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

Batch No. :

**DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS**

**Compiler Construction (CS F363)**

Group Number

11

**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

ID: 2017A7PS0151P Name: Lavanya Soni

ID: 2017A7PS0107P Name: Kushagra Agrawal

ID: 2017A7PS0127P Name: Siddhant Khandelwal

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

1 ast.c 7 hash.c 13 parser.c 19 stack.h

2 ast.h 8 hash.h 14 parser.h 20 symbolTable.c

3 codegen.c 9 lexer.c 15 parserDef.h 21 symbolTable.h

4 codegen.h 10 lexer.h 16 semantics.c 22 tree.c

5 driver.c 11 lexerDef.h 17 semantics.h 23 tree.h

6 grammar 12 makefile 18 stack.c

1. Total number of submitted files: 23 (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): Yes
   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 includes union of Leaf and Non-leaf nodes, where lead node contains tokenInfo as well. The tree node also includes the pointer to appropriate scope and symbol table entry (if applicable).
   2. Symbol Table structure is a tree of scopes and each table pointer has a field that points to the list of symbol table entries. Symbol table entries have relevant fields depending on the type of identifier including the offsets. The scope of each table has the start and end lines of the scope.
   3. array type expression structure has fields like element type of the array, start and end indices of the array for both dynamic and static types.
   4. Input parameters type structure: The scope structure has a field that points to the list of input parameters. This is a list of symbol entries with appropriate fields as described above.
   5. Output parameters type structure: The scope structure has a field that points to the list of output parameters. This is a list of symbol entries with appropriate fields as described above.
   6. Structure for maintaining the three address code(if created) : Our implementation did not require the generation of 3 Address codes. We have been able to implement all the language features without 3 address codes.
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 entry not found
   2. Multiple declarations: symbol entry already present
   3. Number and type of input and output parameters: traversing input/output list and cross-checking list of entries in the function scope
   4. assignment of value to the output parameter in a function checking assignment flags for each output parameter
   5. function call semantics: checking symbol table entries which has appropriate flags covering all the cases
   6. static type checking : verifying type fields from symbol table for each ID(including arrays) in an expression.
   7. return semantics: matching symbol table entries and AST for corresponding return values
   8. Recursion : ensuring no same function call statement is found in present scope using symbol table.
   9. module overloading: symbol table entry already present
   10. 'switch' semantics : checking symbol table entry for switch variable with expected type of case statements.
   11. 'for' and 'while' loop semantics: verifying type of control expression by type extraction, ensuring change in any one ID in control expression in loop body.
   12. handling offsets for nested scopes: resetting OFFSET for every new module, continuing same Offsets for new nested scopes in each module
   13. handling offsets for formal parameters: resetting OFFSET for input parameters and continuing the same for output parameters. Resetting for local variables in the module.
   14. handling shadowing due to a local variable declaration over input parameters: having separate list for input parameter entries and local variable entries in the symbol table for same module.
   15. array semantics and type checking of array type variables: matching bounds with the symbol table entries for static, code generation for run time bound checking
   16. Scope of variables and their visibility : a variable is visible in it’s own scope and sub scopes if not overshadowed in the sub scopes.
   17. computation of nesting depth: maintained nesting level for each variable, 0 for input/output parameters, 1 for local variables and incremented appropriately.
8. Code Generation:
   1. NASM version as specified earlier used (Yes/no): Yes
   2. Used 32-bit or 64-bit representation: 32 bit
   3. For your implementation: 1 memory word = 1 (in bytes)
   4. Mention the names of major registers used by your code generator:

* For base address of an activation record: EBP
* for stack pointer: ESP
* others (specify): EDX (Output List Base Address)
  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), 2 (in bytes)

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

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

* 1. How did you implement functions calls?(write 3-5 lines describing your model of implementation) storing the return address and base address of the caller’s AR (EBP), copying input parameters and preserving space for output parameters in the callee’s AR. Copying the output parameters at appropriate offsets upon return.
  2. Specify the following:
     + Caller's responsibilities: Copying the input parameters and return addresses in the callee’s AR. Copying the output parameters in its own AR upon return.
     + Callee's responsibilities: Setting the registers (EBP, ESP) appropriately for returning back and passing the base address of the output parameters to the caller.
  3. How did you maintain return addresses? (write 3-5 lines): Pushing the return address on the stack before calling the function using the registers EBP and popping them back on return to the caller. Base address of the output parameters is stored with the register EDX.
  4. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee? We have copied values from actual to formal parameter offsets and also reserved space for output parameters in the stack. For using the statically computed offsets, we are moving the base pointer (EBP) to the start of the callee’s activation record.
  5. How is a dynamic array parameter receiving its ranges from the caller? We have maintained the start and end indexes of each dynamic array in the data section of the code and provided a unique index to each dynamic array.
  6. What have you included in the activation record size computation? (local variables, parameters, both): Local variables
  7. register allocation (your manually selected heuristic) :EBP (for base address of AR), ESP (maintaining stack), EDX(Output list base address), EAX, EBX (for storing operands of an expression (temporaries)).
  8. Which primitive data types have you handled in your code generation module?(Integer, real and boolean): Integer, Boolean
  9. Where are you placing the temporaries in the activation record of a function? Placing the temporaries at the top of the stack, pushing and popping while computing the expression as and when required.

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) 3436 and (in seconds) 0.003436
     2. t2.txt (in ticks) 3154 and (in seconds) 0.003154
     3. t3.txt (in ticks) 8279 and (in seconds) 0.008279
     4. t4.txt (in ticks) 7406 and (in seconds) 0.007406
     5. t5.txt (in ticks) 5921 and (in seconds) 0.005921
     6. t6.txt (in ticks) 8616 and (in seconds) 0.008616
     7. t7.txt (in ticks) 4081 and (in seconds) 0.004081
     8. t8.txt (in ticks) 9106 and (in seconds) 0.009106
     9. t9.txt (in ticks) 16318 and (in seconds) 0.016318
     10. t10.txt (in ticks) 4462 and (in seconds) 0.004462
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)

In **code generation**, REAL datatype variables have not been dealt with, as directed.

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 -f elf32 code.asm**

**gcc -m32 code.o -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: **sudo apt install multilib (one time – for using scanf and printf functions)**
3. Declaration: We, Lavanya Soni, Kushagra Agrawal and Siddhant Khandelwal 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 2017A7PS0151P Name: Lavanya Soni

ID 2017A7PS0107P Name: Kushagra Agrawal

ID 2017A7PS0127P Name: Siddhant Khandelwal

Date: 20th April 2020

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

Should not exceed 6 pages.