CSE 4713/6713, Programming Languages
Fall 2010
Lab 5
Due: 30 Nov 2010

Introduction:

In this assignment you will finish your LOL interpreter. You have implemented the following components so far:

- lexical analyzer (lexer, or tokenizer)
- syntax analyzer (parser)
- symbol table manager
- type-checking
- operator associativity and precedence rules
- sequential, selection, and repetition flow control
- input/output

At this point you should have a working interpreter. However, to make our mini-language truly useful we would need to add additional features, such as implementing arrays, functions, and pointers.

Since we don't have enough time to add everything we might like, this semester we will just implement *arrays*.

Implementing array storage:

An array declaration in LOL would be handled by the following syntax rules:

```
2.  <declaration> ::=  <type> IDENT

3.  <type> ::= <simple_type> | <array_type>

4.  <simple_type> ::=   "int" | "float" | "char"

5.  <array_type> ::=  "array" <simple_type> "[" INTLIT { "," INTLIT } "]"
```

In LOL, we can declare an integer array called *myArray* as follows:

```
array int [ 100 ] myArray
```

This array is a single-dimensioned array which holds 100 integers. By default, arrays in LOL are indexed from 0 to *n-1*, so this array is indexed from 0 to 99.

Designators in LOL can be either simple variables or indexed variables, as per the following syntax rule:

```
12.  <designator> ::= IDENT [ "[" <expression> { "," <expression> } "]" ]
```

Assigning a value to an array element would be done using the standard LOL assignment statement:

```
9. <assignment_stmt> ::=  <designator>  "=" <expression>
```

Examples:

```
myArray [ 3 ] = 42
myArray [ i ] = j + 1
```

You should add code to your interpreter to perform the memory allocation necessary to handle arrays when an array declaration is encountered in LOL code.

You will need to modify your symbol table to be able to store the required information for arrays. Your symbol table is already set up to store the base address of the array and the type of the elements stored in the array, so all you need to add is the number of dimensions of the array, and the range (lower and upper index bounds) of each dimension.

We also know the size (number of bytes) that an object of a specific data type occupies. Given knowledge of these parameters, we can write an access function for our compiler to use in indexing into various arrays.

To keep things simple, this semester we will restrict our language to being able to handle only 1-D and 2-D arrays.

Implementing array indexing:

You should add code to your interpreter to permit the LOL programmer to access any element in an array, using row-major ordering. See Appendix B for a discussion of row-major ordering, and for a formula to compute the address of any element in an array.

Write and run your own test case(s) in which you demonstrate that your program correctly handles 1-D and 2-D arrays. Your program should: 1) declare an array, 2) initialize and/or assign values to (at least some of) the elements of the array, and 3) print out the values of (at least some of) the elements of the array.

The Assignment:

Modify your LOL interpreter to implement arrays.

Run your version of the LOL interpreter on my test file, listed in Appendix A below. Print the output.

Run your version of the LOL interpreter on your own test file. Print the output.

Deliverables:

On or before the due date/time:

(1) Email to your instructor and TA a zip file with the source code of all your program files and test files. Also include your lab report document file.

(2) Your lab report should include:

- header page: including class name/number, assignment number, your name, due date, and actual date you are turning in the lab report
- first page: specifying whether you are submitting UNIX code or code for a PC, as well as what compiler should be used, and also including any instructions necessary to compile and execute your code.
- print out of my test file (the example program listed in Appendix A)
- print-out of the output produced by your interpreter for my test file

- print-out of your own LOL program test file
- print-out of the output produced by your interpreter for your own test file, demonstrating that your interpreter correctly handles 1-D and 2-D arrays
- discussion: brief discussion of results, including whether your test cases exhaustively test your arrays, whether your program behaves as expected, problems encountered, etc.

Appendix A:

Example LOL program illustrating the use of arrays:

```
 int lowerBound
 int upperBound
 int i
 int j
 array int [ 10 ] myArray

 lowerBound = 0
 upperBound = 9
 i = lowerBound
while ( i <= upperBound ) {
   myArray [ i ] = 0
  write (  " myArray [ " , i , " ] = " , myArray [ i ] )
   i = i + 1
}

 myArray [ 5 ] = 15
 i = 6
write ( " Please enter an integer value. " )
read ( j )
 myArray [ i ] = j
 myArray [ 7 ] = myArray [ i ] + myArray [ i - 1 ]
if ( myArray [ 7 ] < 100 ) {
  write ( " myArray [7] = " , myArray [ 7 ] )
}

 i = lowerBound
while ( i <= upperBound ) {
  write ( " myArray [ " , i , " ] = " , myArray [ i ] )
   i = i + 1
}
```

Appendix B: Row-Major Order Array Addressing

High-level programming languages facilitate the creation of both single- and multi-dimensional arrays.

