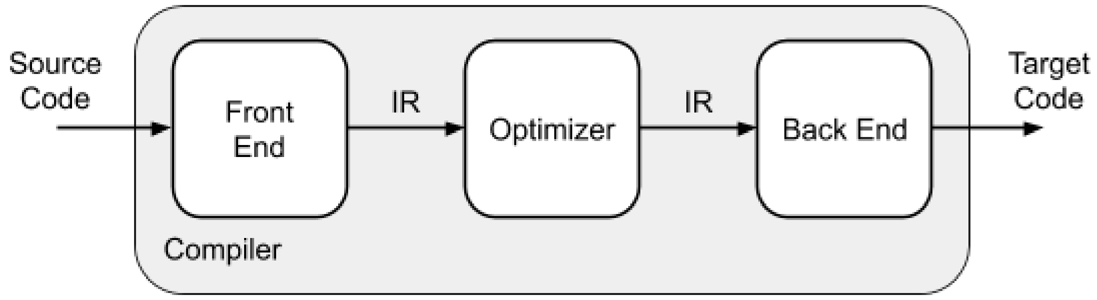
LLVM: Code Generation and Optimization

- Remember from our overview of compilers that the most popular modern compiler design consists of three phases:



- In practice, many popular language implementations, including GCC, do not cleanly implement this three-phase model, being implemented as monolithic applications with significant “leakage” between components.
    - This was particularly true when LLVM was first developed at the University of Illinois at Urbana-Champaign.

- One of the main problems resulting from the construction of these language implementations is that it is hard to reuse components of the compiler in different contexts.
    - For example, it is difficult to reuse the C++ parser from GCC to perform static analysis on code (e.g. for edit-time error detection, i.e. “linting”).

- LLVM is designed as “a collection of modular and reusable compiler and toolchain technologies,” and one of its main purposes is to make it easier to cleanly achieve the three-phase compiler design.

- Two of the main components of LLVM are a set of libraries for performing common optimizations on a program and a set of libraries for generating assembly code from a program for different target machines. LLVM also provides several different front-ends, e.g. for the C/C++/ObjC family of languages.

- The key design feature of LLVM is its intermediate representation (IR), which is used to represent the code across all phases of the compiler.

- The LLVM IR is defined in three isomorphic forms:
    - A textual form, which is a first-class language resembling a low-level RISC-like instruction set.
    - An in memory data structure that is inspected and manipulated by optimizations and the code generator.
    - A binary “bitcode” format that is used to efficiently store LLVM IR on disk.

- In our discussion of LLVM IR, it will be easiest to look at the IR in its textual form. As an example of this, here is a simple function in LLVM IR:

```
define i32 @addRecursive(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @addRecursive(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

- This LLVM IR corresponds to this (inefficient) C/C++ implementation of addition:

```
unsigned addRecursive(unsigned a, unsigned b) {
  if (a == 0) {
    return b;
  }
  return addRecursive(a-1, b+1);
}
```

- In what follows, we’ll explore how to use LLVM’s C++ API to generate LLVM IR. Then, we’ll use the same LLVM C++ API to perform optimizations and generate target code from the IR.

- The optimization and code generation techniques we'll see here can be hooked into any compiler front end from which we can emit LLVM IR. The most common approach to doing this is to implement a front end that generates an AST and then to build LLVM IR based on that AST.

- To simplify the discussion, we won't tie our use of the LLVM C++ API to any particular source language, so we won't use an AST. However, we'll structure our IR generation similarly to the way it would be structured if we were generating IR based on an AST.

First steps for using the LLVM C++ API

- Before we begin, note that the OSU ENGR servers do have a relatively new version (7.0) of LLVM installed, but to use it, you'll need to explicitly specify the version in the LLVM commands you run in the terminal (e.g. `llvm-config-7.0-64`). Otherwise, an older version will be used.

- There are a few essential LLVM objects we'll need at many different places for LLVM IR generation. For simplicity, we'll start out by declaring these objects as static variables at global scope (along with the needed `#include` statements):

```
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/Module.h"

static llvm::LLVMContext TheContext;
static llvm::IRBuilder<> TheBuilder(TheContext);
static llvm::Module* TheModule;
```

- Here's a bit of info about these objects:
    - `TheContext` – This object owns a lot of core LLVM data structures needed during IR generation. A single instance must available to be passed to several of the API methods we'll use.
    - `TheBuilder` – This is a helper object that makes it easier to generate LLVM IR. It keeps track of the current insertion point in the IR and contains lots of methods to generate new instructions.

    - `TheModule` – This is the top-level structure used to contain LLVM IR code. It contains and takes ownership of all generated functions and global variables.

- We can actually allocate the `TheModule` object within `main()`, making sure we free it before we return:

