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 Tables are stored as a n-ary tree of hash tables. Each hash table table is made of the hash table ADT we have used throughout the project. Each symbol table element/node stores lexeme, union of module/identifier/array and requires other flags for parameter passing and conditional loops, and value of the variable, offset and width among many others.**

   c. Array type expression structure: **It stores whether the array is dynamic, what are the lower and upper index, what is the array name and type, along with offset, width etc that are common fields and attributes for all other variables as well.**

   d. Input parameters type structure: **Input parameters are stored as a list of a union of identifiers and arrays with module entry in symbol table. We maintain a flag/field in the symbol table that helps us detect whether a variable is input or output and helps us see whether they are changing, shadowed etc.**

   e. Output parameters type structure: **Output parameters are stored as a list of a union of identifiers and arrays with module entry in symbol table. The parameters also have isAssigned field for semantic checks, whether the variable is assigned something before return 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 operand, argument1, argument2 and result.**

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, finder function gives NULL**

   b. Multiple declarations: **Symbol Table entry already populated with same lexeme**

   c. Number and type of input and output parameters:**Checked with the Symbol table entry for the module name which stores list of input and output parameters**

   d. assignment of value to the output parameter in a function:**The output parameters have a isAssigned field which is populated if they have been assigned at least once in the module body**

   e. function call semantics: **The function exists or not, match the number and type of input and output parameters**

   f. static type checking : **Achieved through the value populated in the symbol table at compile time itself**

   g. return semantics: **Number of return assignments and their type are checked with the symbol table entry of the called module.**

   h. Recursion : **Direct recursion is caught by comparing the called function with current scope**

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

   j. 'switch' semantics : **The type of value in cases is matched with the 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 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. 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 base address and offsets are reset in case of module calls, driver etc, where new calls to scopes are made.**

m. handling offsets for formal parameters: **Formal parameters are in an intermediate stage between the caller and callee activation records. Hence they are accessed by adding an offset (instead of subtracting) to the EBP of the called function. The parameters were pushed in reverse order of occurrence.**

n. handling shadowing due to a local variable declaration over input parameters: **The input and output variables have a tag associated with them. So later in nested scope (inside the function body) when the input parameter is redeclared, it is allowed and given precedence from that point on. But for output variables, semantic error is reported.**

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. For array used without index, loops are created in ASM for iterative assignment (in case array=array of same type is done)**

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 for calculation and temporaries**

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), 4 (in bytes)**
size(real): **4 (in words/ locations), 8 (in bytes)**
size(boolean): **1 (in words/ locations), 4(Padded for double word memory alignment) (in bytes)**

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

We generated param(x) for all parameters that are passed. In the reverseorder of pushing we create intermediate code statements that will trigger the popping of the required number and type of parameters at return time. Then we had a call statement to jump to the module's label. The module (called) uses the input from below its record and places the output values below its own record as well. After returning, the reverse ordered statements created before help us to pop the input values, and store locally the output variables.

g. Specify the following:

- Caller's responsibilities: **Pushing the parameters on the stack and also its own EBP (base address) on the stack. Then placing a call statement to go to callee. Also taking values from above the stack after the function has returned. We make sure that popping happens in the reverse order as the pushing. After returning from the called function, input parameters are simply popped but outputs are also put back into place.**

- Callee's responsibilities: **Take values for the input parameters from below in the stack, use them and put the value of the output variable in the stack below it's activation record.**

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

Mostly we push the current EBP value of the register. The ESP does not change. When returning, ESP is set to the previous value by intermediate codes that help us pop values from stack to make ESP reach the position same as the time of call. Then EBP will be restored from the ESP position - making everything back to normal.

i. How have you maintained parameter passing?

How were the statically computed offsets of the parameters used by the callee? Within the same symbol scope (module) the EBP does not change. So offset will be subtracted from EBP value (to go up in a stack that grows downward).

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 from the memory of the variables that hold the range in the calling functions activation record.

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

Local Variables. We display the values as required, however we internally double word align our implementation.

l. register allocation (your manually selected heuristic) :

We make the code independent of register states and hence code for each intermediate code statement separately, using all registers from scratch

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?
YES. We are storing temporaries in the symbol table. But for display purposes, as specified and required, we display the local variables only.

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 Ubuntu machine, to run the 32 bit NASM on 64 bit machine we used GCC multilib library

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 64-bit machine, install gcc's multilib library.**
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**
