CS 160 Compilers

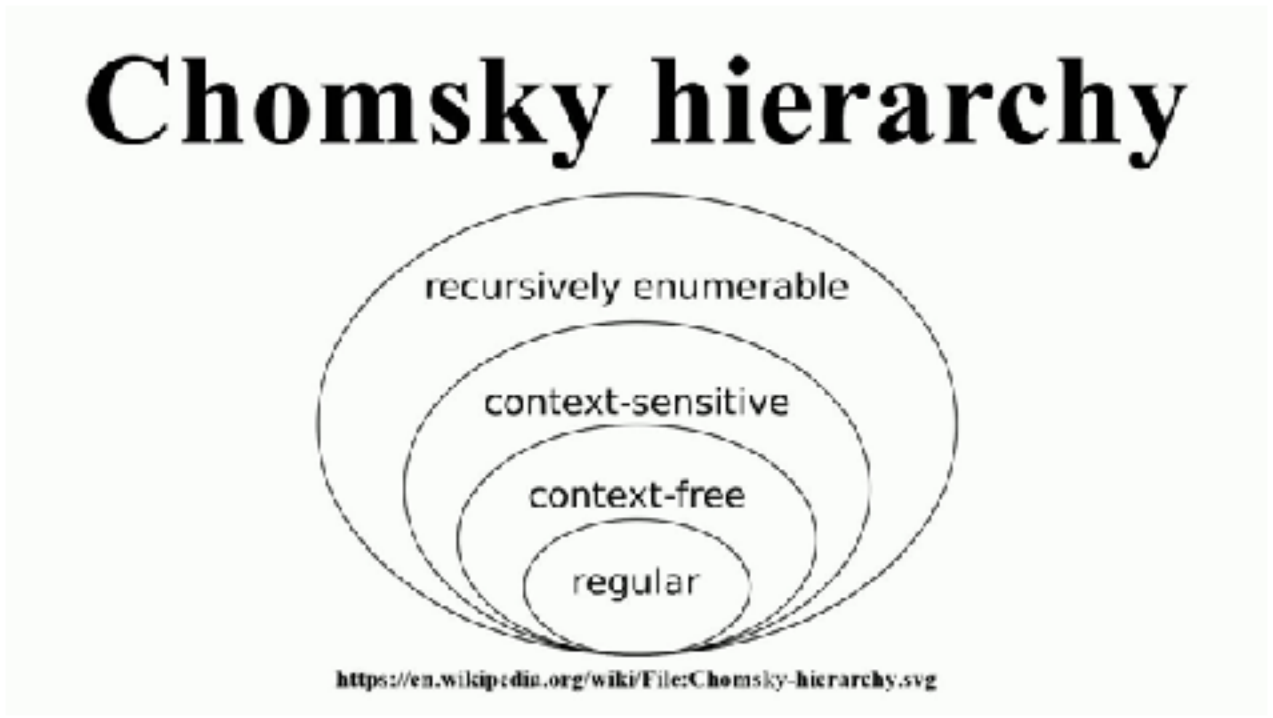
Lecture 14.2: Code Generation & llvmLite

