Week 9: Elements of Compiler Design

Translation to Intermediate Code

Prof. Chris GauthierDickey
University of Denver

Generating LLVM IR

Generating LLVM IR

- We have made it to what is very near code generation
- From the AST, using Frame and our environment information, we will emit LLVM Intermediate Representation (IR) code
- This code can be compiled through the llvm compiler, which is freely available (and already lives on the Macs)

Why LLVM?

- We only have 10 weeks. We could go to a custom IR, and then optimize it, and then do instruction selection, but that's a ton of work in 2 weeks!
  - However, the LLVM IR is now widely used, and from it you can generate code for almost every platform
  - The LLVM IR was designed to be similar to an automatic grammar generator — if you generate the IR, you can compile and optimize for many platforms

The High Level

- LLVM divides code chunks into modules. A module is a translation unit (i.e., file)
- Each module has functions, global variables, and its own set of symbol table entries (note, LLVM handles the symbol tables automatically)
- Each module also has parts that help with binary code generation

Targets

```
; target data layout for Mac, change to m:w instead for Windows
target datalayout = “e-m:o-i64:64-f80:128-n8:16:32:64-S128”
```

- The data layout specifies information about the architecture, in particular endianness, name mangling, floating point/integer sizes, and stack alignment
- The string consists of values separated by ‘-’
- e - little Endian, E - big Endian
- m:o - Mach-O name mangling, m:w - Windows COFF mangling
- i64:64 indicates the natural word size of the machine, 64 bits in this case, with a 64 bit alignment
- f:80:128 - floating point values of 80 and 128 bits. Note this means you can make floating point values of those sizes, 32-bit and 64-bit are always supported on all targets
- n:8:16:32:64 - valid integer sizes
- S128 - natural stack alignment, in bits (so in this case 16 bytes)

Targets

```
; target triple for Mac
target triple = “x86_64-apple-macosx10.10.0”
```

- The triple tells it how to generate machine code—for which architecture in essence
- This triple is different for Mac and Windows (above is for the Mac)
- For testing on Windows, you can just leave this out, llvm will generate the default kind automatically, but warn you
- *Please do not include the target triple, or use the Mac target triple (if you're on a Mac) because I grade on a Mac!*

Types

- Throughout LLVM, you'll see that types are specified
  - *LLVM is statically, strongly typed!*
- This makes generating some kinds of code, like for a static link, a bit of a hassle
- But, the types allow for better optimizations in the end!

First Class Types

- Values of first class types *can only be produced by instructions*, these include:

Integer Types	iN (where N is the number of bits)	i8, i16, i32, i64, i1942652 (haha)	
Floating point types	half (16 bits) float (32 bits) double (64 bits)	fp128 (128 bits, 112-bit mantissa) x86_fp80 (80 bits)	ppc_fp128 (128-bit, 2 64-bit floating points)
Pointer Types	<type>*	i32*, i64*, float*	[4 x i32]* i32 (i32 *) *
Vector Types	< <# elements> x <element type> >	<4 x i32> <4 x float> <2 x i64>	
Label	Labels are just labels, but they're typed	mylabel:	

Other Types

Functions	<return type> (<parameter list>)	i32 (i32) float (i16, i32 *) i32 (i8*, …)	This last example is a varargs call, such as printf
Arrays	[<# elements> x <type>]	[40 x i32] [2 x [3 x [4 x i16]]]	
Structures	%T1 = type { <type list> } %T1 = type ‘<‘ { <type list> } ‘>’		%String = type { i64, i8* } %triple = type <{i32, i32, i32}>
Void	No value and no size type	void	

Basic Blocks

- LLVM code is divided into basic blocks
- Basic blocks begin with an (optional) Label and end with a terminating instruction (*ret, br, switch, indirectbr, invoke, resume, unreachable*)
- If you do not assign a label, one will be assigned for you, and these all look like incrementing numerical temporaries (e.g., %1, %2, …)
- Basic blocks are important because we use them to jump around our code via their labels
- In addition, function bodies are a list of basic blocks

