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: 2
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: 2019B3A70555P	Name: Chirag Gupta
ID: 2019B3A70537P	Name: Dhruv Rawat
ID: 2019B3A70387P	Name: Shreyas Ravishankar Sheeranali
ID: 2019B1A71019P	Name: Swastik Mantry
ID: 2019B3A70593P	Name: Vaibhav Prabhu

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

1 ast.c	7 colorCodes.h	13 lexer.c	19 parserDef.h	25 symbolTable.h	31 t1-t10.txt
2 ast.h	8 driver.c	14 lexer.h	20 Set.c	26 symbolTableDef.h	32 c1-c11.txt
3 astDef.h	9 grammar.txt	15 lexerDef.h	21 Set.h	27 trie.c	33 Coding_Details.pdf
4 codegen.c	10 intermedCodeGen.c	16 makefile	22 stackADT.c	28 trie.h	
5 codegen.h	11 intermedCodeGen.h	17 parser.c	23 stackADT.h	29 FIRST and FOLLOW SETS.pdf	
6 codegenDef.h	12 intermedCodeGenDef.h	18 parser.h	24 symbolTable.c	30 Semantic Rules.pdf	

3. Total number of submitted files: **52**
4. Have you mentioned names and IDs of all team members at the top of each file (and commented well)? (Yes/ no) **Yes**
5. Have you compressed the folder as specified in the submission guidelines? (yes/no) **Yes**

6. Status of Code development: Mention 'Yes' if you have developed the code for the given module, else mention 'No'.
a. Lexer (Yes/No): **Yes**
b. Parser (Yes/No): **Yes**
c. Abstract Syntax tree (Yes/No): **Yes**
d. Symbol Table (Yes/ No): **Yes**
e. Type checking Module (Yes/No): **Yes**
f. Semantic Analysis Module (Yes/ no): **Yes** (reached LEVEL **4** (as per the details uploaded))
g. Code Generator (Yes/No): **Yes**

7. Execution Status:
a. Code generator produces code.asm (Yes/ No): **Yes**
b. code.asm produces correct output using NASM for testcases (C#.txt, #:1-11):
c1.txt, c2.txt, c3.txt, c4.txt, c5.txt, c6.txt, c8.txt, c9.txt, c10.txt
c. Semantic Analyzer produces semantic errors appropriately (Yes/No): **Yes**
d. Static Type Checker reports type mismatch errors appropriately (Yes/ No): **Yes**
e. Dynamic type checking works for arrays and reports errors on executing code.asm (Yes/No): **No**
f. Symbol Table is constructed (yes/no) **Yes** and printed appropriately (Yes/No): **Yes**
g. AST is constructed (yes/ no) **Yes** and printed (yes/no) **Yes**

h. 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**

8. Data Structures (Describe in maximum 2 lines and avoid giving C definition of it)
a. AST Node Structure: struct storing label, the leaf node is it is representing a leaf, a label used for the symbolTable, parent ASTNode, leftmost and rightmost child, and sibling, and an union of type and scope.
b. Symbol Table Structure: Tree of Hash Tables. Only function records at first level, and variables at the other levels. Scope, nesting level, children, max offset, as well as input and output var's symbol table node reference is stored.
c. Array Type Expression Structure: Type of array, unions of lexeme and number for lower and upper bounds, and tags for the union and a flag for minus.
d. Input Parameters Type Structure: A linked-list of records of the input parameters. Record is the symbol table entry.
e. Output parameters type structure: A linked-list of records of the output parameters. Record is the symbol table entry.
f. Structure for maintaining the three address code(if created): Quadruple Table created which is a linked-list of quadruples. A quadruple consists of operator, type of arg1 and arg2, unions of num, rnum, boolean, and symbol table record for arg1 and arg2, union of record for result and module name string.

9. 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.]
a. Variable not Declared: Entry not found in symbol table and its ancestor symbol tables.
b. Multiple declarations: Entry already present in current symbol table.
c. Number and type of input and output parameters: Iterating through input and output record linked list and module_reuse_stmt input and output linked list simultaneously.
d. Assignment of value to the output parameter in a function: Count of number of times a variable is modified check if zero for output params
e. Function Call Semantics: Flag for declared and defined checked appropriately.
f. Static Type Checking: Types of expressions extracted recursively and compared with symbol table entry.
g. Return Semantics: Matching the type with the output param linked list.
h. Recursion: Flag for called checked appropriately.
i. Module Overloading: Module signature for same name already exists in symbol table.
j. 'Switch' Semantics: Default statement checked by checking last child. Hashtable to check for duplicate labels for int and ensure only true and false for bool.
k. 'For' and 'While' loop semantics: Checking iterator variable not on LHS. Checking if any variable in condition modified or not.
l. Handling Offsets for Nested Scopes: Offset relative to function base only. Max Offset maintained for each scope.
m. Handling offsets for formal parameters: First, offsets for input params and then for output params created.
n. Handling shadowing due to a local variable declaration over input parameters: Input params declared in its own symbol table.
o. Array semantics and type checking of array-type variables: Sizes checked for assignment and and bound-checking for static arrays done using symbol table entry of the array.
p. Scope of variables and their visibility: Scope extracted during AST creation using START and END.
q. Computation of nesting depth: Nesting depth stored in the symbol table and computed based on parent.

