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

Group Number

48

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: 2016B4A70487P Name: Puneet Anand

   ID: 2016B1A70703P Name: Mayank Jasoria

   ID: 2016B4A70935P Name: Shubham Tiwari

   ID: 2016B4A70594P Name: Vibhav Oswal

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

1. driver.c	8. parser.h	15. hash_map.c	22. linked_list.h	29. utils.h
2. grammar_new.erp	9. lexerDef.c	16. stack.c	23. ast.c	30. utils.c
3. Makefile	10. parserDef.h	17. stack.h	24. ast.h	31. codegen.h
4. lexer.c	11. n_ary_tree.h	18. queue.c	25. astDef.h	32. codegen.c
5. lexer.h	12. n_ary_tree.c	19. queue.h	26. semanticAnalyzer.c	33. symbolTable.h
6. parser.c	13. hash_map.h	20. linked_list.c	27. semanticAnalyzer.h	34. coding details.pdf
7. symbolTable.c	14. codeGenDef.h	21. stDef.h	28. typeChecker.c	

   Additionally, all 21 testcase files of stage 2 (t1.txt to t10.txt and c1.txt to c11.txt) renamed to include a prefix "testcase_" to the existing testcase files.

3. Total number of submitted files: (All files should be in **ONE** folder named exactly as Group number) **55 [including all testcases and coding details.pdf]. Otherwise 33**
4. Have you mentioned names and IDs of all team members at the top of each file (and commented well)? (Yes/ no) [Note: Files without names will not be evaluated] **Yes**
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, c7.txt, c8.txt, c9.txt, c10.txt, c11.txt

[produces correct output for all test cases]

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: AST node structure contains a union of structures which hold data specific to the AST node, along with a tag field to indicate the type of data contained in the node. It also contains fields for next, previous, parent and child node pointers which help in traversing the abstract syntax tree.
   b. Symbol Table structure: Symbol Table structure is a global hash table for storing function names and its activation record sizes. Each function further has a symbol table for its input and output parameters, and one for its local variables.
   c. array type expression structure: Array type expression is computed using symbol table and AST. It contains fields for module name, scope line number, variable name, whether the array is static or dynamic, its range, and type of its elements.
   d. Input parameters type structure: It is a linked list of input parameters, associated with every function in its symbol table structure.
   e. Output parameters type structure: It is a linked list of output parameters, associated with every function in its symbol table structure.
   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 found.
   b. Multiple declarations: Symbol table entry within same scope already populated.
   c. Number and type of input and output parameters: Traversal of linked list of parameters to compare number and type of parameters.
   d. assignment of value to the output parameter in a function: Check if field in symbol table for variable assignment has been set.
   e. function call semantics: Symbol table entry for function not found.
   f. static type checking: if lower bound and upper bound are of type NUM, then check if lower bounds, upper bounds, and element types of both arrays are same. Trivial check for primitives, synthesized form AST leaf.
   g. return semantics: Symbol table for output parameters empty or populated, assignment operator use.
   h. Recursion: Module name of current scope matches with name of function called.
   i. module overloading: Symbol table entry for function already populated.
   j. 'switch' semantics: Switch variable type and assignment, case labels, and presence of default.

k. 'for' and 'while' loop semantics: loop variable type and assignment (for), expression type and updation of at least one expression variable (while),

l. handling offsets for nested scopes: Nested symbol table structure with AST to manage nested scopes.

m. handling offsets for formal parameters: Using two separate symbol tables.

n. handling shadowing due to a local variable declaration over input parameters: Using a separate symbol table to allow shadowing.

o. array semantics and type checking of array type variables: Type checking as in primitive types, with the addition of static bound checking using the lower and upper bounds present in symbol table record for array, or dynamic bound checking by accessing values from offsets of 'bound' variables.

p. Scope of variables and their visibility: Determined by scope of symbol table in which the entry for variable exists.

q. computation of nesting depth: Nesting level of symbol tables.

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, rcx, rdx as general-purpose registers. r10w and r11w for lower and upper bounds of arrays, xmm0 and xmm1 registers for printing float values, FPU stack for floating point computation.