More on Basic Blocks

- Why even bother?
  - Basic blocks may be reordered for optimization
  - Functions simply contain a list of these without needed to generate a complicated graph with branching
  - You don’t have to do anything special to define them, just use labels as necessary, and when you translate, you’ll end up with terminator instructions

Identifiers

- LLVM have two basic types of identifiers, *global* and *local*.
  - Global identifiers are exactly how they sound: they're global variables accessible by all emitted code
  - Local identifiers are only accessible within the body of a function (so you can't use a local identifier from one function in a different function)

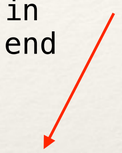
Identifiers

```
let
  ni x is 42
in
end
```

- Global identifiers begin with @

Identifiers

```
let
  ni x is 42
in
end
```

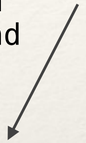


```
@x = global i64 42, align 8
```

- Global identifiers begin with @

Identifiers

```
let
  ni x is 42
in
end
```



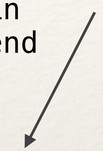
```
@x = global i64 42, align 8
```

- Global identifiers begin with @
- Local identifiers begin with %

Identifiers

- Global identifiers begin with @
- Local identifiers begin with %

```
let
  ni x is 42
in
end
```



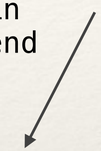
```
@x = global i64 42, align 8
```

```
let
  ni x is 42
in
  let
    neewom addStuff(int y) as int is
      let
        ni z is 3
      in
        x + y + z
  in
    addStuff(6)
end
```

Identifiers

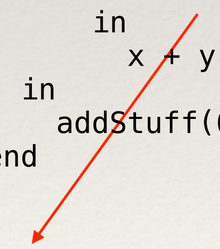
- Global identifiers begin with @
- Local identifiers begin with %

```
let
  ni x is 42
in
end
```



```
@x = global i64 42, align 8
```

```
let
  ni x is 42
in
  let
    neewom addStuff(int y) as int is
      let
        ni z is 3
      in
        x + y + z
  in
    addStuff(6)
end
```



```
%z = alloca i64, align 8
store i64 3, i64* %z
```

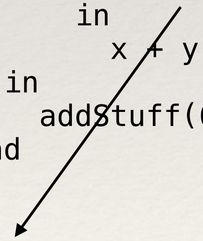
Identifiers

- Notice we demonstrate two things: we can load an integer constant into a global easily

```
let
  ni x is 42
in
end
```

```
@x = global i64 42, align 8
```

```
let
  ni x is 42
in
  let
    neewom addStuff(int y) as int is
      let
        ni z is 3
      in
        x + y + z
  in
    addStuff(6)
end
```



```
%z = alloca i64, align 8
store i64 3, i64* %z
```

Identifiers

- Notice we demonstrate two things: we can load an integer constant into a global easily
- A local may be on the stack, and if it is, it's an address with a pointer type

```
let
  ni x is 42
in
end
```

```
@x = global i64 42, align 8
```

```
let
  ni x is 42
in
  let
    neewom addStuff(int y) as int is
      let
        ni z is 3
      in
        x + y + z
  in
    addStuff(6)
end
```

```
%z = alloca i64, align 8
store i64 3, i64* %z
```

Identifiers

- In addition, notice the ‘align 8’ specification
  - While integer types specify the number of bits, the align specification is in bytes
  - Thus, 64 bits is naturally aligned on 8 byte boundaries

```
let
  ni x is 42
in
end
```

```
@x = global i64 42, align 8
```

```
let
  ni x is 42
in
  let
    neewom addStuff(int y) as int is
      let
        ni z is 3
      in
        x + y + z
  in
    addStuff(6)
end
```

```
%z = alloca i64, align 8
store i64 3, i64* %z
```

Stack vs Register

- Global variables are *always* pointers.
  - For example, @x global i64 42, align 8
    - This stores 42 in the global variable x
    - But what type is x? It's an i64 *, not an i64
    - You have to retrieve the value with a load instruction

