IR Code Generation

Fan Long

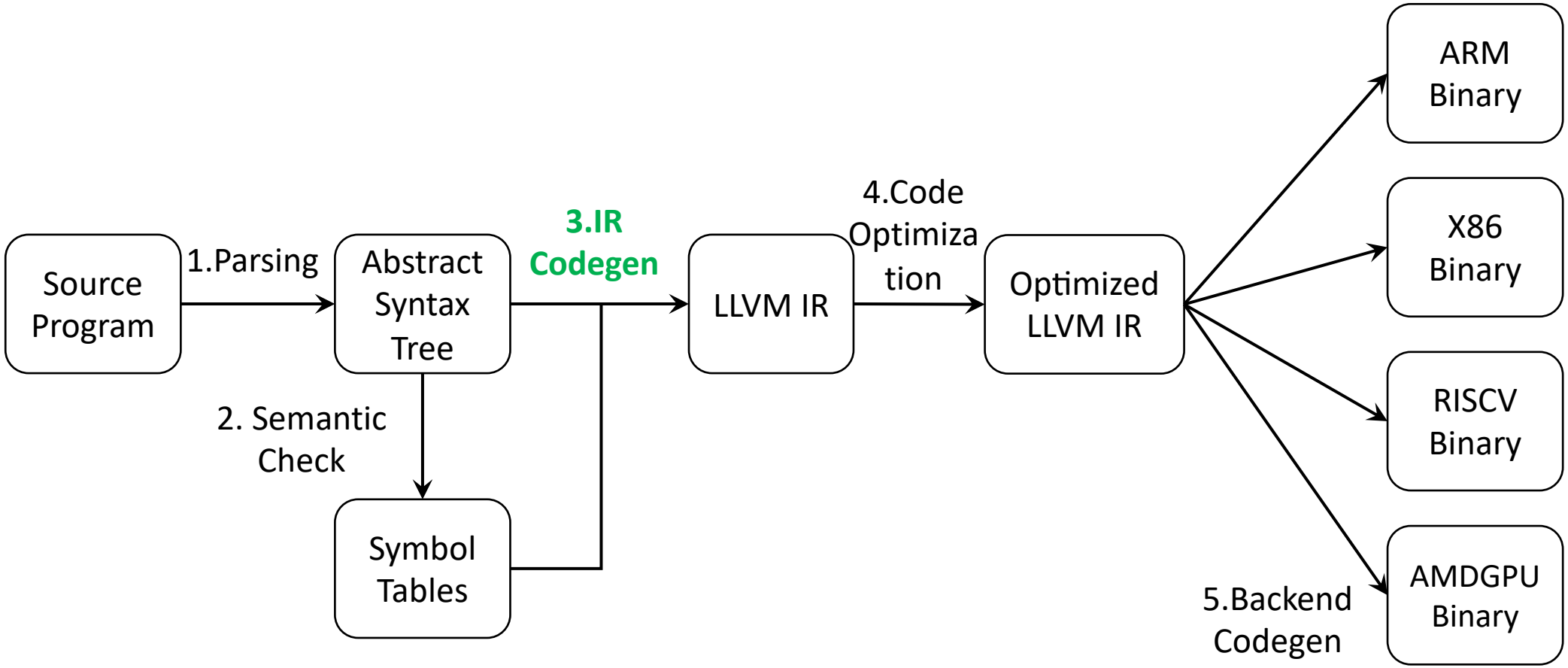
University of Toronto

UNIVERSITY OF TORONTO

Translation and Code Generation

- The ultimate goal of a compiler is to transform a program in some source language into machine instructions for some target machine.
- In many compilers this transformation is a two-step process
  - *Translate* the program from its syntactic representation (parse tree) into some easy to generate intermediate representation (IR), e.g., **LLVM IR**.
  - *Generate Code* for the target machine from the intermediate representation.
- There are a number of advantages to this two-phase approach
  - The IR is usually very easy to generate and manipulate.
  - It is often convenient to perform machine-independent optimization by manipulating the IR.
  - One IR can serve as an interface to code generators for different hardware (e.g., llvm/clang)

Typical LLVM Compiler Workflow



Translation of Programs

- Translation is the process of transforming a program into some intermediate representation.
- Input to the translation process is the representation of the program as produced by the parser *after* it has been subject to semantic analysis.
- Conceptually translation is performed on a parse tree for the program
- Major translation issues are expressions and control structures.
- Translation builds the programs control flow graph.

