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

**Group No.**
**47**

**Compiler Construction (CS F363)**
**II Semester 2019-20**
**Compiler Project (Stage-2 Submission)**
**Coding Details**
**(April 20, 2020)**

---

**1.** IDs and Names of team members

| | |
|---|---|
| ID: 2016B3A70562P | Name: Akash Prasad Kabra |
| ID: 2016B3A70523P | Name: Ayush Rajendra Mungad |
| ID: 2016B4A70520P | Name: Kumar Anant Raj |
| ID: 2016B4A70533P | Name: Mustansir Mohammed Mama |
| ID: 2016B5A70715P | Name: Bhavesh Ranjit Chand |

**2.** Mention the names of the Submitted files :

| | | | |
|---|---|---|---|
| 1. action.txt | 2. ast.c | 3. ast.h | 4. codegen.c |
| 5. codegen.h | 6. driver.c | 7. ast_rules.txt | 8. input_grammar.txt |
| 9. isKeyword.c | 10. langSpec.c | 11. langSpec.h | 12. lexer.c |
| 13. lexer.h | 14. lexerDef.h | 15. makefile | 16. mapping.c |
| 17. mapping.h | 18. parser.c | 19. parser.h | 20. parserDef.h |
| 21. stack.c | 22. symbol_table.c | 23. typecheck.c | 24. typecheck.h |
| 25. typecheck_2nd_pass.c | | | |

And the 21 test case files (t1.txt - t10.txt, c1.txt - c11.txt) provided + this proforma.

**3.** Total number of submitted files: 47

**4.** Have you mentioned your names and IDs at the top of each file (and commented well)? Yes

**5.** Have you compressed the folder as specified in the submission guidelines? Yes

**6.** **Status of Code development**:

a. Lexer: Yes
b. Parser: Yes
c. Abstract Syntax tree: Yes
d. Symbol Table: Yes
e. Type checking Module: Yes
f. Semantic Analysis Module: Yes (reached LEVEL 4 as per the details uploaded)

   g. Code Generator: <u>Yes</u>

7. **Execution Status**:
   a. Code generator produces code.asm : <u>Yes</u>
   b. code.asm produces correct output using NASM for test cases (C#.txt, #:1-11): <u>Yes</u>
   c. Semantic Analyzer produces semantic errors appropriately : <u>Yes</u>
   d. Static Type Checker reports type mismatch errors appropriately .: <u>Yes</u>
   e. Dynamic type checking works for arrays and reports errors on executing code.asm : <u>Yes</u>
   f. Symbol Table is constructed: <u>Yes</u> and printed appropriately : <u>Yes</u>
   g. AST is constructed : <u>Yes</u> and printed : <u>Yes</u>
   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): <u>NA</u>

8. **Data Structures**
   a. **AST node structure:** AST is a child-sibling tree. In addition to information copied from corresponding parse tree node, we have pointers to traverse up/ down the AST, pointers to traverse to left/right sibling AST nodes, an inherited attribute used in AST construction, an attribute used to propagate scope, one for offset computation, and one used to propagate assembly code.
   b. **Symbol Table structure:** ST is implemented as a tree of hash tables. Each ST maintains its nesting level, name of module it belongs to, and it's parent ST. Each ST Entry contains a variable name, the line it was declared at, a representation of its datatype, flags to indicate if the variable is a for loop counter and if it is used in while loop condition. Chaining is used to resolve collisions.
   c. **Array type expression structure:** Contains 2 flags to indicate if the lower and higher bound are static/ dynamic, both the bounds (storing the number in the case of static and storing the variable making up the bound otherwise) and the datatype of its elements.
   d. **Input parameters type structure:** Linked List of records, each containing the variable name and corresponding data type representation.
   e. **Output parameters type structure:** Same as Input parameters.
   f. **Structure for maintaining the three address code(if created) :** IR not implemented. ASM is directly generated after the 2nd type-checking pass.

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 in current & parent scopes
   b. **Multiple declarations:** Current scope's Symbol Table contains entry for the variable name

c. **Number and type of input and output parameters:** Simultaneous traversal and datatype comparison of formal parameter list from function's ST Entry and actual parameter list from 'idList' in 'moduleReuseStmt'. Done in 2nd pass.

d. **Assignment of value to the output parameter in a function:** ST Entries contain value 'assigned', which is incremented when variable is assigned or input by user. After function ends, while traversing 'output_plist', check assigned != 0 for each. Done in 2nd pass.

e. **Function call semantics:** Function's ST Entry contains flags 'declared' & 'defined'. If both flags are set before call, error. If the declaration cannot be removed due to a previous call, 'called' flag is set, and error is not raised on subsequent calls.