Stack vs Register

- When we declare locals, we use % preceding the name
- However, we can declare two types of local variables, stack or frame resident
  - %x = alloca i64, align 8
    - This produces a value in the stack frame, we can store and load to and from it
  - %x = add i64 42, 0
    - This produces a *temporary* or register allocated variable, assigning 42 to it (because we just added two ints)

Stack vs Register

- If a variable escapes, you must place it on the stack
- If it doesn't, you may place it in a register
  - However, we have a major restriction, the LLVM IR uses Static Single Assignment (SSA) form
  - This means that every name can be used only once

Static Single Assignment

- Consider the following code
  - Obviously, the second declaration of x shadows the first (obvious to a human)

```
let
  ni x is 42
  ni x is 23
  ni y is 0
in
  now y is x
end
```

Static Single Assignment

- Now consider if we had to use unique names
  - The compiler can easily optimize away the first x1

```
let
  ni x is 42
  ni x is 23
  ni y is 0
in
  now y is x
end
```

```
let
  ni x1 is 42
  ni x2 is 23
  ni y1 is 0
in
  now y1 is x2
end
```

A more complicated example

- Ignoring the somewhat verbose language description, we have have to rename the variables so that the final value of y is the result of the branch
- Static single assignment only allows one assignment to y, so if we rename, we run into issues:

```
let
  ni x is 42
in
  now x is x - 7;
  let
    ni y is 3
    ni w is 0
  in
    if x <= 35 then
      now y is x - 3
    else
      now y is x - 10
    end;
    now w is y * 2;
    w
  end
end
```

A more complicated example

- First, let's rename x as x1, and x2 (remember we're converting to LLVM IR, so while this program isn't valid in Ni, the equivalent IR would be fine)

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y is 3
    ni w is 0
  in
    if x2 <= 35 then
      now y is x2 - 3
    else
      now y is x2 - 10
    end;
    now w is y * 2;
    w
  end
end
```

A more complicated example

- First, let's rename x as x1, and x2 (remember we're converting to LLVM IR, so while this program isn't valid in Ni, the equivalent IR would be fine)
- Now let's rename w

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      now y is x2 - 3
    else
      now y is x2 - 10
    end;
    now w2 is y * 2;
    w2
  end
end
```

A more complicated example

- First, let's rename x as x1, and x2 (remember we're converting to LLVM IR, so while this program isn't valid in Ni, the equivalent IR would be fine)
- Now let's rename w
- Finally, rename y

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      now y2 is x2 - 3
    else
      now y3 is x2 - 10
    end;
    now w2 is y? * 2;
    w2
  end
end
```

A more complicated example

- First, let's rename x as x1, and x2 (remember we're converting to LLVM IR, so while this program isn't valid in Ni, the equivalent IR would be fine)
- Now let's rename w
- Finally, rename y
  - We're stuck! Should we put y2 or y3 on this line?

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      now y2 is x2 - 3
    else
      now y3 is x2 - 10
    end;
    now w2 is y? * 2;
    w2
  end
end
```

Introducing ϕ

- The ϕ instruction was designed to choose at that time which branch we just came from
- ϕ was so-named by IBM in the 80s when they developed the idea of SSA to stand for ‘phoney’, haha

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      now y2 is x2 - 3
    else
      now y3 is x2 - 10
    end;
    now w2 is y? * 2;
    w2
  end
end
```

Introducing ϕ

- The ϕ instruction was designed to choose at that time which branch we just came from
- ϕ was so-named by IBM in the 80s when they developed the idea of SSA to stand for ‘phoney’, haha
- We insert ϕ to select the right name and assign it to a new name, which can then be used

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      now y2 is x2 - 3
    else
      now y3 is x2 - 10
    end;
    now y4 is ϕ (y2, y3)
    now w2 is y4 * 2;
    w2
  end
end
```

Translating to LLVM IR

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      y2 is x2 - 3
    else
      y3 is x2 - 10
    end;
    now y4 is Φ (y2, y3)
    now w2 is y4 * 2;
    w2
  end
end
```

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
```

