**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI**

Batch No. :

**DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS**

**Compiler Construction (CS F363)**

**II Semester 2022-23**

**Compiler Project (Stage-2 Submission)**

**Coding Details**

**(April 12, 2023)**

**Group number 5 (FIVE)**

*Instruction: Write the details precisely and neatly. Places where you do not have anything to mention, please write NA for Not Applicable.*

1. IDs and Names of team members

ID: 2020A7PS0013P, Name: Kaustab Choudhury

ID: 2020A7PS0078P, Name: Shashank Shreedhar Bhatt

ID: 2020A7PS0110P, Name: Harsh Priyadarshi

ID: 2020A7PS0313P, Name: Hrishikesh Harsh

ID: 2020A7PS1691P, Name: Antriksh Sharma

1. Mention the names of the Submitted files (Include Stage-1 and Stage-2 both)

1. SymbolTable.c 2. SymbolTable.h 3. SymbolTableDef.h 4. Hashing.c

5. TheAST.c 6. TheAST.h 7. TheASTDef.h 8. IntegratedParser.c

9. IntegratedParser.h 10. IntegratedParser.h 11. lexer.c 12. lexerDef.h

13. lexer.h 14. macros.h 15. stackADT.c 16. First\_Follow\_Set.c

17. setADT.c 18. setADT.h 19. commons.h 20. driver.c

21. grammar\_modified.txt 22. coding\_details.docx 23. Code\_Gen.c 24. Code\_Gen.h

25. ts#.txt, #: 1-10 26. t#.txt, #: 1-6 27. c#.txt, #: 1-11 28. DFA.pdf

29. makefile 30. Semantic\_Rules.pdf 31. First\_And\_Follow\_Set.pdf

Note: The semantic testcase files have been renamed to **“ts#.txt”** from **“t#.txt”** to avoid ambiguity between the semantic testcase files and the phase-1 testcase files.

1. Total number of submitted files: 57 (All files should be in **ONE** folder named exactly as Group\_number)
2. Have you mentioned names and IDs of all team members at the top of each file (and commented well)? (Yes/ no) Yes. [Note: Files without names will not be evaluated]
3. Have you compressed the folder as specified in the submission guidelines? (yes/no) Yes.
4. **Status of Code development**: Mention 'Yes' if you have developed the code for the given module, else mention 'No'.
   1. Lexer: Yes
   2. Parser: Yes
   3. Abstract Syntax tree: Yes
   4. Symbol Table: Yes
   5. Type checking Module: Yes
   6. Semantic Analysis Module: Yes (reached LEVEL 4 as per the details uploaded)
   7. Code Generator: Yes
5. **Execution Status**:
   1. Code generator produces code.asm: Yes
   2. code.asm produces correct output using NASM for testcases: c#.txt, #: 1-10.
   3. Semantic Analyzer produces semantic errors appropriately (Yes/No): Yes
   4. Static Type Checker reports type mismatch errors appropriately (Yes/ No): Yes
   5. Dynamic type checking works for arrays and reports errors on executing code.asm (yes/no): Yes.
   6. Symbol Table is constructed Yes and printed appropriately Yes
   7. AST is constructed Yes and printed Yes
   8. Name the test cases out of 21 as uploaded on the course website for which you get the segmentation fault (t#.txt ; # 1-10 and c@.txt ; @:1-11): None
6. **Data Structures** (Describe in maximum 2 lines and avoid giving C definition of it)
   1. AST node structure: Contains a number mentioning the number of children this node has and then an array containing pointers to those children, separately also containing a linked list of nodes in case it needs it. Also contains corresponding parse tree node’s data, and two numbers specifying the scope range.
   2. Symbol Table structure: Contains useful metadata like nesting level, type of symbol table (function, or for loop, etc), a unique ID in case it is a module, and a number of children in case this symbol table points to other symbol tables nested within it. It also contains input-output linked lists which are used for the function symbol tables.
   3. Array type expression structure: *<is\_static, (left\_range, right\_range), element\_data\_type\_enum, element\_data\_type\_string>*. *is\_static* is a number that ranges from 0-3 where 0 specifies completely dynamic array, 1 refers to right range dynamic array, 2 to left range dynamic array, and 3 to completely static array.
   4. Input parameters type structure: *<identifier, type\_record, next\_pointer>*. The *type\_record* is a union to allow both primitives and arrays. The *next\_pointer* refers to the next input parameter in the linked list.
   5. Output parameters type structure: Exactly same as input parameters, as explained above.
   6. Structure for maintaining the three address code (if created): A 2D character array called *quadruple* having dimensions *char [4][21]*.
7. **Semantic Checks:** Mention your scheme NEATLY for testing the following major checks (in not more than 5-10 words)[ Hint: You can use simple phrases such as 'symbol table entry empty', 'symbol table entry already found populated', 'traversal of linked list of parameters and respective types' etc.]
   1. Variable not declared: Whether entry found in current scope’s (or ancestor’s) symbol table.
   2. Multiple declarations: Whether entry already exists in the current scope’s symbol table.
   3. Number and type of input and output parameters: Traversal of AST node’s linked lists accordingly.
   4. Assignment of value to the output parameter in a function: LHS of all assignment type statements (eg. :=, moduleReuse, get\_value, etc.) in the module checked.
   5. Function call semantics: Using the root symbol table to verify whether the called function’s identifier is actually referring to a module and that it has been declared already.
   6. Static type checking: Using symbol table entries.
   7. Return semantics: We are assuming return semantics refers to matching input and output parameters between actual and formal ones. Using appropriate AST node children and symbol table entry of the module to verify input parameters and output parameters match.
   8. Recursion: Using a directed graph of function calls to find out if there are cyclic calls.
   9. Module overloading: Whether module entry already exists in the root symbol table.
   10. 'switch' semantics: Symbol table entry for type checking of switch variable, and checking default and case values accordingly.
   11. 'for' and 'while' loop semantics: Using LHS of all assignment type statements in each case inside the construct’s scope.
   12. Handling offsets for nested scopes: One single global counter is used and updated for every variable, whether nested or global. Offsets are therefore never re-used in our implementation.
   13. Handling offsets for formal parameters: They are treated as the first couple of variables in the module.
   14. handling shadowing due to a local variable declaration over input parameters: Nested symbol table structure helps here. If the variable is found in the current scope’s symbol table, we cease checking its ancestors for the same.
   15. Array semantics and type checking of array type variables: Using symbol table entries for array index type, if identifier. Array’s symbol table entry also specifies the type of its elements, which is used everywhere array elements are used. Symbol table entry for array also specifies its range features, which are used for checking statements of the form *A := B*, where A and B are arrays.
   16. Scope of variables and their visibility: Nested symbol table structure is used.
   17. Computation of nesting depth: Incrementing the nesting depth attribute whenever a child symbol table is created, ie., whenever a nested scope is encountered.
8. Code Generation:
   1. NASM version as specified earlier used: Yes: v2.14.02
   2. Used 32-bit or 64-bit representation: 64-bit, as specified.
   3. For your implementation: 1 memory word = 1 byte. Our offsets are byte-organized for ease of programming.
   4. Mention the names of major registers used by your code generator:

* For base address of an activation record: ESP/RSP
* for stack pointer: ESP/RSP
* others (specify): EAX
  1. Mention the physical sizes of the integer, real and boolean data as used in your code generation module

size(integer): 4 (in words/ locations), 4 (in bytes)

size(real): 16 (in words/ locations), 16 (in bytes)

size(boolean): 4 (in words/ locations), 4 (in bytes)

* 1. How did you implement functions calls?  
     Three steps. First, copying input parameters from caller scope (actual) to callee scope (formal) by moving from offset to offset. Then, executing an unconditional jump to the label of the function and executing the statements written there. Lastly, returning back to callee function and copying the output parameters from caller scope to callee scope.
  2. Specify the following:
     + Caller's responsibilities: Copying of input parameters and executing jump, after ensuring that the semantic checks are validated.
     + Callee's responsibilities: Returning and copying back the output parameters.
  3. How did you maintain return addresses?

The current value of instruction pointer is maintained on the stack when we perform our unconditional jump, so that when we return that address is used to continue execution. This is all encapsulated by NASM’s call-ret logic.

* 1. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee? Simply moving values from offset to offset. For this move operation to be performed correctly, the offset values were used.
  2. How is a dynamic array parameter receiving its ranges from the caller? We have not been able to implement dynamic array passing as parameters.
  3. What have you included in the activation record size computation? (local variables, parameters, both): Local variables and parameters both.
  4. Register allocation (your manually selected heuristic): We used the standard x86\_64 registers offered by NASM, in order of register index.
  5. Which primitive data types have you handled in your code generation module? (Integer, real and boolean): All: integer, real, as well as boolean.
  6. Where are you placing the temporaries in the activation record of a function? In the symbol table of that particular function.

