Tutorial 4: Building Control Flow

ECE 466/566 Spring 2025

February 17, 2025
Due: ~two weeks, to be announced

1. Objective

In this example, we'll look at a simple language that contains control flow, and we'll generate LLVM IR with multiple basic blocks and branches connecting those blocks. This will involve:

- Interpreting a language specification with control flow and function calls.
- Allocation of variables on the stack, and generating loads and stores to access and update their values.
- Devising strategies for constructing control flow while parsing.
- Building a parser that uses mid-rule actions, and associating results with mid-rule actions to hold information that will be needed later.
- Building some more instructions:
    - Branch (br)
    - Conditional Branch (br)
    - Alloca (alloca)
    - Load (load)
    - Store (store)
    - Integer Compare (icmp)
    - Select (select)

2. Tokens and Grammar

To study control flow generation, we'll extend the expr language we had before to support control flow.

You can find the starter code the class repository at tutorials/3. You will need to pull my latest change to get this code.

1. `cd path/to/ncstate_ece566_spring2025`
2. Either stash or commit your changes
    a. `git stash`
    b. `OR:`
    c. `git commit -a -m"my last updates"`
3. Replay your changes on top of mine:
    a. `git pull --rebase`
    Or, if you set up my repo as a remote branch called ece566:
    b. `git pull --rebase ece566 main`
4. `cd tutorials/3`
5. Use CLion to build or build from the command line using make or cmake.
    a. `make`

Or:

```
mkdir build
cd build
cmake ..
cmake --build .
```

Tokens

Keyword/Regular Expression	Token Name
if	IF
while	WHILE
return	RETURN
[a-zA-Z]+	IDENTIFIER
[0-9]+	IMMEDIATE
=	ASSIGN
;	SEMI
,	COMMA
-	MINUS
+	PLUS
*	MULTIPLY
/	DIVIDE
!	NOT
(	LPAREN
)	RPAREN
{	LBRACE
}	RBRACE

Grammar

```
program : function
| program function;
| program SEMI SEMI; // needed since reading from stdin

function: IDENTIFIER LPAREN arglist_opt RPAREN LBRACE stmtlist RBRACE
;

arglist_opt : arglist
| %empty ;

arglist : IDENTIFIER
          | arglist COMMA IDENTIFIER
;

stmtlist :    stmt
           |  stmtlist stmt

;

stmt:   IDENTIFIER ASSIGN expr SEMI              /* expression stmt
*/
      | IF LPAREN expr RPAREN LBRACE stmtlist RBRACE   /*if stmt*/
      | WHILE LPAREN expr RPAREN LBRACE stmtlist RBRACE /*while
stmt*/
      | SEMI /* null stmt */
      | RETURN expr SEMI
;

exprlist_opt : exprlist
|;

exprlist : expr
           | exprlist COMMA expr;


expr:   IDENTIFIER
      | IMMEDIATE
      | IDENTIFIER LPAREN exprlist_opt RPAREN /* function call */
      | expr PLUS expr
      | expr MINUS expr
```

```
      | expr MULTIPLY expr
      | expr DIVIDE expr
      | MINUS expr
      | NOT expr
      | LPAREN expr RPAREN
;
```

Example Programs

Our program must be brace delimited and contain a stmtlist followed by a return statement. The simplest program would look like this:

```
f() { return 0; }
;;
```

Each statement in the statement list will either be an expression statement, an if statement, a while statement or a null statement. So, we could also write a program like this:

```
f() {
    a = 5;
    b = 0;
    if (a) {  b = 10; }
    return b;
}
;;
```

This program creates two variables. Then, if a is non-zero, we set b to 10. Finally, we return the value of b.

We can also write a program with a loop:

```
f() {
    a = 5;
    while (a) {  a = a - 1; }
    return a;
};;
```

We'll also extend the grammar with multiply, divide, and logical negation (!) operators to our expr non-terminal. That way, we can write an if statement that makes a choice on the logical negation of an expression, like this:

```
f() {
   a = -1;
   b = 0;
```

```
   if (!(a+1))  {  b = 10; }
   return b;
} ;;
```

Finally, we also support passing arguments and calling other functions:

```
fibb(n) {
   if (!(n-1)) {
      return 1;
   }
   if (!(n-2)) {
      return 1;
  }
   r = fibb(n-1) + fibb(n-2);
   return r;
}
```

3. Supporting Functions

Rather than putting all the code into one function, we expand our grammar to support arbitrary functions. This means that we need to delay creating our function from the main code used in Tutorial 3 and instead do it when we see a new function.