We begin by the simple translation of the variables

Translating to LLVM IR

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      y2 is x2 - 3
    else
      y3 is x2 - 10
    end;
    now y4 is Φ (y2, y3)
    now w2 is y4 * 2;
    w2
  end
end
```

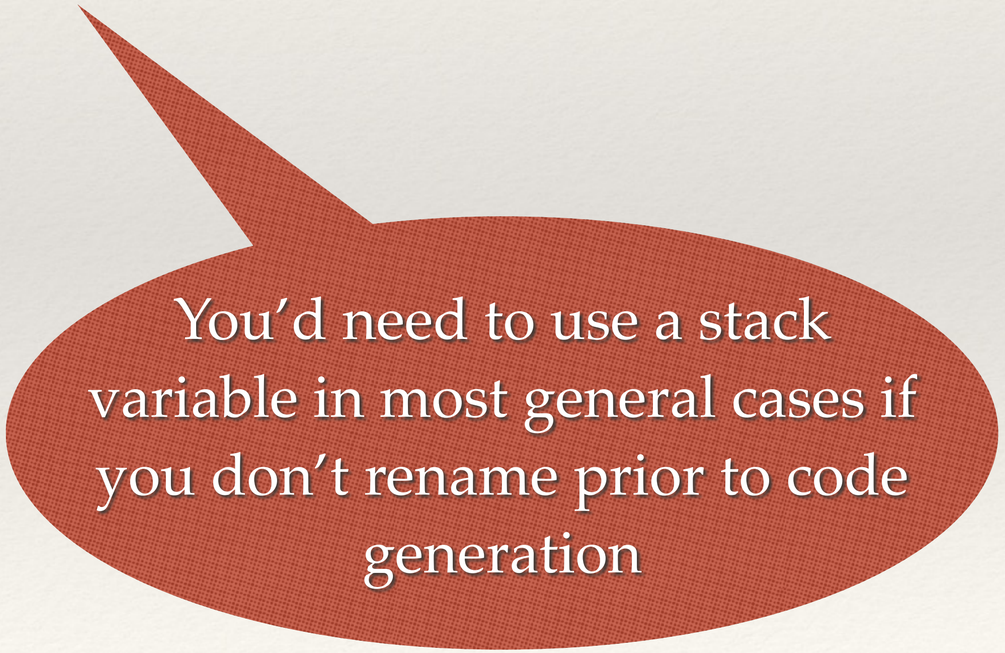
```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
```

Note, we're just using register values, which assumes you renamed everything already

Translating to LLVM IR

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      y2 is x2 - 3
    else
      y3 is x2 - 10
    end;
    now y4 is Φ (y2, y3)
    now w2 is y4 * 2;
    w2
  end
end
```

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
```



Translating a Conditional Branch

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      y2 is x2 - 3
    else
      y3 is x2 - 10
    end;
    now y4 is φ(y2, y3)
    now w2 is y4 * 2;
```

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
; if-then test
%t1 = icmp sle i64 %x2, 35
br i1 %t1, label %L1, label %L2
L1:
  %y2 = sub i64 %x2, 3
  br label L3
L2:
  %y3 = sub i64 %x2, 10
  br label L3
L3:
  %y4 = phi i64 [ %y2, L1 ], [ %y3, L2 ]
  %w2 = mul i64 %y4, 2
```

Next we encode the loop using LLVM's branch

Translating a Conditional Branch

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      y2 is x2 - 3
    else
      y3 is x2 - 10
    end;
    now y4 is φ(y2, y3)
    now w2 is y4 * 2;
  end
```

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
; if-then test
%t1 = icmp sle i64 %x2, 35
br i1 %t1, label %L1, label %L2
L1:
  %y2 = sub i64 %x2, 3
  br label L3
L2:
  %y3 = sub i64 %x2, 10
  br label L3
L3:
  %y4 = phi i64 [ %y2, L1 ], [ %y3, L2 ]
  %w2 = mul i64 %y4, 2
```

This requires using icmp first, followed by a br i1 test

Translating a Conditional Branch

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      y2 is x2 - 3
    else
      y3 is x2 - 10
    end;
    now w2 is y4 * 2
  end
end
```