```
TheModule = new llvm::Module("LLVM_Demo", TheContext);
...
delete TheModule;
return 0;
```

- We'll stick for now to generating code within a single function. We'll start out by generating this function, which we can call `foo`. To generate this function we first need to generate its prototype. In this case, the `foo` function will be `void` type and will take no arguments:

```
llvm::FunctionType* fooFnType = llvm::FunctionType::get(
     llvm::Type::getVoidTy(TheContext), false
);
```

- With the function type prototype, we can create an `llvm::Function` object to represent our `foo` function:

```
llvm::Function* fooFn = llvm::Function::Create(fooFnType,
     llvm::GlobalValue::ExternalLinkage, "foo",
     TheModule);
```

    - The `ExternalLinkage` argument indicates that we'll eventually link our file with other files.

- Next, we will generate an ***entry block*** for our `foo` function.
    - An entry block is the basic block that is run immediately after a function is called.
        - Recall from our discussion on intermediate representations that a basic block is a maximal block of straight-line (i.e. non-branching) code.

- To generate our function's entry block, we will simply create a new `llvm::BasicBlock` object. We can also set our `IRBuilder` to start inserting in the entry block:

  ```
  llvm::BasicBlock* entryBlock =
       llvm::BasicBlock::Create(TheContext, "entry", fooFn);
  TheBuilder.SetInsertPoint(entryBlock);
  ```

- For now, let's simply generate a return operation:

  ```
  TheBuilder.CreateRetVoid();
  ```

- We can conclude by verifying the `foo` function using the LLVM API's `verifyFunction()`. This function checks a generated IR function for consistency and can help identify bugs in code generation.

  ```
  llvm::verifyFunction(*fooFn);
  ```

- Note that to use `verifyFunction()`, we must insert the following `#include` statement:

  ```
  #include "llvm/IR/Verifier.h"
  ```

  - Note here that for older versions of LLVM, like version 3.4.2, the `#include` statement is slightly different:

    ```
    #include "llvm/Analysis/Verifier.h"
    ```

- After this basic setup, we can see what our generated IR looks like by printing out `TheModule`:

  ```
  TheModule->print(llvm::outs(), NULL);
  ```

  - Here, `llvm::outs()` indicates that `TheModule` should be printed to stdout.

Compiling a program that uses the LLVM API

- To compile our “compiler”, we’ll need to make sure to specify all of the needed compiler flags (such as paths to LLVM header files, paths to LLVM libraries, definitions, and so forth) are specified.

- To do this, we can use the `llvm-config` command. This command can be run on the command line to print out all of the compiler flags needed to compile with the LLVM API.

- For example, running the following command in the terminal will print out all of the compiler flags (including needed linker flags library specifications) needed to compile with the LLVM “core” API (the API for IR generation):

  ```
  llvm-config --cppflags --ldflags --libs --system-libs core
  ```

  - Note that the `--system-libs` option is not available for older versions of LLVM, such as 3.4.2, and must be omitted when using those versions.

- In addition, we must make sure to compile under the C++11 standard, many features of which are used by the LLVM API. To do this, we can include the `-std=c++11` option to `g++`.

- Assuming then, that our “compiler” is implemented in a file called `main.cpp`, we can compile it using the following command:

  ```
  g++ -std=c++11 main.cpp                 \
       `llvm-config --cppflags --ldflags  \
            --libs --system-libs core`    \
       -o main
  ```

  - Note here that the `` `...` `` notation indicates that the *output* of the backtick-wrapped command should be added to the command line, as if it was typed.

- Once our “compiler” is compiled, we should be able to run it and see the contents of `TheModule` being printed out, including our (mostly-empty) `foo` function:

```
; ModuleID = 'LLVM_Demo'
source_filename = "LLVM_Demo"

define internal void @foo() {
entry:
  ret void
}
```

Adding instructions to the IR

- To begin to add instructions to our IR, let's start to write some functions that simulate the way code might be generated using an AST.

- All of the functions we'll write will return an `llvm::Value` object. The `Value` class is an important class in the LLVM API. In particular, it is a base class for all classes representing values that can be computed by a program and that can be used as operands to other `Value`s.

- Subclasses of the `Value` class include the `Function` class (which represents individual functions) and the `Instruction` class (which represents individual instructions), among many others.

- All `Value`s have an associated type, and many can be assigned a name.

- To use the `Value` class, we must include the following header file:

  ```
  #include "llvm/IR/Value.h"
  ```

- Let's start by writing a function to return a numeric constant:

  ```
  llvm::Value* numericConstant(float val) {
    return llvm::ConstantFP::get(TheContext,
      llvm::APFloat(val));
  }
  ```

- Here, we see that constant floating-point values are represented using the `ConstantFP` class, which in turn holds the value in an `APFloat` object, which itself holds an arbitrary precision floating point value.

- These constant numerical values don't generate any instructions, so next, let's add a function to compute the binary operation expressions involving addition, subtraction, multiplication, and division.