This should happen in the function rule.

```
function: IDENTIFIER LPAREN arglist_opt RPAREN LBRACE stmtlist RBRACE
```

Once we know the function name and the number of arguments, we can create the function in the LLVM module, something like this:

```
  std::vector<Type*> args;
  for(int i=0; i < number of arguments; i++) {
    args.push_back(i32);
  }

  // Create void function type with no arguments
  FunctionType *FunType =
    FunctionType::get(Builder.getInt32Ty(),args,false);

  // Create a main function
  Function *Function = Function::Create(FunType,
                           GlobalValue::ExternalLinkage,Name of
Function,M);
```

```
//Add a basic block to main to hold instructions
  BasicBlock *BB = BasicBlock::Create(TheContext, "entry",
                                Function);
  // Ask builder to place new instructions at end of the
  // basic block
  Builder.SetInsertPoint(BB);
```

There are two key steps:

- Make sure you scan the name of the IDENTIFIER token and store a copy in a variable. Update the scanner to do this one.
- Then, figure out how many parameters the function will take. We'll assume they are all integers. Associate arglist and arglist_opt with a **`vector<string>*`**.

C++

```
arglist_opt : arglist
{
  $$ = $1;
}
| %empty
{
  $$ = new vector<string>;
};


arglist : IDENTIFIER
{
  $$ = new vector<string>;
  $$->push_back($1); // remember IDENTIFIER
}
| arglist COMMA IDENTIFIER
{
  $$ = $1;
  $$->push_back($3); //remember IDENTIFIER
}
;
```

Then, in the function rule, we can build the function with the appropriate name and argument list length:

C++

```
function: IDENTIFIER LPAREN arglist_opt RPAREN
{ // mid-rule action!!
  Type *i32 = Builder.getInt32Ty();
```

```
  vector<string> &l = *$3;

  std::vector<Type*> args;

  for(int i=0; i<l.size(); i++) {
    args.push_back(i32);
  }

  // Create i32 return function type with arguments
  FunctionType *FunType =
    FunctionType::get(Builder.getInt32Ty(),args,false);

  // Create a main function
  Function *Function = Function::Create(FunType,

GlobalValue::ExternalLinkage,$1,M);

  //Add a basic block to main to hold instructions
  BasicBlock *BB = BasicBlock::Create(TheContext, "entry",
                                      Function);
  // Ask builder to place new instructions at end of the
  // basic block
  Builder.SetInsertPoint(BB);

  // Now we're ready to add statements to function
} /* rest of rule */
LBRACE stmtlist RBRACE
;
```

4. Allocating Variables on the Stack

In comparison with Tutorial 2 and 3, we eliminate the REG terminal and replace it with a general ID that can have an arbitrary single character name, using this regular expression: [a-zA-Z].

Rather than interpreting an ID as a register, we will make space for the variable on the stack. Every time we refer to the variable on the right hand side (RHS) of an ASSIGN, we'll load from memory. When we refer to it on the left hand side (LHS), we'll store to memory. This is much like variables in C, and it matches the assumptions we make for Project 2.

```
a = 0; /* store 0 to a */
b = a; /* load from a, and store to b */
```

To allocate stack space for a and b, we can use the alloca instruction. Then, we can use that address to load or store, like this:

```
%a = alloca i32
%b = alloca i32
```

```
store i32 0, %a
%2 = load i32 %a
store i32 %2, %b
```

Generate Code to Support Variables on the Stack

When a variable is assigned for the first time, then we will emit an alloca instruction and a store to assign its initial value.

C

```
stmt:   ID ASSIGN expr SEMI {
  // Look to see if we already allocated it
  LLVMValueRef var = map_find($1);
  if (var==NULL) {
     // We haven't so make a spot on the stack
     var = LLVMBuildAlloca(Builder,LLVMInt32Type(),$1);
     // remember this location and associate it with $1
     map_insert($1,var);
  }
  // store $3 into $1's location in memory
  LLVMBuildStore(Builder,$3,var);
}
```

C++

```
// Note, put at the top of the file:
std::map<std::string,Value*> idMap;

// In the stmt rule:
stmt:   ID ASSIGN expr SEMI     /* expression stmt */
{
// Look to see if we already allocated it
  Value* var = NULL;
  if (idMap.find($1)==idMap.end()) {
     // We haven't so make a spot on the stack
    var = Builder.CreateAlloca(Builder.getInt32Ty(),
                               nullptr,$1);
     // remember this location and associate it with $1
    idMap[$1] = var;
  } else {
    var = idMap[$1];
  }
  // store $3 into $1's location in memory
  Builder.CreateStore($3,var);
}
```

Next, in the expr rule when we match ID, we need to load it's value.

C

```
expr :  ID
{
  // First, get address
  LLVMValueRef loc = map_find($1);

  // if loc is NULL, we should generate an error

  // Load value
  $$ = LLVMBuildLoad(Builder,loc,$1);
}
```

C++

```
expr : ID
 {
   Value * addr = idMap[$1];
   $$ = Builder.CreateLoad(Builder.getInt32Ty(),addr,$1);
 }
```

After you make these changes, recompile and test on a simple program. If you are using my starter code in the repository, you can just type `make`. If there are no errors, then run the program and type in the following input.