Abstract Syntax Tree Directed Translation

- Perform semantic analysis and code generation by walking the AST.
- Write the code translation as an AST visitor.
- Emit LLVM IR statement by statement and expression by expression.
  - Map global variables to LLVM global variables.
  - Map local variables to LLVM **alloca** stack allocation.
  - Use LLVM virtual registers to hold temporary results.

Global Variable Translation

```
1  #include <stdio.h>
2  int x, y ;
3
4  int main() {
5   int t ;
6   scanf("%d %d",&x,&y);
7   t = x - y ;
8   if (t > 0)
9    printf("x > y") ;
10  return 0 ;
11 }
```

1. LLVM Global Variable
2. **load/store** to access

```
…
2   6 @x = common global i32 0, align 4
    7 @y = common global i32 0, align 4

4  11 define i32 @main() #0 {
   12 entry:
   …
5  14 %t = alloca i32, align 4
   …
6  16 %call = call i32 (i8*, ...)*
        @__isoc99_scanf(…i32* @x,i32* @y)

7  17 %0 = load i32* @x, align 4
   18 %1 = load i32* @y, align 4
   19 %sub = sub nsw i32 %0 %1
   20 store i32 %sub, i32* %t, align 4

8  21 %2 = load i32* %t, align 4
   22 %cmp = icmp sgt i32 %2, 0
   23 br i1 %cmp, label %if.then,
                  label %if.end

9  24 if.then:
   25   %call1 = call i32 … @printf(…
   26   br label %if.end

10 27 if.end:
   28   ret i32 0
```

Local Variable Translation

```
1  #include <stdio.h>
2  int x, y ;
3
4  int main() {
5    int t ;
6    scanf("%d %d",&x,&y);
7    t = x - y ;
8    if (t > 0)
9      printf("x > y") ;
10   return 0 ;
11 }
```

1. Allocate stack space with **alloca**
2. **load/store** to access

```
…
2   6 @x = common global i32 0, align 4
    7 @y = common global i32 0, align 4

4  11 define i32 @main() #0 {
   12 entry:
…
5  14 %t = alloca i32, align 4
…
6  16 %call = call i32 (i8*, ...)*
        @__isoc99_scanf(…i32* @x,i32* @y)
7  17 %0 = load i32* @x, align 4
   18 %1 = load i32* @y, align 4
   19 %sub = sub nsw i32 %0 %1
   20 store i32 %sub, i32* %t, align 4
8  21 %2 = load i32* %t, align 4
   22 %cmp = icmp sgt i32 %2, 0
   23 br i1 %cmp, label %if.then,
                  label %if.end
9  24 if.then:
   25   %call1 = call i32 … @printf(…
   26   br label %if.end
10 27 if.end:
   28   ret i32 0
```

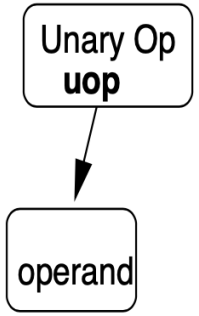
Local Variable Translations

- Using stack allocation to translate local variables is straightforward because it gets around the SSA requirement on temporary register.
- But stack memory is slower than temporary registers…
  - **Don’t worry, we will deal with this during code optimization!**
- LLVM IR parameter variables must satisfy SSA form like registers.
  - Cannot modify parameter varaibles in the function body
- How to handle non-constant parameter?
  - Create a temporary local variable on-stack for each parameter with **alloca**.
  - **store** the original parameter value to the temporary variable.
  - Treat the parameter varaibles as normal local variables in the follow up code translation.

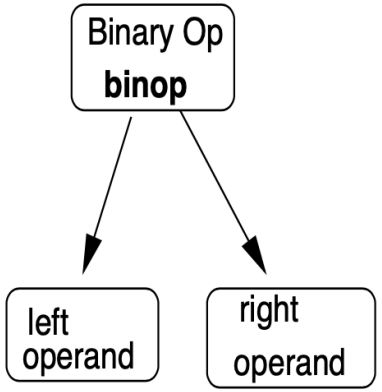
Expression Translation

- Generate IR for expressions with a depth first traversal of the AST that represents the expression.
- Typically generate subexpressions first before the main expression.
- Virtual registers may be required during expression processing to hold intermediate results.
- Use symbol table to track **LLVM alloca** and **global variable** associated with each program variables.
- Record the mapping between expressions to already generated LLVM IR values.

