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

**DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS**

**Compiler Construction (CS F363)**

**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.*

1. IDs and Names of team members

**2017A7PS0067P - Samarth Jain**

**2017A7PS0073P - Srirama Prakhya**

**2017A7PS0171P - Prateek Sharma**

**2017A7PS0118P - Korukanti Shatrugna Rao**

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

**1 lexer.c 7 symbolTable.c 13 codegen.c**

**2 lexer.h 8 symbolTable.h 14 codegen.h**

**3 lexerDef.c 9 symbolTableDef.h 15 AST.c**

**4 parser.c 10 typechecker.c 16 AST.h**

**5 parser.h 11 typechecker.h 17 main.c**

**6 parserDef.h 12 typecheckerDef.h 18 makefile**

1. Total number of submitted files: **18** (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** (all 10 semantic test cases working)
   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): **C1-C9**
   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: **‘node with pointer to symbolTable entry and parent and siblings – same as parse tree node. The ast is just made by deleting and reorganizing nodes from parse tree’**
   2. Symbol Table structure: **‘every ID stored along with scope, type, offset, width and a pointer to other attributes (for arrays – like lower,upper bounds, subtype etc) – every symbol table entry is indexed by its scope and the symbol table entry is a structure which stores all the information regarding the ID’**
   3. array type expression structure: **‘node which contains the type(ARRAY), subtype(NUM, RNUM etc), lower and upper bounds, and isDynamic, along with char arrays to store lexemes (if array is dyanamic), and pointers to the 2 nodes corresponding to the 2 bounds (if the array is dynamic)’**
   4. Input parameters type structure**: ‘stored as a linked list of nodes which hold all the information for that argument’**
   5. Output parameters type structure: **‘stored as a linked list of nodes which hold all the information for that argument’**
   6. Structure for maintaining the three address code(if created): **NA**
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 for that particular variable empty'**
   2. Multiple declarations**: 'symbol table entry for that particular variable in that scope already populated'**
   3. Number and type of input and output parameters: **‘stored in a linked list of the parameters where each node stores all the information regarding that corresponding variable/ID – this helps to directly compare the called functions formal arguments and the passed (actual) arguments’**
   4. assignment of value to the output parameter in a function: ‘**were not able to implement function calling’**
   5. function call semantics: **‘simultaneously traversing the formal and actual arguments and checking their types’**
   6. static type checking: **‘checks if the operands for a particular operation (MUL, DIV, ADD, ASSIGN) etc. are of the correct, compatible types or not…the types of each var/ID are available in their respective symbol table entry that stores all their information like type, offset, width etc.’**
   7. return semantics: **‘simultaneously traversing the return variable list (in caller function) and list of variables the function returns (in callee function) and comparing their types’**
   8. Recursion: **‘checking if present function/module name is same as the module that id being called’**
   9. module overloading: **‘checking if a function with that name has already been defined’**
   10. 'switch' semantics: **‘switch variable stores type to make sure the type is either integer or Boolean – plus we make sure that switch has a new scope of its own and it checks for default if switch variable is INT and check that default is NOT THERE is switch variable is Boolean – switch is stored as list of case statements and default at the end(if present)’**
   11. 'for' and 'while' loop semantics: **‘creating a separate, new scope when encountering a for or while and handling that the loop variable in for is not assigned any value in the ‘for’ and using type checking procedures to make sure the ‘while’ condition gives a suitable value – inside the for or switch, the semantics are like for normal statements’**
   12. handling offsets for nested scopes**: ‘scope count is incremented every time we nest – therefore, each scope has a unique number, the offset that is stored in each variable/ID’s node (present) in symbol table will then be handled accordingly – the “running” offset of that scope is then kept track of using a variable that keeps increasing while we add to it the WIDTH of each variable/ID in that scope’**
   13. handling offsets for formal parameters: **‘even the formal parameters (just stored as a list of the nodes for the IDs) are basically nodes that stores their “width” - the “running” offset in that function can be calculated by adding the widths of these IDs’**
   14. handling shadowing due to a local variable declaration over input parameters: **‘over-writing the symbol table entry for that particular input variable which is now shadowed by the local variable – shadowing of output variables we print an error – we have a marker in the node for that ID to specify if it was declared in input list/ output list OR inside the function (local variable)’**
   15. array semantics and type checking of array type variables: **‘bound checking, type checking are handled for arrays as their subtypes (NUM, RNUM, BOOLEAN) and their upper and lower bounds are also stored (if they are static arrays). Therefore, when type checking or assigning one array to another – we can easily compare their types and bounds. Also if an array variable is in operation with a ID for some other type…we print error. For dynamic arrays, we can check runtime bound/typechecking errors as we KNOW/stored the OFFSET of those 2 bounds (e.g. a[m..n] – we stored pointers to m, n in a’s node)’**
   16. Scope of variables and their visibility: **‘maintaining the corresponding scope for that variable/ID in its struct/node that stores all the information for that node’**
   17. computation of nesting depth: **‘each new scope (created by START – END) is given a new unique scope number. Its parent scope is also stored so we may travel through previous scopes to find the required variables’**
8. Code Generation:
   1. NASM version as specified earlier used (Yes/no): **Yes**
   2. Used 32-bit or 64-bit representation: **64-bit**
   3. For your implementation: 1 memory word = **2 bytes** (in bytes)
   4. Mention the names of major registers used by your code generator:

* For base address of an activation record: MEM + 0
* for stack pointer: RSP
* others (specify): None
  1. Mention the physical sizes of the integer, real and boolean data as used in your code generation module

size(integer): 2 (in bytes)

size(real): 4 (in bytes)

size(booelan): 1 (in bytes)

* 1. How did you implement functions calls?(write 3-5 lines describing your model of implementation) :

We weren’t able to implement function call because of the errors we were getting in the memory

* 1. Specify the following:
     + Caller's responsibilities: NA
     + Callee's responsibilities: NA
  2. How did you maintain return addresses? (write 3-5 lines): NA
  3. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee? NA
  4. How is a dynamic array parameter receiving its ranges from the caller? NA
  5. What have you included in the activation record size computation? (local variables, parameters, both): local variables
  6. register allocation (your manually selected heuristic) : mov and movsx. movsx is used for moving signed byte lower register to higher register by extending its sign for retaining the data.
  7. Which primitive data types have you handled in your code generation module?(Integer, real and boolean): Integer, Boolean
  8. Where are you placing the temporaries in the activation record of a function? NA

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) YES and (in seconds) 0.031690
     2. t2.txt (in ticks) YES and (in seconds) 0.031645
     3. t3.txt (in ticks) YES and (in seconds) 0.033726
     4. t4.txt (in ticks) YES and (in seconds) 0.034149
     5. t5.txt (in ticks) YES and (in seconds) 0.033745
     6. t6.txt (in ticks) YES and (in seconds) 0.031913
     7. t7.txt (in ticks) YES and (in seconds) 0.036527
     8. t8.txt (in ticks) YES and (in seconds) 0.035354
     9. t9.txt (in ticks) YES and (in seconds) 0.039876
     10. t10.txt (in ticks) YES and (in seconds) 0.032391
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)
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 -m64 code.o -o output**

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: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Declaration: We, Samarth Jain, Srirama Prakhya, Prateek Sharma, Korukanti Shatrugna Rao (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]

2017A7PS0067P - Samarth Jain

2017A7PS0073P - Srirama Prakhya

2017A7PS0171P - Prateek Sharma

2017A7PS0118P - Korukanti Shatrugna Rao

Date: 20/04/2020

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

Should not exceed 6 pages.