To run:

```
make
./t4
f() { a = 0; return a; }
```

Then type: `make main.ll`

You should get this in the test.ll file:

```
; ModuleID = 'main.bc'

define i32 @f() {
entry:
  %0 = alloca i32
  store i32 0, i32* %0
  %1 = load i32* %0
  ret i32 %1
}
```

Try a few other test programs and see how it works. Note, you may want to go back and add error checking in case a variable is used that was never assigned.

```
/*
ConstantInt *ci = dyn_cast<ConstantInt>($$);

if ( $$ == Builder.getInt32(0) ) {
  // only be true if $$ is i32 0
}

if (ci != NULL) {
  printf("%ld\n", ci->getZExtValue()); // getSExtValue()
}
*/
```

Generate Code to Support Arguments

Now that we see how we can look-up arguments on the stack using idMap, we can do something similar with function arguments to make them accessible. Go back and insert similar code into the `function` rule that creates an alloca and stores the initial argument value into that location in memory. Also, insert the name into the idMap. Then future requessts of the argument can be handled just like all other variables.

C++

```
// Ask builder to place new instructions at end of the
 // basic block
 Builder.SetInsertPoint(BB);
// After setting insert point
 for (int i=0; i<l.size(); i++) {
   //idMap[l[i]] = Function->getArg(i);

   // Look to see if we already allocated it
   Value* var = NULL;
   if (idMap.find(l[i])==idMap.end()) {
    // We haven't so make a spot on the stack
     var = Builder.CreateAlloca(Builder.getInt32Ty(),
                    nullptr,l[i]);
    // remember this location and associate it with $1
     idMap[l[i]] = var;
   } else {
     yyerror("repeat declaration of same variable!");
     return 1;
   }

   Builder.CreateStore(Function->getArg(i),var);
```

	}

5. Expressions, Logical Negation

To support it, we have to consider what logical negation means. We'll assume the same meaning as in the C language. In the case of a variable x, !x evaluates to 0 if x is non-zero and 1 if x is 0.

We can easily accomplish this using the icmp instruction and comparing x to zero:

C	`expr : NOT expr`
	`{`
	`  LLVMValueRef zero = LLVMConstInt(LLVMTypeOf($2),0,1);`
	`  $$ = LLVMBuildICmp(Builder, LLVMIntEQ, $2,`
	`  zero,"logical.not");`
	`}`

C++	`expr : NOT expr`
	`{`
	`  $$ = Builder.CreateICmpEQ($2,Builder.getInt32(0));`
	`}`

This looks like it should work, but it may not work all the time. `Icmp` produces an `i1` result but every other expression in our grammar produces an `i32` result. If we send the result of logical-not to an add expression, we'll end up with a mix of types, and that will lead to invalid LLVM IR.

To produce the required `i32` result, there are multiple approaches.

Approach 1. Use a select instruction to choose a constant 0 or 1

The select instruction in LLVM is like the ternary operator in C:

```
cond ? true-value : false-value
```

We can use it to select either a true or false value depending on the outcome of the icmp.

C	`expr : NOT expr`
	`{`
	`  LLVMValueRef zero = LLVMConstInt(LLVMTypeOf($2),0,1);`
	`  LLVMValueRef icmp = LLVMBuildICmp(Builder, LLVMIntEQ, $2,`

	zero,"logical.not"); $$ = LLVMBuildSelect(Builder, icmp, // condition LLVMConstInt(LLVMInt32Type(),1,1), // if-true LLVMConstInt(LLVMInt32Type(), 0, 1), // if-false "logical.not"); }

C++	expr : NOT expr
	{
	Value *icmp =
	Builder.CreateICmpEQ($2,Builder.getInt32(0),"logical.not");
	$$ = Builder.CreateSelect(icmp,Builder.getInt32(1),
	
	Builder.getInt32(0),"logical.not");
	}

To test it out, use a simple program like this:

```
f() { a=0; return !a; }
```

You should get something like this:

```
define i32 @f() {
entry:
  %a = alloca i32
  store i32 1, i32* %a
  %a1 = load i32* %a
  %logical.not = icmp eq i32 %a1, 0
  %logical.not2 = select i1 %logical.not, i32 1, i32 0
  ret i32 %logical.not2
}
```

Approach 2. Promote the i1 type to i32 using ZExt

Here we force the result to be `i32` by zero extending it to 32 bits.

C	expr : NOT expr

	{
	  LLVMValueRef zero = LLVMConstInt(LLVMTypeOf($2),0,1);
	  LLVMValueRef icmp = LLVMBuildICmp(Builder, LLVMIntEQ, $2,
	  zero,"logical.not");
	  $$ = LLVMBuildZExt(Builder, icmp,
	LLVMInt32Type(),"logical.not");
	}

C++	expr : NOT expr
	{
	  Value *icmp = Builder.CreateICmpEQ($2,Builder.getInt32(0));
	  $$ = Builder.CreateZExt(icmp,Builder.getInt32Ty());
	}

To test it out, use a simple program like this:

```
f() { a=0; return !a; }
```

You should get something like this:

```
define i32 @f() {
entry:
  %a = alloca i32
  store i32 1, i32* %a
  %a1 = load i32* %a
  %logical.not = icmp eq i32 %a1, 0
  %logical.not2 = zext i1 %logical.neg to i32
  ret i32 %logical.neg2
}
```

6. Generating Control Flow

Now we need to generate control flow for our new statements. This involves at least two steps:

- making basic blocks and
- connecting them with branches.