Translating Arithmetic Operators

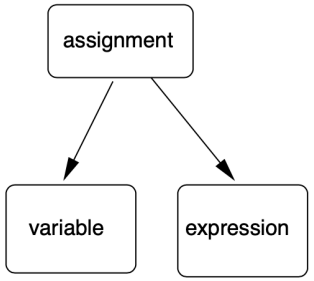



```
gencode(builder, operand);
llvm::Value* subValue = LLVMValueForExpr[operand];
llvm::Value* v = builder->CreateUop(uop, subValue);
LLVMValueForExpr[e] = v;
```




```
gencode(builder, left_operand);
gencode(builder, right_operand);
llvm::Value* lv = LLVMValueForExpr[left_operand];
llvm::Value* rv = LLVMValueForExpr[right_operand];
llvm::Value* v = builder->CreateBinop(binop, subValue);
LLVMValueForExpr[e] = v;
```

Translating Assignment Statements




```
gencode(builder, expression);
llvm::Value* var = getLLVMValueFromSymbolTable(variable.identifier);
llvm::Value* subValue = LLVMValueForExpr[expression];
llvm::Value* v = builder->CreateStore(subValue, var);
LLVMValueForExpr[e] = subValue;
```

- Store *subValue* to the table because the assignment expression in MiniC evaluates to its assigned expression value.
- Need to tweak to handle array expression assignment.
  - Use **getelementptr** instruction to indexing over an array variable

Expression Examples

```
1  #include <stdio.h>
2  int x, y ;
3
4  int main() {
5   int t ;
6   scanf("%d %d",&x,&y);
7   t = x - y ;
8   if (t > 0)
9    printf("x > y") ;
10  return 0 ;
11 }
```

```
…
2   6 @x = common global i32 0, align 4
    7 @y = common global i32 0, align 4

4  11 define i32 @main() #0 {
   12 entry:
…
5  14 %t = alloca i32, align 4
…
6  16 %call = call i32 (i8*, ...)*
        @__isoc99_scanf(…i32* @x,i32* @y)

7  17 %0 = load i32* @x, align 4
   18 %1 = load i32* @y, align 4
   19 %sub = sub nsw i32 %0 %1
   20 store i32 %sub, i32* %t, align 4

8  21 %2 = load i32* %t, align 4
   22 %cmp = icmp sgt i32 %2, 0
   23 br i1 %cmp, label %if.then,
                  label %if.end

9  24 if.then:
   25   %call1 = call i32 … @printf(…
   26   br label %if.end

10 27 if.end:
   28   ret i32 0
```

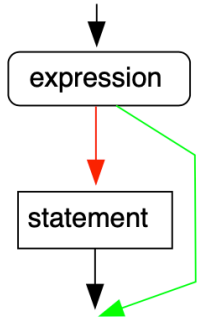
UNIVERSITY OF TORONTO

Basic Block and Control Flow Graph

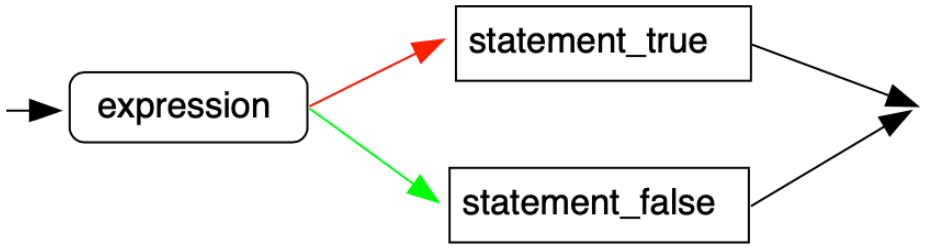
- A basic block is a program is a piece of the program with one entry and one exit. Examples: an expression ; a sequence of non-branching statements (e.g., assignments).
- Branching statements cause a change in the flow of control through a program. Examples: if, for, switch .
- The control flow graph for a program describes all possible flows of control between basic blocks in the program.
- In LLVM, an IR Builder can set the basic block to insert IRs via *SetInsertionPoint()*.