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
; if-then test
%t1 = icmp sle i64 %x2, 35
br i1 %t1, label %L1, label %L2
L1:
  %y2 = sub i64 %x2, 3
  br label L3
L2:
  %y3 = sub i64 %x2, 10
  br label L3
L3:
  %y4 = phi i64 [ %y2, L1 ], [ %y3, L2 ]
  %w2 = mul i64 %y4, 2
```

Notice that for the conditional branch, we must specify labels for where we branch to on true and false

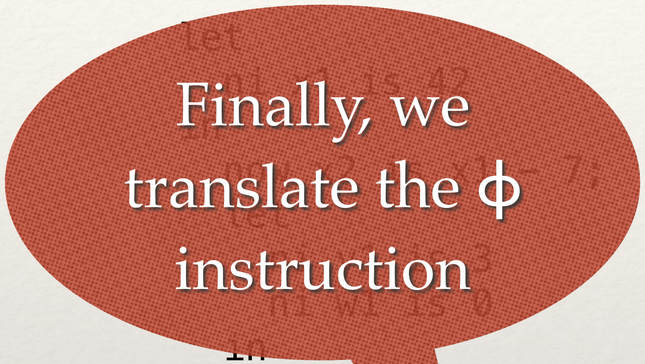
Translating an Unconditional Branch

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      y2 is x2 - 3
    else
      y3 is x2 - 10
    end
    now y4 is phi(y2, y3)
    now w2 = y4 * 2;
  end
end
```

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
; if-then test
%t1 = icmp sle i64 %x2, 35
br i1 %t1, label %L1, label %L2
L1:
  %y2 = sub i64 %x2, 3
  br label L3
L2:
  %y3 = sub i64 %x2, 10
  br label L3
L3:
  %y4 = phi i64 [ %y2, L1 ], [ %y3, L2 ]
  %w2 = mul i64 %y4, 2
```

Furthermore, notice that we have an unconditional branch to exit our different branches. These start with “br label”

Using ϕ




```
    if x2 = 35 then
      y2 is x2 - 3
    else
      y3 is x2 - 10
    end;
    now y4 is ϕ (y2, y3)
    now w2 is y4 * 2;
    w2
  end
end
```

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
; if test
%t1 = icmp sle i64 %x2, 35
br i1 %t1, label %L1, label %L2
L1:
  %y2 = sub i64 %x2, 3
  br label L3
L2:
  %y3 = sub i64 %x2, 10
  br label L3
L3:
  %y4 = phi i64 [ %y2, L1 ], [ %y3, L2 ]
  %w2 = mul i64 %y4, 2
```

Using ϕ

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      y2 is x2 - 3
    else
      y3 is x2 - 10
    end;
    now y4 is ϕ (y2, y3)
    now w2 is y4 * 2;
    w2
  end
end
```

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
; if test
%t1 = icmp sle i64
br i1 %t1, label %L
L1:
  %y2 = sub i64 %x2,
  br label L3
L2:
  %y3 = sub i64
  br label L3
L3:
  %y4 = phi i64 [ %y2, L1 ], [ %y3, L2 ]
  %w2 = mul i64 %y4, 2
```