```
llvm::Value* binaryOperation(llvm::Value* lhs,
     llvm::Value* rhs, char op) {...}
```

- Here, we pass in the left-hand side and the right-hand side of the expression as arguments, along with a char value representing the operation to perform, and we return the resulting value of the expression (which will correspond to an instruction).

- Within the function, we'll first confirm that we indeed have both a left-hand-side value and a right-hand-side value, and we'll return `NULL` if we don't:

```
if (!lhs || !rhs) {
  return NULL;
}
```

- Next, we'll include a `switch` statement to generate and return the correct instruction to perform the specified operation on the given arguments. To do this, we'll make use of our `IRBuilder` object:

```
switch (op) {
  case '+':
    return TheBuilder.CreateFAdd(lhs, rhs, "addtmp");
  case '-':
    return TheBuilder.CreateFSub(lhs, rhs, "subtmp");
  case '*':
    return TheBuilder.CreateFMul(lhs, rhs, "multmp");
  case '/':
    return TheBuilder.CreateFDiv(lhs, rhs, "divtmp");
  default:
    std::cerr << "Invalid operator: " << op << std::endl;
    return NULL;
}
```

- A few things to note about this code:

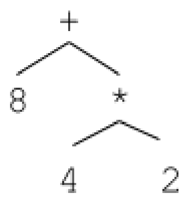
    - The LHS and RHS arguments to the above instructions must have the same type, and the result must be of the same type.
        - In this case, we're using all `float` values, so we don't need to worry.
        - The `CreateF*` methods we used here all take floating-point arguments and generate a floating-point result.
    - The `IRBuilder` is doing a lot here. For example, it knows where in our in-progress IR to insert the instruction, it knows which operands to use for the LHS and RHS values, even if those operands were computed by previous instructions, and it allows us to assign a name to the result of the operation.
        - If we assign the same name twice, LLVM will make the names unique by appending a unique (increasing) numerical suffix.

- Now, back in our compiler's `main()` function (before we generate the `return` instruction), we can use our new functions to generate some IR instructions for the expression `8 + 4 * 2`:

```
lvm::Value* expr1 = binaryOperation(numericConstant(4),
  numericConstant(2), '*');
llvm::Value* expr2 = binaryOperation(numericConstant(8),
  expr, '+');
```

- Importantly, note here how in our first call to `binaryOperation()`, we pass two values generated by `numericConstant()`, while in the second call, we pass one value generated by `numericConstant()` along with the value of our first binary operator expression.

- If we were generating code from an AST, `expr2` would represent an expression whose child was also an expression. The AST would first generate the code for the subexpression (`expr1`) to be used in the higher-level expression. Indeed, the AST would look like this:



- Now, if we compile and run our "compiler", we'll see that the generated `foo` function looks… exactly like it did before:

  ```
  define internal void @foo() {
  entry:
    ret void
  }
  ```

- Why aren't our generated instructions appearing here? To understand this, it helps to understand that LLVM performs an optimization called ***constant folding*** when generating IR.

- The constant folding optimization recognizes instructions like our binary operations above that involve only constant values. Instead of generating instructions to compute these values in the generated code, LLVM itself computes the value of the operation and remembers it so it can use it later in any instructions where the value is referenced.

- In our particular case, LLVM recognized that our first instruction (`4 * 2`) involved two constants that could be folded into the constant value 8. Then, it recognized that our second instruction (`8 + 4 * 2`) also involved two constants that could be folded into the constant value 16. LLVM simply computed that value, *without* adding instructions, and remembered the value (as `expr2`) in case it was needed later. In this case, it wasn't.

- To see some instructions inserted, in other words, we'll have to start using some named values.

Using named values in LLVM instructions

- The instructions above involved register-based values, i.e. values stored in registers. Once we started including function calls and branching instructions, we would have seen some of the values we were computing actually used in instructions.

- Using register-based values is straightforward, but it can become complicated when we start to use branching. That's because register-based values in LLVM are actually single static-assignment (SSA) registers. To be able to make these registers work across branches in the code, we would need to insert ϕ-functions.

- To avoid these challenges, we'll begin using memory-based named values. These values are variables declared on the call stack.

- To use call-stack values in LLVM, we'll need to access a location in memory using load and store instructions. These instructions are provided with the *name* of a particular memory address to be accessed.

- In LLVM, we can create a named piece of memory in the call stack using the `alloca` instruction. For example, here's a simple LLVM IR program that creates a named call-stack variable, then loads it, increments it, and stores the incremented value back into call-stack memory:

```
entry:
  %X = alloca i32
  ...
  %tmp = load i32* %X
  %tmp2 = add i32 %tmp, 1
  store i32 %tmp2, i32* %X
  ...
```

- To begin using call-stack-based named values in our generated IR, we'll need to have a symbol table to map the names to their corresponding `alloca`. We can create a symbol table as a static global variable along with our other globals:

```
static std::map<std::string, llvm::Value*> TheSymbolTable;
```

- Once we have our symbol table in place, we can begin to implement a function to *generate* named values via assignment statements:

```
llvm::Value* assignmentStatement(const std::string& lhs,
     llvm::Value* rhs) {...}
```

- This function takes as arguments a string representing the identifier on the LHS of the assignment and a `Value*` representing the RHS expression. The first thing we'll do in this function is make sure we have a valid RHS:

```
if (!rhs) {
  return NULL;
}
```

- Next, we'll check to see if the LHS identifier already exists in the symbol table. If it doesn't we'll need to generate an `alloca` instruction for it:

  ```
  if (!TheSymbolTable.count(lhs)) {
    ...
  }
  ```

- The way we generate an `alloca` deserves some attention. In particular, if we ensure that all of our `alloca` instructions are executed in their containing function's entry block, we'll later be able to use an LLVM optimization called "mem2reg" that can promote `alloca`s into SSA registers, inserting ϕ-functions as needed.

- We can write a helper function to ensure that all `alloca` instructions are in the entry block of the function:

  ```
  llvm::AllocaInst*
  generateEntryBlockAlloca(const std::string& name) {
    llvm::Function* currFn =
      TheBuilder.GetInsertBlock()->getParent();
    llvm::IRBuilder<> tmpBuilder(&currFn->getEntryBlock(),
      currFn->getEntryBlock().begin());
    return tmpBuilder.CreateAlloca(
      llvm::Type::getFloatTy(TheContext), 0, name.c_str()
    );
  }
  ```

- Here, we create a secondary IR builder that is set up to insert at the beginning of the entry block for the current function, and then we use that builder to insert a `float` `alloca` for the specified name.

- We can use this method back in our `assignmentStatement()` function, inside the `if` block after we have verified that the name on the LHS of the assignment statement does not exist in the symbol table:

  ```
  if (!TheSymbolTable.count(lhs)) {
    TheSymbolTable[lhs] = generateEntryBlockAlloca(lhs);
  }
  ```

- Then, after the `if` statement, once we know we have an `alloca` for the LHS name, we can simply generate an instruction to store the RHS value in the memory corresponding to that LHS name (whose value we'll return from the function):

```
return TheBuilder.CreateStore(rhs, TheSymbolTable[lhs]);
```

- Now, back in our compiler's `main()` function, after our two expressions, we can insert an assignment statement to store the value of `expr2` in a variable `a`:

```
llvm::Value* assignment1 = assignmentStatement("a", expr2);
```

- Now, if we compile and run our "compiler", we'll see the new instructions added to our `foo` function:

```
define internal void @foo() {
entry:
  %a = alloca float
  store float 1.600000e+01, float* %a
  ret void
}
```

- Here, we can also see the result of constant folding. Specifically, the value 16 was computed from our two expressions without explicit instructions in the IR.

- Once we can store values in named variables, we'll want to be able to use those values in expressions. We can do this by incorporating load instructions to access the stored values.

- To do this, we'll create a function to access a variable value:

```
llvm::Value* variableValue(const std::string& name) {
  llvm::Value* val = TheSymbolTable[name];
  if (!val) {
    std::cerr << "Unknown variable: " << name << std::endl;
    return NULL;
  }
  return TheBuilder.CreateLoad(val, name.c_str());
}
```

- Here, we simply create a load statement for the specified variable name, provided we have an `alloca` for that variable in the symbol table.

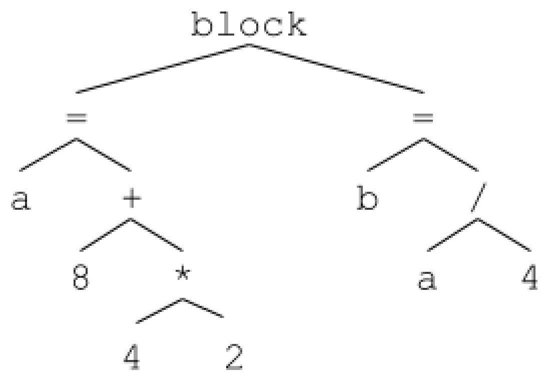
- Again, this function is very similar to a function we might implement to generate IR for a variable expression in an AST. Indeed, we can use this function as part of an expression, re-assigning that expression value to another variable:

```
llvm::Value* expr3 = binaryOperation(variableValue("a"),
  numericConstant(4), '/');
llvm::Value* assignment2 = assignmentStatement("b", expr3);
```

- Compiling and running our "compiler" again with these additional instructions gives us the following IR:

```
define internal void @foo() {
entry:
  %b = alloca float
  %a = alloca float
  store float 1.600000e+01, float* %a
  %a1 = load float, float* %a
  %divtmp = fdiv float %a1, 4.000000e+00
  store float %divtmp, float* %b
  ret void
}
```

- All of this code represents a program with an AST that looks like this:



Branching instructions

- Any full-featured language includes statements that allow the programmer to create branches in control flow, e.g. if/else statements or loops, so let's turn our attention to creating branching instructions in LLVM IR.

- There are two types of LLVM branching instructions we'll explore here:
    - ***Conditional branches*** – These branch to one of two specified basic blocks, depending on the value of a specified condition.
    - ***Unconditional branches*** – These always branch to the same specified basic block.

- We'll investigate these instructions in the context of creating a simple if/else statement.

- We'll start by creating a function to generate an if/else statement:

```
llvm::Value* ifElseStatement() {...}
```

- Note here that our function doesn't take any arguments. To simplify our program this function will simply always generate the same if/else statement, with the same condition and the same statements inside each of the if and else blocks.
    - If we were implementing code generation from an AST, our AST node for an if/else statement would have access to its own children in the AST (i.e. the condition and the if and else blocks) and could recursively generate code for each of those children.

- We'll specifically generate an if statement that looks something like this:

```
if (b < 8) {
  c = a * b;
} else {
  c = a + b;
}
```

- To start, let's generate a condition (`b < 8`) for our if statement. The first thing we'll do is add an additional case in the switch statement in our binary operation function to handle the `<` operator.

- To do this, we can use the IR builder's `CreateFCmpULT()` method (floating-point unordered less-than, with "unordered" indicating that either operand can be NaN). Importantly, this method returns a 1-bit unsigned integer (i.e. a boolean value), which we'll convert back to a floating-point value using the IR builder's `CreateUIToFP()` method, since all of our types in this program are `float`s:

  ```
  case '<':
    lhs = TheBuilder.CreateFCmpULT(lhs, rhs, "lttmp");
    return TheBuilder.CreateUIToFP(lhs,
      llvm::Type::getFloatTy(TheContext), "booltmp");
  ```

- Now, back in our `ifElseStatement()` function, we can generate our condition:

  ```
  llvm::Value* cond = binaryOperation(variableValue("b"),
    numericConstant(8), '<');
  if (!cond) {
    return NULL;
  }
  ```

- Typically, a decision about which way to branch in the control flow is made by comparing a condition value, like the one we just computed, to zero. If the condition is not equal to zero, we take the "if" branch. If it is equal to zero, we take the "else" branch. This gives some flexibility about what kinds of conditions can be used with if statements (for example, allowing us to simply say something like `if (b) {...}`).

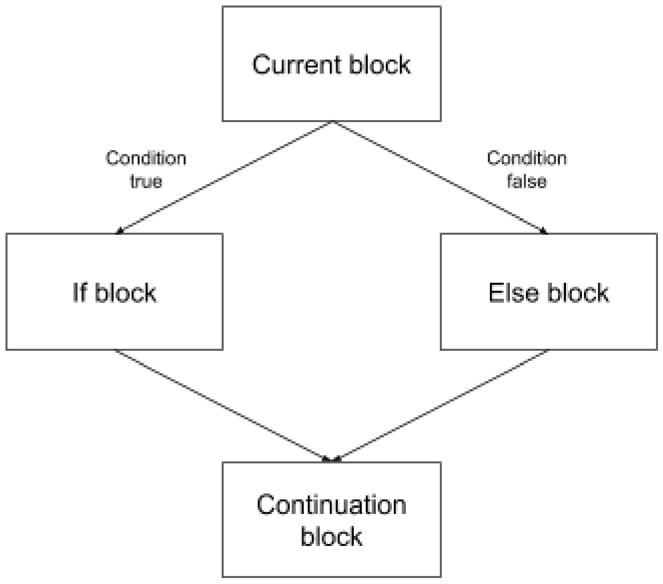
- We can make a comparison to zero using the IR builder's `CreateFCmpONE()` method (floating-point ordered not equal, where "ordered" indicates that neither argument can be NAN):