Control Flow Graph Example

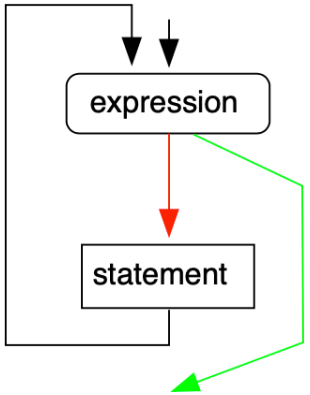
if (expression) statement



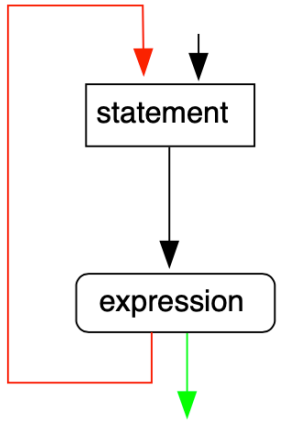
if (expression) stmt_true **else** stmt_false



while (expression) statement



do statement **while** (expression);



Comparison and Short-Circuit Evaluation

- The relational operators < , ≤ , = ,≠ , ≥ , > are usually treated as binary operators that produce a boolean result.
- C and C++ perform short-circuit evaluation on && and ||.
  - Only evaluate as much of a boolean expression as is required to determine its value. Generate branching code instead of arithmetic code.

`A && B`	if A then B else false
`A || B`	if A then true else B

- Why short-circuit evaluate matters?
  - `if (b == 0 || a / b > 1) x=0;`

Translation of Short-circuit Evaluation

- Create two additional basic blocks to represent two possible execution paths:

slow_path — the basic block to execute if we need to evaluate the second sub-expression

out — the exit basic block

- 1. Translate the first sub-expression
- 2. Generate a conditional branch on the sub-expression result value, if true (false) for && (||), jump to the *slow_path,* otherwise jump to *out*
- 3. Set the insertion point to the slow_path basic block.
- 4. Translate the second sub-expression
- 5. Set the insertion point to the out basic block.
- 6. Generate a PHI instruction to merge the results of two paths.

Short-circuit Evaluation Example

(b == 0) || (a / b > 1)

```
  %3 = load i32, i32* @b, align 4
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %10, label %5
5:
  %6 = load i32, i32* @a, align 4
  %7 = load i32, i32* @b, align 4
  %8 = sdiv i32 %6, %7
  %9 = icmp sgt i32 %8, 1
  br label %out
10:
  %11 = phi i1 [ true, %0 ], [ %9, %5 ]
```

(b > 0) && (a / b > 1)

```
 %3 = load i32, i32* @b, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %10
5:
  %6 = load i32, i32* @a, align 4
  %7 = load i32, i32* @b, align 4
  %8 = sdiv i32 %6, %7
  %9 = icmp sgt i32 %8, 1
  br label %10
10:
  %11 = phi i1 [ false, %0 ], [ %9, %5 ]
```

Translate If Statement

- Create three basic blocks, the entrance labels of them are:

then_bb: The basic block for then branch

else_bb: The basic block for else branch

out_bb: The basic block for follow up statements

- 1. Translate the conditional expression
- 2. Generate a conditional branch instruction to jump on the basic block based on the conditional expression value
- 3. Set the insertion point to then_bb and recursively translate the then branch statements.
- 4. Generate an unconditional branch instruction to jump to out_bb

Translate If Statement

- 5. Set the insertion point to else_bb and recursively translate the else branch statements.
- 6. Generate an unconditional branch instruction to jump to out_bb

- If there is no else branch, we can create one less basic block.
- Note that there might be nested control statements inside then/else block, so the branch instructions might not be in the then_bb/else_bb

Translate If Statement Example

```
if (x == 1)
  a = a + 1;
else {
  a = a + 2;
  b = a;
}
```

- 4 is the then_bb
- 7 is the else_bb
- 11 is the out_bb

```
  %2 = load i32, i32* @x, align 4
  %3 = icmp eq i32 %2, 1
  br i1 %3, label %4, label %7
4:                                                ; preds = %0
  %5 = load i32, i32* @a, align 4
  %6 = add nsw i32 %5, 1
  store i32 %6, i32* @a, align 4
  br label %11
7:                                                ; preds = %0
  %8 = load i32, i32* @a, align 4
  %9 = add nsw i32 %8, 2
  store i32 %9, i32* @a, align 4
  %10 = load i32, i32* @a, align 4
  store i32 %10, i32* @b, align 4
  br label %11
11:                                               ; preds = %7, %4
```

