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

Group Number
16

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: 2017A7PS0062P	Name: GANDHI ATITH NIKESKUMAR
ID: 2017A7PS0097P	Name: BURHAN BOXWALLA
ID: 2017A7PS0140P	Name: KESHAV SHARMA
ID: 2017A7PS1180P	Name: SHRAY MATHUR
ID: 2017A7PS1181P	Name: RAJ SANJAY SHAH

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

1 lexerDef.h	7 astDef.h	13 assemblerDef.h	19 Keywords.txt
2 lexer.c	8 ast.c	14 assembler.c	20 ASTWords.txt
3 lexer.h	9 ast.h	15 assembler.h	21 Coding Details Pro Forma
4 parserDef.h	10 symbolDef.h	16 driver.c	22 All test cases from c1-c11.txt
5 parser.c	11 symbolTable.c	17 makefile	23 All test cases from t1-t10.txt
6 parser.h	12 symbolTable.h	18 Grammar.txt	24 tags
25 Test.txt			

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 test cases (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: Each node contains a pointer to its parent, first child, next sibling, previous sibling, number of children, a union of non-terminal name/pointer to lexeme, pointer to symbol table entry and a union for storing the type. The union for type stores the data type for 'ID' terminals and for Array IDs, it even stores the start and end index of the array. In case of non-terminals, this union helps to propagate the type of its corresponding sub-tree.

   b. Symbol Table structure: Each scope has its own symbol table and pointer to parent symbol table for nested scope, Function name corresponding to it, pointer to the next symbol table with same level,pointer to child symbol table, level and other metadata explained in the below points.

   c. array type expression structure: For an array variable its corresponding symbol table entry has the following fields populated with meaningful values :- array start, array end lexeme/number, element type, variable name, check whether the array is static or dynamic, offset, line number from where its scope starts and size.

   d. Input parameters type structure: Linked list of parameters with each node containing variable name, data type, pointer to next node, offset, size, tag for array type.

   e. Output parameters type structure: Linked list of parameters with each node containing variable name, data type, pointer to next node, offset and size

   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 Parameter lists and Matching

   d. assignment of value to the output parameter in a function : Marking if getvalue or assignment statement found for corresponding output parameter variable

   e. function call semantics: Traversal of parameter list and matching types, Redundant declaration of module handled by comparing the line number of declaration, definition and first use.

   f. static type checking : Comparing Start and End Indices in case of arrays, Checking the type of left and right subtree for arithmetic, logical and relational operators, comparing left and right hand side type for assignment statements.

   g. return semantics: Traversing output_plist, optional ast node and matching corresponding types.

   h. Recursion: Recursion not allowed by matching caller's name to callee's name.

   i. module overloading: Symbol Table entry found. No redefinition allowed.

j. 'switch' semantics : Case statement with real not allowed, boolean can't have default, integer must have default: done by traversing AST and checking number of children. Type checking done by traversing caseStmt subtree.

k. 'for' and 'while' loop semantics: Integer type for For loop variable, boolean type for while expression by traversing AST, Traversing For loop sub-tree to ensure no assignment or getvalue statement on for loop iterator, compared lower and higher range in case of for expression.

l. handling offsets for nested scopes: Reset offsets only for new function definition. Else keep adding in the function's offset.

m. handling offsets for formal parameters: Included in function's activation record and working fine. Hence input_plist has an offset of 0.

n. handling shadowing due to a local variable declaration over input parameters: Made a separate Symbol Table between Function and Main scope.

o. array semantics and type checking of array type variables: Only integer type array indices, Bound checking by comparing with start and end index, Type and Bound compared between array types for assignment statement (eg A:=B), Compared end and start index bounds of array to ensure start <= end, and no relational, arithmetic or logical operations allowed on them

p. Scope of variables and their visibility : Variable usage valid from declaration line to corresponding scope's END keyword.

q. computation of nesting depth: Traversing the symbol table and incrementing count till we reach root symbol Table.

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 = 2 (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): RSI for storing Caller Function's Base Pointer

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

size(integer):	4 (in locations),	4 (in bytes)
size(real):	8 (in locations),	8 (in bytes)
size(boolean):	1 (in locations),	1 (in bytes)

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

During Call, RSI, RBP store the base address of Caller and Callee respectively. Traversing the parameter lists of input, values are copied corresponding to their offsets and only the start address copied for array type variables. Pushed the value of RSI to store base address. While returning, the stack pointer is first decremented, the original base pointer is popped out of the stack into RSI. RBP now contains Callee's base pointer and RSI contains Caller's. Traversing the return list, values are copied into the caller's variables using the two base pointers. Finally RSI is moved into RBP.

g. Specify the following:

   - Caller's responsibilities: To copy variables to the Callee function's Input parameters, push the base pointer and make a new stack frame for Callee to execute.

