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

Group Number

32

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:2016B4A70622P	Name:Satwadhi Das
ID:2016B4A70166P	Name:Aaryan Kapoor
ID:2016B3A70534P	Name:Ishan Joglekar
ID:2016B4A70632P	Name:Satyansh Rai
ID:2016B5A70560P	Name:Parth Misra

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

1 lexerDef.h
2 lexer.h
3 lexer.c
4 parserDef.h
5 parser.h
6 parser.c
7 astDef.h
8 ast.h
9 ast.c
10 symbolTableDef.h
11 symbolTable.h
12 symbolTable.c
13 typeCheckerDef.h
14 typeChecker.h
15 typeChecker.c
16 codeGeneration.h
17 codeGeneration.c
18 grammarHelper.c
19 makefile
20 NewGrammar.txt
21 t1.txt - t10.txt
22 c1.txt - c11.txt
23 Grammar.txt
24 lexerHelper.c
25 newdriver.c
26 Codingdetails

***NOTE: typechecker.c contains both typechecking module and semantic analyser module.**

3. Total number of submitted files: 45 (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): c 1-10
   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): c11

8. **Data Structures** (Describe in maximum 2 lines and avoid giving C definition of it)

a. AST node structure: contains unions corresponding to number of children it has/whether it is a treenode, type which is an enum corresponding to its name, and link to next node in AST

b. Symbol Table structure: Function Table corresponding to each function connected to a tree of corresponding symbol tables, each contained 200 hash keys, parent and child links and scope

c. array type expression structure: Structure contains datatype denoted by integer, start and end index which is a union for Num/ST entry, and rangekind which indicated static or dynamic

d. Input parameters type structure: linkedlist of a structure named var_l which contains name, type, size, offset, flag (for marking assigned output parameters) and link to next parameter

e. Output parameters type structure:linkedlist of a structure named var_l which contains name, type, size, offset, flag (for marking assigned output parameters) and link to next parameter

f. Structure for maintaining the three address code(if created) : Not created

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: output parameter structure has flag

e. function call semantics: function table entry has status and line number when first encountered

f. static type checking : comparing corresponding symbol table entries which have datatype

g. return semantics: activation record stores return offsets as subtracted from base pointer

h. Recursion : passing current function to typechecker

i. module overloading: function table entry already found populated

j. 'switch' semantics : Case Statements linkedlist checked based on switch variable

k. 'for' and 'while' loop semantics: maintaining active for and while variables

l. handling offsets for nested scopes: oset is a global variable that goes into various function calls

m. handling offsets for formal parameters: resetting offset before input and after corresponding output list

n. handling shadowing due to a local variable declaration over input parameters: Input parameters maintained in function table, checked on declaration

o. array semantics and type checking of array type variables: type maintained in symbol table and range maintained for comparison

p. Scope of variables and their visibility : checking current and parent chain of symbol tables for entry

q. computation of nesting depth: recursive call in which child number and relevant symbol table is passed

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 = 1 (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

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), 2 (in bytes)
size(real): not implemented in codegen in words/ locations), not implemented in codegen (in bytes)
size(boolean): 1 (in words/ locations), 1 (in bytes)

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

Once we reach a module reuse statement, input parameters are passed by placing on top of the stack followed by the caller's base pointer. The base pointer and stack pointer are then updated as required by the callee. After function execution, we go back to caller activation record and restore caller's base pointer and caller's stack pointer. Using these two values and return parameters offset, the valued is stored where required.

g. Specify the following:

- Caller's responsibilities: Putting actual parameters on top of stack
- Callee's responsibilities: Saving original, updating base pointer with new location, updating output values in callers memory and restoring callers base pointer after execution

h. How did you maintain return addresses? (write 3-5 lines): Once we reach a module reuse statement, input parameters are passed by placing on top of the stack followed by the callers base pointer. After function execution, we go back to caller activation record and restore caller's base pointer and caller's stack pointer. Using these two values and return parameters offset, the valued is stored where required.

i. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee? Caller copies the value of the input parameter on top of its stack. Callee uses offset with respect to new base pointer

j. How is a dynamic array parameter receiving its ranges from the caller? It copies value from actual parameter

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

l. register allocation (your manually selected heuristic) : Registers allocated memory as and when required, no optimization done

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

n. Where are you placing the temporaries in the activation record of a function? On top of stack

11. **Compilation Details**:

a. Makefile works (yes/No): YES

b. Code Compiles (Yes/ No): YES

c. Mention the .c files that do not compile: N/A

d. Any specific function that does not compile: N/A

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) 1747 and (in seconds) 0.001747
    - ii. t2.txt (in ticks) 1656 and (in seconds) 0.001656
    - iii. t3.txt (in ticks) 2492 and (in seconds) 0.002492
    - iv. t4.txt (in ticks) 2228 and (in seconds) 0.002228
    - v. t5.txt (in ticks) 2209 and (in seconds) 0.002209
    - vi. t6.txt (in ticks) 3004 and (in seconds) 0.003004
    - vii. t7.txt (in ticks) 3239 and (in seconds) 0.003239
    - viii. t8.txt (in ticks) 3561 and (in seconds) 0.003561
    - ix. t9.txt (in ticks) 3748 and (in seconds) 0.003748
    - x. t10.txt (in ticks) 1894 and (in seconds) 0.001894

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): Not printing parse tree incase of syntax error
15. Are you availing the lifeline (Yes/No): No
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 codegen.asm && gcc -no-pie -g codegen.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: Starting line for various scopes has been taken from the lines before the start token, for example the like in which function is declared, where while and for are written etc. This has been done as start is not stored in AST to save memory, and will not affect any semantic checks.
19. Declaration: We, Satwadhi Das, Aaryan Kapoor, Ishan Joglekar, Satyansh Rai, Parth Misra (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:2016B4A70622P	Name:Satwadhi Das
ID:2016B4A70166P	Name:Aaryan Kapoor
ID:2016B3A70534P	Name:Ishan Joglekar
ID:2016B4A70632P	Name:Satyansh Rai
ID:2016B5A70560P	Name:Parth Misra

Date: 20/04/2020

Should not exceed 6 pages.