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

Batch No. :

**DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS**

**Compiler Construction (CS F363)**

Group Number

**8**

**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: 2017A7PS0093P Name: Ayush Jain

ID: 2017A7PS0025P Name: Bharat Bhargava ID: 2017A7PS0117P Name: Satvik Golechha

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

1 ast.c 7 driver.c 13 lexer.h 19 symbol\_table.c

2 ast.h 8 enum.h 14 Makefile 20 symbol\_table.h

3 bool.c 9 grammar.txt 15 parser.c 21 type.h

4 bool.h 10 hash.c 16 parser.h 22\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5 code\_gen.c 11 hash.h 17 semantic\_analyser.c 23\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6 code\_gen.h 12 lexer.c 18 semantic\_analyser.h 24\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Total number of submitted files: 21 (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, Within symbol table module itself
   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-C11)
   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 (as soon as an error comes, the execution terminates).
   6. Symbol Table is constructed (yes/no) Yes and printed appropriately (Yes /No): Yes (Except start and end line numbers for scope, which has been handled implicitly in our implementation)
   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): N/A
6. **Data Structures** (Describe in maximum 2 lines and avoid giving C definition of it)
   1. AST node structure: It removes unnecessary nodes (i.e., nodes which won’t be used in future) from the generated parse tree and produces a syntax tree occupying less memory and suitable enough to perform semantic analysis and code generation subsequently.
   2. Symbol Table structure: There are 2 symbol tables: Identifier Symbol Table (ID-ST: local to a function) and Global Symbol Table (G-ST, containing names of all the functions defined in our program). The ID-ST contains identifier entries, which will be accessed by hashing. We build a hierarchical structure of ID-STs for handling nesting by keeping pointers to symbol tables for nested scopes. The global symbol table contains names for defined functions, whose entries will also be accessed using hashing. Collision has been resolved by chaining in both tables.
   3. array type expression structure: Includes the enum for base type (real/integer/boolean), the begin index, the end index and whether the array is dynamically allocated memory or not.
   4. Input parameters type structure: Linked list of a structure comprising of the name of the parameter, the type of parameter and whether the parameter is assigned a value or not.
   5. Output parameters type structure: Linked list of a structure comprising of the name of the parameter, the type of parameter and whether the parameter is assigned a value or not (very important from the point of view of semantic analysis)
   6. Structure for maintaining the three address code(if created) : We performed code generation directly; did not go through intermediate code generation
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 : Lookup in the id symbol table; entry not present
   2. Multiple declarations: Lookup in the id symbol table; entry already populated there
   3. Number and type of input and output parameters: Traversing list of actual and formal parameters simultaneously
   4. assignment of value to the output parameter in a function: Lookup of parameters; checking “is\_assigned” flag
   5. function call semantics: Matching the number and type of parameters by traversal, checking whether the called function is defined, checking declare-define related semantics
   6. static type checking : Lookup in the symbol table and parameters’ list; matching type
   7. return semantics: Checking whether each output parameter is assigned a value or not
   8. Recursion : Matching the executing function’s name with the name of function called
   9. module overloading: Lookup in the global symbol table; entry already found ⬄ overloading/redefinition
   10. 'switch' semantics : Checking type of switch variable; Boolean switch variable ⬄ no default; Integer switch variable ⬄ default compulsory; Ignore entire case if switch variable has type real/array. For boolean, no. of cases = 2 (true & false, strictly).
   11. 'for' and 'while' loop semantics: Checking while condition should be boolean, checking whether for loop variable is assigned a value within the loop or not.
   12. handling offsets for nested scopes: Offsets start from 1 in every id symbol table, in strong correlation with our code generation procedure
   13. handling offsets for formal parameters: Lookup for the offset stored in the structure designed for parameters.
   14. handling shadowing due to a local variable declaration over input parameters: Performed two separate lookups, the first one being on the local id symbol table and then on the input parameters list.
   15. array semantics and type checking of array type variables: Checked out-of-bounds and types of elements through lookup on the structure specifically designed for each array variable declared, storing the base type and high/low indices
   16. Scope of variables and their visibility : Automatically maintained through the hierarchical structure of symbol table of identifiers
   17. computation of nesting depth: No check (supports nesting depth till exhaustion of memory)
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 = 8(in 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): rdi, rsi, rdx, rcx, r8-r15 for function call and return
  1. Mention the physical sizes of the integer, real and boolean data as used in your code generation module

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

size(real): 1 (in ~~words~~/ locations), 4 (in bytes)

size(boolean): 1 (in ~~words~~/ locations), 1(in bytes)

NOTE: It doesn’t matter at all what the sizes of the data types is; for every push/pop operation in NASM, we would have to push (pop) 8 bytes on (from) the stack, which can neither be avoided nor generalized. The sizes would have had a profound effect in reducing memory, had the sizes occupied by variables of any of these datatypes been more than 8 bytes.