Yu Feng

Spring 2023

A typical flow of a compiler

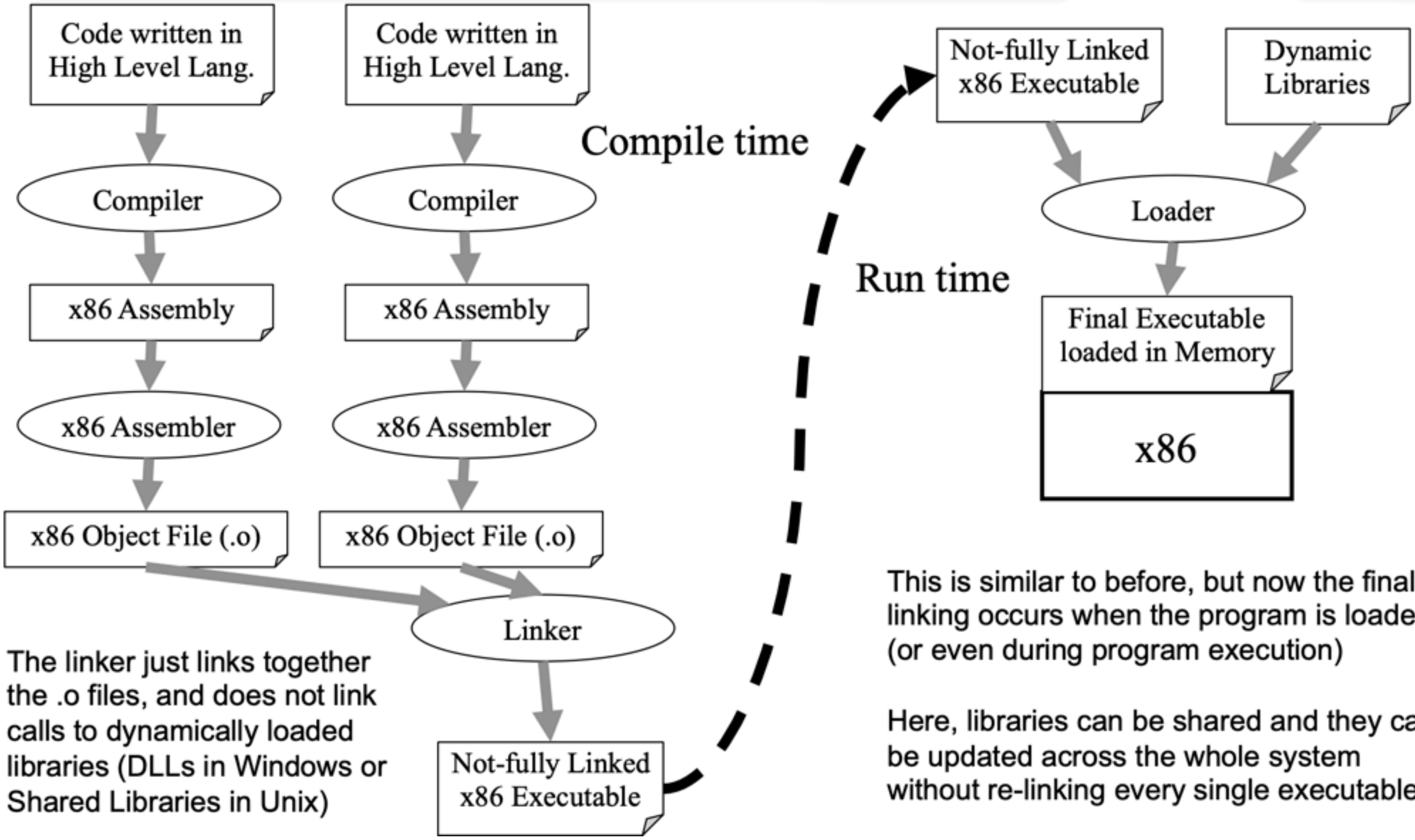
$$\begin{tikzcd}
{} \arrow[r, "\textit{Program}"] & \boxed{\text{Lexical Analysis}} \arrow[r, "\textit{Tokens}", "\textit{Regular}"'] & \boxed{\text{Syntax Analysis}} \arrow[r, "\textit{ASTs}", "\textit{CFG}"'] & \boxed{\text{Semantic Analysis}} \arrow[r, "\textit{IRs}", "\textit{???}"'] & \boxed{\text{Code Generation}} \arrow[r, "\textit{Executable}"] & {}
\end{tikzcd}$$



Code Generation

- To generate actual code that can run on a processor (such as gcc) or on a virtual machine (such as javac) we need to understand what code for each of these machines looks like.
- Rather than worry about the exact syntax of a given assembly language, we instead use a type of pseudo-assembly that is close to the underlying machine.
- In this class, we need to worry about 2 different types of code
    - Stack based code: Similar to the Java Virtual Machine
    - Register-based code: Similar to most processors (x86, Sparc, ARM)