Translate If Statement Example (Nested)

```
if (x == 1)
  a = a + 1;
else {
  if (b == 0) a = a + 2;
  b = a;
}
```

- 7 and 10 are basic blocks for the inner if stmt

```
  %2 = load i32, i32* @x, align 4
  %3 = icmp eq i32 %2, 1
  br i1 %3, label %4, label %7
4:
  %5 = load i32, i32* @a, align 4
  %6 = add nsw i32 %5, 1
  store i32 %6, i32* @a, align 4
  br label %15
7:
  %8 = load i32, i32* @b, align 4
  %9 = icmp eq i32 %8, 0
  br i1 %9, label %10, label %13
```

```
10:
  %11 = load i32, i32* @a, align 4
  %12 = add nsw i32 %11, 2
  store i32 %12, i32* @a, align 4
  br label %13
13:
  %14 = load i32, i32* @a, align 4
  store i32 %14, i32* @b, align 4
  br label %15
15:
```

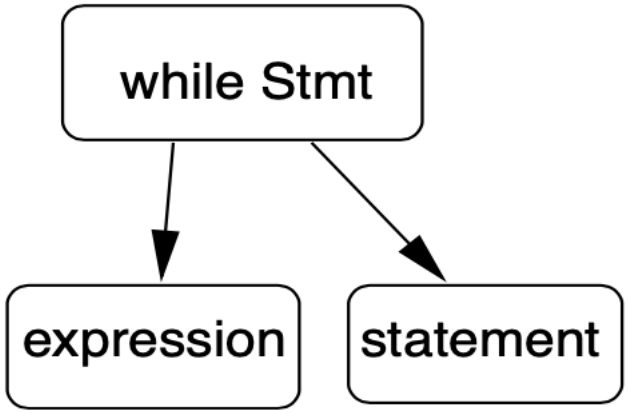
Translate While Loop

- Create three basic blocks, the entrance labels of them are:

cond_bb: The basic block for evaluating the condition

body_bb: The basic block for the while body

out_bb: The basic block for follow up statements



- 1. Set cond_bb as the insertion point.
- 2. Translate the conditional expression.
- 3. Generate a conditional branch instruction that jumps to body_bb or out_bb based on the value of the conditional expression
- 4. Set body_bb as the insertion point.
- 5. Translate the while body statement.
- 6. Generate an unconditional branch instruction to jump to cond_bb

Translate While Statement Example

```
while (x > 0) {
  a = a + 1;
  x = x / 2;
}
```

- 2 is the cond_bb
- 5 is the body_bb
- 10 is the out_bb

```
2:                                                ; preds = %5, %0
  %3 = load i32, i32* @x, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %10

5:                                                ; preds = %2
  %6 = load i32, i32* @a, align 4
  %7 = add nsw i32 %6, 1
  store i32 %7, i32* @a, align 4
  %8 = load i32, i32* @x, align 4
  %9 = sdiv i32 %8, 2
  store i32 %9, i32* @x, align 4
  br label %2

10:                                               ; preds = %2
```

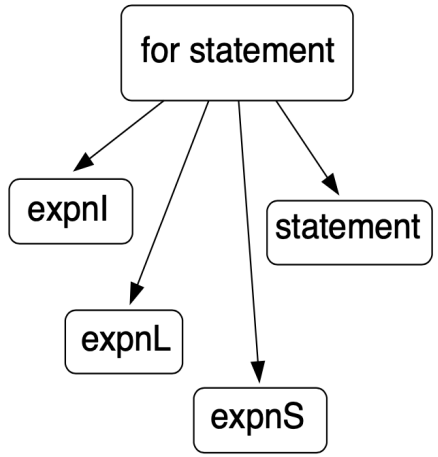
Translate For Loop

- Create three basic blocks, the entrance labels of them are:

cond_bb: The basic block for evaluating the condition

body_bb: The basic block for the for body

out_bb: The basic block for follow up statements



- 1. Translate the initilization expresion.
- 2. Set cond_bb as the insertion point.
- 3. Translate the conditional expression.
- 4. Generate a conditional branch instruction that jumps to body_bb or out_bb based on the value of the conditional expression
- 5. Set body_bb as the insertion point.

