From: Vandana Agarwal . (via Nalanda)
Sent: 10 April 2023 11:18
To: RAMAKANT PANDURANG TALANKAR .
Subject: 11551001318_CS F363_JAN_2023: Stage 2: Some updates and explanations

11551001318_CS F363_JAN_2023 » Forums » Announcements » Stage 2: Some updates and explanations

Stage 2: Some updates and explanations
by Vandana Agarwal . - Monday, 10 April 2023, 11:05 AM

1. Please use the 64-bit representation in NASM. The latest version of NASM is not stable as reported by the teams. Therefore, you are advised to use 2.14.02 which works well for the UBUNTU version used by us.

2. You can use standard printf and scanf functions in your generated code. Make sure that complete code as generated by your compiler should be in code.asm file. It is expected that the templates for printing of data of variables or reading data for variables of different data types, such as integer, real, boolean or array type, should be part of the assembly code of the function. Separate C files for such implementations will not be accepted.

3. Include in your test cases the semantic of verifying that at least one of the conditional variables used in while loop construct is assigned a value. If none of the conditional variables is assigned any value within the nested scope of while should be treated as error.

4. You can use one complete word for representing a boolean value. Use 1 for reading true value and 0 for reading false value. However, printing of the value of a boolean variable should be strings true and false, and not 1 and 0. Use messages comprising of strings appropriately.

5. All compile time errors should be reported till the end of the user source code. You should not exit the code after reporting few errors. Rather, keep reporting the errors till the end of user source code.

6. Scope information must be collected in three different forms - name of the function, pairs of line numbers of start and end for a scope, and nesting level. These should be reflected in the symbol table output as per the format given earlier.

7. A switch statement construct should be handled with extra care. If the type of the switch variable is an array or a real number, you should not visit its corresponding sub-tree for reporting of additional errors such as case value and the presence or absence of the default statement.

8. Implementing code generation for function calls using static or dynamic array variables is complex and students are advised to first complete other requirements and take up this only if they have sufficient code confidence and available time.

9. Error reporting should be line-wise if the error is present at that line. Any error due to the absence of the construct is needed to be reported using the scope lines of that block e.g. lines

22-45, none of the variables of while loop is assigned a value, or a default is missing in a switch case statement, and so on.

10. Placing of the input parameters of array type by the caller: The caller places ***three*** values implicitly for the array parameter (say A)

- the address of the first element of A (say B), and the
- values of lower and higher subranges (say m,n)

Static array: An input parameter of array type with integer range values (say 10 and 20, in an array variable declared as declare A: array[10..20] of integer) is required to pass information (by the caller to the callee) about the base address of the first element of the array along with the range values (low and high) for future bound checking by the callee. In total the information passed comprises of one address (base of A = base address of caller + C + offset of A) and two integer constants (low = 10 and high = 20 - available at compile time in the symbol table). Remember that the array elements data remains in the caller's memory and only its base reference is passed to the callee. This imitates pass by reference parameter passing method.

Dynamic array: Consider a dynamic array - declare A: array [low..high] of integer. Procedure almost remains same except that the range values (low and high) need to be copied by the caller from their associated memory locations (low and high values - accessed from locations of low and high, populated at run time) to the memory designated for the input parameters.

11. Calculation of width of array input parameters and local variables of array type (for symbol table as well as for code generation): This is calculated different from width of local variable of array type.

- Width of **input parameter** of array type - static or dynamic: 1(base address)+ 2*sizeof(int) [refer B, m and n above]
- Width of **local variable** of array type - **static**: 1(base address) + (high-low +1) * sizeof(array element)
- Width of **local variable** of array type- **dynamic:** 1(base address) [Note: we allow the elements to be populated after all fixed sized variables. Assumption: any array to be passed as a parameter to the callee should have been populated before the call].

Remember that the widths of local variables of array type are different for static and dynamic arrays.

See this post in context