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): **4** (in words/ locations), **4** (in bytes)
size(booelan): **1** (in words/ locations), **1** (in bytes)

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

1. Push and assign a new stack frame (activation record).
2. Copying input parameters from caller to callee using offsets of actual arguments relative to the caller's base, to formal parameters using offsets relative to callee's base.
3. Executing a call instruction to the function. The function executes as needed
4. Return using ret. [The functions themselves will always clear any extra content that they add to the stack, to reset it to the state which it was it when the function was called]
5. Remove the callee's stack frame (activation record), reset these markers to demarcate the caller's activation record, then copy all output parameters from callee to caller using relative offsets from the callee's base and caller's base respectively

g. Specify the following:

- Caller's responsibilities: Passing input parameters to callee's activation record, retrieving output parameters from callee's activation record.
- Callee's responsibilities: Ensure that stack operations are balanced, assigning values to all output parameters

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

The call instruction pushes the instruction pointer into the stack. On executing ret, the value of the instruction pointer is restored by popping it off the stack.

i. How have you maintained parameter passing? How were the statically computed offsets of the parameters used by the callee? Caller passes the parameters to callee, and retrieves output parameters from callee. The statically computed offsets of input and output parameters are used to compute their addresses relative to the base pointer for use within the function.

j. How is a dynamic array parameter receiving its ranges from the caller? Along with memory for its pointer, extra memory is allocated to store its lower and upper bounds. The caller simply populates the lower and upper bounds fields into the callee's activation record.

k. What have you included in the activation record size computation? (local variables, parameters, both): Only local variables. (Parameters are treated to be within the activation record during the initial step of parameter passing, but outside it thereafter.)

l. register allocation (your manually selected heuristic): rax, rbx, rcx ,and rdx have been used as general purpose registers, rcx has additionally been used as a loop counter, rdx has been used to store rbp-offset at various places. r10w and r11w have been used for storing the lower and upper bounds of arrays. Additionally, rdx and r9 have been used for storing the base pointer + offset couple, respectively.

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

n. Where are you placing the temporaries in the activation record of a function? At the top of the activation record, after all the local (including nested) 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) **Yes**
12. Execution time for compiling the test cases [lexical, syntax and semantic analyses including symbol table creation, type checking and code generation] : **Below are average of 3 tests**
    i. t1.txt (in ticks) **2069.33** and (in seconds) **0.002069**
    ii. t2.txt (in ticks) **1836.67** and (in seconds) **0.001836**
    iii. t3.txt (in ticks) **2635.67** and (in seconds) **0.002635**
    iv. t4.txt (in ticks) **2667.00** and (in seconds) **0.002667**
    v. t5.txt (in ticks) **2371.00** and (in seconds) **0.002371**
    vi. t6.txt (in ticks) **4278.00** and (in seconds) **0.004278**
    vii. t7.txt (in ticks) **4878.33** and (in seconds) **0.004878**
    viii. t8.txt (in ticks) **5181.33** and (in seconds) **0.005181**
    ix. t9.txt (in ticks) **5948.67** and (in seconds) **0.005948**
    x. t10.txt (in ticks) **1665.67** and (in seconds) **0.001665**
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 code.asm
gcc -no-pie 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:
    (i). Floating point expression evaluation has been implemented
    (ii). Instead of taking 1 memory width (= 1 byte) for pointers, we took 8 as the width to store complete 64-bit addresses. Consequently, our array variables will consume 7 bytes more memory compared to the suggested specification (and therefore the widths and offsets do not align as per numbers in the given sample symbol table, but are perfectly aligned in terms of the order of variables). It follows that the memory allocated to pointers for dynamic arrays in input parameter list will be of 12 bytes, 2 each for lower and upper bound respectively.
19. Declaration: We, Puneet Anand, Mayank Jasoria, Shubham Tiwari, Vibhav Oswal (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: 2016B4A70487P	Name: Puneet Anand
ID: 2016B1A70703P	Name: Mayank Jasoria
ID: 2016B4A70935P	Name: Shubham Tiwari
ID: 2016B4A70594P	Name: Vibhav Oswal

Date: 20th April, 2020

Should not exceed 6 pages.