10. Code Generation:
a. NASM version as specified earlier used (Yes/no): **Yes**
b. Used 32-bit or 64-bit representation: **64-bit representation**
c. For your implementation: 1 memory word = **8** (in bytes)
d. 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): **rax, rbx, rcx, rdx, rsi, rdi, xmm0, xmm1, xmm2, xmm3**

e. Mention the physical sizes of the integer, real and boolean data as used in your code generation module

Size (Integer): **1** (in words/ locations), **8** (in bytes)

Size (Real): **1** (in words/ locations), **8** (in bytes)

Size (Boolean): **1** (in words/ locations), **8** (in bytes)

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

Push all the registers onto the stack. Push all actual parameters to the stack when the function is called. Pop the actual parameters and store them at the formal parameters when in the function. Execution the function lines. Push the formal output parameters onto the stack and return. Pop the formal output parameters and store them at the actual output parameters location where the function was called. Pop all the registers to restore the state.

g. Specify the following:

▪ Caller's responsibilities: Push the actual input parameters. Call the function. Pop the formal output parameters and store them at the actual output parameters' location.

▪ Callee's responsibilities: Pop the actual input parameters and store them at the formal input parameters' location. Execute the function code. Push the formal output parameters onto the stack. Return address to be maintained in a separate reserved integer which is stored in at the beginning and pushed to stack at the end.

h. How did you maintain return addresses?: Return address is stored in a separate reserved integer by fetching it from the stack at the beginning of the function call and storing it lated at the end of the function.

i. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee?: Parameters are passed by putting them on the stack. They are later popped of the stack one-by-one in the callee and stored at the statically-computed offsets of the formal parameters.

j. How is a dynamic array parameter receiving its ranges from the caller?: N/A

k. What have you included in the activation record size computation? (local variables, parameters, both):

While printing, we have only included the actual variables used, but in code generation, the complete activation record (including the temporaries) is used to allocate memory.

l. Register Allocation (your manually selected heuristic):

RAX and RBX for storing operands of expressions.

RCX and RDX for storing loop variables.

RBP for storing base pointer.

RSP for storing stack pointer.

m. Which primitive data types have you handled in your code generation module? (Integer, real and boolean):

Integer, Real, and, Boolean

n. Where are you placing the temporaries in the activation record of a function?: On the top of the actual variables used.

11. Compilation Details:

a. Makefile works (yes/No): **Yes**

b. Code Compiles (Yes/ No): **Yes**

c. Mention the .c files that do not compile: **None**

d. Any specific function that does not compile: **None**

e. Ensured the compatibility of your code with the specified versions [GCC, UBUNTU, NASM] (yes/no) **Yes**

12. Execution time for compiling the test cases [lexical, syntax and semantic analyses including symbol table creation, type checking and code generation] :

i. t1.txt (in ticks) 2129 and (in seconds) 0.002129

ii. t2.txt (in ticks) 1907 and (in seconds) 0.001907

iii. t3.txt (in ticks)	3904	and (in seconds)	0.003904
iv. t4.txt (in ticks)	2523	and (in seconds)	0.002523
v. t5.txt (in ticks)	3338	and (in seconds)	0.003338
vi. t6.txt (in ticks)	4043	and (in seconds)	0.004043
vii. t7.txt (in ticks)	5415	and (in seconds)	0.005415
viii. t8.txt (in ticks)	5848	and (in seconds)	0.005848
ix. t9.txt (in ticks)	3850	and (in seconds)	0.003850
x. t10.txt (in ticks)	2522	and (in seconds)	0.002552

13. Driver Details: Does it take care of the TEN options specified earlier?(yes/no): **Yes**
14. Specify the language features your compiler is not able to handle (in maximum one line)
In code generation, dynamic arrays have not been handled and functions cannot have an array as an input parameter.
15. Are you availing the lifeline (Yes/No): **Yes**
16. 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 -o code.o && gcc -no-pie code.o -o code
./code
17. 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
18. Any other point you wish to mention:
It is important to compile with **-no-pie** flag in gcc and we are using C scanf and printf functions.
19. Declaration: We, Chirag Gupta, Dhruv Rawat, Shreyas Ravishankar Sheeranali, Swastik Mantry, Vaibhav Prabhu, 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.

ID: 2019B3A70555P	Name: Chirag Gupta
ID: 2019B3A70537P	Name: Dhruv Rawat
ID: 2019B3A70387P	Name: Shreyas Ravishankar Sheeranali
ID: 2019B1A71019P	Name: Swastik Mantry
ID: 2019B3A70593P	Name: Vaibhav Prabhu

Date: 13th April 2023 Group number: **2**