Finding the address of an element of a single-dimension array is easy, since memory is also a 1-dimensional array. Each array has a starting address, and the index is added to the starting address to get the address of the element in the array (assuming that the first index is 0, and that each element is just a single byte).

For example, assume you have an array called CharArray, a single-dimension array of ASCII characters (1 byte per character), indexed from 0 to 255. It would be declared in Pascal as:

```
var CharArray : array  [0..255] of char;
```

If its base address (the address of the first element in the array) is 1000, then the address of CharArray[0] is 1000, the address of CharArray[1] is 1001, the address of CharArray[2] is 1002, and you can see the pattern. The address of CharArray[10] is 1000 + 10, or 1010. Our formula is very simple:

```
address of Array[i] = base_address + i
```

What if we have an array called RealArray, a single-dimension array of floating-point numbers (4 bytes per number), indexed from 0 to 100? Assume its starting address is 1000. In that case, array element RealArray[1] is located 4 bytes beyond the starting address, at location 1004; RealArray[2] is located 8 bytes beyond the starting address, at 1008; and so on. Our formula is:

```
address of Array[i] = base_address + i * data_type_size_in_bytes
```

Finally, suppose RealArray is defined in Pascal as follows:

```
var RealArray : array  [2..8] of real;
```

Here our lower bound of the array is not 0; instead, it is 2. This means that RealArray[2] is NOT located 8 bytes beyond the starting address, at 1008; rather, it is located at the starting address, 1000, because it is the first element in the array. It is RealArray[4] which is located at 1008. It is easy to see that here we need to subtract the lower bound of our array from the index of the element, i, that we want to find the address of, and then multiply the result of the subtraction by the data type's size in bytes in order to find our offset from the starting address. So our formula becomes:

```
address of Array[i] = starting_address + (i – lower_bound) *
size_in_bytes
```

Multi-dimensional arrays are more complicated. Suppose we have a 2-dimensional array (like a matrix, with rows and columns). Since memory is really 1-dimensional, we somehow must constrain the 2-dimensional array into a 1-dimensional memory structure. We will still have a starting address for the array, and the array will still take up as many bytes of memory as before, but we need to map the 2 dimensions into a single one.

There are two standard methods to do this: row-major order and column-major order. In row-major order, the rows in the 2-D array are stored in order. Thus, the first row of the 2-D array is put into memory beginning at the starting address, and then the next row is taken and put into memory, and so on until all rows are stored. In column-major order, the array columns are stored in successive order. Most programming languages, including C and Java, store arrays in row-major order, but some languages, such as Fortran, store them in column-major order.

Suppose array A is defined in Pascal as follows:

```
var A : array  [2..4, 6..9] of real;
```

In this array there are 3 rows and 4 columns. It would be indexed as:

A[2,6]	A[2,7]	A[2,8]	A[2,9]
A[3,6]	A[3,7]	A[3,8]	A[3,9]
A[4,6]	A[4,7]	A[4,8]	A[4,9]

If we are storing this array in (one-dimensional) memory in row-major order, it would be stored in the following order (with the memory addresses increasing from left to right):

A[2,6]	A[2,7]	A[2,8]	A[2,9]	A[3,6]	A[3,7]	A[3,8]	A[3,9]	A[4,6]	A[4,7]	A[4,8]	A[4,9]

Note that, in row major order, the numbers change like an odometer on a car – the rightmost number changes most rapidly.

What if we have a multi-dimensional array of more than two dimensions? For multi-dimensional arrays, row-major order always stores arrays with the leftmost index being the first grouping, and so on to the right. The rightmost index again varies the fastest. Column-major order processes the indices right to left, with the leftmost index varying the fastest.

