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

Group Number

30

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: **2017A7PS0050P**	Name: **JIVAT NEET KAUR**
ID: **2017A7PS0088P**	Name: **VAISHNAVI KOTTURU**
ID: **2017A7PS0104P**	Name: **SARGUN SINGH**
ID: **2017A7PS0160P**	Name: **ADITI MANDLOI**
ID: **2017A7PS0227P**	Name: **JASPREET SINGH**

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

1 ast.c	**2** idTable.c	**3** parser.c	**4** symbol.h
5 ast.h	**6** intermediateCode.c	**7** parser.h	**8** symbolTableDef.h
9 codeGen.c	**10** intermediateCode.h	**11** parserDef.h	**12** utils.c
13 codeGen.h	**14** lexer.c	**15** semanticAnalyzer.c	**16** utils.h
17 driver.c	**18** lexer.h	**19** semanticAnalyzer.h	**20** symbol.c
22 funcTable.c	**22** lexerDef.h	**23** t1.txt	**24** t2.txt
25 t3.txt	**26** t4.txt	**27** t5.txt	**28** t6.txt
29 t7.txt	**30** t8.txt	**31** t9.txt	**32** t10.txt
33 c1.txt	**34** c2.txt	**35** c3.txt	**36** c4.txt
37 c5.txt	**38** c6.txt	**39** c7.txt	**40** c8.txt
41 c9.txt	**42** c10.txt	**43** makefile	**44** coding_Details(stage 2).pdf
45 c11.txt	**46** Sample_Symbol_table.txt	**47** README.md	**48** grammar.txt

3. Total number of submitted files: **48** (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** [Note: Files without names will not be evaluated]
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): **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**
    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): **-NA-**
8. **Data Structures** (Describe in maximum 2 lines and avoid giving C definition of it)
    a. AST node structure: **It has pointer to symbol table if it's a leaf. It has a union for selecting a leaf node or non terminal node and a tag isLeaf for the union. Leaf node has token t and enum value representing datatype. Non terminal node has enum values of the symbols used for representing non-terminals and datatype. The AST node also has pointers to child, parent and sibling.**
    b. Symbol Table structure: **Symbol table is made using a hash table using polynomial hashing. Module symbol table has a tree of symbol tables for nested scoping of local variables.**
    c. Array type expression structure: **Stored the datatype using enum and range variable stored in union of symbol table entry and num depending on static/dynamic nature of array.**
    d. Input parameters type structure: **It has the name of variable, its datatype, flags to check if it's an array, if it is assigned and pointer to the next parameter.**
    e. Output parameters type structure: **It has the name of variable, its datatype, flags to check if it's an array, if it is assigned and pointer to the next parameter.**
    f. Structure for maintaining the three address code(if created) : **A quadruple has been used which stores the names, pointers to AST nodes, pointers to symbol table of the 3 arguments (2 operands and 1 result), the symbol for operator used and pointer to next quadruple.**
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: **Using a flag which is updated when moduleReuseStmt, (get_value) ioStmt or assignmentStmt is encountered.**
    e. function call semantics: **Traversal of linked list of input and output parameters of the callee and the input parameters in moduleReuseStmt to check if number and types match. Check module symbol table to see if module is declared/defined.**

f. static type checking : **Traversal of expression subtree and extraction of type from symbol table**

g. return semantics: **Traversal of linked list of output parameters and the optional part of moduleReuseStmt to match the number and types of parameters**

h. Recursion : **Module name should not occur in a moduleReuseStmt in its own definition.**

i. module overloading: **Module table entry already found populated (module already defined).**

j. 'switch' semantics : **Switch variable boolean or integer only, if boolean then no default case else if integer then default case checked by traversal.**

k. 'for' and 'while' loop semantics: **Traversal of iterativeStmt subtree to check the assigned flag of respective loop variables. The while loop condition checked to be Boolean type by AST traversal.**

l. handling offsets for nested scopes: **Offset incremented with respect to a module definition only using a global variable called OFFSET.**

m. handling offsets for formal parameters: **Different scope for formal parameters in the module symbol table.**

n. handling shadowing due to a local variable declaration over input parameters: **Different scope for input parameters where local variable symbol table is its child.**

o. array semantics and type checking of array type variables: **Type and bound checking done using respective symbol table array entries for static arrays with static indices at compile time. Rest of the checking done at runtime.**