This becomes "phi", and has a list of first class typed values and Labels

Using ϕ

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      y2 is x2 - 3
    else
      y3 is x2 - 10
    end;
    now y4 is ϕ (y2, y3)
    now w2 is y4 * 2;
    w2
  end
end
```

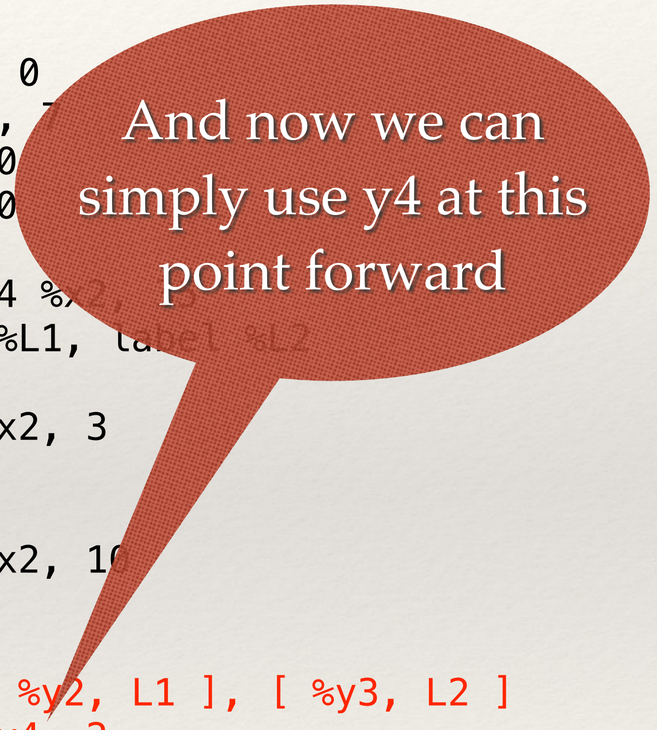
```
%x1 = add i64 42
%x2 = sub i64
%y1 = add i6
%w1 = add
; if test
%t1 = icmp
br i1 %t1,
L1:
  %y2 = sub i
  br label L3
L2:
  %y3 = sub i64 %x2, 10
  br label L3
L3:
  %y4 = phi i64 [ %y2, L1 ], [ %y3, L2 ]
  %w2 = mul i64 %y4, 2
```

This list can be longer than two tests, and each pair is the value to copy for assignment, and the label of the predecessor block

Using ϕ

```
let
  ni x1 is 42
in
  now x2 is x1 - 7;
  let
    ni y1 is 3
    ni w1 is 0
  in
    if x2 <= 35 then
      y2 is x2 - 3
    else
      y3 is x2 - 10
    end;
    now y4 is ϕ (y2, y3)
    now w2 is y4 * 2;
    w2
  end
end
```

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
; if test
%t1 = icmp sle i64 %x2, 35
br i1 %t1, label %L1, label %L2
L1:
  %y2 = sub i64 %x2, 3
  br label L3
L2:
  %y3 = sub i64 %x2, 10
  br label L3
L3:
  %y4 = phi i64 [ %y2, L1 ], [ %y3, L2 ]
  %w2 = mul i64 %y4, 2
```



Translating Binary Operators

- We've already seen how to use some binary operators
- The integer arithmetic ones are as expected: add, sub, mul, sdiv
- These take a type, and two operands which must be first class types

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
; if test
%t1 = icmp sle i64 %x2, 35
br i1 %t1, label %L1, label %L2
L1:
  %y2 = sub i64 %x2, 3
  br label L3
L2:
  %y3 = sub i64 %x2, 10
  br label L3
L3:
  %y4 = phi i64 [ %y2, L1 ], [ %y3, L2 ]
  %w2 = mul i64 %y4, 2
```

Translating Binary Operators

- Note that as a computation, they must be assigned to a local of some sort
- This can then be stored in a stack value if needed

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
; if test
%t1 = icmp sle i64 %x2, 35
br i1 %t1, label %L1, label %L2
L1:
  %y2 = sub i64 %x2, 3
  br label L3
L2:
  %y3 = sub i64 %x2, 10
  br label L3
L3:
  %y4 = phi i64 [ %y2, L1 ], [ %y3, L2 ]
  %w2 = mul i64 %y4, 2
```

Translating Binary Operators

- We also have bitwise binary operators:
  - shl (for logical shift left)
  - lshr (for logical shift right)
  - ashr (for arithmetic shift right, which keeps the signed value)
  - and (logical and)
  - or (logical or)
  - xor (logical xor)

```
%x1 = add i64 42, 0
%x2 = sub i64 %x1, 7
%y1 = add i64 3, 0
%w1 = add i64 3, 0
; if test
%t1 = icmp sle i64 %x2, 35
br i1 %t1, label %L1, label %L2
L1:
  %y2 = sub i64 %x2, 3
  br label L3
L2:
  %y3 = sub i64 %x2, 10
  br label L3
L3:
  %y4 = phi i64 [ %y2, L1 ], [ %y3, L2 ]
  %w2 = mul i64 %y4, 2
```