Creating Basic Blocks

There are two functions, in particular, that we're gonna need from the LLVM API to make this happen. We need a function that creates new basic blocks and adds them to the current function, and we need to control which basic block the builder places instructions inside. Often, we'll do something like this:

C	// Make a new block

	`LLVMBasicBlockRef newblock = LLVMAppendBasicBlock(Fn, "new.bb");`
	
	`// Position Builder at end of new block`
	`LLVMPositionBuilderAtEnd(Builder, newblock);`

C++	`BasicBlock *BB = BasicBlock::Create(TheContext, "new.bb", Function);`
	
	`// Position Builder at end of new block`
	`Builder.SetInsertPoint(BB);`

Note, when we make a new block it starts out empty. So, the end of the block is the same as the beginning, and we start inserting at the end to fill up the block with instructions.

Branch Instructions

Whenever we add blocks, we must insert branch instructions to say how they are connected.

Important

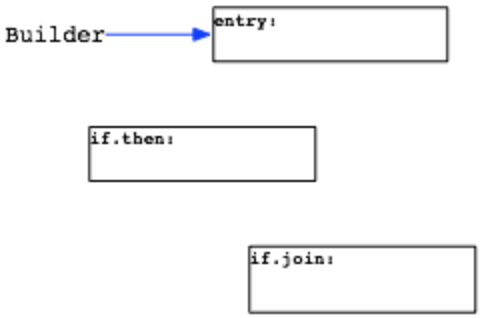
A branch or some kind of terminator instruction must always be the last instruction of a basic block otherwise the IR is illegal.

Examples of terminator Instructions: direct branch (br), conditional branch (br), return (ret)

For example, when we find an if-statement, we'll need to generate two new basic blocks: the then-block and the converge-block that joins the true and false edges in the control flow graph. Something like this:

C	`LLVMBasicBlockRef thenblock = LLVMAppendBasicBlock(Fn, "if.th");` `LLVMBasicBlockRef joinblock = LLVMAppendBasicBlock(Fn, "if.jn");`
C++	`BasicBlock *then = BasicBlock::Create(TheContext, "if.th", Function);` `BasicBlock *join = BasicBlock::Create(TheContext, "if.jn", Function);`

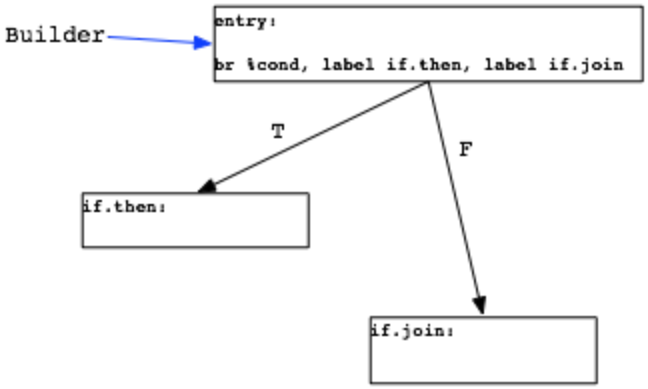
The Builder will already be inside some block--probably the entry block, as shown below.



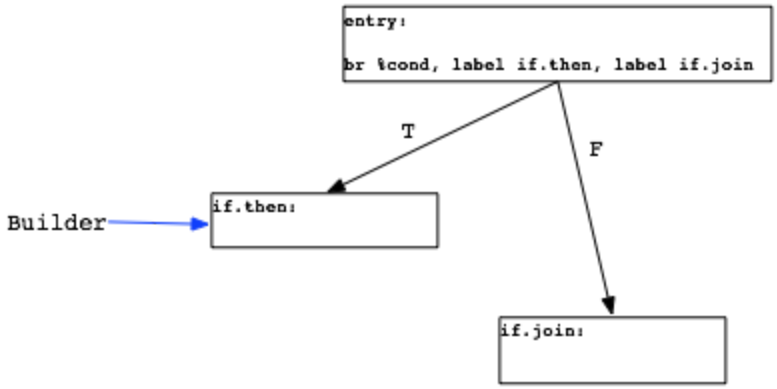
So, we just need to make a branch that selects which way to go. Let's assume that val has the condition value already:

C	`// val = LLVMBuildICmp ...` `LLVMValueRef branch = LLVMBuildCondBr(Builder, val, thenblock, joinblock);`
C++	`// val = Builder.CreateICmp(...);` `Value* branch = Builder.CreateCondBr(val, then, join);`

Now the graph looks like this:



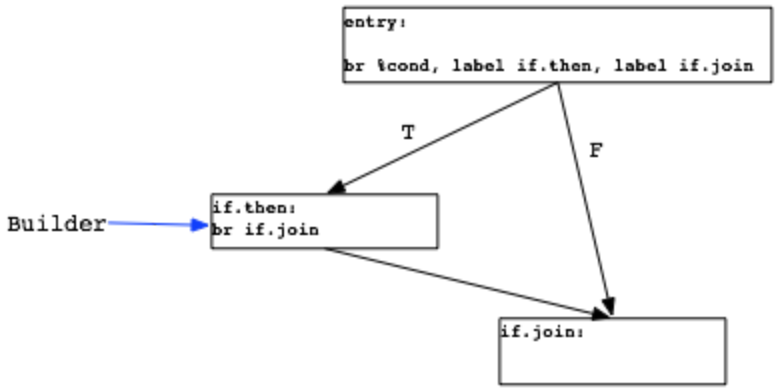
C	`LLVMPositionBuilderAtEnd(Builder,thenblock);` `// Now, generate code in if.then block`
C++	`Builder.SetInsertPoint(then);` `// Now, generate code in if.then block`



This creates the branch that links the entry block with the if.then block and if.join block. Then, at the end of the thenblock, we need an unconditional branch that connects the thenblock to the joinblock:

C	`LLVMValueRef branch = LLVMBuildBr(Builder,joinblock);` `LLVMPositionBuilderAtEnd(Builder, joinblock);`
C++	`Builder.CreateBr(join);` `Builder.SetInsertPoint(join);`

Now the control flow graph is complete:



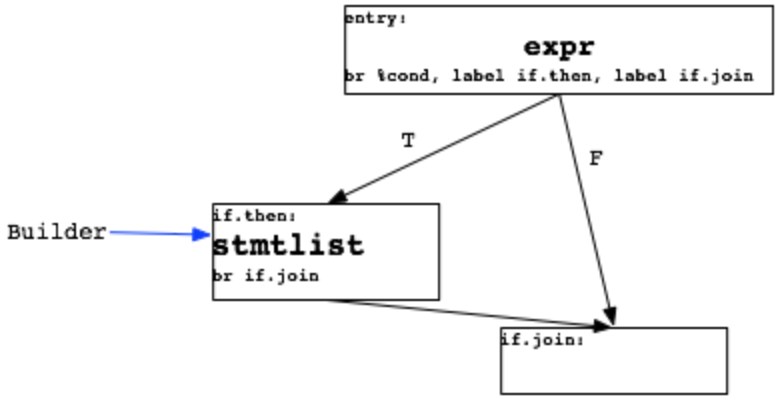
7. How to build control flow during parsing: mid-rule actions

Up until now, we've had a fairly simple grammar and that allowed us to execute actions after a rule was fully matched. But, that won't work for control flow. Consider the following rule in our grammar:

```
stmt:  IF LPAREN expr RPAREN LBRACE stmtlist RBRACE
{
      // what happens here?
```

```
      // what do we put in here?
}
;
```

Which basic block should have `expr` and which should have the `stmtlist`? Conceptually, it should look like this:



To properly generate code for this rule, we need to make two basic blocks: the then-block and join block. Also, we need for the stmtlist to be placed in the then-block.

But, this leads to a problem. Recall that before a rule is completely matched, each non-terminal in the rule will have already matched and executed its action. That means it's too late to put the stmtlist in a new basic block. In fact, the statements will be put wherever the Builder was previously pointing, which would be in the `entry` block. So, if we delay creation of the blocks until the entire rule is matched, the stmtlist will end up in the wrong place.

We need to exploit a feature of Bison called a mid-rule action to generate control flow correctly. Mid-rule actions let us execute some code in the middle of the rule rather than waiting until the end when it's too late, like this:

```
stmt:  IF LPAREN expr RPAREN {

  // first midrule action
  //  1. Make join block and then block
  //  2. Create icmp for $3
  //  3. insert conditional branch
  //  4. position builder in then-block
  //  5. remember join block somewhere (global?)

  //  Now stmtlist will be put in the then-block.

} LBRACE stmtlist RBRACE {
```

```
  // final action
  //  1. find join block
  //  2. insert branch to join block
  //  3. position builder in join block
  //  done with if!

};
```

The mid-rule actions execute as the if-statement is parsed, not at the end of parsing of the statement. This let's us set up the then-block and join-block after the `expr` sub-rule is matched.

Implement the if-statement control flow

We'll implement the actions for the if-statement according to the pseudo-code shown above.

C

```
stmt:  IF LPAREN expr RPAREN {
  // first mid-rule action
  //  1. Make join block and then block
  LLVMBasicBlockRef then = LLVMAppendBasicBlock(Fn,
                           "then.block");
  LLVMBasicBlockRef join = LLVMAppendBasicBlock(Fn,
                           "join.block");

  //  2. Create icmp to evaluate if $3 is true or not:
  LLVMValueRef zero = LLVMConstInt(LLVMTypeOf($3),0,1);
  LLVMValueRef cond = LLVMBuildICmp(Builder, LLVMIntNE, $3,
                                    zero,"cond");
  //  3. insert conditional branch
  LLVMBuildCondBr(Builder,cond,then,join);
  //  4. position builder in then-block
  LLVMPositionBuilderAtEnd(Builder,then);
  //  5. remember join block somewhere, declare new global

  BBjoin = join; /* declare BBjoin at top of file */
  //  Now stmtlist will be put in the then-block. Excellent!
} LBRACE stmtlist RBRACE {
  // final action
  //  1. find join block
  LLVMBasicBlockRef join = BBjoin;
  //  2. insert branch to join block
  LLVMBuildBr(Builder,join);
  //  3. position builder in join block
```

```
  LLVMPositionBuilderAtEnd(Builder,join);
  //  done with if
};
```

