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

Group Number
31

Compiler Construction (CS F363)
II Semester 2019-20
Compiler Project (Stage-2 Submission)
Coding Details
(April 21, 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:	**2016B4A70933P**	Name:	**MADHUR PANWAR**
ID:	**2016B3A70528P**	Name:	**TUSSANK GUPTA**
ID:	**2016B4A70580P**	Name:	**SALMAAN SHAHID**
ID:	**2016B3A70549P**	Name:	**APURV BAJAJ**
ID	**2016B5A70452P**	Name:	**HASAN NAQVI**

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

1 README.txt	**16 ast.c**	**31 errorPtr_stack.c**
2 makefile	**17 ast.h**	**32 errorPtr_stack.h**
3 grammar.txt	**18 astDef.h**	**33 generic_stack.c**
4 gr_hr.txt	**19 symbolHash.c**	**34 generic_stack.h**
5 keywords.txt	**20 symbolHash.h**	**35 hash.c**
6 tokens.txt	**21 symbolTable.c**	**36 hash.h**
7 nonTerminals.txt	**22 symbolTable.h**	**37 set.c**
8 lexer.c	**23 symbolTableDef.h**	**38 set.h**
9 lexer.h	**24 printSymTable.c**	**39 util.c**
10 lexerDef.h	**25 typeCheck.c**	**40 util.h**
11 parser.c	**26 typeCheck.h**	**41 config.h**
12 parser.h	**27 codeGen.c**	**42 driver.c**
13 parserDef.h	**28 codeGen.h**	**43 archive.c**
14 treeNodePtr_stack.c	**29 error.c**	**44 archive.h**
15 treeNodePtr_stack.h	**30 error.h**	**45 coding_Details(stage 2).pdf**

Semantic analysis and type checking testcases

46 t1.txt	**51 t6.txt**
47 t2.txt	**52 t7.txt**
48 t3.txt	**53 t8.txt**
49 t4.txt	**54 t9.txt**
50 t5.txt	**55 t10.txt**

Code Generation test cases

56 c1.txt	**62 c7.txt**
57 c2.txt	**63 c8.txt**
58 c3.txt	**64 c9.txt**
59 c4.txt	**65 c10.txt**
60 c5.txt	**66 c11_corrected.txt**
61 c6.txt	**67 c11_original.txt**

3. Total number of submitted files: **67** (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): **None**

8. **Data Structures** (Describe in maximum 2 lines and avoid giving C definition of it)
    a. **AST node structure**: AST Node struct contains grammar symbol (as label by which it is referred), (start, end) line numbers for scope information, *pointers* to tokenInfo (from lexer), corresponding symbol table entry (in case it is ID node) and *pointers* to the parent, sibling and child AST Node of the node.
    b. **Symbol Table structure:** Symbol Table structure contains a hash table of symbol table nodes, Pointers to a nested scope, parent scope and sibling scope symbol tables, a pointer to the START node of that scope in the AST, scope's function name and the size occupied by the scope. Each symbol table node contains a lexeme and a union of structure for function information and structure for variable information.
    c. **array type expression structure:** The variable type structure contains an Enum for the base type, an Enum for the kind of array, width of that array or variable, and unions of symbol table entry and integer for the start index and the end index.
    d. **Input parameters type structure:** An input parameter node contains the lexeme of the node, line number of its declaration, a structure for variableType, the offset of that node and Enums for ease in performing ‘for’ and ‘while’ loop semantic checks.
    e. **Output parameters type structure:** An output parameter node contains the lexeme of the node, line number of its declaration, a structure for variableType, the offset of that node, Enums for ease in performing ‘for’ and ‘while’ loop semantic checks and a boolean flag to know whether that variable was ever assigned or not.
    f. Structure for maintaining the three address code(if created) : **NA**