x86 C Compiler

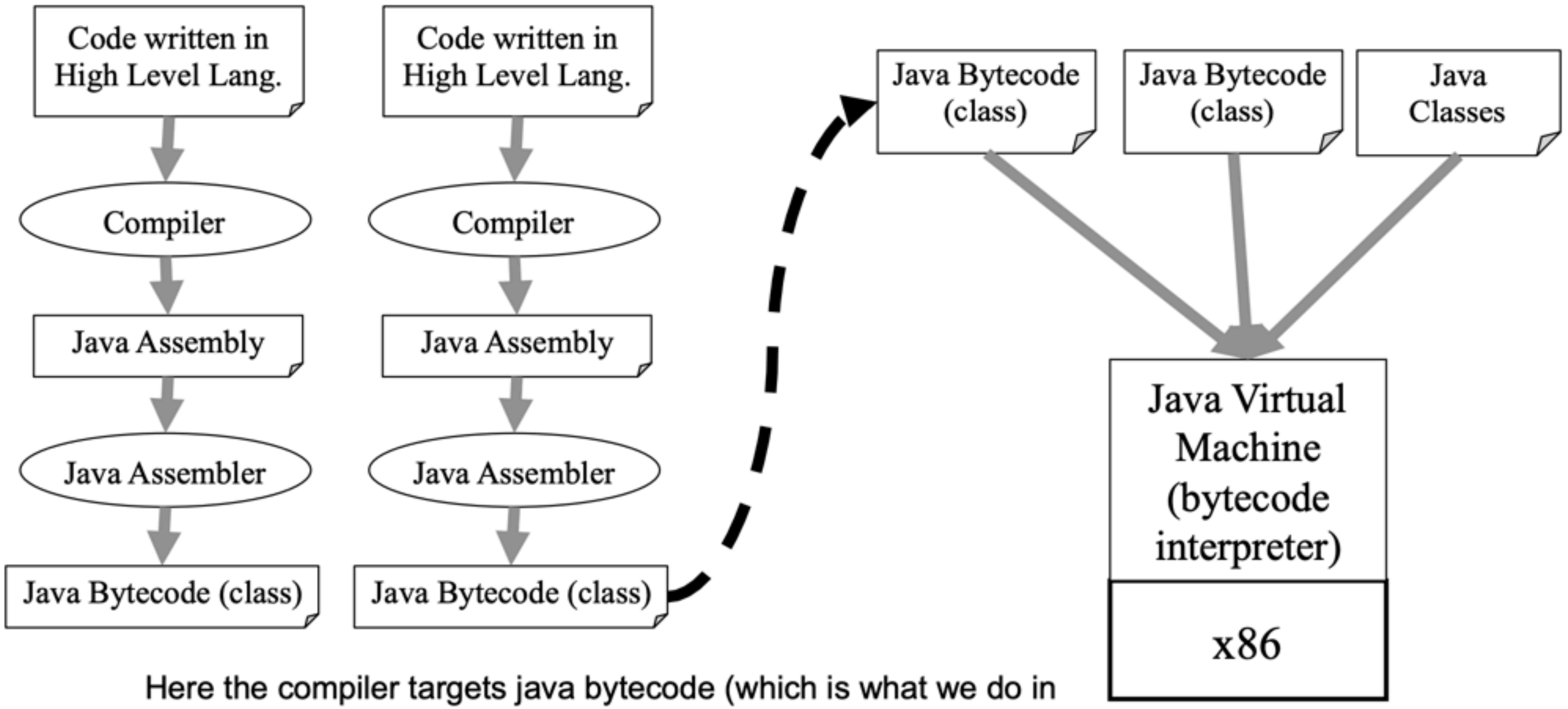

Code written in High Level Lang.
Compiler
x86 Assembly
x86 Assembler
x86 Object File (.o)
Code written in High Level Lang.
Compiler
x86 Assembly
x86 Assembler
x86 Object File (.o)
Linker
Not-fully Linked x86 Executable
Compile time
Run time
Not-fully Linked x86 Executable
Dynamic Libraries
Loader
Final Executable loaded in Memory
x86

The linker just links together the .o files, and does not link calls to dynamically loaded libraries (DLLs in Windows or Shared Libraries in Unix)

This is similar to before, but now the final linking occurs when the program is loaded (or even during program execution)

Here, libraries can be shared and they can be updated across the whole system without re-linking every single executable

Java Compiler



Here the compiler targets java bytecode (which is what we do in this class) and the bytecode is then run on top of the Java Virtual Machine (JVM). The JVM really just interprets (simulates) the bytecode like any scripting language. Because of this, any java program compiled to bytecode is portable to any machine that someone has already ported the JVM too. No need to recompile.

Register-based Machine

- Each instruction can have at most three operands
- We have to break large statements into little operations that use temporary variables
  - X=(2+3)+4 turns into to T1=2+3; X=T1+4;
- Temporary variables store the results at the internal nodes in the AST
- Assignments
  - x := y
  - x := y op z *op: binary arithmetic or logical operators*
  - x := op y *op: unary operators (minus, negation, integer to float)*
- Branch
  - goto L *execute the statement with labeled L next*
- Conditional Branch
  - if x relop y goto L relop: <, =, <=, >=, ==, !=
    - if the condition holds, we execute statement labeled L next
    - if the condition does not hold, we execute the statement following this statement next

