P3 report

Ruoyu Wu

ASTExprStmt

Just invoke the `emitIR()` of its `mExpr`.

ASTCompoundStmt

Invoke the `emitIR()` of each of the declarations, and then each of the statements.

ASTConstantExpr

Depending on the type (i.e., Int or Char) of the constant, we use `ConstantInt::get()` to get the value and return.

ASTReturnStmt

In this function, we use LLVM's `IRBuilder` to emit IR code. We use `CreateRet` and `CreateRetVoid` of `IRBuilder`, depending on if there is a return value or not.

ASTBinaryMathOp

We first create the lhs and rhs with `emitIR()`, then we invoke the corresponding create function in `IRBuilder` to generate IR.

ASTBinaryCmpOp

It is similar to `ASTBinaryMathOp`, except for using another set of `IRBuilder`'s API and returning the 32-bit extension of the result.

Creating Stack Space for Local Variables

In `ScopeTable::emitIR()`, we create an `alloca` instruction, according to the type. If the identifier is a function argument, we also create a `store` instruction to store the argument to the stack space.

Reading/Writing to Local Variables

In `readFrom`, we create a `load` instruction to read from the corresponding stack space, and in `writeTo`, we create a `store` instruction to write to the corresponding stack space.

ASTIncExpr and ASTDecExpr

We use `readFrom` to build the instruction to read the value, and crate an increase/decrease instruction, and store the value back to the identifier's stack space, using `writeTo`.

ASTNotExpr

We use `ICmpEQ` to compare the value with zero, and use `ZExt` to extend it to a 32-bit integer.

ASTWhileStmt

We firstly create three basic basic blocks, for while-condition, while-body, and while-end. We create a branch instruction from the predecessor to the condition block. For condition block, we move the basic block pointer (i.e., `ctx.mBlock`) to the condition block, emit IR of the condition expression, and create a conditional branch whose jump target depends on the value of the emitted IR (i.e., `true` or `false`). For while body block, we just emit the IR and create a branch to the condition block. At the end, we move the basic block pointer to the end block.

This function took me a long time, and after carefully checking the generated bitcode (by checking `.ll` converted by `llvm-dis`), I realized that there was a typo in `ASTIncExpr` and `ASTDecExpr` – I typed `builder.getInt32(1)` as `builder.getInt32(32)`.

ASTAssignStmt

We just write the rhs's emitted IR to the lhs (i.e., identifier).

ASTIfStmt

It is similar to `ASTWhileStmt`. We firstly create three basic blocks, for if-then block, else-then block (optional) and end block. We emit the condition and insert the conditional branch, to jump to different blocks depending on the condition's value. Then, we emit the IR for each block, and carefully moving the basic block pointer and IRBuiler's insert point. It is not hard after having the experience of implementing `ASTWhileStmt`.