BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

Group Number
13

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

ID: 2017A7PS0096P **Name: Sahil Dubey**

ID: 2017A7PS0105P **Name: Rohit Milind Rajhans**

ID: 2017A7PS0109P **Name: Saujas Adarkar**

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

1. ast_def.c	18. grammar.txt	35. semantic_analyzer_def.h	52. t5.txt
2. ast.c	19. intermediate_code_def.h	36. semantic_analyzer.c	53. t6.txt
3. ast.h	20. intermediate_code.c	37. semantic_analyzer.h	54. t7.txt
4. code_gen.c	21. intermediate_code.h	38. stack.c	55. t8.txt
5. code_gen.h	22. label.h	39. stack.h	56. t9.txt
6. c1.txt	23. Language specifications.pdf	40. symbol_table_def.h	57. t10.txt
7. c2.txt	24. lexer_def.h	41. symbol_table.c	58. coding_Details
8. c3.txt	25. lexer.c	42. symbol_table.h	
9. c4.txt	26. lexer.h	43. testcase.txt	
10. c5.txt	27. lookup_table.c	44. tree.c	
11. c6.txt	28. lookup_table.h	45. tree.h	
12. c7.txt	29. makefile	46. type_extractor.c	
13. c8.txt	30. parser_def.h	47. type_extractor.h	
14. c9.txt	31. parser.c	48. t1.txt	
15. c10.txt	32. parser.h	49. t2.txt	
16. c11.txt	33. README.md	50. t3.txt	
17. driver.c	34. Sample_Symbol_table.txt	51. t4.txt	

3. Total number of submitted files: **58** (All files should be in **ONE** folder named exactly as Group number)
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 test cases (C#.txt, #:1-11): **Yes**
    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): **Yes**
    f. Symbol Table is constructed (yes/no) Yes and printed appropriately (Yes /No): **Yes, Yes**
    g. AST is constructed (yes/ no) Yes and printed (yes/no) **Yes, 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- **Has links to parent, child and next AST node. Leaf AST node stores information about lexical tokens which have line number, lexemes and tokens.**
    b. Symbol Table structure:- **Each scope has a separate symbol table. Each symbol table has a list of slots into which variables are inserted based on a hash of their ASCII values**.
    c. Array type expression structure:- **Array has an array datatype of its elements. Its indices are stored as a union of an identifier symbol and a number**
    d. Input parameters type structure:- **They are stored in a separate scope**. **Non-array input parameters have their corresponding type and are used similar to other function variables while array input parameters have pointer type**
    e. Output parameters type structure:- **They are stored in a separate scope. Type is the same as that declared. Behave similar to other function variables.**
    f. Structure for maintaining the three address code(if created) :- **Tuple which contains a single operator, and 3 strings corresponding to the operands, and a pointer to the next tuple**

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 :- **Symbol table entry empty**
    b. Multiple declarations:- **Symbol table entry already found populated**
    c. Number and type of input and output parameters:- **Traversal of linked list of parameters and respective types**
    d. assignment of value to the output parameter in a function:- **Assignment by either assignment statement, get_value, or by module reuse statement**
    e. function call semantics:- **Actual and formal Input and output parameters' counts and types same**
    f. static type checking :- **For arithmetic and relational operators, both integer or both real operands. For boolean operators, both boolean operands**
    g. return semantics:- **Output parameters' count and types same as formal parameters**
    h. Recursion :- **Name of function being called is same as current scope**
    i. module overloading:- **Symbol table for function already populated**
    j. 'switch' semantics :- **Switch condition variable datatype check. If boolean, true and compulsory. Default for integer condition, no default for boolean**

k. 'for' and 'while' loop semantics:- **For loop variable integer datatype, redeclaration and assignment check, while loop variables modification check, redeclaration check**

l. handling offsets for nested scopes:- **Computed along as part of offset of the function**

m. handling offsets for formal parameters:- **Computed separately, not as part of the main function**

n. handling shadowing due to a local variable declaration over input parameters:- **Different scope for input parameters**

o. array semantics and type checking of array type variables:- **Arrays can be assigned directly only if type and bounds match. Compile time-bound checking for static declared arrays. Run time-bound checking for dynamic arrays**

p. Scope of variables and their visibility :- **Using scope in which they are defined**

q. computation of nesting depth:- **Increment level by one when moving one scope deeper**

10. Code Generation:

a. NASM version as specified earlier used (Yes/no): **Yes**

b. Used 32-bit or 64-bit representation: **64-bit**

c. For your implementation: 1 memory word = **2** (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, rbp, al, rbx, ax, bx, bl, cx, rdi, rsi, eax**

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

size(integer):	**2**	(in words/ locations),	**4**	(in bytes)
size(real):	**4**	(in words/ locations),	**8**	(in bytes)
size(boolean):	**1**	(in words/ locations),	**2**	(in bytes)

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

Before a function call, all input parameters are copied (in case of array base address is copied) to the callee's scope. Formal parameters are stored at the beginning of callee's activation record (after return address). Also, the addresses of actual output parameters are saved in the callee's AR. On call, the base address of caller is pushed on stack. While returning, the computed outputs are stored in required addresses. Caller's base address is popped from stack.

g. Specify the following:

- Caller's responsibilities: **Formal parameters are stored at the beginning of callee's activation record. Addresses of actual output parameters are saved in the callee's activation record**
- Callee's responsibilities: **Copy formal output parameters into actual parameters when returning from call**

h. How did you maintain return addresses? (write 3-5 lines):

Return address of caller is stored on the stack, just before the beginning of the activation record of the callee. On function return, the return address is then popped from the stack and stored in the base register.

i. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee? **The parameters have been passed using pass by value(for non-array type parameters) and pass by reference(for array type parameters). Computed offsets were used relative to callee's base address.**

j. How is a dynamic array parameter receiving its ranges from the caller? **Populated by caller during function call at run time in space reserved in the activation record of callee.**

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

l. register allocation (your manually selected heuristic) : **Free register assigned**

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? **After the variables used in the function**

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 3896 (in ticks) and 0.003896 (in seconds)
    ii. t2.txt 3056 (in ticks) and0.003056 (in seconds)
    iii. t3.txt 3229 (in ticks) and 0.003229 (in seconds)
    iv. t4.txt 3989 (in ticks) and 0.003989 (in seconds)
    v. t5.txt 4901 (in ticks) and 0.004901 (in seconds)
    vi. t6.txt 5087 (in ticks) and 0.005087 (in seconds)
    vii. t7.txt 6187 (in ticks) and 0.006187 (in seconds)
    viii. t8.txt 7145 (in ticks) and 0.007145 (in seconds)
    ix. t9.txt 7843 (in ticks) and 0.007843 (in seconds)
    x. t10.txt 2830 (in ticks) and 0.002830(in seconds)

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): **None**
15. Are you availing the lifeline (Yes/No): **No**
16. Write the 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]:-
    To compile code.asm in 64-bit:
    nasm -f elf64 code.asm -o code.o && gcc -no-pie -o exe code.o -lc
    or
    make executable
    To run:
    ./exe
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: **None**

19. Declaration: We, **Sahil Dubey, Rohit Rajhans and Saujas Adarkar**, 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: 2017A7PS0096P **Name: Sahil Dubey**

ID: 2017A7PS0105P **Name: Rohit Milind Rajhans**

ID: 2017A7PS0109P **Name: Saujas Adarkar**

Date: 19/04/2020

Should not exceed 6 pages.