C++

```
stmt: IF LPAREN expr RPAREN
{
  //  1. Make join block and then block

  BasicBlock *then = BasicBlock::Create(TheContext,
                     "if.then",TheFunction);
  BasicBlock *join = BasicBlock::Create(TheContext,
                     "if.join",TheFunction);
  //  2. Create icmp to evaluate if $3 is true or not:
  //  3. insert conditional branch
  Builder.CreateCondBr(
            Builder.CreateICmpNE($3,Builder.getInt32(0)),
            then,join);
  //  4. position builder in then-block
  Builder.SetInsertPoint(then);
  //  5. remember join block somewhere, declare new global
  BBjoin = join;
  // now, stmtlist will be put inside then block
}
    LBRACE stmtlist RBRACE   /*if stmt*/
{
  // merge back to join block
  //  1. find join block
  //  2. insert branch to join block
  Builder.CreateBr(BBjoin);
  //  3. position builder in join block
  Builder.SetInsertPoint(BBjoin);
}
```

Compile and test your code. Try a simple program like this:

```
./t4
{ a=1; if (a) { a=2; } return a; }
```

You should get output like this:

```
; ModuleID = 'main.bc'
```

```
define i32 @main() {
entry:
  %a = alloca i32
  store i32 1, i32* %a
  %a1 = load i32* %a
  %cond = icmp ne i32 %a1, 0
  br i1 %cond, label %then.block, label %join.block

then.block:                                       ; preds = %entry
  store i32 2, i32* %a
  br label %join.block

join.block:                                       ; preds = %then.block, %entry
  %a2 = load i32* %a
  ret i32 %a2
}
```

Will our implementation work on a nested if statement?

```
{ a=0; b=0; if (a) { if (b) { a=1; } } return a; }
```

Try it out and see!

What did you see? Hopefully, you saw that it didn't work. The main problem is that we used a single global variable to track multiple join points. Clearly one variable can't track multiple values. We need a stack, at a minimum, to track the information. Since if-statements will be strictly nested, we could make a stack that keeps track of the next join point. Each time we parse an if-statement, we push its information on the top of the stack. As we exit parsing each if-statement, we can pop the last join point off the top of the stack---revealing the next one. A stack is general enough to work in every case, and is a reasonable approach.

Fortunately, Bison provides a more elegant solution that allows us to use the parse tree directly. Each mid-rule action is treated like a positional argument in the rule, and we can associate an output with each of these mid-rule actions. But, since there's no way to associate a %type command with a mid-rule action, we have to explicitly specify a field in the union, like this:

```
%union {
  char * id;
  int imm;
  LLVMValueRef val;
  LLVMBasicBlockRef bb;
}
```

```
$<bb>$ = joinblock;
```

Then, in the final action, we can grab the reference to the joinblock using a positional argument for the mid-rule action, $5. The mid-rule action is the fifth position of the rule, so we can access the output of the rule using $5. This yields the following new implementation:

```
stmt:  IF LPAREN expr RPAREN {
  // first mid-rule action
  //  1. Make join block and then block
  LLVMBasicBlockRef then = LLVMAppendBasicBlock(Fn,"then.block");
  LLVMBasicBlockRef join = LLVMAppendBasicBlock(Fn,"join.block");

  //  2. Create icmp to evaluate if $3 is true or not:
  LLVMValueRef zero = LLVMConstInt(LLVMTypeOf($3),0,1);
  LLVMValueRef cond = LLVMBuildICmp(Builder, LLVMIntNE, $3,
                                    zero,"cond");
  //  3. insert conditional branch
  LLVMValueRef br = LLVMBuildCondBr(Builder,cond,then,join);
  //  4. position builder in then-block
  LLVMPositionBuilderAtEnd(Builder,then);
  //  5. remember join block as output of mid-rule action
  $<bb>$ = join;
  //  Now stmtlist will be put in the then-block. Excellent!

} LBRACE stmtlist RBRACE {
  // final action
  //  1. find join block
  LLVMBasicBlockRef join = $<bb>5;
  //  2. insert branch to join block
  LLVMBuildBr(Builder,join);
  //  3. position builder in join block
  LLVMPositionBuilderAtEnd(Builder,join);
  //  done with if
};
```

Why is this better than a single global variable? Well, we essentially get the advantage of storing a basic block in some nodes of the parse tree within if-statements. Regardless of how many if-statements are nested, each one is guaranteed to have its own mid-rule action in the parse tree where we can save the joinblock. It's as good as having a stack and doesn't require us to implement an additional data structure.

Compile and test your code and make sure that it supports nested if-statements.

