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

Batch No. :

**DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS**

**Compiler Construction (CS F363)**

**Group Number**

**4**

**II Semester 2019-20**

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

**Coding Details**

**(April 20, 2020)**

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

IDs and Names of team members

ID: 2016B1A70280P Name: Akshina Jindal

ID: 2017A7PS0276P Name: Kavya Gupta

ID: 2016B4A70322P Name: Satyavrat Sharma

ID: 2016B4A70509P Name: Nikita N. Singh

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

1. ast.c 7. coding\_Details(stage2).doc 13. parserDef.h 19. typeChecker.c

2. codeGeneration.c 8. lexer.h 14. semanticAnalyser.c 20. typeChecker.h

3. codeGeneration.h 9. lexerDef.h 15. semanticAnalyser.h 21. t1.txt-t12.txt

4. driver.c 10. makefile 16. symbolTable.c 22. c1.txt-c11.txt, ST.txt

5. grammar\_latest.txt 11. parser.c 17. symbolTable.h 23. Coding\_details.docx

6. lexer.c 12. parser.h 18. symbolTableDef.h

1. Total number of submitted files: **45** (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/No): **YES**
   2. Parser (Yes/No): **YES**
   3. Abstract Syntax tree (Yes/No): **YES**
   4. Symbol Table (Yes/ No): **YES**
   5. Type checking Module (Yes/No): **YES**
   6. Semantic Analysis Module (Yes/ no): **YES**  (reached LEVEL **4** as per the details uploaded)
   7. Code Generator (Yes/No): **YES**
5. **Execution Status**:
   1. Code generator produces code.asm (Yes/ No): **YES**
   2. code.asm produces correct output using NASM for testcases (C#.txt, #:1-11): **ALL (C1-11.TXT)**
   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/no) **YES**  and printed appropriately (Yes /No): **YES**
   7. AST is constructed (yes/ no) **YES** and printed (yes/no) **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: **ASTRight, ASTParent and ASTChild parameters have been added to the Parse Tree Node structure itself.**
   2. Symbol Table structure: **Hash of hash tables where outer table corresponding to each scope and inner table contains the symbol table entries.**
   3. array type expression structure: **(ARRAY, start\_index, end\_index)**
   4. Input parameters type structure: **array of type expressions**
   5. Output parameters type structure: **array of type expressions**
   6. Structure for maintaining the three address code(if created) : **Not created three address code**
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 : **symbol table entry empty and variable not part of declaration statements**
   2. Multiple declarations: **symbol table entry already found populated for that particular scope**
   3. Number and type of input and output parameters: **Traverse input, output ‘sequence preserved’ (type) parameter list separately on encountering module reuse statement during AST traversal (modular code)**
   4. assignment of value to the output parameter in a function: **Save in ST if variable is assigned a value during AST traversal**
   5. function call semantics: **While traversing input parameter list in (c)**
   6. static type checking : **Evaluate expression type by traversing AST, use type info from ST and known rules**
   7. return semantics: **While traversing output parameter list in (c)**
   8. Recursion : I**f ‘ancestor static scope’ is same module as in reuse statement (traversing static parent)**
   9. module overloading: **module entry already exists in the function table**
   10. 'switch' semantics : **When switch encountered, find type of variable from ST, traverse AST accordingly, pass new scope for subtree**
   11. 'for' and 'while' loop semantics: **Similar to previous approach in (j) for type check; separate modular code (traverse it’s AST subtree and check all type of assignment) called for other semantic checks**
   12. handling offsets for nested scopes: **each scope (while/for/switch) has a separate entry in the outer hash table and hence separate ST.**
   13. handling offsets for formal parameters: **as soon as we encounter input\_plist in AST traversal offset is set to zero and subsequently formal parameter’s entries are populated in the symbol table.**
   14. handling shadowing due to a local variable declaration over input parameters: **Store local parameters of module and input parameter list separately, save line number of valid declaration, search ST accordingly**
   15. array semantics and type checking of array type variables: **Incorporated with primitive type checking, separate cases for lvalueID and lvalueARR; type and index from ST; check if operator with array operand**
   16. Scope of variables and their visibility : **Compare current scope, line number and the same for variable declaration stored in ST; search from present scope to static parent recursively (if not found)**
   17. computation of nesting depth: **each scope has nesting Level parameters which is calculated depending on the number of Parents that scope has until parent scope is NULL. Each scope has parameters children and parent in the function table.**
8. Code Generation:
   1. NASM version as specified earlier used (Yes/no): **YES**
   2. Used 32-bit or 64-bit representation: **64 bits**
   3. For your implementation: 1 memory word = **2** (in bytes)
   4. Mention the names of major registers used by your code generator:

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

size(integer): **2** (in words/ locations), **4** (in bytes)

size(real): **4** (in words/ locations), **8** (in bytes)

size(booelan): **1** (in words/ locations), **2** (in bytes)

* 1. How did you implement functions calls?(write 3-5 lines describing your model of implementation) **Since recursion is not allowed, we have reserved space for only 1 variable copy of each module in the data section. Whenever the function is called the reserved variables are assigned values accordingly. We do not implement using stack . Only the function calls are implemented using activation records.**
  2. Specify the following:
     + Caller's responsibilities: **store the values of the actual input parameters to the formal parameter variables. After return, assign the ret values to the actual return parameters.**
     + Callee's responsibilities: **Align the stack, execute the module statements and return to the caller’s address.**
  3. How did you maintain return addresses? (write 3-5 lines): **Return address is a pushed into the stack on each function call. Call <module\_name> handles this.**
  4. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee? **The caller assigns the actual parameter values to the formal ones before the function call.**
  5. How is a dynamic array parameter receiving its ranges from the caller? **Each array has 3 values reserved: address, bound\_in and bound\_out. For a dynamic array parameter, these 3 values of actual parameters are assigned to the dynamic array address, bound\_in and bound\_in.**
  6. What have you included in the activation record size computation? (local variables, parameters, both): **Both**
  7. register allocation (your manually selected heuristic) : **RAX for adrress, RBX for offsets,**
  8. Which primitive data types have you handled in your code generation module?(Integer, real and boolean): **Integer and boolean**
  9. Where are you placing the temporaries in the activation record of a function? **Data section**

1. **Compilation Details**:
   1. Makefile works (yes/No): **YES**
   2. Code Compiles (Yes/ No): **YES**
   3. Mention the .c files that do not compile: **NONE**
   4. Any specific function that does not compile: **NONE**
   5. Ensured the compatibility of your code with the specified versions [GCC, UBUNTU, NASM] (yes/no) **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) 6191and (in seconds) 0.006191
     2. t2.txt (in ticks) 5149 and (in seconds) 0.005149
     3. t3.txt (in ticks) 10178 and (in seconds) 0.010178
     4. t4.txt (in ticks) 6889 and (in seconds) 0.006889
     5. t5.txt (in ticks) 11211 and (in seconds) 0.011211
     6. t6.txt (in ticks) 12255 and (in seconds) 0.012255
     7. t7.txt (in ticks) 14182 and (in seconds) 0.014482
     8. t8.txt (in ticks) 13941 and (in seconds) 0.013941
     9. t9.txt (in ticks) 23814 and (in seconds) 0.023814
     10. t10.txt (in ticks) 7549 and (in seconds) 0.007549
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): **Floating point arithmetic**
5. Are you availing the lifeline (Yes/No): **NO**
6. 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 -f elf64 code.asm && gcc -no-pie code.o && ./a.out**

**(Either execute as a single command OR as three different commands separated by && in order)**

1. **Strength of your code**(Strike off where not applicable): (a) correctness (b) completeness (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: **N/A**
3. Declaration: We *Akshina Jindal, Kavya Gupta, Nikita N. Singh, Satyavrat Sharma* (your names) 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: 2016B1A70280P Name: Akshina Jindal

ID: 2017A7PS0276P Name: Kavya Gupta

ID: 2016B4A70322P Name: Satyavrat Sharma

ID: 2016B1A70509P Name: Nikita N. Singh

Date: **20th April, 2020**

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

Should not exceed 6 pages.