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

As soon as the function is called, the instruction pointer (for return address) is automatically pushed onto the stack. We then push rbp, make space for input and output parameters, push rbp again two times and then make space for local variables (for we know the number and types of local variables in a function, which are stored in the corresponding symbol table). On return, we first destroy the local variables and then the parameters. Finally, just before return, we pop the original base pointer into rbp.

* 1. Specify the following:
     + Caller's responsibilities: Pass on the input parameters for callee in registers.
     + Callee's responsibilities: Keep space for input and output parameters and modify memory locations keeping track of the computations performed with the parameters/ local variables.
  2. How did you maintain return addresses? (write 3-5 lines): As mentioned in Q.10 f., we push the value of rbp on the stack, just after we call a function. We ensure that the statements executed subsequently do not modify this memory location. Upon completion of the execution of the function, we pop into rbp the previous base pointer (the original base value is thus preserved) and return to the next instruction to be executed.
  3. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee?: We first ensured that no statement other than “module reuse statement” required any dedicated register. Then, when we call a function, we put all the input parameters to be passed on from the caller in registers (thus implicitly getting an upper bound on the number of input parameters which can be passed). In the callee’s activation record, we’ve reserved the space for parameter allocation, where we directly put (or, rather, save the states) the values of input parameters and retrieve from the values of the output parameters into registers. We then ensure that these registers finally put the values in appropriate memory locations in caller’s activation record.
  4. How is a dynamic array parameter receiving its ranges from the caller? For each variable present in the range construct in the AST, we have allotted space in the stack. So, we’re directly receiving the value of that (range) variable from the stack itself.
  5. What have you included in the activation record size computation? (local variables, parameters, both):

Local variables only (for parameters, we’re simply reserving constant space in the stack)

* 1. register allocation (your manually selected heuristic) : No heuristic employed. For statements other than module reuse statement, it doesn’t matter at all what registers we’re using, as per our implementation. For the module reuse statement, though, we have maintained dedicated registers for saving values of input parameters, which are rdi, rsi, rdx, rcx, r8, r9, r10, r11, r12, r13, r14, r15 in order.
  2. Which primitive data types have you handled in your code generation module?(Integer, real and boolean): Integer and Boolean. The instructions for handling real constants and variables are found to have varied outputs in our implementation (may be, due to inadequate support of 8087 on our devices, for 80x86 family cannot handle FP arithmetic on its own). At the same time, the constraints of the stack and the size of the registers for FP arithmetic require unnecessary pushes and pops and breaking a single constant into two, hence we cannot perform an operation directly in memory (especially the qword/ dword operations).
  3. Where are you placing the temporaries in the activation record of a function?: We’ve used registers for temporaries. We haven’t placed them in the activation record.

1. **Compilation Details**:
   1. Makefile works (yes/No): yes
   2. Code Compiles (Yes/ No): Yes
   3. Mention the .c files that do not compile: N/A
   4. Any specific function that does not compile: N/A
   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) 60804.00 and (in seconds) 0.06080
     2. t2.txt (in ticks) 71510.00 and (in seconds) 0.07151
     3. t3.txt (in ticks) 68050.00 and (in seconds) 0.06805
     4. t4.txt (in ticks) 71742.00 and (in seconds) 0.07174
     5. t5.txt (in ticks) 74694.00 and (in seconds) 0.07469
     6. t6.txt (in ticks) 73864.00 and (in seconds) 0.07386
     7. t7.txt (in ticks) 74510.00 and (in seconds) 0.07451
     8. t8.txt (in ticks) 74276.00 and (in seconds) 0.07428
     9. t9.txt (in ticks) 76685.00 and (in seconds) 0.07669
     10. t10.txt (in ticks) 73701.00 and (in seconds) 0.07370
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)

Handling real numbers during code generation; printing start and end line numbers of scope.

1. Are you availing the lifeline (Yes/No): No (already availed)
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 elf64 code.asm && gcc code.o –o code && ./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: N/A
3. Declaration: We, Ayush Jain, Bharat Bhargava and Satvik Golechha (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: 2017A7PS0093P Name: Ayush Jain

ID: 2017A7PS0025P Name: Bharat Bhargava ID: 2017A7PS0117P Name: Satvik Golechha

Date: 20/4/2020

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

Should not exceed 6 pages.