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 :** No entry in current scope, parent scope and all ancestor scope symbol tables including the function's input and output lists.
    b. **Multiple declarations:** Entry in the current scope symbol table or in the function's output list if current scope is the function's local variable root scope.
    c. **Number and type of input and output parameters:** Stored in Function Table entry of every module.
    d. **Assignment of value to the output parameter in a function:** An entry in the output list node of a variable keeps track whether it was assigned or not. The output list is traversed and checked after processing of a function.
    e. **function call semantics:** Actual and formal parameters matched for type checks by using symbol table entries
    f. **static type checking :** Symbol Table entries checked for match of bounds, bottom up computation of expression types traversing the AST
    g. **return semantics:** Return variables' type expressions matched with corresponding receiving variables in moduleReuseStatement in caller (using symbol table nodes' information).
    h. **Recursion :** If the current scope function's name matches with the name of the function used in the moduleReuse statement, then it is reported as a Recursion error.
    i. **module overloading:** Module status DEFINED in Function table entry.
    j. **'switch' semantics :** data type of switch variable used to mandate DEFAULT and case values' repetition checked (list search) [integer variable] or exactly one TRUE and FALSE [boolean variable], relevant AST Nodes used.
    k. **'for' and 'while' loop semantics:** FOR: iterator redeclaration or assignment (Symbol Table scopes' searched), WHILE: mandate assignment of some condition variable by algorithm (loop level, and variable assignment level tracked).
    l. **handling offsets for nested scopes:** global offset maintained for a function scope, incremented at every declaration.
    m. **handling offsets for formal parameters:** global offset maintained for a input/output variables' scope, incremented at every declaration. This is reset to zero at the start of function scope.
    n. **handling shadowing due to a local variable declaration over input parameters:** input variables' not searched while checking for redeclaration inside module scope (Output variables' checked: redeclaration disallowed).
    o. **array semantics and type checking of array type variables:** Array.baseType and Statically available bounds matched, OOB error thrown appropriately, Matching type expressions from type checker (bottom up tree traversal) for LHS and RHS array use expressions in assignment.
    p. **Scope of variables and their visibility :** Declaration in a scope (Symbol Table) visible in hierarchically nested scopes (Symbol Table). Symbol table entries store (start, end) line numbers of its scope.
    q. **computation of nesting depth:** I_O_variables_depth = 0, Module_scope_depth = 1 (Base cases) , internal_scope_depth = parent_scope_depth + 1;

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 bytes** (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): **RBX (base of Input/Output Lists) ,**
     R8, R9, R10, R11, R12, R13 (computation)

e. Mention the physical sizes of the integer, real and boolean data as used in your code generation module
size(integer): **2 words** (in words/ locations), **4 bytes** (in bytes)
size(real): **4 words** (in words/ locations), **8 bytes** (in bytes)
size(boolean): **1 word** (in words/ locations), **2 bytes** (in bytes)

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

The caller places the input parameters on the stack and then reserves space for the output parameters. It then pushes RBP, RBX to preserve them (base pointers for caller's Activation Record/IOList), and then sets RBX to the base of IOList of callee. Finally it calls the function, placing the instruction pointer on stack (implicitly). The callee stores the stack pointer in RBP (base of callee's activation record now). It reserves space for its local variables on the stack. At the end of function, it removes its activation record from stack, and returns.

g. Specify the following:

- **Caller's responsibilities:** placing the input parameters, reserving place for output parameters, preserving its own base pointers, copying the values of output parameters into local variables (after the control returns).
- **Callee's responsibilities:** restoring the stack by removing its activation record, placing the output parameters in the IOList area, returning control.

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

After parameters have been assigned space, caller pushes RBP, RBX to preserve them (base pointers for its own Activation Record/IOList), and sets RBX to the base of IOList of callee. It calls the function, placing the instruction pointer on stack (implicitly). The callee stores the stack pointer in RBP (base of its activation record now). It reserves space for its local variables on the stack. At the end of function, it removes its activation record from stack, and returns using the address which is now at top of stack.

i. **How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee?** Parameters are placed in IOList area using the statically computed offsets of parameters. Callee uses these to reference the parameters with respect to RBX.

j. **How is a dynamic array parameter receiving its ranges from the caller?** The callee places the base address, lower bounds, and upper bounds in the 5 width reserved by dynamic array parameter.

