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

Compiler Construction (CS F363)

II Semester 2021-22

Compiler Project (Stage-2 Submission)

Group 18

Coding Details

(April 16, 2022)

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:2019A7PS0029P	Name:Sankha Das
ID:2019A7PS0051P	Name:Pratham Neeraj Gupta
ID:2019A7PS0063P	Name:Madhav Gupta
ID:2019A7PS0243P	Name:Meenal Gupta
ID:2019A7PS1138P	Name:Yash Gupta

1. Mention the names of the Submitted files (Include Stage-1 and Stage-2 both)
    1. astDef.h
    2. astGenerator.c
    3. astGenerator.h
    4. buffer.c
    5. buffer.h
    6. c1.txt
    7. c2.txt
    8. c3.txt
    9. c4.txt
    10. c5.txt
    11. c6.txt
    12. c7.txt
    13. c8.txt
    14. codegen.c
    15. codeGenerator.c
    16. codeGenerator.h
    17. codeGeneratorDef.h
    18. Coding Details - Group 18.pdf
    19. driver.c
    20. DFA.pdf

21. finalgrammar.txt
22. First and Follow.pdf
23. FirstAndFollow.c
24. FirstAndFollow.h
25. firstSets.txt
26. followSets.txt
27. globalDef.h
28. icgGenerator.c
29. icgGenerator.h
30. icgGeneratorDef.h
31. keywordMap.c
32. keywordMap.h
33. keywords.txt
34. lexer.c
35. lexer.h
36. lexerDef.h
37. Makefile
38. p1.txt
39. p2.txt
40. p3.txt
41. p4.txt
42. parser.c
43. parser.h
44. parserDef.h
45. parserTable.csv
46. README.md
47. SemanticRules.pdf
48. SemanticAnalyzer.c
49. SemanticAnalyzer.h
50. s1.txt
51. s2.txt
52. s3.txt
53. s4.txt
54. s5.txt
55. stack.c
56. stack.h
57. symbolTable.c
58. symbolTable.h
59. symbolTableDef.h
60. t1.txt
61. t2.txt
62. t3.txt
63. t4.txt
64. t5.txt

65. t6.txt
66. TypeChecker.c
67. TypeChecker.h
68. typing.c
69. typing.h

1. Total number of submitted files: 69 (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] Yes
3. Have you compressed the folder as specified in the submission guidelines? (yes/no) Yes

1. **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: variant records not handled - was giving errors for other records if incorporated)
   g. Code Generator (Yes/No):No (Only Intermediate Code being fully generated, ASM not fully)

1. **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): No (read, write giving nasm compilation issue)
   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 variant records with tagged union and reports errors on executing code.asm (yes/no): dynamic types detected
   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 17 as uploaded on the course website for which you get the segmentation fault (p#.txt ; # 1-4, s$.txt; $ 1-5, and c@.txt ; @:1-8): c3+
2. **Data Structures** (Describe in maximum 2 lines and avoid giving C definition of it)
   a. AST node structure
      1. Type: the operation/tag of that AST Node
      2. Flags: leaf node / linked list node / normal node
      3. Pointers to children, data and next (for linked list) nodes
   b. Symbol Tables structure:
      1. Table metadata - identifier, total width
      2. An hashmap (with separate chaining) of table entries
      3. Symbol tables are stored in a linked list so a next pointer
   c. Record type expression structure:

The actual values accessed are retrieved from the linked list of corresponding ast nodes for type checking and semantic analysis.

d. Data structure for global variables:
    1. Just a normal symbol table with identifier as "GLOBAL"

e. Variant record type expression structure:

    Checking if tagValue field is present and one and only one of the fields is of type Union.

a. Input parameters type structure:
    1. Linkedlist of parameters
    2. Datatype of each parameter, its identifier, and a pointer to the next node
b. Output parameters type structure:

    Linkedlist of parameters

    Datatype of each parameter, its identifier, and a pointer to the next node

c. Structure for maintaining the three address code(if created):
    1. Pentuple - contains the Operation, Result, 2 Arguments, jump Label or immediate value
d. Any other interesting data structures used:
    1. Linked List of symbol tables that helps to print them
    2. Type Table - to store the data type including those of records

1. **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 already has an identifier in scope
    c. Number and type of input and output parameters: Line <line_no> : function <func_name> requires <input_number_of_parameters> input parameters, but <output_number_of_parameters> supplied
    d. assignment of value to the output parameter in a function: Function <function_name>: does not update output parameter - <output_parameter_name>.
    e. function call semantics:

        - Line <line_no> : Function <function_name> not defined.

        - Line <line_no> : Recursion not allowed!

Line <line_no> : function <function_name> should be defined before <function_name

Line <line_no>: function <function_name> requires <number_of_formall_parameters> input parameters, but <number_of_actual_parameters> supplied\n

Line <line_no>: function <function_name> has <number of formal_parameters> output parameters, but <number of actual parameters> seen.

Line <line_no> : Input parameter mismatch for <func_name> for <output_parameter> - expected <formal type> but got <actual type>.

<output_parameter_name>: Output parameter mismatch.\n

f. static type checking :Line <line_no>: Addition/subtraction: real or integer operands required.

   -Accordingly, for division/multiplication, relational, boolean, and assignment in similar style

g. return semantics: Function should return %d values, but returns %d values.
    - Mismatch in the return list.
h. Recursion : Line %d : Recursion not allowed!
i. module overloading:Error line <line_no>: Function <function_name> already defined.
j. if-then-else semantics : *already checked in boolean expressions*
k. handling offsets for local variables (starting with 0, integer size =2, real size =4 for symbol table purpose):Yes
l. handling offsets for formal parameters: 2 linked list used for input and output formal parameters of each function
m. handling global variable declaration over local variables and input-output parameters: Line <line-no>: Error - <var_name> already declared in global scope
n. Record semantics and static type checking: Accessing the field types from a field linked list for each record.
a. Variant record semantics and dynamic type checking: Not handled
a. Scope of variables and their visibility : Line <line_number>: Identifier <identifier_name> unrecognized - undeclared or unaccessible.
b. handling nesting depth of variables in Boolean expression in while loop for assignment of an expression to one of the guard variables:Error: Looping variables in while not changed.

1. **Compiler passes description (Mention the details of information collected/populated/worked upon at each traversal of the whole AST):**
    a. Pass 1: Names of constructed data types retrieved from their definitions;
    b. Pass 2: Type, width, and offset calculations for constructed data types
    c. Pass 3: Handling the declaration statements to populate the respective symbol tables (reporting errors for function overloading, redeclarations in same scope, redeclarations of globals)
    d. Pass 4: Type Checking and Semantic Analysis (function calls, while looping, all expressions, assignments, returns, function called before its definition appears)

1. **Code Generation:**

a. NASM version as specified earlier used (Yes/no):_______2.15.05_________________

b. Used 32-bit or 64-bit representation:______________64_______________________________

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: ____EBP______________
- for stack pointer:______________ESP_______________________
- others (specify):________EAX (arithmetic operations), EBX (data storage)___________________________

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

    size(integer): __________1_____________(in words/ locations), ____2___________(in bytes)
    size(real): _________2_________________(in words/ locations), _______4________(in bytes)

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

Sections in the data segment were reserved for the functions' local variables and input output parameters. Whenever a function is called, the offset of the function called is stored in a register separate from EBP, while it is used for storing the offset of the function in which the function was called. Before shifting the IP by using 'call statement', the values are copied into the input parameter memory locations. After the called function returns, the values present in the called function's output function parameters are copied one by one in the calling function's actual parameters assigned to them.

a. Specify the following:
    - Caller's responsibilities: Copy all actual params' values into the callee's function params memory location, and copy all functional output params to caller's actual output parameters.
    - Callee's responsibilities: Push calling function's EBP value into stack, and restore it before returning control
    - How did you maintain return addresses? (write 3-5 lines): using the offsets of the variables passed as function output parameters by the callee, which are designated a location in the data segment, the return addresses are maintained when the control is handed back to the caller.

a. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee? _____the offsets were used to address the memory locations in the data segment where the function params were copied by the caller________
b. What have you included in the activation record size computation? (local variables, parameters, both): both local variables and parameters
c. Choice of registers (your manually selected heuristic only) ____________________________________
____
d. Which primitive data types have you handled in your code generation module?(Integer and real): ___________Integer____
e. Where are you placing the temporaries in the activation record of a function?

   in the same symbol table as all declared variables during intermediate code generation. Code generation does not distinguish between temporaries and declared variables.

f. Write your method of code generation for dynamic type checking for tagged union data type. Not implemented

1. **Compilation Details**:
   a. Makefile works (yes/No): Yes
   b. Code Compiles (Yes/ No): Yes
   c. Mention the .c files that do not compile: All files compile
   d. Any specific function that does not compile: All functions intended to execute compile
   e. Ensured the compatibility of your code with the specified versions [GCC, UBUNTU, NASM] (yes/no)GCC,Ubuntu (18.04 && 20.04) Yes, GCC 7.5 and Ubuntu 20.04

2. Execution time for compiling the test cases [type checking (p1-p4.txt), semantic analyses including symbol table creation (s1-s5.txt), and code generation (c1-c8.txt)] :
   i. p1.txt (in ticks) _____3136____________ and (in seconds) _______0.003136_____
   ii. p2.txt (in ticks) _____3626____________ and (in seconds) _______0.003626_____
   iii. p3.txt (in ticks) _____3864____________ and (in seconds) _______0.003864_____
   iv. p4.txt (in ticks) _____3933____________ and (in seconds) _______0.003933_____
   v. s1.txt (in ticks) _____3621____________ and (in seconds) _______0.003621_____
   vi. s2.txt (in ticks) _____4465____________ and (in seconds) _______0.004465_____
   vii. s3.txt (in ticks) _____5385____________ and (in seconds) _______0.005385_____
   viii. s4.txt (in ticks) _____4976____________ and (in seconds) _______0.004976_____
   ix. s5.txt (in ticks) _____4255____________ and (in seconds) _______0.004255_____
   x. c1.txt (in ticks) _____2089____________ and (in seconds) _______0.002089_____
   xi. c2.txt (in ticks) _____3009____________ and (in seconds) _______0.003009_____
   xii. c3.txt (in ticks) ______________________ and (in seconds) ____________________
   xiii. c4.txt (in ticks) ______________________ and (in seconds) ____________________
   xiv. c5.txt (in ticks) ______________________ and (in seconds) ____________________
   xv. c6.txt (in ticks) ______________________ and (in seconds) ____________________

xvi. c7.txt (in ticks) ____________________ and (in seconds) ____________________
xvii. c8.txt (in ticks) ____________________ and (in seconds) ____________________

1. **Driver Details**: Does it take care of the **ELEVEN** options specified earlier?(yes/no):Yes
2. Specify the language features your compiler is not able to handle (in maximum one line)

Variant records, Partially implemented assembly

1. Are you availing the lifeline (Yes/No): Lifeline used in stage 1
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 code partially implemented

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: We have attempted to create a comprehensive compiler (e.g. providing the user with not only error messages but also warning messages in case of type checks with unsafe)
1. Declaration: We, Madhav Gupta, Meenal Gupta, Pratham Neeraj Gupta, Sankha Das, and Yash Gupta (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: 2019A7PS0063P	Name: Madhav Gupta
ID: 2019A7PS0243P	Name: Meenal Gupta
ID: 2019A7PS0051P	Name: Pratham Neeraj Gupta
ID: 2019A7PS0029P	Name: Sankha Das
ID: 2019A7PS1138P	Name: Yash Gupta

Date: 16/04/2022

Should not exceed 6 pages.