Announcements

Stage 2: Some updates and explanations

◄ Stage 2: Deadline extended (Final)

Stage 2: Revised test cases (semantic analysis) ►

Display replies in nested form

Stage 2: Some updates and explanations
by Vandana Agarwal . . - Wednesday, 15 April 2020, 9:47 AM

1. You can use either the 64-bit or 32-bit representation in NASM based assembly code.

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 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. This will carry a total of only **4 marks** (out of total 45 marks reserved for stage 2).

Explanation for code generation for function calls using static or dynamic array variables

9. In code generation module, as discussed earlier about the placement of actual parameter values, place them after the local variables are placed. You can use stack pointer which points at the top of the activation record of the caller before it calls the callee function. The actual parameters, when caller calls the callee function, can be placed according to their relative offsets (starting with 0) from the stack pointer. Allow the stack to grow till all input and output parameters find their positions, and keep incrementing the stack pointer. Once the actual parameters are placed, then the callee's activation record can be started using current stack pointer contents. This stack pointer contents of the starting address of the callee, is then copied into the base pointer (of course after preserving base pointers previous content of starting address of the caller) for accessing local variables of the callee therafter. For more information, please refer the lecture details on code generation.

10. In view of 9, assign offset 0 to the first input parameter, and keep assigning for all input parameter in the sequence from left to right in terms of their positions in the user code. Then carry forward the current offset to use for the output parameters offset computation. I will upload a sample symbol table display soon so that the offsets initialization for parameters will be more clear. Additionally, the first local variable also gets a offset of 0 for access relative to the base pointer.

11. The crux of points 9 and 10 can be understood in the following way.

Caller: Accessing of physical locations for placing the actual parameters by the caller is relative to stack pointer content (after all local variable) and accessing of local variables is relative to its base pointer.

Callee: Accessing of physical locations for using the values placed by the caller in terms of actual input parameters or for populating the output parameters by the callee is relative to its current base pointer content (these locations are even before callee's activation record stared, basically these are the locations common to both caller and callee) and accessing of local variables of the callee is relative to its base pointer as was the case with caller.

12. Placing input parameters of array type by the caller:

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.

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 (l**ow and high values - accessed from locations of low and high, populated at run time**) to the memory designated for the input parameters.

13. **Width of array input parameter (for symbol table as well as for code generation)**: This should be 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)**
- 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]

I will upload the sample symbol able for offsets of array variables - declared as input parameter and as local variable. Remember that widths of local variables of array type are different for static and dynamic arrays as was discussed in the class.

Please inform me if any point is not clear.

Permalink

◄ Stage 2: Deadline extended (Final) Stage 2: Revised test cases (semantic analysis) ►

Contact us : nalanda@pilani.bits-pilani.ac.in, Location : Media Lab, Room Number : 3231, SDET, FD-III, Pilani Campus