Concepts of Programming Languages, CSCI 305, Fall 2021
Lexical Analyzer for MiniC

Updated: Nov. 12, 2021

Sept. 13 – Regular expression for each token and for comments, 10%
Oct. 8 - DFA, 10%
Oct. 22 - Lexical analyzer in Python, along with human readable table, 40%
Nov. 12 - Lexical analyzer in C#, along with human readable table, ~~20%~~ 40%
Dec. 3 - Lexical analyzer in Scheme, along with human readable table, ~~20%~~ , Extra credit, a maximum of 10% points on your overall course grade.

MiniC

"C programming is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system. C is the most widely used computer language. It keeps fluctuating at number one scale of popularity along with Java programming language, which is also equally popular and most widely used among modern software programmers."[1]

https://www.tutorialspoint.com/cprogramming/index.htm

Facts about C

- C was invented to write an operating system called UNIX.
- C is a successor of B language which was introduced around the early 1970s.
- The language was formalized in 1988 by the American National Standard Institute (ANSI).
- The UNIX OS was totally written in C.
- Today C is the most widely used and popular System Programming Language.
- Most of the state-of-the-art software have been implemented using C.

Sample C and MiniC Program

```
int main() {
  /* my first program in C */
  printf("Hello, World! \n");

  return 0;
}
```

MiniC is a subset of C. This program is also legal in MiniC.

Semicolons

In a MiniC program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

[1] Much of this information is taken directly from https://www.tutorialspoint.com/cprogramming/index.htm. To avoid clutter, for the remainder of the document, direct quotes will not be set off in quotes.

Comments

Comments are ignored by the MiniC compiler so should not be returned by the lexical analyzer. They start with /* and terminate with the characters */. ~~You cannot have comments within comments and comments cannot occur within a string or character literals.~~

Any characters allowed in the language is also allowed within comments.

Identifiers

A MiniC identifier is a name used to identify a variable, function, or any other user-defined item. An identifier must start with a letter A to Z, a to z, or an underscore '_'. The remainder of the identifier name can contain characters a-z, A-Z, _ or digits. Identifiers names cannot be reserved words (e.g. return, int, float, t, f etc.).

MiniC does not allow punctuation characters such as @, $, and % within identifiers. C is a case-sensitive programming language. Thus, *Manpower* and *manpower* are two different identifiers in C.

Keywords/Reserved Words

Keywords have a special meaning in a language, and are part of the syntax. Reserved words are words that cannot be used as identifiers (variables, functions, etc.), because they are reserved by the language. In practice most keywords are reserved words and vice versa. In MiniC, and C, "t", "f", "T" and "F" are not keywords, but they are reserved words. The lexical analyzer should return boolLiteral (see the list of tokens below) for "t", "f", "T" and "F". The remaining reserved words may not be used as constants or variables or any other identifier names. In general, these keywords must be recognized by the lexical analyzer and they correspond to tokens with the same name.

break	else	long	switch
bool	false	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

Whitespace in MiniC

Whitespace is the term used in MiniC to describe blanks (0x20), tabs (0x09), newline characters (0x0d, 0x0a) and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement

```
int age;
```

there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement fruit = apples + oranges;

no whitespace characters are necessary between fruit and =, or between = and apples.

Line Endings in MiniC

Text files created on DOS/Windows machines have different line endings than files created on Unix/Linux. DOS uses carriage return and line feed ("\r\n", or 0x0d 0x0a) as a line ending, while Unix uses just line feed ("\n", 0x0a). You can assume that all MiniC programs will be written on Windows so the “\r\n” sequence will be used.

MiniC programs

MiniC programs contain one function, called ‘main’, which is a void function or returns an integer. A void function has no return value. This function is called by the operating system and the integer returned indicates whether the program executed correctly or not. MiniC programs may have additional function definitions.

Tokens in MiniC programs may be surrounded by any number of whitespace characters, including none. Identifiers, Booleans, character literals, string literals, number literals are delimited by whitespace, semicolon, opening and closing parentheses (opening before another token and closing after another token), operations, comparators and more.

Tokens in MiniC programs

The following tokens are in MiniC programs so must be recognized by the lexical analyzer. In addition, the lexical analyzer must recognize each of the keywords. The lexical analyzer will need to recognize comments and whitespace, so they can be thrown away.

Token Name	Description	Symbol
addOp	Addition and subtraction binary operators	+, -
assignOp	Assignment operators	=, +=, -=, *=, /=
boolLiteral	Boolean value	t, T, f, F
char	Character	'a ', 'B ', '\\ ', '\ ' ', '\"'
comma	Comma	,
comparator	Comparison operator	~~=,~~ >, >=, <, <=, ==, !=
floatLiteral	Floating point value	3.14159, 314159E-5
incrOp	Increment or decrement operator	++, --
identifier	Identifier	main, printf, _x, myFunc
intLiteral	Integer value	0, 212, 215u, 0xFeel, 075, 30ul
lbrace	Left brace	{
logicalNot	Logical unary operator	!
lparen	Left parenthesis	(

multOp	Multiplication and division operators	*, /
rbrace	Right brace	}
rparen	Right parenthesis	)
semicolon	Semicolon	;
string	string	"Hello\tWorld\n\n", "hello, dear", "hello, "

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal. An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. Notice that integer literals in MiniC cannot be signed.

