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

Group Number
21

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: 2017A7PS0134P	Name: S Hariharan
ID: 2017A7PS1174P	Name: Praveen Ravirathinam
ID: 2017A7PS1195P	Name: Anirudh Srinivasan Chakravarthy
ID: 2016B2A70770P	Name: Honnesh Rohmetra

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

1 driver.c	7 semantictester.c	13 symboltableutils.c	19 lexerDef.h	25 parserutils.c
2 codegen.c	8 typeExtractor.c	14 symboltableutils.h	20 hash.c	26 parserutils.h
3 codegen.h	9 typeExtractor.h	15 ast.c	21 hash.h	27 parsertester.c
4 codegentester.c	10 symboltable.c	16 ast.h	22 parser.c	28 lexertester.c
5 semantics.c	11 symboltable.h	17 lexer.c	23 parser.h	29 makefile
6 semantics.h	12 symboltableDef.h	18 lexer.h	24 parserDef.h	30 grammar.txt

3. Total number of submitted files: 30 (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): c1.txt, c2.txt, c3.txt, c4.txt, c5.txt, c6.txt, c8.txt, c9.txt, c10.txt
   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): c7.txt (malloc abort), c11.txt

8. **Data Structures** (Describe in maximum 2 lines and avoid giving C definition of it)
   a. AST node structure: Each node contains a union for non-terminal or terminal data (with a marker to distinguish), datatype information, child and sibling pointers, pointer to the symbol table for the scope (for, while, module etc), pointer to symbol table entry (arithmetic/boolean expressions), start and end line number (for scope information). We also store for identifiers whether it is a redeclaration or not (to flag relevant errors).

   b. Symbol Table structure: Each entry in the table contains identifier information, data type, offset, marker for array or primitive type, flag to indicate whether variable has been updated (for output parameters), flag to mark loop variable (to prevent assigning loop variable). Symbol table is a list of buckets corresponding to different hash values, which stores start and end line number, parent, child and sibling pointers (to maintain hierarchy), and each bucket contains a linked list of entries.

   c. array type expression structure: Primitive datatype, lower and upper bounds (as tokens), flag whether dynamic or static array.

   d. Input parameters type structure: A linked list where each entry contains datatype information, identifier information (token), and a pointer to the next parameter in the list.

   e. Output parameters type structure: A linked list where each entry contains datatype information, identifier information (token), and a pointer to the next parameter in the list.

   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: symbol table entry not present

   b. Multiple declarations: symbol table entry already populated

   c. Number and type of input and output parameters: traversal of AST and parameter linked list.

   d. assignment of value to the output parameter in a function: marker in symbol table whether parameter updated or not.

   e. function call semantics: declaration and definition line number comparison with current line number, type and number check between formal and actual parameters.

   f. static type checking: compare AST datatype fields (use symbol table entries for arrays).

   g. return semantics: traversal of output parameters to check whether not updated.

   h. Recursion: compare names of current symbol table and called symbol table.

   i. module overloading: entry already exists in function symbol table.

   j. 'switch' semantics: AST datatype check of switch variable (INT, BOOLEAN - accordingly determine presence/absence of default statement). Then proceed for type check on each case value, don't traverse subtree if error.

   k. 'for' and 'while' loop semantics: type check (FOR – integer, WHILE - boolean) using AST (WHILE) and symbol table (FOR) datatype. Loop variable symbol table entry must indicate not updated.

l. handling offsets for nested scopes: continue with counter of parent scope, add new child to hierarchy of symbol tables.

m. handling offsets for formal parameters: start from 0, assign for input and output parameters, then proceed to local variables.

n. handling shadowing due to a local variable declaration over input parameters: NA

o. array semantics and type checking of array type variables: symbol table array type information comparison (bound check for static).

p. Scope of variables and their visibility: hierarchy of symbol tables.

q. computation of nesting depth: children pointers of symbol tables.

