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

Group Number
33

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. ID and Names of team members

ID: **2017A7PS0023P** Name: **Akshit Khanna**

ID: **2017A7PS0077P** Name: **Aryan Mehra**

ID: **2017A7PS0429P** Name: **Vipin Baswan**

ID: **2017A7PS0030P** Name: **Swadesh Vaibhav**

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

1 - ast.c	**7 - codegen.c**	**13 - nasmcode.c**	**19 - symbolTable.c**
2 - ast.h	**8 - codegen.h**	**14 - nasmcode.h**	**20 - symbolTable.h**
3 - astDef.h	**9 - codegenDef.h**	**15 - nasmcodeDef.h**	**21 - symbolTableDef.h**
4 - parser.c	**10 - lexer.c**	**16 - semantics.c**	**22 - driver.c**
5 - parser.h	**11 - lexer.h**	**17 - semantics.h**	**23 - grammer.txt**
6 - parserDef.h	**12 - lexerDef.h**	**18 - semanticDef.h**	

3. Total number of submitted files: **23** (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 (Highest)** 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): **All 11**
    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 (we got all correct executions)**

8. **Data Structures** (Describe in maximum 2 lines and avoid giving C definition of it)
   a. AST node structure: **It is a tree node that has the child and sibling pointer. The element itself is a union of internal node and leafnode. The internal node has label, line numbers etc. Leaves have lexeme, line number, type, value etc.**
   b. Symbol Table structure: **Symbol Table is stored as a n-ary tree of hash tables. Each hash table table is made of the hash table ADT which we have used throughout the project. Each symbol table element/node stores lexeme, union of module/identifier/array and various flags required for semantic checks, offset and width (among many others things)**
   c. Array type expression structure: **It stores whether the array is dynamic, what are the lower and upper indices (they may be static nums or lexemes of the ranges), what is the array name and type, along with offset, width etc (and those which are common fields and attributes for all other variables as well).**
   d. Input parameters type structure: **Each input parameter is stored as a union of Identifier and Array type. All input parameters are stored as a list of a union of identifiers and arrays. This list is present in the Module hash table in the symbol table's n-ary tree.**
   e. Output parameters type structure: **Similar to (d), output parameters are stored as a list of a union of identifiers and arrays with module entry in symbol table (Though array part of the union is actually never used as syntactically output parameter can't be an array, union greatly simplifies implementation). The parameters also have isAssigned field for semantic checks, whether the variable is assigned something before returning or not.**
   f. Structure for maintaining the three address code(if created) : **A structure named Quad is created to store Intermediate Representation of the code. Quad consists of 4 fields: operator, argument1, argument2 and result. Appropriate tags are kept to know whether the field of quad has an immediate data or a variable.**

9. **Semantic Checks:** Mention your scheme NEATLY for testing the following major checks -*[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 : **Not found in Symbol Table or it exists but hasn't been traversed in the AST yet (checked using a flag which is set once a variable's declaration is traversed in AST). In both cases, function returns NULL. Search happens through a stack of hash tables maintained on the fly.**
   b. Multiple declarations: **Symbol Table entry already populated with same lexeme (Done during symbol table population)**
   c. Number and type of input and output parameters: **Hash into the Symbol table entry for the module which stores a list of input and output parameters. Now, do matching appropriately.**
   d. assignment of value to the output parameter in a function: **The output parameters have a isAssigned flag which is set if they have been assigned at least once in the module body. While exiting from the module, we check all the output parameters to give appropriate error (if it exists).**
   e. function call semantics: **Check whether function exists or not. Check if its definition exists, then check whether definition is below or above the call (using a flag, same as the one used in part a). If definition is below the call, check whether declaration exists or not. Then, match the number and type of input**

and output parameters. Give appropriate msgs at every error. This also helps in detection of redundant declarations.