Register-based Machine

```
if (x < y)
  x = 5*y + 5*y/3;
else
  y = 5;
x = x + y;
```

Variables can be represented with their locations in the symbol table

```
           if x < y goto L1
           goto L2
L1:        t1  := 5 * y
           t2  := 5 * y
           t3  := t2 / 3
           x  := t1 + t2
           goto L3
L2:        y  := 5
L3:        x  := x + y
```

Temporaries: temporaries correspond to the internal nodes of the syntax tree

- Three-address code instructions can be represented as an array of

  quadruples: operation, argument1, argument2, result
  triples: operation, argument1, argument2
  (each triple implicitly corresponds to a temporary)

Stack-based Machine

- Stack based code uses the stack to store temporary variables
- When we evaluate an expression *(E+E)*, it will take its arguments off the stack, add them together and put the result back on the stack.
- (2+3)+4 will *push 2; push 3; add; push 4; add*
- The machine code for this is a bit more ugly but the code is actually easier to generate because we do not need to handle temporary variables.

Stack-based Machine

```
if (x < y)
   x = 5*y + 5*y/3;
else
   y = 5;
x = x+y;
```

```
     load x
     load y
     iflt L1
     goto L2
L1:  push 5
     load y
     multiply
     push 5
     load y
     multiply
     push 3
     divide
     add
     store x
     goto L3
L2:  push 5
     store y
L3:  load x
     load y
     add
     store x
```

pushes the value at the location x to the stack

pops the top two elements and compares them

pops the top two elements, multiplies them, and pushes the result back to the stack

stores the value at the top of the stack to the location x

JVM: A stack machine

- JVM interpreter executes the bytecode on different machines
- JVM has an operand stack which we use to evaluate expressions
- JVM provides 65,535 local variables for each method The local variables are like registers so we do not have to worry about register allocation
- Each local variable in JVM is denoted by a number between 0 and 65535 (x and y in the example will be assigned unique numbers)

Stack-based v.s. Register-based

- Register-Based code:
    - Good - Compact representation
    - Good - “Self contained”: has inputs, outputs, and operation all in one “instruction”
    - Bad - Requires lots of temporary variables
    - Bad - Temporary variables have to be handled explicitly
- Stack Based Code:
    - Good – No temporaries, everything is kept on the stack
    - Good – It is easy to generate code for this
    - Bad – Requires more instructions to do the same thing

Expressions

- We are targeting a register-based machine
- We need to evaluate expressions assuming a very limited set of available registers (No register allocation).
- To generate code for an expression we will do a recursive traversal in post-order (that is, visit the children first, then generate code for the parent).

Expressions

- Let's start with a simple example expression: `(1 + 2) * (3 - 4)`

```
call generate_aexp(* node, left):
  call generate_aexp(+ node, left):
    call generate_aexp(1 node, left):
      emit "mov 1 LEFT_REG"
    call generate_aexp(2 node, right):
      emit "mov 2 RIGHT_REG"
    emit "add RIGHT_REG LEFT_REG"
 call generate_aexp(- node, right):
   call generate_aexp(3 node, left):
     emit "mov 3 LEFT_REG"
   call generate_aexp(4 node, right):
     emit "mov 4 RIGHT_REG"
   emit "sub RIGHT_REG LEFT_REG" "mov LEFT_REG RIGHT_REG"
 emit "mul RIGHT_REG LEFT_REG"
```

```
mov 1 LEFT_REG
mov 2 RIGHT_REG
add RIGHT_REG LEFT_REG
mov 3 LEFT_REG
mov 4 RIGHT_REG
sub RIGHT_REG LEFT_REG
mov LEFT_REG RIGHT_REG
mul RIGHT_REG LEFT_REG
```

What is the problem?