Translate For Loop

- 6. Translate the for body statement.
- 7. Translate the step expression.
- 8. Generate an unconditional jump to cond_bb

Translate For Statement Example

```
for (i = 0; i < x; i = i + 1)
  a = a + i;
```

- %2 is the register holding the stack address for the variable i
- 3 is the cond_bb
- 7 is the body_bb
- 14 is the out_bb

```
  store i32 0, i32* %2, align 4
  br label %3
3:                                                ; preds = %7, %0
  %4 = load i32, i32* %2, align 4
  %5 = load i32, i32* @x, align 4
  %6 = icmp slt i32 %4, %5
  br i1 %6, label %7, label %14
7:                                                ; preds = %3
  %8 = load i32, i32* @a, align 4
  %9 = load i32, i32* %2, align 4
  %10 = add nsw i32 %8, %9
  store i32 %10, i32* @a, align 4
  %12 = load i32, i32* %2, align 4
  %13 = add nsw i32 %12, 1
  store i32 %13, i32* %2, align 4
  br label %3
14:                                               ; preds = %3
```

Translate Break and Continue

- Many language have special control flow statements inside loops such as **break** and **continue**.
- To translate **break**, the code translator will record the exit_bb label inside a loop and generate an unconditional jump to exit_bb
- To translate **continue** for while loop, it will generate an unconditional jump back to cond_bb.
- To translate **continue** for for loop, the code translate need to create one additional basic block for the step expresion only saperated from the body_bb, and generate unconditional jump to this additional basic block.

Translate Function Declarations

- Perform semantic analysis on function header.
- Record formal parameter declarations in symbol/type table.
- Record type of return value for a function.
- For the function body, create an entrance basic block and emit prologue code to set up run time environment for the function.
- Process declarations for local variables.
- Emit epilogue code as required.
- Record the generated IR handle in the symbol table.

Prologue and Epilogue

- The prologue is typically emitted at the head of each routine.
  - Allocate resources
  - For native code, typically saving registers that might be modified
  - The typical prologue for MiniC in LLVM IR is to setup the temporary variable space at the stack for non-constant parameters.

- The epilogue is invoked to trigger a return from the function.
  - Deallocate resources
  - Return from the function
  - The typical epilogue for MiniC in LLVM IR is to return the default value defined by the language for the function if there is no return statement in the end.

Return and Terminator Caveat in LLVM IR

- Return statements should be directly translated into return instructions in LLVM IR.

- **Terminator Caveat**: Every basic block in LLVM can only have one terminator instruction at the end!
  - Return instruction, conditional branch, unconditional branch, etc.
  - If you attempt to insert more than one terminator instruction, the IRBuilder in LLVM will not fail but insert the instruction in the wrong place in llvm 11.0!
- You may want to skip some of the branch statement if the previous statement is already a terminator when translating if/for statements.

Argument-Passing Method

- Formal parameters in function definition are just place holders, replaced by arguments during a function call.
- *Parameter passing* - matching of arguments with formal parameters when a routine call occurs. But... what does A[i] mean: a name? a value? an address?
- Possible interpretations:
  - Call by Value: pass value of A[i]
  - Call by Reference: pass location of A[i] (i.e., &A[i] in C++)

Translate Call Expressions

- Call expresions should be translated into call instructions in LLVM IR.
- LLVM call instruction assume pass-by-value semantic. For call-by-value cases, the translation is straightforward.
- For call-by-reference cases, we should modify the function signature to convert the parameter into its corresponding pointer type and then pass the address pointer of the argument variable.

Call Example

```
void foo(int a, int* b) {
    *b = a;
}
int main() {
    foo(x, &y);
    return 0;
}
```

- passing value of x and the pointer location of y

```
define void @foo(i32 %0, i32* %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32*, align 8
  store i32 %0, i32* %3, align 4
  store i32* %1, i32** %4, align 8
  %5 = load i32, i32* %3, align 4
  %6 = load i32*, i32** %4, align 8
  store i32 %5, i32* %6, align 4
  ret void
}
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = load i32, i32* @x, align 4
  call void @foo(i32 %2, i32* @y)
  ret i32 0
}
```

Struct and Classes

- Struct is user defined aggregate types. It can be directly mapped to aggregate types in LLVM IR.
- A class-like construct is available in a number of languages.
- Classes are used to isolate some data and a collection of procedures and functions that operate on the data. i.e as an *abstract data type*
- Classes allow separate compilation in large software systems. This is the primary use of classes in C++ and modules in Modula-3
- Classes provide *information hiding* so that data internal to the class is not visible outside of the class. Information hiding is an important tool in allowing parts of a software system to be constructed and maintained separately.

