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

**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 :** 20

*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: 2019B4A70572P Name: Rajan Sahu**

**ID: 2019B4A70638P Name: Yash Goyal**

**ID: 2019B4A70652P Name: Ayush Agarwal**

**ID: 2019B4A70656P Name: Vasu Swaroop**

**ID: 2019B4A70744P Name: A. Sudarshan**

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

1 driver.c 7 grammar.txt 11 t1.txt - t10.txt 17 c1.txt - c10.txt

2 lexer.c 8 lexer.h 12 lexerDef.h 18 STAGE-1.pdf (DFA+Grammar)

3 parser.c 9 parser.h 13 parserDef.h 19 STAGE-2.pdf (Semantic rules AST)

4 ast.c 10 ast.h 14 codegen.c 20 codegen.h

5 semanticAnalyzer.c 15 semanticAnalyzer.h 21 semanticAnalyzerDef.h

6 IR\_codegen.c 16 IR\_codegen.h 22 IR\_codegendef.h

23 codegendef.hs

1. Total number of submitted files: 40 (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 \_\_\_\_ 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 test cases (C#.txt, #:1-11): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
   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): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
   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): **NA**
6. **Data Structures** (Describe in maximum 2 lines and avoid giving C definition of it)
   1. **AST node structure**: parseTreeNode, traversal pointers and node name. The traversal pointers are used during symbol table population. Node name is used for identification during symbol table population and semantic checks.
   2. **Symbol Table structure**: There is a global table of function entry. A function entry has a root and current symbol table pointer. The current pointer is used to keep a track of the current depth of the symbol table and root is used to access the symbol table. The symbol table has type expressions for each entry of variable, offsets, etc.
   3. **Array type expression structure**: The type expression has data type and array\_data. Data Type is “array” in appropriate cases. the array\_data is filled with the type and bound of the array in case it is an array.
   4. **Input parameters type structure**: type\_exp is used for all the variables
   5. **Output parameters type structure**: type\_exp is used for all the variables
   6. **Structure for maintaining the three address code**(if created) : Not created
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** : Finds if the variable exists in the symbol table, if not found goes one level up. If still not found, search in the input and output list.
   2. **Multiple declarations**: Handled during symbol table population. Special case to allow input list variable so that it is allowed to be shadowed.
   3. **Number and type of input and output parameters**: Type checking for each parameter. If any of formal/actual parameter still left to be typechecked and either is empty, number mismatch
   4. **Assignment of value to the output parameter in a function:** isChanged variable for each type expression. In module definition, the value is checked for each output parameter
   5. **Function call semantics**: Input and output parameters are matched appropriately. (Null output list used in an assignment has not been handled)
   6. **Static type checking** : During array access:- bound checking, During assignment and and operations the right and left side type expressions are checked/matched/propagated.
   7. **Return semantics**: Check if the variables in the output list have been changed via isChanged and lastChanged flag and return appropriate error
   8. **Recursion** : Check if the function being called as the current function
   9. **Module overloading**: During function\_entry population, check for prior existence of another function
   10. **'Switch' semantics**: The case ID is type checked. If error found dont check the statements
   11. **'for' and 'while' loop semantics**: The statements constructs update lastLineChanged. Then if any of the variable in while/for conditional is changed/not changed, propagate isChanged and handle appropriately
   12. **handling offsets for nested scopes**: Added all the offsets and child offsets are changed accordingly.
   13. **handling offsets for formal parameters**: Before the declaration of the other parameters, the formal parameters are assigned appropriate offsets.
   14. **handling shadowing due to a local variable declaration over input parameters**:Handled using isChanged parameter. If the line where the variable is defined is greater than the line where the variable is used, then we use the type of the input parameter list.
   15. **Array semantics and type checking of array type variables**: Type checking and Bound checking of the index. The type of the array is extracted and propagates as a temporary type expression for further type expression matching.
   16. **Scope of variables and their visibility**: search for the variable at the nesting depth, if not found, go one level above until the variable is found. Finally check the input and output list.
   17. **Computation of nesting depth**: using Huffman coding scheme to compute the path of the corresponding symbol table
8. Code Generation:
   1. NASM version as specified earlier used (Yes/no): **YES (2.16.01 was not available in suggested Ubuntu 20.04 with gcc -9, so we have used 2.14.02 and updated this information to Dr. Vandana Agarwal)**
   2. Used 32-bit or 64-bit representation: **64-bit representation**
   3. For your implementation: 1 memory word = **2 bytes**
   4. Mention the names of major registers used by your code generator:

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

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

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

size(boolean): **0.5 words** (in words/ locations), **1 byte**(in bytes)

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

**Not being handled**

* 1. Specify the following:
     + Caller's responsibilities: **NA**
     + Callee's responsibilities: **NA**
  2. How did you maintain return addresses? (write 3-5 lines): Function codegen was a bit complex in comparison to other constructs. We did not have time to brainstorm further about this construct and hence couldn’t implement code generation for functions.
  3. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee? **Not handled**
  4. How is a dynamic array parameter receiving its ranges from the caller? **Not handled**
  5. What have you included in the activation record size computation? (local variables, parameters, both): **both**
  6. register allocation (your manually selected heuristic) : We have used all the 8 registers (rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp), due to the requirements of the constructs.
  7. Which primitive data types have you handled in your code generation module?(Integer, real and boolean): **Integer, Real and Boolean**
  8. Where are you placing the temporaries in the activation record of a function? We are placing the temporaries in the symbol table structure. We are finding the current level in the symbol table where that temporary variable needs to be declared.

1. **Compilation Details**:
   1. Makefile works (yes/No): **YES**
   2. Code Compiles (Yes/ No): **YES**
   3. Mention the .c files that do not compile: **NA**
   4. Any specific function that does not compile: **NA**
   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) 31250 and (in seconds) 0.031250 s (SEMANTIC)
     2. t2.txt (in ticks) 46875 and (in seconds) 0.046875 s
     3. t3.txt (in ticks) 15625 and (in seconds) 0.015625 s
     4. t4.txt (in ticks) 15625 and (in seconds) 0.015625 s
     5. t5.txt (in ticks) 31250 and (in seconds) 0.031250 s
     6. t6.txt (in ticks) 15625 and (in seconds) 0.015625 s
     7. t7.txt (in ticks) 15625 and (in seconds) 0.015625 s
     8. t8.txt (in ticks) 46875 and (in seconds) 0.046875 s
     9. t9.txt (in ticks) 78125 and (in seconds) 0.078125 s
     10. t10.txt (in ticks) 31250 and (in seconds) 0.03123 s
     11. c1.txt (in ticks) 15625 and (in seconds) 0.015625 s (CODE GENERATION)
     12. c2.txt (in ticks) 46875 and (in seconds) 0.046875 s
     13. c9.txt (in ticks) 78125 and (in seconds) 0.078125 s
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)**

The language

1. Are you availing the lifeline (Yes/No): **NO**
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 -o code code.o && ./code**

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. Since function overloading is not allowed, our overall implementation continues to check all the statements within, hence it throws errors in case of undeclared variables.
   2. Some errors are thrown out of order because they are caught in different modules
   3. Intermediate code generation is working accurately. Few of Code generation constructs could not be completely handled in nasm due to which some code generation test cases could not be run.
3. **Declaration**: We, Rajan Sahu, Yash Goyal, Ayush Agarwal, Vasu Swaroop, A. Sudarshan from group 20 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: 2019B4A70572P Name: Rajan Sahu**

**ID: 2019B4A70638P Name: Yash Goyal**

**ID: 2019B4A70652P Name: Ayush Agarwal**

**ID: 2019B4A70656P Name: Vasu Swaroop**

**ID: 2019B4A70744P Name: A. Sudarshan**

Date: **12/04/2023** Group number **20**

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

Should not exceed 6 pages.