Implement the While-statement Control Flow

The control-flow structure for the while loop is strikingly similar to the if-statement. One difference is that the end of the loop body branches back to the condition rather than branching to the join block. The other difference is that the conditional expression must execute in it's own block. The former difference is fairly trivial, but the latter requires a somewhat different approach compared with the if-statement. As a result, we end up with 2 mid-rule actions, one that creates a the conditional expression block and one that sets up the body.

```
stmt: WHILE LPAREN
{
    // 1. Make a block for the cond
    // 2. Insert a branch from the current block to cond
    // 3. Move the builder to the cond block
    // 4. Remember the cond block ($<bb>2)
}
expr RPAREN
{
    // 1. Make the block for the body
    // 2. Make the join block
    // 3. Build a conditional branch to body and join block
    // 4. Position builder in the body
    // 5. Remember the join block ($<bb>6)
}
LBRACE stmtlist RBRACE
{
    // 1. Make an unconditional branch back to $<bb>2
    // 2. Move builder to $<bb>6
}
;
```

The final code looks like this:

C	WHILE
	{
	  // 1. Make a block for the cond
	  LLVMBasicBlockRef cond =
	LLVMAppendBasicBlock(Fn,"while.cond");
	  // 2. Insert a branch from the current block to cond
	  LLVMBuildBr(Builder,cond);
	  // 3. Move the builder to the cond block

	LLVMPositionBuilderAtEnd(Builder, cond);
	// 4. Remember the cond block ($<bb>2)
	$<bb>$ = cond;
	} LPAREN expr RPAREN {
	// 1. Make the block for the body
	LLVMBasicBlockRef body =
	LLVMAppendBasicBlock(Fn,"while.body");
	// 2. Make the join block
	LLVMBasicBlockRef join =
	LLVMAppendBasicBlock(Fn,"while.join");
	
	// 3. Build a conditional branch to body and join block
	LLVMValueRef zero = LLVMConstInt(LLVMTypeOf($4),0,1);
	LLVMValueRef cond = LLVMBuildICmp(Builder, LLVMIntNE, $4,
	zero,"cond");
	LLVMValueRef br = LLVMBuildCondBr(Builder,cond,body,join);
	
	// 4. Position builder in the body
	LLVMPositionBuilderAtEnd(Builder,body);
	// 5. Remember the join block ($<bb>$)
	$<bb>$ = join;
	}
	LBRACE stmtlist RBRACE
	{
	// 1. Make an unconditional branch back to $<bb>2
	LLVMBuildBr(Builder,$<bb>2);
	// 2. Move builder to $<bb>6
	LLVMPositionBuilderAtEnd(Builder, $<bb>6);
	}

C++	stmt: WHILE
	{
	BasicBlock *expr =
	BasicBlock::Create(TheContext,"w.expr",TheFunction);
	Builder.CreateBr(expr);
	Builder.SetInsertPoint(expr);
	$<bb>$ = expr;
	}
	LPAREN expr RPAREN
	{
	BasicBlock *body =
	BasicBlock::Create(TheContext,"w.body",TheFunction);
	BasicBlock *exit =
	BasicBlock::Create(TheContext,"w.exit",TheFunction);

```
  Builder.CreateCondBr(Builder.CreateICmpNE($4,
Builder.getInt32(0)),body,exit);
  Builder.SetInsertPoint(body);
  $<bb>$ = exit;
}
LBRACE stmtlist RBRACE
{
  Builder.CreateBr($<bb>2);
  Builder.SetInsertPoint($<bb>6);
}
```

Compile and test your code. Try this program:

```
f{a=10; while(a) { a = a-1; } return a; }
```

You should get:

```
define i32 @main() {
entry:
  %a = alloca i32, align 4
  store i32 10, i32* %a, align 4
  br label %w.expr

w.expr:                                           ; preds = %w.body,
%entry
  %a1 = load i32, i32* %a, align 4
  %0 = icmp ne i32 %a1, 0
  br i1 %0, label %w.body, label %w.exit

w.body:                                           ; preds = %w.expr
  %a2 = load i32, i32* %a, align 4
  %expr.MINUS.expr = sub i32 %a2, 1
  store i32 %expr.MINUS.expr, i32* %a, align 4
  br label %w.expr

w.exit:                                           ; preds = %w.expr
  %a3 = load i32, i32* %a, align 4
  ret i32 %a3
}
```

Try out simple while-loops, nested while-loops, and nested if-statements and while loops. If you added this code in the right way, it should all work!

8. Function Calls

The only thing remaining to implement are function calls. These are fairly straightforward with all of our other functionality. `Builder.CreateCall(F,argslist)` will make the call for us.

We need to associate a exprlist with a vector of values, much the same as we did to creat the argument list:

C++

```
exprlist_opt : exprlist
{
  $$ = $1;
}
| %empty
{
  $$ = new vector<Value*>;
}
;

exprlist : expr
{
  $$ = new vector<Value*>;
  $$->push_back($1);
}
| exprlist COMMA expr
{
  $$ = $1;
  $$->push_back($3);
}
;
```

Then, we can use that to pass the arguments to the Builder to create the call.