Expressions

- We have to create memory locations to hold temporary values during expression evaluation.

```
call generate_aexp(* node, tmp_num = 0):
  call generate_aexp(+ node, tmp_num = 1):
    call generate_aexp(1 node, tmp_num = 2):
      emit "mov 1 RESULT_REG"
    insert _tmp1 into symbol table
    emit "store RESULT_REG [_tmp1]"
    call generate_aexp(2 node, tmp_num = 2):
      emit "mov 2 RESULT_REG"
    emit "ld [_tmp1] OTHER_REG" "add OTHER_REG RESULT_REG"
    remove _tmp1 from symbol table
  insert _tmp0 into symbol table
  emit "store RESULT_REG [_tmp0]"
  call generate_aexp(- node, tmp_num = 1):
    insert _tmp1 into symbol table
    call generate_aexp(3 node, tmp_num = 2):
      emit "mov 3 RESULT_REG"
    emit "store RESULT_REG [_tmp1]"
    call generate_aexp(4 node, tmp_num = 2):
      emit "mov 4 RESULT_REG"
    emit "ld [_tmp1] OTHER_REG" "sub RESULT_REG OTHER_REG" "mov OTHER_REG RESULT_REG"
    remove _tmp1 from symbol table
  emit "ld [_tmp0] OTHER_REG" "mul OTHER_REG RESULT_REG"
  remove _tmp0 from symbol table
```

```
mov 1 RESULT_REG
store RESULT_REG [_tmp1]
mov 2 RESULT_REG
ld [_tmp1] OTHER_REG
add OTHER_REG RESULT_REG
store RESULT_REG [_tmp0]
mov 3 RESULT_REG
store RESULT_REG [_tmp1]
mov 4 RESULT_REG
ld [_tmp1] OTHER_REG
sub RESULT_REG OTHER_REG
mov OTHER_REG RESULT_REG
ld [_tmp0] OTHER_REG
mul OTHER_REG RESULT_REG
```

Let’s Generalize It

- Let's generalize the algorithm for arbitrary arithmetic expressions

```
generate_aexp(AST* node, int tmp_num = 0) {
  if (node is a constant number <n>) { emit "mov <n> RESULT_REG";
return; }
  if (node is a variable <x>) { emit "ld [x] RESULT_REG"; return; }

  // node must be one of +,-,*
  generate_aexp(node->left, tmp_num+1);
  insert _tmp<tmp_num> into symbol table;
  emit "store RESULT_REG [_tmp<tmp_num>]";
  generate_aexp(node->right, tmp_num+1);
  emit "ld [_tmp<tmp_num>] OTHER_REG";

  // left-hand value is in OTHER_REG, right-hand value is in RESULT_REG
  if (node is +) { emit "add OTHER_REG RESULT_REG"; return; }
  if (node is -) { emit "sub RESULT_REG OTHER_REG"; emit "mov OTHER_REG
RESULT_REG"; return; }
  emit "mul OTHER_REG RESULT_REG";

  remove _tmp<tmp_num> from symbol table;
}
```

https://sites.cs.ucsb.edu/~emre/cs160/#orgfa5615b

Assignment

- We evaluate the right-hand side using generate_aexp, which puts the result in RESULT_REG, then store the result to the memory location for the left-hand side variable.

```
generate_assign(lhs, rhs) {
  generate_aexp(rhs);
  emit "store RR [lhs]";
}
```

Non-nested Conditionals

- Conditionals without a nested scope

```
if (x < 2) { x := 1; } else { x := 2; }
```

```
generate_if(node) {
  <n> = fresh index;
  generate_rexp(node->guard);
  emit "cmp 0 RESULT_REG";
  emit "jmpe IF_FALSE_<n>";
  generate_block(node->true_branch);
  emit "jmp IF_END_<n>";
  emit "IF_FALSE_<n>:";
  generate_block(node->false_branch);
  emit "IF_END_<n>:";
}
```

```
    ld [x] RR
    store RR [_tmp0]
    mov 2 RR
    ld [_tmp0] OR
    cmp RR OR
    setlt RR
    cmp 0 RR
    jmpe IF_FALSE_0
    mov 1 RR
    store RR [x]
    jmp IF_END_0
IF_FALSE_0:
    mov 2 RR
    store RR [x]
IF_END_0:
```

Nested Conditionals

- When the code generator enters that new scope:
  - see how many declared variables there are
  - adjust the stack pointer accordingly
  - initialize the new memory locations to 0
  - update symbol table to map the newly declared variables to their offsets
- and when we leave the new scope we need to adjust things back the way they were:
  - reset the stack pointer to its old position
  - restore the symbol table to its old value

Nested Conditionals

```
generate_block(node) {
  old_symbol_table = symbol_table;
  stack_size = node->num_declared_variables * 4; // because 4-byte
integers
  emit "sub <stack_size> STACK_REG";
  insert_in_symbol_table(symbol_table, node->declared_variables);
  for each var in node->declared_variables { emit "store 0 [var]"; }
  .
  .
  .
  emit "add <stack_size> STACK_REG";
  symbol_table = old_symbol_table;
}
```