  ```
  cond = TheBuilder.CreateFCmpONE(cond, numericConstant(0),
    "ifcond");
  ```

- With the condition computed, we can start to work on branching. Aside from the current basic block, there are three different basic blocks associated with an if/else statement:
  - **The if block** – the block to which we branch if the condition is true (i.e. not equal to zero).

- **The else block** – the block to which we branch if the condition is not true (i.e. equal to zero).
- **The continuation block** – the block to which control returns from both the if block and the else block. If there is no else block associated with the if statement, the continuation block also acts as the “else” block.

- This looks something like this:



- Our next step will be to create LLVM BasicBlock objects to represent these three blocks:

```
llvm::Function* currFn =
  TheBuilder.GetInsertBlock()->getParent();
llvm::BasicBlock* ifBlock =
  llvm::BasicBlock::Create(TheContext, "ifBlock", currFn);
llvm::BasicBlock* elseBlock =
  llvm::BasicBlock::Create(TheContext, "elseBlock");
llvm::BasicBlock* continueBlock =
  llvm::BasicBlock::Create(TheContext, "continueBlock");
```

- Note here that we add the if block to the end of the current function, but we don’t add either the else block or the continue block. This is OK. We will still be able to refer to these blocks, and we’ll add them into the function later.

- With these `BasicBlock` objects in place, we can create the conditional branch instruction to evaluate our condition and select the direction control should flow:

  ```
  TheBuilder.CreateCondBr(cond, ifBlock, elseBlock);
  ```

- Next, we'll generate the IR for our if and else blocks. Again, if we were generating IR from an AST, we would have references here to the actual AST nodes representing the if block, and the else block, so we could recursively generate IR for them. Here, however, we'll just hard-code some instructions.

- Let's start with the IR for our if block:

  ```
  TheBuilder.SetInsertPoint(ifBlock);
  llvm::Value* aTimesB = binaryOperation(variableValue("a"),
    variableValue("b"), '*');
  llvm::Value* ifBlockStmt =
    assignmentStatement("c", aTimesB);
  TheBuilder.CreateBr(continueBlock);
  ```

- Note here that at the end of the block, we add an unconditional branch to the continuation block.

- We can do something similar to generate code for our else block:

  ```
  currFn->getBasicBlockList().push_back(elseBlock);
  TheBuilder.SetInsertPoint(elseBlock);
  llvm::Value* aPlusB = binaryOperation(variableValue("a"),
    variableValue("b"), '+');
  llvm::Value* elseBlockStmt =
    assignmentStatement("c", aPlusB);
  TheBuilder.CreateBr(continueBlock);
  ```

- Note that we start here by inserting the else block into the current function.

- We'll conclude by inserting the continuation block into the current function and setting the IR builder's insertion point to the continuation block:

  ```
  currFn->getBasicBlockList().push_back(continueBlock);
  TheBuilder.SetInsertPoint(continueBlock);
  ```

- Note that we won't explicitly add code into the continuation block. Rather, we'll simply leave the insertion point set to the continuation block so whatever code is generated next will be inserted here.

- The return value for the `ifElseStatement()` function is not important in our specific case, since we won't be using this value elsewhere. We'll simply return the continuation block to serve as a non-NULL return value:

  ```
  return continueBlock;
  ```

- Now, back in `main()`, we can make a call to `ifElseStatement()` to generate instructions for our if/else statement:

  ```
  llvm::Value* ifElse = ifElseStatement();
  ```

- Then, if we recompile and run our "compiler", we'll see that our `foo` function now contains the instructions for our if/else statement:

  ```
  define internal void @foo() {
  entry:
    %c = alloca float
    %b = alloca float
    %a = alloca float
    store float 1.600000e+01, float* %a
    %a1 = load float, float* %a
    %divtmp = fdiv float %a1, 4.000000e+00
    store float %divtmp, float* %b
    %b2 = load float, float* %b
    %lttmp = fcmp ult float %b2, 8.000000e+00
    %booltmp = uitofp i1 %lttmp to float
    %ifcond = fcmp one float %booltmp, 0.000000e+00
    br i1 %ifcond, label %ifBlock, label %elseBlock

  ifBlock:                     ; preds = %entry
    %a3 = load float, float* %a
    %b4 = load float, float* %b
    %multmp = fmul float %a3, %b4
    store float %multmp, float* %c
    br label %continueBlock
  ```