C++

```
expr: IDENTIFIER LPAREN exprlist_opt RPAREN
{
  Function *F = ... // look up function from name
  $$ = Builder.CreateCall(F,*$3);
}
```

We still need a way to figure out the function we are calling using the name. The basic idea is that we need a map from the function name to the Function object we created. There are several ways of achieving this. The easiest is using the Module (M):

```
M->getFunction(name); // returns null on undeclared function
```

This will work if we have created the function already. But, there a number of things that could go wrong here. What are they? Consider adding some checks that prevent or help us avoid seg-faults when those problems occur.

Questions

1. How did we create local variables on the stack? How were they accessed?
   We created local variables on the stack by implementing the "alloca" instruction. These were accessed through the "store" and "load" parts of the code.
2. What mechanisms were needed to construct control flow?
   The mechanisms that were needed to construct the control flow include initializing back edges in loops, and using "br" to handle the branch instructions in LLVM IR.
3. How did we use mid-rule actions; why is it better to use mid-rule results than a global variable?
   We used mid-rule actions in the code specifically in the middle of a rule in the expr.y file (ex: breaking up LBRACE stmlist RBRACE from the full list of actions and adding specific code necessary to that part). It is better to use mid-rule results than a global variable because it allows for certain sections of the script to compile/compute prior to the specific rule completing.
4. What kind of variable did we use to a create a kind of simple symbol table?
   The variable we use to create a simple symbol table is "map<string, Value*> idMap;". This map is able to hold the variable addresses associated with the simple symbol table.
5. What kinds of problems can arise when we try to generate the code to call functions?
   Some problems that may arise when we try to generate the necessary code to call functions include issues in aligning the stack, saving the correct register values, and misdirection from pointers throughout the code.

Appendix: Helpful Functions in the LLVM C Module

Note, this appendix is most helpful for C language APIs.

These functions position the Builder at a specific location for instruction creation. Usually, we just need to position the Builder at the end of a basic block using LLVMPositionBuilderAtEnd:

void	**LLVMPositionBuilder** (**LLVMBuilderRef** Builder, **LLVMBasicBlockRef** Block, **LLVMValueRef** Instr)
void	**LLVMPositionBuilderBefore** (**LLVMBuilderRef** Builder, **LLVMValueRef** Instr)
void	**LLVMPositionBuilderAtEnd** (**LLVMBuilderRef** Builder, **LLVMBasicBlockRef** Block)

LLVMBasicBlockRef	**LLVMGetInsertBlock** (**LLVMBuilderRef** Builder)

This function adds an empty basic block to the function for us, but it always adds them to the end of the basic block list within the function:

LLVMBasicBlockRef	**LLVMAppendBasicBlock** (**LLVMValueRef** Fn, **const** char ***Name**)

Sometimes, when generating control flow, you want to insert new blocks in the middle. So, we can do that with this function:

LLVMBasicBlockRef	**LLVMInsertBasicBlock** (**LLVMBasicBlockRef** InsertBeforeBB, **const** char ***Name**)

A Bit About Branches

Conditional branches need an argument of type i1. This is a 1-bit integer: in other words, a boolean. There are a few ways to create this. But, usually, we'll use the `icmp` instruction. It can perform a comparison of an integer to zero (or not zero) which is useful for our if and while statements.

LLVMValueRef	**LLVMBuildICmp** (**LLVMBuilderRef**, **LLVMIntPredicate** Op, **LLVMValueRef** LHS, **LLVMValueRef** RHS, **const** char ***Name**)

enum
LLVMIntPredicate

Enumerator:

LLVMIntEQ equal

LLVMIntNE	not equal
LLVMIntUGT	unsigned greater than
LLVMIntUGE	unsigned greater or equal
LLVMIntULT	unsigned less than
LLVMIntULE	unsigned less or equal
LLVMIntSGT	signed greater than
LLVMIntSGE	signed greater or equal
LLVMIntSLT	signed less than
LLVMIntSLE	signed less or equal

Definition at line **318** of file **Core.h**.

Usually, it looks something like this:

```
// Get a zero that matches the type of lhs
LLVMValueRef zero = LLVMConstInt(LLVMTypeOf(lhs),0);
LLVMValueRef icmp = LLVMBuildICmp(Builder, LLVMIntNE, lhs,
                                  zero,"ifcond");
```

When you use icmp, it's important to remember that the two arguments, lhs and rhs, must be of the same type in LLVM IR. For example, they should both be of i32 type or i32* or i32 or i8. If you have a type mismatch on integers, you can correct it by using the sign-extend or zero-extend operations:

LLVMValueRef	**LLVMBuildZExt** (**LLVMBuilderRef**, **LLVMValueRef** Val, **LLVMTypeRef** DestTy, **const** char ***Name**)
LLVMValueRef	**LLVMBuildSExt** (**LLVMBuilderRef**, **LLVMValueRef** Val, **LLVMTypeRef** DestTy, **const** char ***Name**)

But, if you have a type mismatch otherwise, you would need to generate the appropriate instructions to convert between the types. This doesn't often come up, so it's not worth worrying about except in special cases. But, you will encounter this in Project 2.

It's also important to remember that it will always produce a result that's of type i1.