f. static type checking : **Achieved through the data populated in the symbol table at compile time itself (like type, ranges of arrays). Done at different constructs like module call, assignment, expressions, etc.**

g. return semantics: **Offset counter is reset (so that the offsets now start from 0). Also, it is checked whether all output parameters have been assigned something in the moduleDef or not.**

h. Recursion : **Direct recursion is caught by comparing the called function name with name of the current scope (which is simply inherited as the name of the function inside which you are currently traversing)**

i. module overloading:**Symbol table entry for the functions already populated with that name**

j. 'switch' semantics : **Type of switch variable checked. The type of value in cases is checked according to type of switch variable. Default statement is checked if required in switch depending on the type of switch variable.**

k. 'for' and 'while' loop semantics: **Index variable is marked (by setting isIndex flag) and checked if it is changed in the for body. The lower index of 'for' must be less than or equal to the higher index. The index variable must be integer. The expression in while's condition must be of boolean type. At least one of the variables in while must change during while loop.**

l. handling offsets for nested scopes: **As far as the module does not change, the offsets keep increasing. The offsets are reset only when we change the scope "at the module level" (like traversing another module or traversing moduleDef after traversing the module's input and output parameters).**

m. handling offsets for formal parameters: **Formal parameters are in an intermediate stage between the caller and callee activation records. Offsets for them start from 0 and keep on increasing until we reach moduleDef (then offset is reset to 0). At runtime, they are accessed by adding an offset (instead of subtracting) to the EBP. The parameters are pushed in reverse order (at runtime)**

n. handling shadowing due to a local variable declaration over input parameters: **Input and output parameters are present in a different symbol table (technically, in the parent of moduleDef's symbol table). Hence, shadowing is easily achieved for input parameters. If we encounter some declaration, we check whether the variable exists in the output parameter list stored in the current function's symbol table. If yes, give error as shadowing of output parameters is illegal**

o. array semantics and type checking of array type variables: **Whenever used, index of array should be in accordance with array bounds (compile time check for static and runtime check for dynamic arrays). The array name should be declared beforehand, index should always be a number/integer. This takes place in various constructs like module call, expressions, assignments, etc. For A := B kind of statements, type and bounds are matched. For dynamic arrays, it is also checked that lower range entered is numerically smaller or equal to upper range entered and none of them can be a negative number.**

p. Scope of variables and their visibility : **We maintain a stack of scopes that helps us nest the scope with ease on the fly. We find the variable in the stack of scopes. Variables visibility is thus in all nested scopes ahead, as it should be.**

q. computation of nesting depth: **Simply recursive and passed from parent symbol tale to child table and keeps incrementing as it goes down in the tree of scopes/symbol tables.**

10. Code Generation:

a. NASM version as specified earlier used (Yes/no): **YES**
b. Used 32-bit or 64-bit representation: **32-bit**
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: **EBP**
- for stack pointer: **ESP**
- others (specify): **Mostly EAX,AX, EBX, BX, ECX, CX, EDX, DX, st0, st1**

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

size(integer): **2 (in words) or 4 (in bytes)**
size(real): **4 (in words) or 8 (in bytes)**
size(boolean): **1 (in words) or 2 (in bytes). In implementation, 4 bytes (Padded for double word memory alignment required in 32-bit code)**

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

Actual parameters (both input and output) are pushed on the top of the activation record of the calling function. EBP is made to point to current ESP. Old EBP is the pushed. Then "call" happens and NASM implicity pushes return address. Then, memory for called function is allocated. Before returning, allocated memory is deallocated by adding to ESP the size of the called function's activation record (i.e ESP becomes EBP-8). Then, RET happens and NASM returns us to appropriate instruction. Then, EBP is restored by popping from the stack. The value of output parameters is copied to corresponding local vars memory and all the pushed parameters are then popped and we return back to the initial state of calling function's activation record (the state just before the call).

g. Specify the following:

- Caller's responsibilities: **Pushing the parameters on the stack (in an appropriate order, so that called function can access them using offsets stored with it) and also its own EBP (frame pointer) on the stack. Then a call statement is executed to go to callee. After execution returns to instruction after call, the caller restores EBP and copies the value of formal output parameters to its corresponding local vars. Now, the pushed values are popped.**