Translating Functions

- Functions are fairly easy to translate to LLVM IR because they follow a similar definition as to how they are displayed in Ni.
- Here we have a simple Ni language function

```
// add one to x
neewom incr(int x) as int is
  x + 1
```

Translating Functions

- To create a function, we use *define*

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- To create a function, we use *define*
- For Ni, all functions will also have the *nounwind* attribute—this means they do not throw exceptions

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- As with Ni, we declare the return type

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- As with Ni, we declare the return type
- And the name of the function is a global, while the formal parameter is a local

```
// add one to x
neewom incr(int x) as int is
  x + 1
```

```
// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- Every function in LLVM has an entry block, whether you label it or not

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- Every function in LLVM has an entry block, whether you label it or not
- Unlike other labels, *you cannot jump to the first block in a function*

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- This is simply because on entry to a function, LLVM must emit certain code—if you need to jump to the entry, what you really need is a *call*, i.e., a function call

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- Note that the first thing we do is copy the formal parameters into temporaries—this is because we have no idea if the function will be recursive
- Also note that formal parameters look like they're on the stack by default

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- We allocate space for %x

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- We allocate space for %x
- Then we copy it to our local stack space

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- We allocate space for %x
- Then we copy it to our local stack space
- Then we move it to a register so we can operate on it

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- We allocate space for %x
- Then we copy it to our local stack space
- Then we move it to a register so we can operate on it
- We operate on it, store the result

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Translating Functions

- And return it using a *ret* instruction

```
// add one to x
neewom incr(int x) as int is
  x + 1

// add one to x
define i64 @incr(i64 %x) nounwind {
entry:
  %1 = alloca i64, align 8
  store i64 %x, %1 align 8
  %2 = load i64* %1, align 8
  %3 = add nsw i64 %2, 1
  ret i64 %3
}
```

Recursive Functions

```
// add one to x
neewom sum(int n) as int is
  if n == 0 then
    0
  else
    1 + sum(n - 1)
  end
```

- Recursive functions differ little from regular functions, except that you have a function call

Recursive Functions

```
// add one to x
neewom sum(int n) as int is
  if n == 0 then
    0
  else
    1 + sum(n - 1)
  end
```

```
define i64 @sum(i64 %n) nounwind {
entry:
  %0 = icmp eq i64 %n, 0
  br i1 %0, label %exit, label %recurse

recurse:
  %1 = sub i64 %n, 1
  %2 = call i64 @sum(i64 %1)
  %3 = add i64 %2, %n
  ret i64 %3

exit:
  ret i64 0
}
```

We define a label for the entry into our function

- Recursive functions are not much different from regular functions, except that you have a function call

Recursive Functions

```
// add one to x
neewom sum(int n) as int is
  if n == 0 then
    0
  else
    1 + sum(n - 1)
  end
```

```
define i64 @sum(i64 %n) nounwind {
entry:
  %0 = icmp eq i64 %n, 0
  br i1 %0, label %exit, label %recurse

recurse:
  %1 = sub i64 %n, 1
  %2 = call i64 @sum(i64 %1)
  %3 = add i64 %2, %n
  ret i64 %3

exit:
  ret i64 0
}
```

We start with the branch, if n is 0, we conditionally branch

- Recurs[illegible]m regular functions, except that you have a functi[illegible] call

Recursive Functions

```
// add one to x
neewom sum(int n) as int is
  if n == 0 then
    0
  else
    1 + sum(n - 1)
  end
```

```
define i64 @sum(i64 %n) nounwind {
entry:
  %0 = icmp eq i64 %n, 0
  br i1 %0, label %exit, label %recurse

recurse:
  %1 = sub i64 %n, 1
  %2 = call i64 @sum(i64 %1)
  %3 = add i64 %2, %n
  ret i64 %3

exit:
  ret i64 0
}
```

We have two options to branch, to the exit (because n is 0) or to the recursive part, which we label

- Recurs[illegible]m regular functions, except that you have a functi[illegible]

Recursive Functions

```
// add one to x
neewom sum(int n) as int is
  if n == 0 then
    0
  else
    1 + sum(n - 1)
  end
```

