Fixes from Previous Assignment:

The 4 failing test cases from hw2 were resolved in this version of my code. Reason behind the failed test cases was the use of mSymbols.getIdentifier(), which lead to segmentation faults, instead of the getVariable() function that performed type safety checks.

ASTExprStmt

Following the comment left there since the ASTExpr node can hold several different node types defined by the grammar (e.g. ASTLogicalOr, ASTLogicalAnd, ASTBinaryMathOp, etc.), we simply need to recursively rely on the underlying subclass's emitIR() implementation through polymorphism. Therefore, all that needs to be done here is call mExpr->EmitIR(ctx) and return nullptr, as expression statements don't produce a value themselves – they just execute the expression for its side effects.

ASTCompoundStmt

Like ASTExprStmt, we just care about calling emit on the list of declarations and statements without needing to return any value.

ASTConstantExpr

We need to convert the constant number to an LLVM ConstantInt object since LLVM IR works with LLVM objects and not plain C++ integers. The hierarchy for LLVM nodes roughly follows Value -> User -> Constant -> ConstantInt, so our retVal can be the result of ConstantInt::get. In order to get the right LLVM type for ConstantInt, we have to check mType of the AST which either can be int or char.

ASTReturnStmt

The return types for a function can be either void or non-void. For this implementation we need to create an IRBuilder since we are creating an LLVM instruction without relying on another function to handle it. If the function is non-void (i.e mExpr is not null), we use build.CreateRet on the IR of mExpr. Otherwise, we simply call build.CreateRetVoid to avoid segmentation faulting from dereferencing a NULL expr.

ASTBinaryMathOp

First, create the IRBuilder. Then we generate the IR for the LHS and RHS expressions before creating and returning the appropriate instruction based on the operation: add(+), sub(-), mult(*), div(/), or mod(%), using a switch statement and respective create functions. The name hints I chose here are just the names of the operators listed above.

ASTBinaryCmpOp

Identical to ASTBinaryMathOp except it is for integer comparisons with the operations being < (lt), > (gt), == (eq), != (neq) and using CreateZExt since there are no boolean types in USCC and we need to extend the 0 or 1 to i32 for conversion to int type.

SymbolTable::ScopeTable::emitIR

First, we check whether the identifier represents an array. If it does, we pass true to llvmType() so it is treated as a pointer type. This is necessary because arrays that reach the else case are function parameters, and in C, arrays are passed by reference. Next, we perform the same allocation steps as in the array case, but we adjust the memory alignment depending on the data type: 1 byte for char, 4 bytes for int, and 8 bytes for pointers. For function parameters, we must then copy the parameter's value into the newly allocated stack space to ensure pass-by-value semantics. Finally, we record the allocated address for every identifier, including function parameters, so subsequent references can correctly access their corresponding LLVM memory locations.

Identifier::WriteTo and Identifier::ReadFrom

For the else cases, we simply construct the store or load instruction for the local variables. We assume that the address is non-null as the identifier should be pointing to a valid address if these functions are called.

ASTIncExpr and ASTDecExpr

Read the identifier using load instruction from calling readFrom then increment or decrement the value accordingly and create the respective add or sub instruction from that before writing back to the identifier by calling writeTo.

ASTNotExpr

We want to evaluate the falseness of the expression, so we first mExpr->emitIR to evaluate it. Then we use ICmpEq to compare against ctx.mZero (zero constant) before zero extending to correctly return an integer cast boolean of the falseness of the expression.

ASTWhileStmt

To follow the structure of while statement outlined by the document, we create basic blocks for while.cond, while.body, and while.end. As mentioned in the document, we need to unconditionally branch to the while.cond basic block, this is done with the createBr(condBlock). After that we must change the mBlock to condBlock before evaluating the condition expression. Since we have changed blocks, we must create a new IRBuilder for the current condition block. When that is done, we create an ICmpNe instruction so we can branch to the body block if the condition is non-zero or non-falsy;

otherwise, it goes to the end block. Next, we just generate the while loop body instructions and end the basic block with a branch to the condition block again until it turns false; in which case, we set the mBlock to the endBlock basic block.

ASTAssignStmt

Simple as the homework document suggests, we just need to emitIR for the expression and use writeTo for the value of the expression to the identifier.

ASTIfStmt

The structure of the if statement is very similar to the while statement. It follows the same procedure of creating the basic blocks first for thenBlock, endBlock, and elseBlock (if there is an mElseStmt). However, it is important to first evaluate the condition of the expression before creating the IRBuilder since functions like ASTLogicalAnd can change the mBlock themselves and we want to ensure instructions are in the correct order. The other difference from while statement is the branching which includes the elseBlock, if it exists, along with the endBlock; otherwise, it just includes the thenBlock and endBlock.