Announcements

Stage 2: Driver, Makefile and execution details

◄ Stage 2: Code generation test cases

Stage 2: Deadline extended (Final) ►

Display replies in nested form

Stage 2: Driver, Makefile and execution details
by Vandana Agarwal . . - Monday, 13 April 2020, 8:44 AM

1. Driver

Your driver must have the following **TEN** choices

Press option for the defined task

0. To exit from the loop (Ask for the choices in a loop)

1. Lexer: For printing the token list generated by the lexer (on the console)

2. Parser: For parsing to verify the syntactic correctness of the input source code and to produce parse tree (On Console)

3. **AST:** For printing the Abstract Syntax Tree in appropriate format. Also specify the traversal order at the beginning. (On Console)

4. Memory: For displaying the amount of allocated memory and number of nodes to each of parse tree and abstract syntax tree for the test case used. The format should be as per the example given below

Parse tree Number of nodes = 150 Allocated Memory = 1024 Bytes

AST Number of nodes = 30 Allocated Memory = 200 Bytes

Compression percentage = ((1024-200)/1024)*100

(use sizeof() to compute size of allocated memory while allocate memory during construction of these trees)

5. Symbol Table: For printing the Symbol Table giving following information (ten in number) for each variable identifier at each line using formatted output.***[Use width of variables of type integer as 2, of real as 4 and of boolean as 1 for printing the symbol table]***

- *Variable name*
- *Scope - module name*
- *scope - line number pairs of start and end of the scope*
- *width (if a variable is of array type, then add 1 to total requirement for all elements of an array for holding address of the first element)*
- *is array*
- *if array, whether static or dynamic*
- *if array, range variables or number lexemes (e.g. [m, n], [p, q], [10, 20] etc.)*
- *type of element*
- *offset*
- *nesting level (for an input or output parameter, level= 0, local variable in function definition, level = 1, any variable inside a nested scope should get its level incremented appropriately)*

For example (not specific to any test case, mentioned two variables for example)

B switch_var_demo1 36- 56 43 yes static array [10, 30] integer 26 2

a read_array 19-27 4 no --- --- real 54 3

and so on

if an entry is not applicable, then mark three hyphen marks as ---

Once the symbol table is completely populated, use **printSymbolTable()** to print the contents of that by traversing the hierarchical hash tables and print appropriately. It is absolutely acceptable that the variables may not appear in the same order as they appeared in the test case code, provided their computed offsets genuinely verify the ordering of variables in the test case. However, variable sequence from one scope should not be overlapped with that of another scope of the same nesting level (say sibling). One scope's (say S) variables along with all its children nested scopes variables should be printed before you print the variables from a sibling scope of S.

6. Activation record size (fixed, excluding system related): For printing the total memory requirement (sum total of widths of all variables in the function scope) for each function. The format is as follows

function1 18

compute_sum 34

.... and so on

7. **Static and dynamic arrays:** For printing the type expressions and width of array variables in a line for a test case for the following information.Separate entries using formatted output (e.g. %10d, %15.4f and so on)

- Scope - module name
- scope - line number pairs of start and end of the scope
- Name of array variable
- whether static or dynamic
- range variables or number lexemes
- type of element

Example format

switch_var_demo1	36- 56	B	static array	[10, 30]	integer
switch_var_demo1	36- 56	E	static array	[4, 10]	integer
var_demo_array	178-200	b4	static array	[100, 150]	boolean

.....

.....

module name	67- 119	A	dynamic array	[m, n]	real

and so on.....

8. Errors reporting and total compiling time: For compiling to verify the syntactic and semantic correctness of the input source code If the code is syntactically incorrect, report all syntax errors only. If the code is syntactically correct, then report all type checking and semantic errors. Also print (on the **console**) the total time taken by your **integrated compiler**. Print both total_CPU_time and total_CPU_time_in_seconds (as mentioned earlier)

9. Code generation: For producing assembly code (Linux based NASM will be used for execution) (assuming that there is no syntactic, semantic or type mismatch error in the test cases).

Perform actions appropriately by invoking appropriate functions. All lexical, syntax and semantic errors including type mismatch errors must be reported appropriately on the console (Standard output) ONLY and not in any file unless otherwise specified.

The very first line on the console, as your compiler code executes, should contain a message regarding the status of your work such as

LEVEL #: Message

where # is any one index in {1,2,3,4} and the message is Symbol table/type Checking/ Semantic rules module(s) work(s)/ handled static and dynamic arrays in type checking and code generation (specify one or many as applicable). The level specification is according to the total number of semantic rules you could successfully implement. The

measure for the total number of semantic rules (type checking and semantic) is as per the number of errors you could target. However, This level of message does not count on the syntax errors (which however are to be reported if exist).

Specify LEVEL as

1 , if less than or equal to 5 ERRORs out of all errors could be successfully handled

2 , if 6-10 ERRORs could be implemented

3 , if 11-15 ERRORs could be implemented

4 , if 16 or more ERRORs could be implemented

Note: If you print a message,

LEVEL 2: Symbol table/ AST/ Semantic Rules modules work.

This means that you were able to implement the symbol table, AST and semantic rules handling 6-10 semantic errors in total, but you could not implement type checking successfully (subject to verification).

The complete ERROR list (with line numbers first) should be printed on the console if the code is syntactically or semantically incorrect, else **print a message** [Essential when there is no error]

Code compiles successfully.....

2. Compilation:

The name of the make file should be makefile only as I will avoid using -f option always to make your file named something else (that includes searching for the file which is time taking). The evaluation of your code will be done using the GCC version as specified earlier. Please ensure compatibility.

NOTE: The test cases t#.txt and c#.txt (# :1-10) were provided only as a support and they may not be complete with respect to all aspects of the implementation. Students are advised to create own test cases for verifying the correctness of their compiler code.

3. Execution

The command line argument for execution of the driver should be as follows, for example

$./compiler testcase.txt code.asm

where compiler is the executable generated after linking all the files (your makefile should be absolutely correct). Also, the testcase.txt is the input source code (in the language you are implementing) file to be compiled. code.asm is the output file containing the assembly language code (NASM -64 bits compatible) equivalent to the input source code. All errors including lexical errors, syntax errors and semantic errors should be displayed on the console with line numbers. Testing of the Resultant code in assembly language is done on a simulator NASM 2.14.02. The Instruction Set Architecture of NASM should be used for the target code.

****

◄ Stage 2: Code generation test cases

Stage 2: Deadline extended (Final) ►

© 2019-2020 Centre for Software Development,SDET Unit, BITS-Pilani, India.

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

Navigation

Administration

Important Links

Contact Us

Events

Calendar

Upcoming events

MOBILE APP LOGIN