Batch No. :

Group No. : 10

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

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
    a. ID: 2017A7PS0143P Name: Ujjwal Gandhi
    b. ID:2017A7PS0101P Name: Atmadeep Banerjee
    c. ID:2017A7PS0179P Name: Mayank Jain
    d. ID:2017A7PS0157P Name: Aditya Mithal
2. Mention the names of the Submitted files (Include Stage-1 and Stage-2 both)

1 ast.c	12 c9.txt	23 Makefile	34 t5.txt
2 ast.h	13 c10.txt	24 parser.c	35 t6.txt
3 astDef.h	14 c11.txt	25 parserDef.h	36 t7.txt
4 c1.txt	15 code_gen.c	26 semanticAnalyser.c	37 t8.txt
5 c2.txt	16 codeGen.h	27 symbolTable.c	38 t9.txt
6 c3.txt	17 driver.c	28 symbolTable.h	39 t10.txt
7 c4.txt	18 exprn_code_gen.c	29 symbolTableDef.h	40 parser.h
8 c5.txt	19 grammar.txt	30 t1.txt	41 PROFORMA.pdf
9 c6.txt	20 lexer.c	31 t2.txt	
10 c7.txt	21 lexer.h	32 t3.txt	
11 c8.txt	22 lexerDef.h	33 t4.txt	

3. Total number of submitted files: **41** (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'.
    1. Lexer (Yes/No): **Yes**
    2. Parser (Yes/No): **Yes**
    3. Abstract Syntax tree (Yes/No): **Yes**

4. Symbol Table (Yes/ No): **Yes**
5. Type checking Module (Yes/No): **Yes**
6. Semantic Analysis Module (Yes/ no): **Yes** (reached LEVEL **4** as per the details uploaded)
7. Code Generator (Yes/No):**Yes**

7. **Execution Status**:
    1. Code generator produces code.asm (Yes/ No): **Yes**
    2. code.asm produces correct output using NASM for testcases (C#.txt, #:1-11): **(C#1-11)**
    3. Semantic Analyzer produces semantic errors appropriately (Yes/No):**Yes**
    4. Static Type Checker reports type mismatch errors appropriately (Yes/ No): **Yes**
    5. Dynamic type checking works for arrays and reports errors on executing code.asm (yes/no): **Yes**
    6. Symbol Table is constructed (yes/no) **Yes** and printed appropriately (Yes /No): **Yes**
    7. AST is constructed (yes/ no) **Yes** and printed (yes/no) **Yes**
    8. 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)
    1. **AST node structure**: Contains following variables:label (enum value), isLeaf (flag), pointer to firstChild, sibling, parent, numberOfChildren, pointer to syntaxTreeNode
    2. **Symbol Table structure**: Each symbolTableEntry contains name, type, start & end index (int for static & entry for dynamic), offset, width, flags for isArray, isAssigned, isReturn, isStatic. =>Each entry is inserted in a hashTable (hashed on name). There is a tree of such hashTables for scopes. Each tree is inserted in a moduleHashTable (hashed on module name).
    3. array type expression structure: In ast, Varidnum node has first child as variable name, and second child as index if array. In symbol table entry the info is stored using flags as described earlier.
    4. Input parameters type structure: In ast, each inputparameter node has 2 children, first type, second id. There is a linked list of such nodes. In symbol table entry the info is stored using flags as described earlier.
    5. Output parameters type structure:In ast, each type node has id as its child. There is a linked list of such nodes. In symbol table entry the info is stored using flags as described earlier.
    6. 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.]

1. Variable not Declared : variable entry not found in symbol table.
2. Multiple declarations: variable entry already present in symbol table.
3. Number and type of input and output parameters: Traversal of linked list of actual and formal parameters and compare
4. assignment of value to the output parameter in a function: Check isAssigned flag in symbolTable entry at end of module.
5. function call semantics: (isUsed) flag to check the call has been made or not, isDefined flag to check whether module defined or not.
6. static type checking : For every operation, datatypes of all operands are checked; error raised for invalid combinations
7. return semantics : checked if returned variable is assigned and compared with the type of variable used in caller function.
8. Recursion : compare callee function name with current function name(stored in moduleHash Node)
9. module overloading: module entry already present in module hash table
10. 'switch' semantics: compare for the valid type of switch variable with all case values and handling for default.
11. 'for' and 'while' loop semantics: check is isAssigned flag at end of scope for all variables in the expression
12. handling offsets for nested scopes: Offset starts from 0 for local variables and are in continuation for all internal scopes.
13. handling offsets for formal parameters: input parameter's starting offset 0, return parameters' offsets continuing after input parameters.
14. handling shadowing due to a local variable declaration over input parameters: Local variables symbol table is child of input variables symbol table. When using a variable check in current scope, then in parent.
15. array semantics and type checking of array type variables:
16. Scope of variables and their visibility: checking symbolTable Node for current scope,if not found then recursive checking on parent SymbolTableNode.
17. computation of nesting depth: Depth of the symbolTable node in its module tree.