10. Code Generation:

a. NASM version as specified earlier used (Yes/no): No (we were only able to get 2.13.02 working – as indicated by email only one member had a functioning Linux machine)
b. Used 32-bit or 64-bit representation: 64-bit
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): R8- lower array bound, R9- upper bound, R10- array index, R15- FOR loop variable
e. Mention the physical sizes of the integer, real and boolean data as used in your code generation module
size(integer): 1 (in words/ locations), 8 (in bytes)
size(real): 1 (in words/ locations), 8 (in bytes)
size(booelan): 1 (in words/ locations), 8 (in bytes)

f. How did you implement functions calls?(write 3-5 lines describing your model of implementation)
Before the call, caller pushes input parameters and then output parameters onto stack (in reverse order). Output parameters are placed symbolically to reserve space for when the callee places these values. The function call is now performed, and the callee picks up input parameters from the shared space and copies their values to the offsets defined for formal parameters. Before return from the callee, the output parameters are placed into the reserved space (which was done by caller. The caller now places output parameters from this shared space into the offsets for these variables.

g. Specify the following:

- Caller's responsibilities: Pushing input and output parameters (to reserve space) onto the stack in the shared space, placing returned output parameters back to actual locations.

- Callee's responsibilities: Pick input parameters from shared space to initialize formal parameters, and place return values into shared space for output parameters.

h. How did you maintain return addresses? (write 3-5 lines): On entering a function, push RBP onto the stack which was pointing to base of the caller’s activation record. Now set RBP to the base of callee’s activation record. While returning simply pop from the location storing the base address of the caller’s activation record (i.e RBP).

i. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee? Offsets are assigned first to input then output parameters, followed by local variables in a module. The caller places the parameters (input, output) into the shared space before the call, and the callee now picks these parameters to initialize input parameters into their respective offset locations, and places the output parameters from the offset locations back into the shared space.

j. How is a dynamic array parameter receiving its ranges from the caller? NA

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

l. register allocation (your manually selected heuristic) : R8 for data storage and array lower bound, R9 for array upper bound, R10 for array index, R15 for FOR loop variable.

m. Which primitive data types have you handled in your code generation module?(Integer, real and boolean): integer, Boolean (not tested real).

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

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) No (as indicated by email only one member had a functioning Linux machine)
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) 1337 and (in seconds) 0.001337
    ii. t2.txt (in ticks) 1162 and (in seconds) 0.001162
    iii. t3.txt (in ticks) 1449 and (in seconds) 0.001449
    iv. t4.txt (in ticks) 1635 and (in seconds) 0.001635
    v. t5.txt (in ticks) 1661 and (in seconds) 0.001661
    vi. t6.txt (in ticks) 2386 and (in seconds) 0.002386
    vii. t7.txt (in ticks) 2068 and (in seconds) 0.002068
    viii. t8.txt (in ticks) 2449 and (in seconds) 0.002449
    ix. t9.txt (in ticks) 2506 and (in seconds) 0.002506
    x. t10.txt (in ticks) 1533 and (in seconds) 0.001533

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) Dynamic arrays as parameters, shadowing of input parameters.
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 -f elf64 code.asm
gcc -no-pie -m64 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: Updated Grammar, and hence AST, is not left associative, therefore 2*3 – 56 + 3 is treated as 2*3 - (56+3). Code generated for c7.txt is correct but unable to test due to malloc and scanf issues in NASM(as indicated by email, but we were unable to reach a resolution over email). In addition as discussed over email, two of our members were unable to contribute due to network restriction and unavailability of laptop repair shops. One member had a MacOS making code generation all the more difficult for Linux due to different NASM specifications. We have implemented to the best of our ability given these challenging circumstances and we request for your due consideration towards our situation during evaluation.

19. Declaration: We, Anirudh Srinivasan Chakravarthy, S Hariharan, Praveen Ravirathinam and Honnesh Rohmetra, 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 2017A7PS0134P	Name: S Hariharan
ID 2017A7PS1174P	Name: Praveen Ravirathinam
ID 2017A7PS1195P	Name: Anirudh Srinivasan Chakravarthy
ID 2016B2A70770P	Name: Honnesh Rohmetra

Date: 20/04/2020

Should not exceed 6 pages.