Class Implementation Issues

- Initialization and finalization of class instances.
  - When should it happen?
  - Can it be guaranteed to happen? e.g., assignment of class variables, class instances embedded in other constructs.
  - What is the correct order of class initialization?
  - What is the correct order of class finalization?
- Code translation component may often
- Nested class declarations.

Kinds of Classes

- The difficulty in implementing a class depends on the kind of class, single instance, multiple instance or template multiple instance.
- Single Instance Classes There is exactly one instance of the classes internal data and routines. Examples Static Classes in Java
- Multiple Instances Classes There may be multiple instances of the classes internal data but only one instance of the class member functions. Examples Classes in C++ , Classes in Java.
- Template Multiple Instance Classes The class is parameterized by constant and type information (templates) which can be used to specialize each instance of the class.. Usually requires multiple copies of the classes data *and member functions*. Examples: parameterized Classes in C++ and Java

Generic Class

class {

Import declarations

Export declarations

Internal constants, types and variables

Exported constants and types

Internal functions and procedures

Exported functions and procedures

}

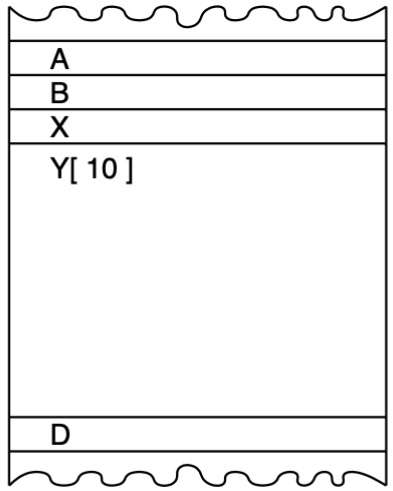
Implementing Class Imports and Exports

- Managing imports and exports is a symbol table management issue.
- The body of a class is a separate scope like the body of a routine.
- Create a type table entry for describing classes even if classes aren't treated like types in the language. The type table entry will be similar to the ones used for records
- The names exported by a class are linked together in the symbol table in a list that is pointed to from the classes type table entry. This list is used outside of the class to resolve references to exported items.
- Symbol table entries for items imported into a class can be copied into the symbol table of the class.

Single Instance Class Data

- Treat class like a struct variable.
- Allocate storage in enclosing major scope, i.e., class data is handled var P, Q .. . like minor scope data.
- Member functions are children of enclosing major scope
- Control access to class data via symbol table lookup management

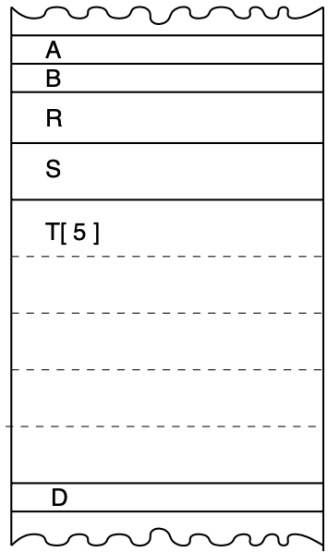
```
var A, B: integer
static class C1 {
    var X, Y[10]: integer
    func foo(…) { … }
}
var D : boolean
```



Multiple Instance Classes

- Treat class like a struct type
- Allocate storage separately for each class instance variable.
- For space efficiency, generate one instance of each member function.
- Give member functions access to class instance data via an extra parameter or a hidden pointer for the class (i.e., this pointer).

```
var A, B: integer
class C2 {
    var X: integer
    func foo(…) { … }
}
var R, S, T[5] : C2
var D : boolean
```



Multiple Instance Template Classes

- Treat each class template instance as a distinct struct type
- **Option 1**: Expand the class definition similar to C macros.
  - This is the C++ approach.
  - Advantage: straightforward
  - Disadvantage: Put more weight on preprocessors. Slow compilation. Hard to provide good diagnose error message for semantic analysis.
- **Option 2**: Use symbol table to track and maintain different instances. Build a type system around the generic template type parameters.
  - Advanrage: more elegant solutions and better diagnose message.
  - Disadvantage: need a complicated type system

Q/A?