- Callee's responsibilities: Copy returned variables to caller's actual variables, bring back the base pointer by popping and bring back original stack pointer.

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

An array called ModuleSpace in the data section contains the amount moved up by the stack pointer(In short contains amount of memory used by dynamic arrays + Activation record size). During function call, we increase the index for this array and store the activation record plus dynamic size in the new index. After the function completes,we move down the stack pointer by the size stored in that index of Module Space, decrement the index and hence come back to the caller's index and perform operations there.

i. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee?

By storing base pointers of both functions temporarily in RSI and RBP, we are traversing both lists and copying variables during both calling time and return time using the offsets computed. Each variable is accessed by the address of the (function's base pointer - the offset).

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

Each array has two additional spaces storing it's bounds which are copied into callee's variable location unless the formal parameter is a static array parameter, then the bounds are checked wrt Caller's array.

k. What have you included in the activation record size computation? (local variables, parameters, both): Both. But we have printed excluding parameters according to printing requirements

l. register allocation (your manually selected heuristic) :

RAX and RBX register used for temporary calculations, RSI and RDI register used for array index calculation, RCX register used for Loop counter, RDX for printing purposes ,RBP and RSP as per standard definition.

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?
We have pushed the temporary evaluation on top of the stack of current activation record. When needed, values are popped out and used.

11. **Compilation Details**:
    a. Makefile works (yes/No): YES
    b. Code Compiles (Yes/ No): YES
    c. Mention the .c files that do not compile: NONE
    d. Any specific function that does not compile: NONE
    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. c1.txt (in ticks) : 920 and (in seconds) : 0.000920
    ii. c2.txt (in ticks) : 895 and (in seconds) : 0.000895
    iii. c3.txt (in ticks) : 731 and (in seconds) : 0.000731
    iv. c4.txt (in ticks) : 1127 and (in seconds) : 0.001127

v. c5.txt (in ticks) : 571 and (in seconds) : 0.000571

vi. c6.txt (in ticks) : 548 and (in seconds) : 0.000548

vii. c7.txt (in ticks) : 1098 and (in seconds) : 0.001098

viii. c8.txt (in ticks) : 800 and (in seconds) : 0.000800

ix. c9.txt (in ticks) : 649 and (in seconds) : 0.000649

x. c10.txt (in ticks) : 970 and (in seconds) : 0.000970

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)
    Operations on Real Valued Variables (Arithmetic/Relational).

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 -l code.lst code.asm
    gcc -no-pie -m64 -o code code.o
    ./code
    If the 2nd argument i.e the name of the .asm file is not mentioned, the code is automatically generated in code.asm. If the assembly code is made in file 'code.asm', 'make run' command in the terminal will run the file code.asm using the above 3 instructions automatically.

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:

    Semantic Check is strongly robust with close to 35 Unique Semantic Checks.

    We have included curly braces '{' and '}' which act as START and END lexemes respectively like in the case of C. Also, we have incorporated get_value and print functions even for real data types, including arrays of both static and dynamic type.

    We have incorporated code for Type checking and Semantic rules alongside Symbol table creation in the file symbolTable.c. Hence, some Semantics are handled during Symbol table creation and there are additional two traversals of the Abstract Syntax Tree for the remaining Semantic Analysis and Type Checking respectively.

    We have stored output_plist in the same symbol table as the main scope of the function (as the input parameter list could have been shadowed but not output parameter list, we created an intermediate symbol table only for input parameter list), however output_plist's scope level will be printed out as 0 (same as that of input_plist). This results in some overlap in the 0s and 1s while printing the scope level column, which is only caused because of the global scope of the function (printed as 1) and output_ plist (printed as 0). All other scope levels are printed without any discontinuities in ascending order. This can be checked by running the file Test.txt.

19. Declaration: We, Gandhi Atith Nikeshkumar, Burhan Boxwalla, Keshav Sharma, Shray Mathur and Raj Sanjay Shah 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: 2017A7PS0062P Name: Gandhi Atith Nikeshkumar
ID:2017A7PS0097P Name: Burhan Boxwalla
ID:2017A7PS0140P Name: Keshav Sharma
ID:2017A7PS1180P Name: Shray Mathur
ID: 2017A7PS1181P Name: Raj Sanjay Shah

Date: 20th April 2020

Should not exceed 6 pages.