Just to be clear, let's look at a small 3-D array. Suppose we have a 3-D array, B, and array B is defined in Pascal as follows:

```
var B : array  [2..4, 5..6, 8..9] of real;
```

In this array there are 3 rows, 2 columns, and 2 planes. It would be indexed as:

B[2,5,8]	B[2,6,8]		B[2,5,9]	B[2,6,9]
B[3,5,8]	B[3,6,8]		B[3,5,9]	B[3,6,9]
B[4,5,8]	B[4,6,8]		B[4,5,9]	B[4,6,9]

If we are storing this array in row-major order, it would be stored in memory in the following order (where memory addresses increase from left-to-right):

B[2,5,8]	B[2,5,9]	B[2,6,8]	B[2,6,9]	B[3,5,8]	B[3,5,9]	B[3,6,8]	B[3,6,9]	B[4,5,8]	B[4,5,9]	B[4,6,8]	B[4,6,9]

So, what is the general formula for computing the address of element i in a 3-dimensional array? Assume that the array index variables are *row*, *column* and *plane* for dimensions 1, 2, and 3, respectively. Assume:

```
base_address = address of Array[lower_bound_row, lower_bound_column, lower_bound_plane]
size_row = ( upper_bound_row – lower_bound_row ) + 1 ; similarly for column and plane
bytes  = number of bytes occupied by each element
```

Then

```
address of Array[i, j, k] = base_address + bytes * offset
```

where

```
offset =  ((i – lower_bound_row) * size_plane + (j – lower_bound_column))  *
size_column + (k  - lower_bound_plane)
```

It may be easier to understand this formula if `offset` is rewritten as:

```
(i – lower_bound_row) * size _column * size_plane  +
(j – lower_bound_column) * size_plane +
(k  - lower_bound_plane)
```

Let's see if this works. Assume a base address of 2103 for the first element of array B above. The offset of B[3, 6, 8] in the array above should be:

```
(i – low_row) * size _col * size_plane  + (j – low_col) * size_plane + (k  - low_plane)

( 3 – 2 )   *  2     *  2       +  ( 6 – 5  ) *  2      + ( 8 -  8   )=

               4                +           2           +      0    = 6
```

Assume a base address of 2103 for the first element of array B above, and that each element requires 1 byte of storage. Then the address of the first element, B[2, 5, 8] is 2103. By our formula, the address of element B[3, 6, 8] is:

```
address of Array[i, j, k] = 2103 + 1 * 6 = 2109
```

You can count it out on the memory representation of the array above and see that this formula is correct. If we are storing 4-byte real numbers instead of 1-byte ASCII characters, then bytes = 4, and the address of element B[3, 6, 8] will be 2103 + 24 + 2127.

Finally, we can give a universal formula for computing the address of an element in an array of any arbitrary dimension *d* which is stored in row-major order in linear memory. Assume that the indices are represented by $index_1$, $index_2$, . . ., $index_d$, and that the size of each dimension is given by $size_1$, $size_2$, . . . $size_d$. Then:

$$\text{address of Array}[index_1, index_2, \ldots, index_d] = \text{base\_address} + \text{bytes} * \text{offset}$$

where

$$\text{offset} = (index_d - lower\_bound_d) + size_d * ((index_{d-1} - lower\_bound_{d-1}) + size_{d-1} * (\ldots + size_2 * (index_1 - lower\_bound_1)))$$

CSE 4713/6712 Programming Languages
Lab 5 Grading Form

Title page:	
	Include class name/number, assignment number, your name, due date, and actual date you are turning in the lab report [5 pts]
Compilation instructions:	
	Include a page specifying the compiler used and instructions for how to compile. [5 pts]
Copy of my test file:	
	Include a print out of my test file (the example program listed in Appendix A). [5 pts]
Output for test file # 1:	
	Include a print-out of the output produced by your interpreter for my test file. [25 pts]
Copy of your test file:	
	Include a print-out of your own LOL program test file. [5 pts]
Output for test file #2:	
	A print-out of the output produced by your interpreter for your own test file which demonstrates that your interpreter can implement 1-D and 2-D arrays correctly:
	Print-out demonstrates example of 1-D array [30 pts]
	Print-out demonstrates example of 2-D array [10 pts]
	Label the relevant sections of your output that demonstrate these program components. [5 pts]
Discussion	
	Brief discussion of results, including whether your test cases exhaustively test arrays, whether your program behaves as expected, problems encountered, etc. [10 pts]
Lateness:	
	Take off 10 points for each day the lab report was late.

Total: _____