```
elseBlock:                   ; preds = %entry
  %a5 = load float, float* %a
  %b6 = load float, float* %b
  %addtmp = fadd float %a5, %b6
  store float %addtmp, float* %c
  br label %continueBlock

continueBlock:               ; preds = %elseBlock, %ifBlock
  ret void
}
```

- Here, we can see that each basic block has a comment that lists the block's control-flow predecessors.

Generating target machine code

- Let's explore generating target machine code from the IR we've constructed. Before we start, we'll need to add the following include statements to our code, since we'll be using components from these headers:

```
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Support/TargetRegistry.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/FileSystem.h"
#include "llvm/Target/TargetOptions.h"
#include "llvm/Target/TargetMachine.h"
```

- LLVM supports many different target architectures, and you can compile to any of them. To specify the architecture we want to target, we'll use a ***target triple*** string, which takes the form `<arch><sub>-<vendor>-<sys>-<abi>`.
  - If you have `clang` installed, you can see the target triple string for your machine by running the following command:
    `clang --version | grep Target`
  - For OSU's ENGR servers, the target triple string is `x86_64-redhat-linux-gnu`.

- Let's use the machine on which we're running as the compilation target. Luckily, we don't need to hard-code the target triple string for this target. In `main()`, after our IR is generated, we can add this line to get the target triple string:

  ```
  std::string targetTriple =
    llvm::sys::getDefaultTargetTriple();
  ```

- Next, we'll initialize LLVM's target code emitters. We can simply initialize all targets to make things easy for ourselves:

  ```
  llvm::InitializeAllTargetInfos();
  llvm::InitializeAllTargets();
  llvm::InitializeAllTargetMCs();
  llvm::InitializeAllAsmParsers();
  llvm::InitializeAllAsmPrinters();
  ```

- With the targets initialized, we can use our target triple string to get an `llvm::Target` object for our machine:

  ```
  std::string error;
  const llvm::Target* target =
    llvm::TargetRegistry::lookupTarget(targetTriple, error);
  if (!target) {
    std::cerr << error << std::endl;
    return 1;
  }
  ```

- In addition to the `Target` object, we also need an `llvm::TargetMachine` object, which provides a complete description of the machine we're targeting, including the specific CPU and specific processor features (e.g. SSE).
  - We can see a list of the specific CPUs and features LLVM supports for the x86 target my running this command:

    ```
    llc -march=x86 -mattr=help
    ```

- We'll just use the "generic" CPU with no special features:

```
std::string cpu = "generic";
std::string features = "";
llvm::TargetOptions options;
llvm::TargetMachine* targetMachine =
  target->createTargetMachine(targetTriple, cpu, features,
    options, llvm::Optional<llvm::Reloc::Model>());
```

- Next, we'll configure `TheModule` to give it information about the target machine. This will benefit us when we want to perform optimizations on our code:

  ```
  TheModule->setDataLayout(targetMachine->createDataLayout())
  ;
  TheModule->setTargetTriple(targetTriple);
  ```

- Now we're ready to emit object code for our target machine. We'll start by opening the file to which we'll write the object code:

  ```
  std::string filename = "foo.o";
  std::error_code ec;
  llvm::raw_fd_ostream file(filename, ec,
    llvm::sys::fs::F_None);
  if (ec) {
    std::cerr << "Could not open output file: "
      << ec.message() << std::endl;
    return 1;
  }
  ```

- Finally, we can write the target-machine object code to this file. To do this, we'll make use of an LLVM mechanism that makes a "pass" over the IR. We'll use a `PassManager` object to manage this pass:

  ```
  llvm::legacy::PassManager pm;
  ```

- We'll add a single pass to this `PassManager` object to emit object code to the specified file. We need to explicitly specify that we're emitting object code. We could emit assembly code instead, if we wished:

```
llvm::TargetMachine::CodeGenFileType type =
  llvm::TargetMachine::CGFT_ObjectFile;
if (targetMachine->addPassesToEmitFile(pm, file,
    NULL, type)) {
  std::cerr << "Can't emit object file for target machine"
    << std::endl;
  return 1;
}
```

- Finally, we can run the pass on our module and close the output file:

```
pm.run(*TheModule);
file.close();
```

- To confirm that our object code was successfully written, we can write a short C program that calls our `foo()` function. Before we get to the C program, let's modify our "compiler" to return the value of the variable `c`, so we can see `foo()` in action.

- First, we'll have to modify the return type of `foo()`:

```
llvm::FunctionType* fooFnType = llvm::FunctionType::get(
  llvm::Type::getFloatTy(TheContext), false
);
```

- Then, after our IR is generated, we'll return the value of `c` instead `void`:

```
TheBuilder.CreateRet(variableValue("c"));
```

- Now, we can write our simple C program to call `foo()` and print its return value:

```
#include "stdio.h"

extern float foo();

int main() {
  float x = foo();
  printf("Return from foo(): %f\n", x);
}
```

- Then, assuming that C program is saved in the file test.c, we should be able to successfully compile and run it, making sure to link with our generated object file:

  ```
  gcc test.c foo.o -o test
  ./test
  ```

- We should see the program print this line:

  ```
  Return from foo(): 64.000000
  ```

Performing optimizations

- We'll finish by applying some optimizations to our IR. To do this, we'll need to include these headers in our code:

  ```
  #include "llvm/Transforms/Scalar.h"
  #include "llvm/Transforms/Scalar/GVN.h"
  #include "llvm/Transforms/Utils.h"
  ```

- LLVM provides many ready-made optimizations that can be included easily into a compiler. The compiler writer can also write their own optimizations. Here, we'll apply just a few of LLVM's optimizations.

- Applying optimizations is done in passes, similar to the way we ran a pass over our module above to write object code to a file.

- LLVM supports whole-module optimization passes that analyze as large a body of code as possible (this could be a whole file, or optimizations can be run at link time to optimize over much of the program). It also supports per-function optimization passes that analyze just a single function.

- Here, we'll explore adding some per-function optimizations. To do this, we'll create a new `FunctionPassManager` object. We can do this in `main()` right after our call to `verifyFunction()`:

  ```
  llvm::legacy::FunctionPassManager* fpm =
    new llvm::legacy::FunctionPassManager(TheModule);
  ```

- Next, we'll add some specific optimization passes to the pass manager. For now, we'll just add a single optimization that tries to promote in-memory variables (declared with `alloca` instructions) to registers:

  ```
  fpm->add(llvm::createPromoteMemoryToRegisterPass());
  ```

- With our optimization added to the pass manager, we can simply run the pass manager on our function:

  ```
  fpm->run(*fooFn);
  ```

- If we compile and run our "compiler" now, we can see that this optimization does an effective job at simplifying our code by completely eliminating `alloca` instructions:

  ```
  define float @foo() {
  entry:
    %divtmp = fdiv float 1.600000e+01, 4.000000e+00
    %lttmp = fcmp ult float %divtmp, 8.000000e+00
    %booltmp = uitofp i1 %lttmp to float
    %ifcond = fcmp one float %booltmp, 0.000000e+00
    br i1 %ifcond, label %ifBlock, label %elseBlock

  ifBlock:                     ; preds = %entry
    %multmp = fmul float 1.600000e+01, %divtmp
    br label %continueBlock

  elseBlock:                   ; preds = %entry
    %addtmp = fadd float 1.600000e+01, %divtmp
    br label %continueBlock

  continueBlock:               ; preds = %elseBlock, %ifBlock
    %c.0 = phi float [ %multmp, %ifBlock ],
      [ %addtmp, %elseBlock ]
    ret float %c.0
  }
  ```

- Interestingly, we can see in `continueBlock` that because the value of `c` depends on whether control flowed from `ifBlock` or from `elseBlock`, LLVM needed to insert a ϕ-function to keep the IR in SSA form.

- There are many other effective optimizations available, so let's add a few more. We'll start by adding an optimization that tries to combine redundant instructions:

  ```
  fpm->add(llvm::createInstructionCombiningPass());
  ```

- Next, we'll add an optimization that tries to reassociate arithmetic expressions (for example 4 + (x + 5) → x + (4 + 5)) to make other optimizations, like constant folding, easier to find

  ```
  fpm->add(llvm::createReassociatePass());
  ```

- Next, we'll run the "global value numbering" optimization, which tries to eliminate redundant instructions:

  ```
  fpm->add(llvm::createGVNPass());
  ```

- Finally, we'll add an optimization that tries to simplify the control flow graph by deleting unreachable blocks, etc.:

  ```
  fpm->add(llvm::createCFGSimplificationPass());
  ```

- Now if we recompile and run our "compiler", we can see that adding these additional optimizations was highly effective, reducing our whole `foo()` function down to only a return statement!

  ```
  define float @foo() {
  entry:
    ret float 6.400000e+01
  }
  ```