Examples:

```
212         /* Legal */
215u        /* Legal */
215Lu       /* Legal */
0xFeeL      /* Legal */
0           /* Legal */
00          /* Legal but 0 is octal */
000         /* Legal but 00 is octal */
012         /* Legal but 12 is octal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
-212        /* Not legal, MiniC does not allow signed numbers */
```

Other examples of integer literals:

```
85          /* decimal */
0213        /* octal */
0x4b        /* hexadecimal */
30          /* int */
30u         /* unsigned int */
30l         /* long */
30ul        /* unsigned long */
```

Float Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. While representing floating point literals, you must include the decimal point, the exponent, or both. When giving the decimal point, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

The integer part, fractional part and exponential part, if given, must be in decimal without the signed or long indicator. While neither integer literals or float literals can be signed in MiniC, the exponent portion of exponential numbers may be signed.

Examples:

```
3.14159       /* Lega, decimal point only */
314159E-5     /* Legal, exponent only */
0.0           /* Legal, */
0.0e0         /* Legal, both decimal point and exponent */
000.0         /* Legal */
12.           /* Legal */
.12           /* Legal */
.12E-5        /* Legal */
12.E-5        /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

Character Literals

Character literals are enclosed in single quotes, e.g., 'x'. These can be a plain character (e.g., 'x'), or an escape sequence (e.g., '\t').

Escape sequences allowed:

Escape sequence	**Meaning**
\\	\ character
\'	' character
\"	" character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline, also form feed
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

While the whitespace characters space (0x20), horizontal tab (0x09), carriage return (0x0d) and line feed (0x0a) may appear in MiniC source files, they cannot be characters in MiniC.

String Literals

String literals or constants are enclosed in double quotes ". A string optionally contains characters that are similar to character literals: plain characters, escape sequences, and space (0x20). While the whitespace characters space (0x20), horizontal tab (0x09), carriage return (0x0d) and line feed (0x0a) may appear in MiniC source files, only the space (0x20) can appear within strings.

Legal Characters

a-z, A-Z, 0-9, period, comma, underscore, +, -, *, \, <, >, !, =, /, ', ", (,), {, }, ;, space (0x20), horizontal tab (0x09), carriage return (0x0d) and line feed (0x0a).

The plain characters:

a-z, A-Z, 0-9, period, comma, underscore, +, -, *, \, <, >, !, =, /, ', ", (,), {, }, ;

along with whitespace:

space (0x20), horizontal tab (0x09), carriage return (0x0d) and line feed (0x0a)

are the only characters allowed in MiniC programs. An error should be flagged if a character not on this list appears in MiniC source code.

Sample MiniC Program with Output

Here is a sample MiniC program:

```
int  main(int argc)  {
        int x, y, z;
        x=0;
        y=1;
        z = sum(x,y);
        z += mult(x,y);
        return z;
}

int sum(int a, int b) {
        return a+b;
}

int mult(int a, int b) {
        return a*b;
}
```

```
Output:
Token: int              Lexeme: int
Token: identifier       Lexeme: main
Token: lparen           Lexeme: (
Token: int              Lexeme: int
Token: identifier       Lexeme: argc
Token: rparen           Lexeme: )
Token: llbrace          Lexeme: {
Token: int              Lexeme: int
Token: identifier       Lexeme: x
Token: comma            Lexeme: ,
Token: identifier       Lexeme: y
```

```
Token: comma          Lexeme: ,
Token: identifier     Lexeme: z
Token: semicolon      Lexeme: ;
Token: identifier     Lexeme: x
Token: assignOp       Lexeme: =
Toke: intLiteral      Lexeme: 0
Token: semicolon      Lexeme: ;
Token: identifier     Lexeme: y
Token: assignOp       Lexeme: =
Toke: intLiteral      Lexeme: 1
Token: semicolon      Lexeme: ;
              .
              .
              .
```

Assignment

Create a table-driven lexical analyzer for MiniC which uses a scanning table to identify and remove comments, and to identify tokens. Once an identifier is found, a keyword/reserved word lookup table is allowed to identify these tokens. To create the scanning table, begin by defining regular expressions for each token. From that create an NFA that recognizes all tokens, a DFA, a minimal DFA, and the scanning table.

The interface to your Python, C# and Scheme programs should:

Purpose of the lexical analyzer: Describe the purpose of the lexical analyzer and how to use it.

Paths to relevant tables: Obtain paths to a scanning, token and keyword table. Allow the user to start the program without selecting any files (i.e. using tables at default locations). However, also allow the user to specify alternate paths for one or more of the tables. Provide a file picker so the user is not required to type the path of the files. Inform the user if scanning is attempted but one or more of the tables is not present. Provide useful error messages.

Perform multiple tests: Allow the user to repeatedly choose a test file containing a MiniC program. If the user is in the middle of scanning a MiniC program when they request scanning a different MiniC program, warn the user that scanning the current program is not complete and all the user to choose what they want to do.

Token by token: When a MiniC program is being scanned, allow the user (eventually this will be the parser) to repeatedly request a token, until the MiniC program has been scanned or the user decides to quit scanning. After each request, your program should display the token identified and the lexeme associated with the token, and allow the user to request the next token.

Source code errors: When the analyzer encounters an error in the MiniC program that it is scanning, the analyzer should display an error message that it helpful to the owner of the MiniC program. The error should display what character(s) caused the error. The analyzer should allow the user to continue scanning the program, once the error message has been displayed.

EOF in source code: Recognize the end of the source program and inform the user that the end of file has been reached.

For full credit:

- Your program must be well commented. It should use descriptive variable, class and method names. Comments should describe each class and the methods within the classes. The inputs and outputs of a method should be described.
- Your program should be well designed.
- Your tables should not be hard coded.
- Paths to your tables are not hard coded (see "Paths to relevant tables" above.)

You cannot use the Unix tool lex for this assignment.