Project 3 Report, Srinjoy Dutta

- ASTExprStmt: This was a very simple function as shown in the instructions. Since the expression might perform operations but not a value needed by any part of the program, it is enough to call emitIR and return a nullptr.
- ASTCompoundStmt: There are declarations followed by statements. So first I loop through all the declarations and call emitIr then I do the same for statements. I do this because the node doesn't return any value for compound statements so I can just return a nullptr at the end.
- ASTConstantExpr: If it is an integer, then set retVal to llvm::Type::getInt32Ty as specified or if it is a char set it to llvm::Type::getInt8Ty you are setting it to 32 or 8 as Int is 32 bit and char is 8 bit. Then return retVal.
- ASTReturnStmt: The IRBuilder is used to generate IR instruction with the current basic block ctx.mBlock where the return instruction will be inserted. I check if the return statement has an expression or not if it is null then return type is void so I use builder.CreateRetVoid() to represent that. For non void returns, I first call emitIr on the mExpr to generate the IR for that expression. Then I call builder.CreateRet() with that for the LLVM IR.
- ASTBinaryMathOp: Now we are generating the LLVM IR for binary math operations by generating the IR for the LHS and RHS, then creating an IRBuilder to insert the resulting operations instruction to the current basic block. I do a switch based on the type of operation. For addition I generate an add instruction same methodology for subtraction, multiplication, signed division, and signed remainder.

Then I return the resulting value. Lastly represent in a 32 bit int using the CreateZExt function.

- ASTBinaryCmpOp: Kind of follows the same process as MathOp. First generate the LLVM IR code for LHS and RHS by calling emitIR. Then I make an IRBuilder to insert a comparison instruction to the current basic block. I use a switch statement again for mOp and use the respective builder commands for EqualTo, NotEqual, LessThan, and GreaterThan. Lastly I represented in a 32 bit int using the CreateZExt function.
- ScopeTable::emitIR: Here my goal is to allocate stack memory for all local variables. So first I check if the Identifier already has an address which would be if it is a function parameter which means it would already be assigned a value but we still need to allocate space on the stack and copy the parameter's value into the allocated space. Whether the variable is a function param or a regular local variable, CreateAlloca allocates memory for the variable on the stack. llvmType is used to determine the correct type for allocation. If the variable is a function parameter, after allocating stack space for it the existing value is copied into the allocated space using CreateStore. Lastly set the address of the allocated space back into the Identifier for later reference during IR generation.
- readFrom: This function loads the value of a variable from its memory location and returns it and handles both arrays and regular variables. Firstly, check if the variable is an array. Then you can directly return the memory address of the array as getting the address is more important than the value at the address. Otherwise

get the value from the memory address and use CreateLoad to load the value stored there. Lastly return the value.

- writeTo: The function first stores a value in the variable's memory location. THe function handles both arrays and regular variables. First if the variable is an array, set the array's address using setAddress. If it is a non array, use the IRBuilder to generate a CreateStore instruction which stores the given value into the memory address retrieved by getAddress().
- ASTIncExpr: The goal is to increment the value of a variable. First I read the current variable. Then I increment the variable depending on whether it is an int value or a char value by 1. Then I write the incremented value back
- ASTDecExpr: This node decrements the value of a variable in a similar way. The current value is first read from memory. Depending on if type is int or char, value is decremented by 1 using CreateSub and then I write the value back.
- ASTNotExpr: This performs a logical NOT on the result of a sub expression. It is first evaluated by calling emitIR on the sub expression. This generates the IR code for the sub expression and stores the result in subExprVal. The logical NOT compares the result of the sub expression to 0 using CreateICMPEQ to check if the sub-expr result is 0 generating a boolean value. Then the zero extended result is returned as the final result.
- ASTWhileStmt: Basic blocks in the LLVM are a sequence of instructions with a single entry and exit point. The loop has 3 basic blocks, the whileCond which evaluates the loop's condition, the whileBody which contains the loop body, and the whileEnd which is executed once the loop exits. Before the program has to jump to

the whileCond block before checking the loop condition. The initial CreateBr instruction ensures that execution will jump to the condition block. The condition expression is evaluated using emitIR which generates the LLVM IR for the condition. The value is compared with ctz.mZero which represents 0 using the CreateICmpNE instruction, the loop proceeds to the body then, otherwise it exits to the whileEnd. The CreateCondBr generates a conditional branch based on the evaluated condition. If true, branch to whileBody else to whileEnd. In the loop body, emitIr is called on mLoopStmt which emits the IR for the statements within the loop. After execution, the program jumps back to the whileCond to recheck the condition forming the cycle.

- ASTAssignStmt: emitIr is called to generate the LLVMIR for the RHS. THen returns the computed value in exprVal. Then the function stores the value in the memory address associated with the identifier.
- ASTIfStmt: First create the basic blocks such as ifThen, ifElse, and ifEnd. The condition mEpr is evaluated to determine which branch to take. A conditional branch instruction CreateCondBr is created based on whether an else statement exists. If the condition is true. It jumps to ifThen, and if false, it jumps to ifElse. If an else statement exists, the block for ifElse is executed. The instruction for the else blocks are emitted and the control is transferred to the ifEnd block. The then block is always executed if the condition evaluates to true. After executing the block, control to ifEnd. if End is sealed and set as current block to ensure subsequent instructions are placed after the if else structure.