f. **Static type checking :** Comparison of datatypes in ST Entries of variables. Skip static bound check if either operand is/belongs to a dynamic array.

g. **Return semantics:** Same as check for assignment of output parameters.

h. **Recursion :** Flag 'current' is set in the function's ST Entry on entering it, check this flag when function called.

i. **Module overloading:** Function's entry exists in ST with 'defined' flag set.

j. **'switch' semantics :** Switch variable can't be real. If integer, then default compulsory. If boolean, then default undesired, and both 'case true' and 'case false' must appear once only.

k. **'for' and 'while' loop semantics:**
   - FOR loop sets 'isLoopVar' flag for loop counter var, which prevents it from being assigned.
   - WHILE loop: No. of times expression variable is assigned before and after entering while loop is compared to check if it is assigned.

**l. Handling offsets for nested scopes:** Computed relative to module itself. Start offset of a variable in nested scope is equal to the offset of the last declared variable in parent scope + it's width.

**m. Handling offsets for formal parameters:** Computed relative to module itself. Input parameters' offset starts with 0,then output parameters and then local variables.

**n. Handling shadowing due to a local variable declaration over input parameters:** Input parameters declared in nesting level 0, while output parameters & local variables are declared from nesting level 1 onwards. So, shadowing is allowed.

o. **Array semantics and type checking of array type variables:**
   - For array assignment (A:=B;) :- Compare element type at compile time. If both static, compare bounds at compile time, else at run time.
   - For array element access :- If A is static & A[NUM], then compile time bound check. If A is dynamic OR A[var], then run time bound check.
   - For array input parameter:- Compare only element type at compile time.

**p. Scope of variables and their visibility :** Scope of variable is corresponding to the closest start-end block. If the variable is declared outside that start-end block, it's scope is computed by traversing to parent symbol tables.

**q. Computation of nesting depth:** Computed at time of ST creation and stored as an attribute of the ST itself.

**10. Code Generation:**

a. NASM version as specified earlier used : <u>Yes</u>

b. Used 32-bit or 64-bit representation: <u>64-bit representation</u>
c. For your implementation: 1 memory word = <u>8</u> (in bytes)
d. Mention the names of major registers used by your code generator:
    - For base address of an activation record: <u>rbp</u>
    - for stack pointer: <u>rsp</u>
    - others (specify): <u>rsi - format specifier string for printf and scanf</u>, <u>rdi - parameter for printf and scanf.</u>
e. Mention the physical sizes of the integer, real and boolean data as used in your code generation module
    - size(integer): <u>1</u> (in words/ locations), <u>8</u> (in bytes)
    - size(real):<u>1</u> (in words/ locations), <u>8</u> (in bytes)
    - size(boolean): <u>1</u> (in words/ locations), <u>8</u> (in bytes)

f. How did you implement function's calls?
Above the caller function's stack, space is allocated for input parameters, and then for output parameters above that. This space in stack can be called a "shared space". Then the caller's base pointer is pushed and the return address is pushed (by call instruction), which are assumed to be a part of callee's stack. So, callee's local variables start from [rbp-16].

g. Specify the following:
    - **Caller's responsibilities:**
        1. Push actual input parameters in reverse order in shared space.
        2. Decrement rsp by size required for output parameters, thus allocating shared space for them.
        3. Push rbp.
        4. Call function.
        5. Pop rbp to restore current stack space.
        6. Copy values of output parameters from shared space to locations of actual output parameters.
        7. Pop actual input parameters to restore stack to its state before Step 1.
    - **Callee's responsibilities:**
        1. Set current space's base as rbp = rsp - 16.
        2. Allocate space for self's static activation record.
        3. Copy input parameters from shared space to locations of formal input parameters. If the formal input array parameter has static bounds, check bounds against the passed array. If it has dynamic bounds, simply copy the given bound.
        4. Generate code for statements inside.
        5. Copy formal output parameters to their allocated shared space.
        6. Restore stack pointer to rsp = rbp + 16.

h. How did you maintain return addresses? :

        - Caller is required to push it's base pointer before the function call, so that the start of its stack space may be restored after the function call. *By design, 'CALL' instruction pushes Instruction pointer and 'RET' pops it automatically.* So, callee needs to ensure that the stack pointer is restored to its original state from when we entered callee. This is required so that now the return address is at top of stack.
        - To handle the case when we need to exit because of run time error: A space 'initst' has been allocated in the bss section, which is populated with rsp as soon as we enter main. Since a Run Time Error may occur in any module at any time, to exit a program properly, the value from 'initst' is copied into rsp before exiting, so that proper return to OS address is at top of stack.