1. **Compilation Details**:
   1. Makefile works (yes/No): Yes.
   2. Code compiles: Yes.
   3. Mention the .c files that do not compile: None.
   4. Any specific function that does not compile: No.
   5. Ensured the compatibility of your code with the specified versions [GCC, UBUNTU, NASM]: Yes.
2. Execution time for compiling the test cases [lexical, syntax and semantic analyses including symbol table creation, type checking and code generation] :
   * 1. t1.txt (in ticks) \_\_\_\_\_\_\_2612\_\_\_\_\_\_\_ and (in seconds) \_\_\_\_\_\_0.002612\_\_\_\_\_\_
     2. t2.txt (in ticks) \_\_\_\_\_\_\_2421\_\_\_\_\_\_\_\_\_\_ and (in seconds) \_\_\_0.002421\_\_\_\_\_\_\_\_\_
     3. t3.txt (in ticks) \_\_\_\_\_\_\_2599\_\_\_\_\_\_\_\_\_\_ and (in seconds) \_\_\_0.002599\_\_\_\_\_\_\_\_\_\_
     4. t4.txt (in ticks) \_\_\_\_\_\_\_2818\_\_\_\_\_\_\_\_\_\_ and (in seconds) \_\_\_0.002812\_\_\_\_\_\_\_\_\_\_
     5. t5.txt (in ticks) \_\_\_\_\_\_\_4355\_\_\_\_\_\_\_\_\_\_ and (in seconds) \_\_\_0.004355\_\_\_\_\_\_\_\_\_
     6. t6.txt (in ticks) \_\_\_\_\_\_\_4703\_\_\_\_\_\_\_\_\_\_ and (in seconds) \_\_\_0.004703\_\_\_\_\_\_\_\_\_
     7. t7.txt (in ticks) \_\_\_\_\_\_\_5597\_\_\_\_\_\_\_\_\_\_ and (in seconds) \_\_\_0.005597\_\_\_\_\_\_\_\_\_
     8. t8.txt (in ticks) \_\_\_\_\_\_\_5316\_\_\_\_\_\_\_\_\_\_ and (in seconds) \_\_\_0.005316\_\_\_\_\_\_\_\_\_
     9. t9.txt (in ticks) \_\_\_\_\_\_\_8030\_\_\_\_\_\_\_\_\_\_ and (in seconds) \_\_\_0.008030\_\_\_\_\_\_\_\_\_
     10. t10.txt (in ticks) \_\_\_\_\_\_2502\_\_\_\_\_\_\_\_\_\_\_ and (in seconds) \_\_\_0.002502\_\_\_\_\_\_\_\_\_
3. **Driver Details**: Does it take care of the **TEN** options specified earlier?(yes/no): Yes.
4. Specify the language features your compiler is not able to handle (in maximum one line)

Redundant module declarations in the form of *declare .. define .. use*. We are not handling this particular type of semantic check. Apart from this, a few code generation features are also missed out (not handled), for instance, cases like dynamic arrays having negative identifier ranges.

1. Are you availing the lifeline (Yes/No): Yes
2. Write exact command you expect to be used for executing the code.asm using NASM simulator [We will use these directly while evaluating your NASM created code]  
   nasm -felf64 code.asm && gcc -no-pie code.o && ./a.out

*Note: The filename need not always be “code.asm”. It depends on the command line input given.*

1. **Strength of your code** (Strike off where not applicable): (a) Correctness ✓ (b) ~~Completeness~~ (*a few* code generation features missed out) (c) Robustness ✓ (d) Well documented ✓ (e) Readable ✓ (f) Strong data structure ✓ (f) Good programming style (indentation, avoidance of goto stmts etc) ✓ (g) Modular ✓ (h) Space and time efficient ✓
2. Any other point you wish to mention: Thank you for giving us the opportunity to work on this project. Although it was hectic and difficult at times, we are sure that we learnt many things through this project – including programming on a large codebase (improving our programming skill), the underlying workings of compilers, teamwork and collaboration. Albeit extremely time consuming, working on this project was a fruitful experience.
3. Declaration: We, Kaustab Choudhury, Shashank Shreedhar Bhatt, Harsh Priyadarshi, Hrishikesh Harsh, and Antriksh Sharma declare that we have put our genuine efforts in creating the compiler project code and have submitted the code developed only by our group. We have not copied any piece of code from any source. If our code is found plagiarized in any form or degree, we understand that a disciplinary action as per the institute rules will be taken against us and we will accept the penalty as decided by the department of Computer Science and Information Systems, BITS, Pilani. [Write your ID and names below]

*ID: 2020A7PS0013P, Name: Kaustab Choudhury*

*ID: 2020A7PS0078P, Name: Shashank Shreedhar Bhatt*

*ID: 2020A7PS0110P, Name: Harsh Priyadarshi*

*ID: 2020A7PS0313P, Name: Hrishikesh Harsh*

*ID: 2020A7PS1691P, Name: Antriksh Sharma*

Date: April 13th, 2023, Group number Five

---------------------------------------------------------------------------------------------------------------------------------------------

Should not exceed 6 pages.