While Loops

- Conditionals without a nested scope

```
while (x < 3) { x := x + 1; }
```

$\downarrow$

```
generate_while(node) {
  <n> = fresh index;
  emit "WHILE_START_<n>:";
  generate_rexp(node->guard);
  emit "cmp 0 RESULT_REG";
  emit "jmpe WHILE_END_<n>";
  generate_block(node->body);
  emit "jmp WHILE_START_<n>";
  emit "WHILE_END_<n>:";
}
```

```
WHILE_START_0:
    ld [x] RR
    store RR [_tmp0]
    mov 3 RR
    ld [x] OR
    cmp RR OR
    setlt RR
    cmp 0 RR
    jmpe WHILE_END_0
    ld [x] RR
    store RR [_tmp0]
    mov 1 RR
    ld [_tmp0] OR
    add OR RR
    store RR [x]
    jmp WHILE_START_0
WHILE_END_0:
```

OCaml vs LLVM

"let - in" and
OCaml-style identifiers:

```
let tmp1 = add 3L 4L in
```

•OCaml-style "let-rec"
and functions for blocks:

```
let rec entry () =
  let tmp1 = …
and foo () =
  let tmp2 = …
```

•OCaml-style global variables:

```
let varX = ref 0L
```

Omits let/in and prefixes local identifiers with %:

```
%tmp1 = add i64 3, i64 4
```

•Uses lighter-weight colon notation:

```
entry:
  %tmp1 = …
foo:
  %tmp2 = …
```

•Prefixes globals with @

```
define @X = i64 0
```

Example LLVM Code

```
#include <stdio.h>
#include <stdint.h>

int64_t factorial(int64_t n) {
  int64_t acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

```
define @factorial(%n) {
  %1 = alloca
  %acc = alloca
  store %n,  %1
  store 1,  %acc
  br label %start

start:
  %3 = load %1
  %4 = icmp sgt %3, 0
  br %4, label %then, label %else

then:
  %6 = load %acc
  %7 = load %1
  %8 = mul %6, %7
  store %8, %acc
  %9 = load %1
  %10 = sub %9, 1
  store %10, %1
  br label %start