- Callee's responsibilities: **Take values for the input and output parameters from the stack (using [EBP+offset], use them and put the value of the output variable in the stack (again using [EBP+offset]). Also, it deallocates its activation record by changing ESP. Then, it executes RET to return to appropriate instruction (done by NASM).**

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

Return address is implicity pushed by NASM on to the stack when "CALL" instruction is executed. Similarly, it is again implicity popped by NASM when "RET" is executed. All we do is to ensure that ESP is in correct position all the time and push the old EBP on the stack and restore the EBP when required.

i. How have you maintained parameter passing?

Firstly the output parameters are pushed in reverse order, then input parameters are pushed (again in reverse order). Also, for arrays (both dynamic and static), the following are pushed (in this order only): upper index, lower index, base address of the array. This order of pushing facilitates access of these parameters in the called function (using [EBP+offset]). So, array is pass by reference and everything else is pass by value. Copying the value of output parameters to appropriate locations is caller's responsibility.

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

At the runtime the lower and upper indices of the array are passed to the callee function as parameters only (copied from the memory of the variables that hold the range in the calling functions activation record and pushing them on the stack, along with the base address of the array).

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

It includes both temporaries and local variables. "But for printing, we are just printing the size of the activation record excluding local variables (just for the sake of uniformity)."

l. register allocation (your manually selected heuristic) :

Due to the time constraints, we haven't used any optimization as such in register allocation. We execute each intermediate instruction by reading the values from the activation record and putting them in the registers and then use instructions on those registers. Finally, put the output back in the activation record.

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

n. Where are you placing the temporaries in the activation record of a function?
We place the temporaries after storing control information (return address, old EBP) and local variables of the function.

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 (All compile)**

d. Any specific function that does not compile: **NONE (All compile)**

e. Ensured the compatibility of your code with the specified versions [GCC, UBUNTU, NASM] (yes/no)

YES, We use NASM 2.14. On 64-bit ubuntu machine, to run the 32 bit NASM we used GCC multilib library. (Please refer readme.txt)

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) **10620** and (in seconds) **0.01062**

ii. t2.txt (in ticks) **9622** and (in seconds) **0.009622**

iii. t3.txt (in ticks) **11911** and (in seconds) **0.011911**

iv. t4.txt (in ticks) **6151** and (in seconds) **0.006151**

v. t5.txt (in ticks) **11879** and (in seconds) **0.011879**

vi. t6.txt (in ticks) **13430** and (in seconds) **0.013430**

vii. t7.txt (in ticks) **15797** and (in seconds) **0.015797**

viii. t8.txt (in ticks) **17142** and (in seconds) **0.017142**

ix. t9.txt (in ticks) **21637** and (in seconds) **0.021637**

x. t10.txt (in ticks) **9772** and (in seconds) **0.009772**

13. **Driver Details**: Does it take care of the **TEN** options specified earlier?(yes/no): **YES (all ten)**

14. Specify the language features your compiler is not able to handle (in maximum one line)
Our compiler implements all required features according to guidelines.

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 elf -F dwarf -g code.asm
    gcc -m32 code.o -o code
    ./code

17. **Strength of your code**(Strike off where not applicable): **(a) correctness (b) completeness (c) robustness (d) Well documented (e) readable (f) good data structures (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: **To run 32-bit NASM code on a 64-bit machine, install gcc's multilib library (please see readme.txt for more details).**

19. Declaration: We, **Akshit Khanna, Aryan Mehra, Vipin Baswan and Swadesh Vaibhav** (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**: 2017A7PS0023P** Name**: Akshit Khanna**

    ID**: 2017A7PS0077P** Name**: Aryan Mehra**

    ID**: 2017A7PS0429P** Name**: Vipin Baswan**

    ID**: 2017A7PS0030P** Name**: Swadesh Vaibhav**

    Date: **20th April 2020**