10. Code Generation:

1. NASM version as specified earlier used (Yes/no): **Yes**
2. Used 32-bit or 64-bit representation: **64-bit**
3. For your implementation: 1 memory word = 2 (in bytes)
4. 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, r9, r10, r11, r12, r13, r14, r15, rax, rbx, rsi, rdi**

5. Mention the physical sizes of the integer, real and boolean data as used in your code generation module
   size(integer): **2** bytes, (words/locations)
   size(real): **4** bytes, **2** (words/locations)
   size(boolean): **1** bytes, **0.5** (words/locations)
6. How did you implement functions calls?(write 3-5 lines describing your model of implementation)

   Store stack pointer in a temporary variable, push address of return parameters to stack, push actual parameters to stack (if array, push high, low, base address) and call the module. In the module, push rbp, (after execution leave, ret), then load stack pointer back with the temporary variable.

7. Specify the following:
   - Caller's responsibilities: storing stack pointer, pushing actual parameters, loading stack pointer back.
   - Callee's responsibilities: pushing rbp(base pointer), after execution-leave, ret.

8. How did you maintain return addresses? (write 3-5 lines): Temporary Variables stores the stack pointer and loads it back after the function execution.

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

   The actual parameters were pushed before the call. The input Parameters are accessed using rbp+16+offset(static) (16 due to 2 addresses pushed), output parameters are pushed by callee function and accessed using rbp-(offset - sum of size of all input parameters).

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

    8 byte Address for the dynamic array is pushed along with values of (2bytes) low and (2 bytes) high indexes from the caller side.

11. What have you included in the activation record size computation? (local variables, parameters, both): **local variables**
12. register allocation (your manually selected heuristic) :Using any available register according to size

13. Which primitive data types have you handled in your code generation module?(Integer, real and boolean): Integer, boolean
14. Where are you placing the temporaries in the activation record of a function? common temporaries for all functions defined in data segment

11. **Compilation Details**:
    1. Makefile works (yes/No): **Yes**
    2. Code Compiles (Yes/ No): **Yes**
    3. Mention the .c files that do not compile: **None**
    4. Any specific function that does not compile: **None**
    5. 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] :
    1. t1.txt (in ticks) 724 and (in seconds) 0.000724
    2. t2.txt (in ticks) 1933 and (in seconds) 0.001933
    3. t3.txt (in ticks) 2835 and (in seconds) 0.002835
    4. t4.txt (in ticks) 2673 and (in seconds) 0.002673
    5. t5.txt (in ticks) 2992 and (in seconds) 0.002992
    6. t6.txt (in ticks) 4035 and (in seconds) 0.004035
    7. t7.txt (in ticks) 3098 and (in seconds) 0.003098
    8. t8.txt (in ticks) 4726 and (in seconds) 0.004726
    9. t9.txt (in ticks) 8555 and (in seconds) 0.008555
    10. t10.txt (in ticks) 1769 and (in seconds) 0.001769

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 -o code.out code.o

    ./code.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 (g) **Good programming style** (indentation, avoidance of goto stmts etc) (h) **modular** (i) **space and time efficient**

18. Any other point you wish to mention: Width of the base pointer of array in input parameters is assumed to be 64-bits (8 bytes) as we're using 64-bit NASM. This being an implementation concern, the width of **formal parameter arrays** in the symbol table is hence **12 bytes (**Base address[8]+start index[2]+end index[2]) and **not 5 bytes** as mentioned in the sample symbol table.

19. Declaration: We, **Ujjwal Gandhi, Aditya Mithal, Mayank Jain, Atmadeep Banerjee** 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: 2017A7PS0143P Name: Ujjwal Gandhi
    ID:2017A7PS0101P Name: Atmadeep Banerjee
    ID:2017A7PS0179P Name: Mayank Jain
    ID:2017A7PS0157P Name: Aditya Mithal

    Date: 21/04/2020