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

   Shared space between caller and callee stack is used to pass both input and output parameters. Actual parameters' statically computed offsets are used and their values are copied in the shared space. Formal Parameters are populated by copying from the shared space to the formal parameters' static offsets.

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

   Since parameters are pushed in reverse order, for array input parameter, the caller first pushes higher bound, then lower bound, then address of the first element. Callee first retrieves the address, then the lower bound value, then the higher bound value. If the formal parameter array is static, the actual bounds are compared against formal bounds. If it is dynamic, the actual bounds are copied to formal bounds.

k. What have you included in the activation record size computation? : Parameters, Local Variables & Temporary Variables
l. register allocation (your manually selected heuristic) :

   We have used r8 to r15 for all construct- based computations. rax is used by default wherever 'MUL' and 'DIV' instructions are used. rsi & rdi are reserved for scanf and printf calls. rbp and rsp are reserved for memory addressing. Registers are pushed and popped appropriately wherever an external function is called.

m. Which primitive data types have you handled in your code generation module? : Integer & Boolean
n. Where are you placing the temporaries in the activation record of a function? :

   Placed depending on the offset of the temporary. The offsets of temporaries are also calculated during the first type checking pass on AST.

11. **Compilation Details**:
    a. Makefile works : Yes
    b. Code Compiles : 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

12. **Execution time for compiling the test cases** [lexical, syntax and semantic analyses including symbol table creation, type checking and code generation] :
    a. t1.txt (in ticks) 5038 and (in seconds) 0.005038
    b. t2.txt (in ticks) 3456 and (in seconds) 0.003456
    c. t3.txt (in ticks) 6637 and (in seconds) 0.006637
    d. t4.txt (in ticks) 3566 and (in seconds) 0.003566
    e. t5.txt (in ticks) 7000 and (in seconds) 0.007
    f. t6.txt (in ticks) 9976 and (in seconds) 0.009976
    g. t7.txt (in ticks) 9108 and (in seconds) 0.009108
    h. t8.txt (in ticks) 10551 and (in seconds) 0.010551
    i. t9.txt (in ticks) 13536 and (in seconds) 0.013536
    j. t10.txt (in ticks) 4991 and (in seconds) 0.004991
    k. c1.txt (in ticks) 2424 and (in seconds) 0.002424
    l. c2.txt (in ticks) 2145 and (in seconds) 0.002145
    m. c3.txt (in ticks) 1596 and (in seconds) 0.001596
    n. c4.txt (in ticks) 2766 and (in seconds) 0.002766
    o. c5.txt (in ticks) 1994 and (in seconds) 0.001994
    p. c6.txt (in ticks) 1622 and (in seconds) 0.001622
    q. c7.txt (in ticks) 2718 and (in seconds) 0.002718
    r. c8.txt (in ticks) 1248 and (in seconds) 0.001248
    s. c9.txt (in ticks) 1703 and (in seconds) 0.001703
    t. c10.txt (in ticks) 1757 and (in seconds) 0.001757
    u. c11.txt (in ticks) 1598 and (in seconds) 0.001598

13. **Driver Details**: Does it take care of the **TEN** options specified earlier? : Yes
14. Specify the language features your compiler is not able to handle (in maximum one line): NA
15. Are you availing the lifeline: 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]
    a. make run , OR
    b. nasm -felf64 code.asm && gcc -no-pie -o exec code.o && ./exec

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:**

    We had removed START and END nodes from AST construction stage. Instead, new scope was derived on encountering moduleDef, iterativeStmt, or conditionalStmt. So, we were not able to preserve line number pairs of scope, which is thus reflected in symbol table printing. As a result, even though all our semantic checks have been successful, some of their line numbers are different:

    a. For switch construct, if default is required(in the case of INT switch variable) but is *absent*, then the *semantic error would be printed on the "switch(var)" line number* and not on the line number of the "end" symbol.
    b. If output parameters of a module are not assigned, then the error is reported on the 1st line of module definition instead of the last line.

19. Declaration: We, Akash Kabra, Ayush Mungad, Bhavesh Chand, Kumar Anant Raj, and Mustansir Mama 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.

| | |
|---|---|
| ID: 2016B3A70562P | Name: Akash Prasad Kabra |
| ID: 2016B3A70523P | Name: Ayush Rajendra Mungad |
| ID: 2016B4A70520P | Name: Kumar Anant Raj |
| ID: 2016B4A70533P | Name: Mustansir Mohammed Mama |
| ID: 2016B5A70715P | Name: Bhavesh Ranjit Chand |

Date: 20/04/2020

-----------------------------------------------------------------------------------------------------------------------------