case) letters or digits, beginning with a letter. Identifiers are case sensitive (i.e. the identifier **ABC** is different from **Abc**). 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 strings taken together (the size of the string table). If defined, these limits should be defined as follows.

```
#define LIMIT1   500
#define LIMIT2   4096
```

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 base 10 number. A string constant is a sequence of characters surrounded also by single quotes, e.g. 'Hello, world'. 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 support by the MINI-JAVA language are $\backslash n$ for newline, $\backslash t$ for tab, $\backslash'$ for the single quote and $\backslash\backslash$ for the backslash. Any other character following a backslash is not treated as 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 about some tokens to the parser (the lexeme). This extra information is passed to the parser as a single value, namely an integer, through a global variable as described below. For integer constants, the numeric value of the constant is passed. In order to allow other passes of the compiler to access the original identifier lexeme, the lexical analyzer passes an integer uniquely identifying an identifier (other than reserved words). 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 (pointer) into a *string table* created by the lexical analyzer to record the lexemes. Same identifiers should return the same index.

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. In the case of integer and string constants, store the value into the global integer variable *yylval*. *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*.

In the case of identifiers and string constant, yylval contains an index into the string table that contains the real string followed by a null('\0') character. The same index should be returned for the same identifier that appear at different places. Similarly the same index is returned for the same string. Also identifiers and string constants need not be differentiated in the string table (i.e. abc and "abc" can have the same index in the string table).

Reserved words may be handled as regular expressions or stored as part of the id table. For example, reserved words may be pre-stored in the string table so your program can determine a reserve word from an identifier by the section of the table in which the lexeme is found. Efficiency should be a factor in the management of the lexical and string table.

You are to write a routine *ReportError* 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 used to allow the lexical analyzer to return token numbers symbolically. In order to avoid using token names that are reserved or significant in C or in the parser, the token names have been specified for you in Figure 1.

The parser and the lexical analyzer must agree on the token number to ensure correct communication between them. The token numbers can be chosen by you, as the compiler writer, or, by default, by *Yacc* (a parser generator to be used in the next assignment). The default token number for a literal character (i.e. a one-character token) is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257 in the order they are declared in your Yacc specification. Regardless of how token numbers are chosen, the end-marker must has token number 0 or negative, and thus your lexical analyzer must return a 0 (or a negative) as a token number upon reaching the end of input.

Token name	Symbolic Name	Token Name	Symbolic Name
ANDnum	&&	CLASSnum	**class**
ASSGNnum	:=	COMMAnum	,
DECLARATIONnum	**declarations**	DIVIDEnum	/
DOTnum	.	ELSEnum	**else**
ENDDECLARATIONSnum	**enddeclarations**	EQnum	==
EQUALnum	=	GEnum	>=
GTnum	>	ICONSTnum	*integerconstant*
IDnum	*identifier*	IFnum	**if**
INTnum	int	LBRACEnum	{
LBRACnum	[	LEnum	<=
LPARENnum	(	LTnum	<
METHODnum	**method**	MINUSnum	-
NEnum	!=	NOTnum	!
ORnum	||	PLUSnum	+
PROGRAMnum	**program**	RBRACEnum	}
RBRACnum	]	RETURNnum	**return**
RPARENnum	)	SCONSTnum	*stringconstant*
SEMInum	;	TIMESnum	*
VALnum	**val**	VOIDnum	**void**
WHILEnum	**while**	EOFnum	*end of file*

Figure 1: Defined Tokens in Mini Java.

Temporary driver

In order to test your lexical analyzer without a parser, you will have to write a simple driver program which calls your lexical analyzer and print each token with its value as the input is scanned. For ease in combining the lexical analyzer and parser in the second assignment, the lexical analyzer function should be put in a file by itself. The following shows the structure of a driver. If you use it, please remember to break from the endless loop after recognizing the end of file token.

```
main()
{...    while (1) {
                switch (yylex()) {
                case ICONSTnum: ...
                }
        }
...}
```

Error handling

Your lexical analyzer should recover from all malformed lexemes, as well as such things as string constants that extend across a line boundary or comments that are never terminated. **Specifically**, an identifier which starts with a digit is considered to be an error and should be reported.

An example program with output

The program:

```
/* Example 1: A hello world program */
program xyz;
class Test {
    method void main() {
        System.println('Hello World !!!');
    }
}
```

The output of Lexical Analyzer:

Line	Column	Token	Index_in_String_table
2	8	PROGRAMnum	
2	12	IDnum	0
2	13	SEMInum	
3	6	CLASSnum	
3	11	IDnum	4
3	13	LBRACEnum	
4	11	METHODnum	
4	16	VOIDnum	
4	21	IDnum	9
4	22	LPARENnum	
4	23	RPARENnum	
4	25	LBRACEnum	
5	15	IDnum	14
5	16	DOTnum	
5	23	IDnum	21
5	24	LPARENnum	
5	41	SCONSTnum	29
5	42	RPARENnum	
5	43	SEMInum	
6	6	RBRACEnum	
7	2	RBRACEnum	
8	1	EOFnum	

String Table : xyz Test main System println Hello World !!!

Assignment submission

When you are done, create a gzipped tarball of your commented source files. You **must** include a file that shows how to compile/execute your code – named *Readme.txt*. Preferably, include a makefile named *Makefile*. The submission should be a compressed file that contains your project source code and readme (no executable please). On Linux, this can be done with the command "tar zcvf USERNAME_proj1.tar.gz *". Copy your archive to the directory: ∼wahn/submit/2210/.

Make sure you name the file with your username, and that you have your name in the comments of your source file. Note that this directory is insert-only, you may not delete or modify your submission once in the directory. You can however check that the file has been copied by listing the contents of that directory. If you have made a mistake, before the deadline, resubmit with a number suffix like USERNAME_proj1_2.tar.gz. Your most recent submission will be graded.