else:
  %12 = load %acc
  ret %12
}
```

Real LLVM

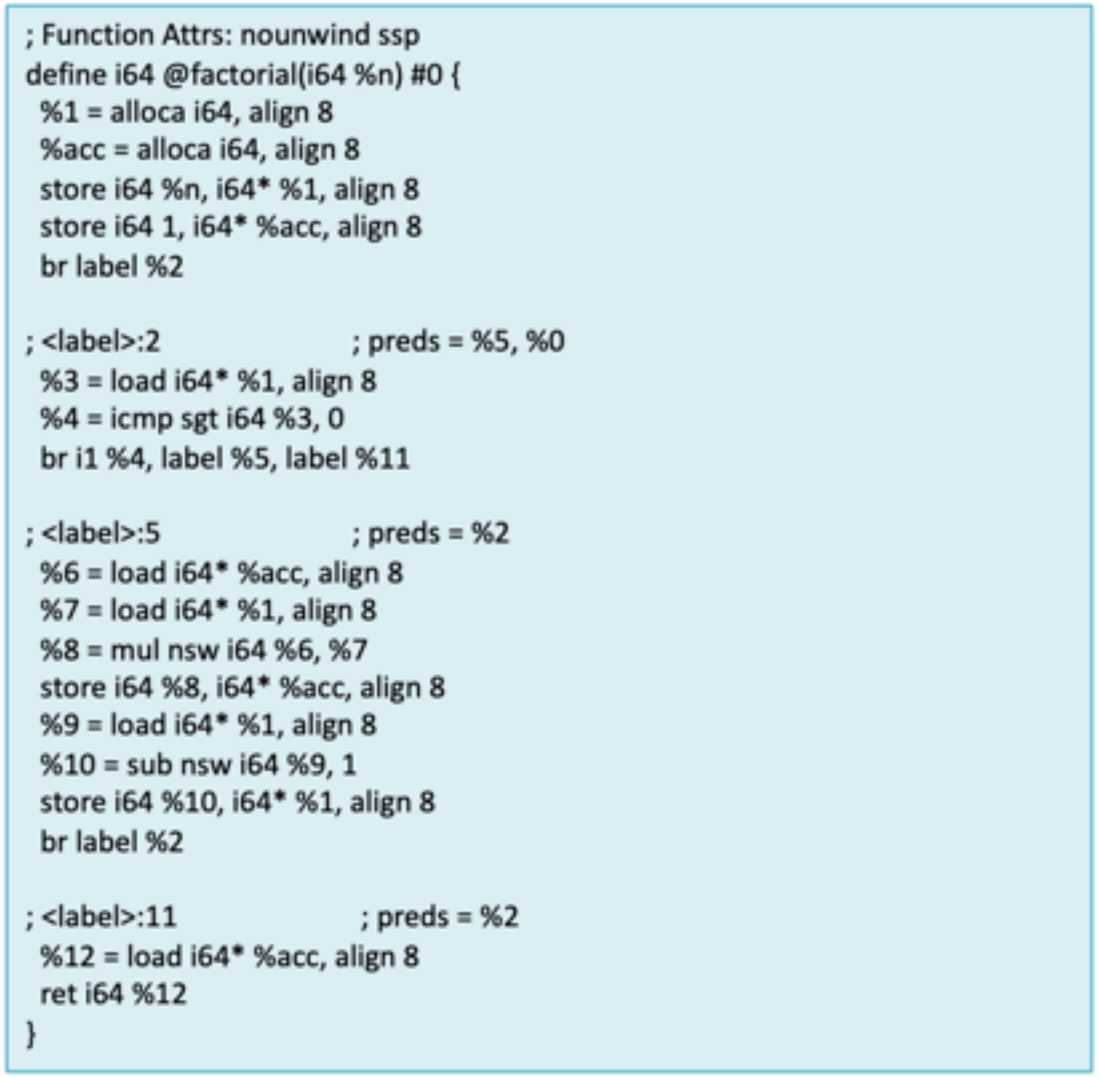
- Decorates values with type information

  i64

  i64*

  i1
- Permits numeric identifiers
- Has alignment annotations
- Keeps track of entry edges for each block:

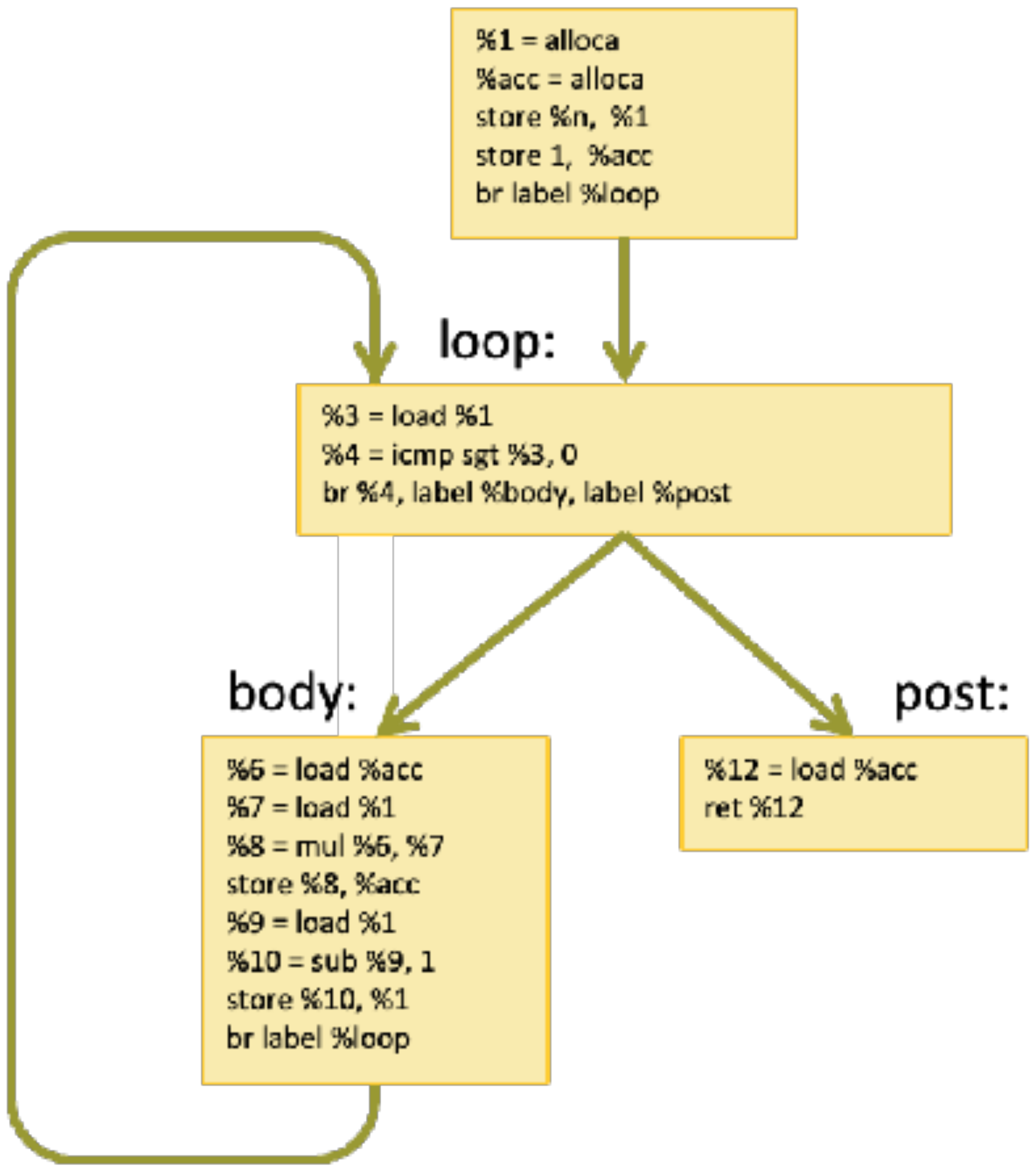
  preds = %5, %0



