

CS245 Lab 4: Symbol Tables & Memory Allocation

In this lab, you will be extending your `translator` project to produce translations for expressions with variables. As in the previous lab, you are not required to do any lexical analysis beyond simply reading each token with my scanner files. Thus, you do not have to place any restrictions on the variable names. (Note that variable names in Scheme can include a variety of characters that would not be allowed in a C++ variable name, such as “-”). As in the previous lab, all the test input will have spaces between consecutive tokens.

Your translator should identify, and report an error for, illegal variable uses and declarations. For this lab, you may limit the number of declarations in any one “`let`” to two.

Your translator should convert any references to variables into references to a global array called `MEM`. You may assume that `MEM` will be large enough to hold all the variables in any input program you are asked to translate. Each variable initialization should be followed by a “`,`”, so that the output of your translator is a legal C++ expression (my test program will declare and initialize the `MEM` array, and provide the “`input`” function, so that it can run your result).

As in the previous lab, you may include unnecessary parentheses in your output, but you must include enough parenthesis to ensure that your C++ output will produce the same value that the input would produce in Scheme.

Thus, the following examples should work as shown below:

INPUT	TRANSLATED RESULT (or <i>error</i>)
<code>6 6</code>	<code>6 6</code>
<code>(+ 5 1)</code>	<code>5 + 1</code>
<code>(let ((a 5)) a)</code>	<i>a undeclared variable “a”</i>
<code>(let ((a (input “enter_a”))) a)</code>	<code>MEM[0]=5, MEM[0]</code>
<code>(let ((a 5)) (+ a 1))</code>	<code>MEM[0]=5, MEM[0]+1</code>
<code>(+ (let ((a 5)) a) 1)</code>	<code>(MEM[0]=5, MEM[0]) + 1</code>
<code>(+ (let ((a 5)) a) a)</code>	<i>undeclared variable “a”</i>
<code>(let ((a 5) (b 2)) (+ a b))</code>	<code>MEM[0]=5, MEM[1]=2, MEM[0]+MEM[1]</code>
<code>(let ((a 5)) (let ((b 2)) (+ a b)))</code>	<code>MEM[0]=5, MEM[1]=2, MEM[0]+MEM[1]</code>
<code>(let ((a 5) (a 2)) (+ a a))</code>	<i>“a” is declared twice in the same let</i>
<code>(let ((a 5)) (let ((a 2)) (+ a a)))</code>	<code>MEM[0]=5, MEM[1]=2, MEM[1]+MEM[1]</code>
<code>(let ((a 5))</code>	
<code>(let ((b (+ a 1)))</code>	<code>MEM[0]=5, MEM[1]=MEM[0]+1, MEM[0]+MEM[1]</code>
<code>(+ a b)))</code>	

You will need to add a symbol table data structure that records variable names and locations in memory. For this course, you should perform your symbol table processing in the *imperative* paradigm rather than the pure functional approach: create a procedure that traverses your AST in the appropriate order, making changes to the AST as necessary (for example, adding information at a variable *use* about the memory location of that variable) — the procedure should update and use a symbol table as it goes (this can be passed around the tree as a reference parameter, or stored in a global variable (ugh)). In CS350 in the spring, we will discuss how this problem can be solved in the pure functional approach and via *aspect-oriented* programming.

For this lab, you should feel free to either create your own symbol table class, re-use something you wrote from CS206, research and use the C++ “standard template library”, or get a symbol table class from some other source, as long as you put in comments to make clear where you got your class.