p. Scope of variables and their visibility : **Mainly using nested symbol tables along with line number pairs stored during AST formation.**

q. computation of nesting depth: **Using depth of symbol table tree structure.**

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, r8, rdi, rsi**

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), **16** (in bytes)
size(real): **4**	(in words/ locations), **32** (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): **Actual parameters placed after the local variables in the activation record of caller in reverse order. After making the function call, base pointer is pushed onto the stack, stack pointer is copied into the base pointer and space is allocated on the stack for the activation record of the callee. Callee extracts actual parameter values from the common caller/callee space, performs computations and places the values of output parameters into the common space. Base pointer is restored and return back to the caller.**

g. Specify the following:

        - Caller's responsibilities: **Place the actual parameters relative to stack pointer after placing local variables, access local variables relative to base pointer and extract output parameter values after returning from callee.**
        - Callee's responsibilities: **Access physical locations relative to current base pointer for extracting the actual parameter values placed by caller and access local variables relative to its base pointer. After computation, place output parameter values in the common caller/callee region.**

h. How did you maintain return addresses? (write 3-5 lines): **Call statement is executed which pushes the address of the next instruction onto the stack, following which the current base pointer is pushed onto the stack. Once execution of the callee is over, the stack pointer is restored by subtracting the function activation record size. Base pointer is popped from the stack and the return statement returns control back to the caller.**

i. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee? **Created quadruples for pushing actual parameters in reverse order by the caller. The callee uses the statically computed offsets of the parameters relative to the current base pointer to extract the actual input parameter values placed below its activation record in the common caller/callee space.**

j. How is a dynamic array parameter receiving its ranges from the caller? **Using 5 memory locations in the caller/callee space, 1 for base address of the array in the caller's activation record, 2 for start index value(integer) and 2 for the end index(integer).**

k. What have you included in the activation record size computation? (local variables, parameters, both): **Included fixed sized local variables (one location for base addresses for dynamic arrays).**

l. register allocation (your manually selected heuristic): **Registers rax and rbx used as operands for arithmetic, logical and relational computations. Register rcx is used as a counter for looping. Register rdi is used for storing the format string for printf and scanf. Register rsi is used for storing arguments for format string.**

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

n. Where are you placing the temporaries in the activation record of a function?: **After placing all the local variables of 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: **-NA-**
    d. Any specific function that does not compile: **-NA-**
    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) **243043.000000** and (in seconds) **0.243043**
    ii. t2.txt (in ticks) **176898.000000** and (in seconds) **0.176898**
    iii. t3.txt (in ticks) **320356.000000** and (in seconds) **0.320356**
    iv. t4.txt (in ticks) **383498.000000** and (in seconds) **0.383498**
    v. t5.txt (in ticks) **225421.000000** and (in seconds) **0.225421**

vi. t6.txt (in ticks) **450352.000000** and (in seconds) **0.450352**

vii. t7.txt (in ticks) **634036.000000** and (in seconds) **0.634036**

viii. t8.txt (in ticks) **702787.000000** and (in seconds) **0.702787**

ix. t9.txt (in ticks) **1112198.000000** and (in seconds) **1.112198**

x. t10.txt (in ticks) **599175.000000** and (in seconds) **0.599175**

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):
-NA-

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 -o output.o code.asm**
    - **gcc output.o**
    - **./a.out**

    NOTE: We compiled our code on Ubuntu 16.04 machine. On other machines like Ubuntu 18.04, sometimes "gcc output.o -no -pie" is required to compile code.asm.

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:
    a. **Handled divide by zero exception for integers.**
    b. **Runtime type checking for statements containing array elements and assignment statements for arrays has also been done.**

19. Declaration: We, **Jivat Neet Kaur, Vaishnavi Kotturu, Sargun Singh, Aditi Mandloi and Jaspreet Singh** (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: **2017A7PS0050P**	Name: **JIVAT NEET KAUR**
ID: **2017A7PS0088P**	Name: **VAISHNAVI KOTTURU**
ID: **2017A7PS0104P**	Name: **SARGUN SINGH**
ID: **2017A7PS0160P**	Name: **ADITI MANDLOI**
ID: **2017A7PS0227P**	Name: **JASPREET SINGH**

Date: **21 April 2020**