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

l. **register allocation (your manually selected heuristic) :** RSP for stack pointer, RBP for base pointer of an activation record, RBX for base of parameters of the function, registers R8, R9, R10 and R11 as temporaries for calculations, R12, R13 whose values are preserved by function calls (call-save registers) for implementing loops.

m. **Which primitive data types have you handled in your code generation module?(Integer, real and boolean):** Integer, Real and Boolean constants and variables for *get_value(..)* and *print(..)*;

Integer and Boolean arithmetic; Arrays of base type Integer and Boolean *(it was announced that there would be no expressions/arrays of base type real in code generation test cases)*.

n. **Where are you placing the temporaries in the activation record of a function?** Temporaries are temporarily placed on the stack during code generation and removed after use. No space is being reserved for them.

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] :

Semantic analysis and type checking test cases:

i.	t1.txt (in ticks)	**1512.00**	and (in seconds)	**0.001512**
ii.	t2.txt (in ticks)	**1204.00**	and (in seconds)	**0.001204**
iii.	t3.txt (in ticks)	**2373.00**	and (in seconds)	**0.002373**
iv.	t4.txt (in ticks)	**1966.00**	and (in seconds)	**0.001966**
v.	t5.txt (in ticks)	**2142.00**	and (in seconds)	**0.002142**
vi.	t6.txt (in ticks)	**2985.00**	and (in seconds)	**0.002985**
vii.	t7.txt (in ticks)	**3346.00**	and (in seconds)	**0.003346**
viii.	t8.txt (in ticks)	**3544.00**	and (in seconds)	**0.003544**
ix.	t9.txt (in ticks)	**5369.00**	and (in seconds)	**0.005369**
x.	t10.txt (in ticks)	**1382.00**	and (in seconds)	**0.001382**

Code Generation test cases:

xi.	c1.txt (in ticks)	**612.00**	and (in seconds)	**0.000612**
xii.	c2.txt (in ticks)	**766.00**	and (in seconds)	**0.000766**
xiii.	c3.txt (in ticks)	**1020.00**	and (in seconds)	**0.001020**
xiv.	c4.txt (in ticks)	**1178.00**	and (in seconds)	**0.001178**
xv.	c5.txt (in ticks)	**1191.00**	and (in seconds)	**0.001191**
xvi.	c6.txt (in ticks)	**1453.00**	and (in seconds)	**0.001453**
xvii.	c7.txt (in ticks)	**798.00**	and (in seconds)	**0.000798**
xviii.	c8.txt (in ticks)	**686.00**	and (in seconds)	**0.000686**
xix.	c9.txt (in ticks)	**757.00**	and (in seconds)	**0.000757**
xx.	c10.txt (in ticks)	**891.00**	and (in seconds)	**0.000891**

xxi. c11_corrected.txt (in ticks) **1565.00** and (in seconds) **0.001565**

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 (compiler implements all the language features)_____
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 -felf64 code.asm && gcc code.o && ./a.out

    _____

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: **c11.txt had syntactic and semantic errors, so we have included two files in the Group folder, namely c11_original.txt (The original file provided on Nalanda) and c11_corrected.txt (The file with all the errors rectified such that it successfully compiles now).**
19. Declaration: We, **MADHUR PANWAR, TUSSANK GUPTA, SALMAAN SHAHID, APURV BAJAJ** and **HASAN NAQVI** 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:	**2016B4A70933P**	Name:	**MADHUR PANWAR**
ID:	**2016B3A70528P**	Name:	**TUSSANK GUPTA**
ID:	**2016B4A70580P**	Name:	**SALMAAN SHAHID**
ID:	**2016B3A70549P**	Name:	**APURV BAJAJ**
ID:	**2016B5A70452P**	Name:	**HASAN NAQVI**

Date: **21/04/2020**