```
define i64 @sum(i64 %n) nounwind {
entry:
  %0 = icmp eq i64 %n, 0
  br i1 %0, label %exit, label %recurse

recurse:
  %1 = sub i64 %n, 1
  %2 = call i64 @sum(i64 %1)
  %3 = add i64 %2, %n
  ret i64 %3

exit:
  ret i64 0
}
```

If n is not 0, we subtract 1 from n, and call sum(n - 1). We then add 1 and return.

- Recurs[illegible] m regular functions, except that you have a functio[illegible]

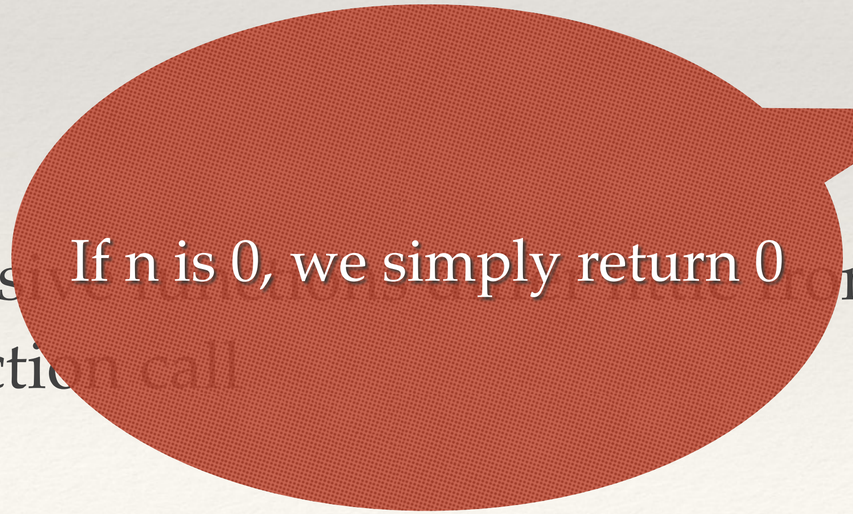
Recursive Functions

```
// add one to x
neewom sum(int n) as int is
  if n == 0 then
    0
  else
    1 + sum(n - 1)
  end
```

```
define i64 @sum(i64 %n) nounwind {
entry:
  %0 = icmp eq i64 %n, 0
  br i1 %0, label %exit, label %recurse

recurse:
  %1 = sub i64 %n, 1
  %2 = call i64 @sum(i64 %1)
  %3 = add i64 %2, %n
  ret i64 %3

exit:
  ret i64 0
}
```



- Recurs m regular functions, except that you have a functi n call

Recursive Functions

```
// add one to x
neewom sum(int n) as int is
  if n == 0 then
    0
  else
    1 + sum(n - 1)
  end
```

```
define i64 @sum(i64 %n) nounwind {
entry:
  %0 = icmp eq i64 %n, 0
  br i1 %0, label %exit, label %recurse

recurse:
  %1 = sub i64 %n, 1
  %2 = call i64 @sum(i64 %1)
  %3 = add i64 %2, %n
  ret i64 %3

exit:
  ret i64 0
}
```

Note that we didn't put this into tail-recursive form, which we could have

- Recurs[illegible] m regular functions, except that you have a functio[illegible]

Recursive Functions

```
// add one to x
neewom sum(int n) as int is
  if n == 0 then
    0
  else
    1 + sum(n - 1)
  end
```

```
define i64 @sum(i64 %n) nounwind {
entry:
  %0 = icmp eq i64 %n, 0
  br i1 %0, label %exit, label %recurse

recurse:
  %1 = sub i64 %n, 1
  %2 = call i64 @sum(i64 %1)
  %3 = add i64 %2, %n
  ret i64 %3

exit:
  ret i64 0
}
```

- Recursive functions differ little from regular functions, except that you have a function call

References

- http://llvm.org/docs/LangRef.html
- http://www.aosabook.org/en/llvm.html
- http://llvm.lyngvig.org/Articles/Mapping-High-Level-Constructs-to-LLVM-IR