```
; Function Attrs: nounwind ssp
define i64 @factorial(i64 %n) #0 {
  %1 = alloca i64, align 8
  %acc = alloca i64, align 8
  store i64 %n, i64* %1, align 8
  store i64 1, i64* %acc, align 8
  br label %2

; <label>:2                        ; preds = %5, %0
  %3 = load i64* %1, align 8
  %4 = icmp sgt i64 %3, 0
  br i1 %4, label %5, label %11

; <label>:5                        ; preds = %2
  %6 = load i64* %acc, align 8
  %7 = load i64* %1, align 8
  %8 = mul nsw i64 %6, %7
  store i64 %8, i64* %acc, align 8
  %9 = load i64* %1, align 8
  %10 = sub nsw i64 %9, 1
  store i64 %10, i64* %1, align 8
  br label %2

; <label>:11                       ; preds = %2
  %12 = load i64* %acc, align 8
  ret i64 %12
}
```

Example CFG

define @factorial(%n) {



```
%1 = alloca
%acc = alloca
store %n,  %1
store 1,  %acc
br label %loop
```

```
%3 = load %1
%4 = icmp sgt %3, 0
br %4, label %body, label %post
```

```
%6 = load %acc
%7 = load %1
%8 = mul %6, %7
store %8, %acc
%9 = load %1
%10 = sub %9, 1
store %10, %1
br label %loop
```

```
%12 = load %acc
ret %12
```

}

LL Basic Blocks and Control-Flow Graphs

- LLVM enforces (some of) the basic block invariants syntactically.
- Representation in OCaml:

```
type block = {
        insns : (uid * insn) list;
        term  : (uid * terminator)
}
```

- A *control flow graph* is represented as a list of labeled basic blocks with these invariants:
  - No two blocks have the same label
  - All terminators mention only labels that are defined among the set of basic blocks
  - There is a distinguished, unlabeled, entry block:

```
type cfg = block * (lbl * block) list
```

LL Storage Model: Locals

- Several kinds of storage:
  - Local variables (or temporaries): `%uid`
  - Global declarations (*e.g.*, for string constants): `@gid`
  - Abstract locations: references to (stack-allocated) storage created by the `alloca` instruction
  - Heap-allocated structures created by external calls (*e.g.*, to `malloc`)

- Local variables:
  - Defined by the instructions of the form `%uid = …`
  - Must satisfy the ***static single assignment*** invariant
    - *Each `%uid` appears on the left-hand side of an assignment only once in the entire control flow graph.*
  - The value of a `%uid` remains unchanged throughout its lifetime
  - Analogous to "`let %uid = e in` …" in OCaml
- Intended to be an abstract version of machine registers.

LL Storage Model: alloca

- alloca instruction allocates stack space and returns a reference to it.
  - The returned reference is stored in local:

    %ptr = alloca type
  - The amount of space allocated is determined by the type

- The contents of the slot are accessed via the load and store instructions:

```
%acc = alloca i64               ; allocate a storage slot
store i64 341, i64* %acc        ; store the integer value 341
%x = load i64, i64* %acc        ; load the value 341 into %x
```

- Gives an abstract version of stack slots

References

- llvm.org
- llvmLite: https://www.cs.princeton.edu/courses/archive/spring19/cos320/hw/llvmlite.shtml