Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel

Code Generation

Overview

- Intermediate Representations
  - There is more that one way to represent code as it is being generated, analyzed, and optimized (we use ASTs)
- How code runs
  - The way code runs on a machine depends on if the code is compiled or interpreted, and if it is statically or dynamically linked
- Code Generation
  - Three-address code and stack code
  - Dealing with Boolean values and control (such as loops)
  - Arrays

Intermediate Representations

Graphical IRs
- Abstract Syntax Trees (AST)
- Directed Acyclic Graphs (DAG)
- Control Flow Graphs (CFG)

Linear IRs
- Static Single Assignment Form (SSA)
- Stack Machine Code
- Three Address Code

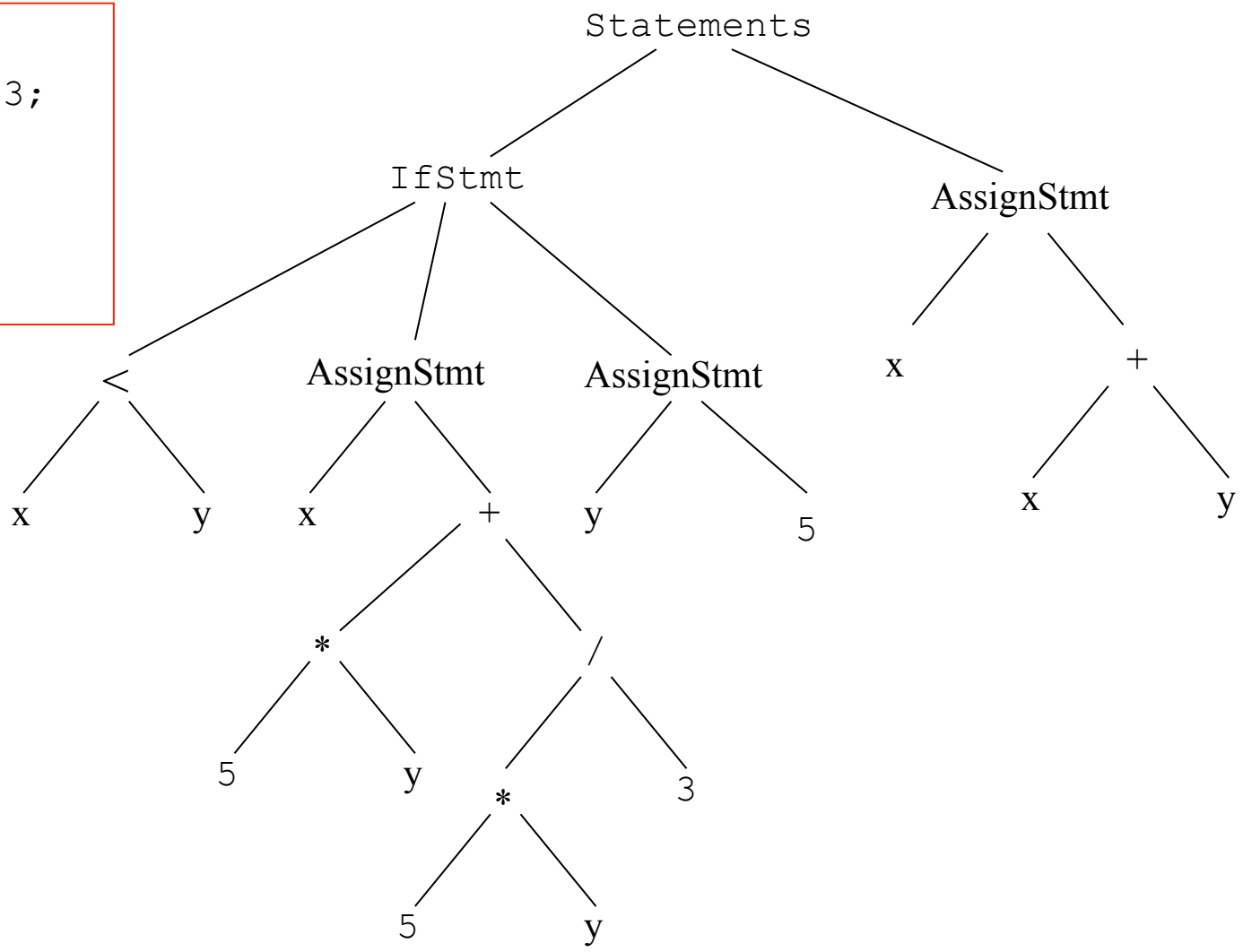
high-level

low-level

- Hybrid approaches mix graphical and linear representations
  - MIPS compilers use ASTs for loops if-statements and array references
  - Use three-address code in basic blocks in control flow graphs

Abstract Syntax Trees (ASTs)

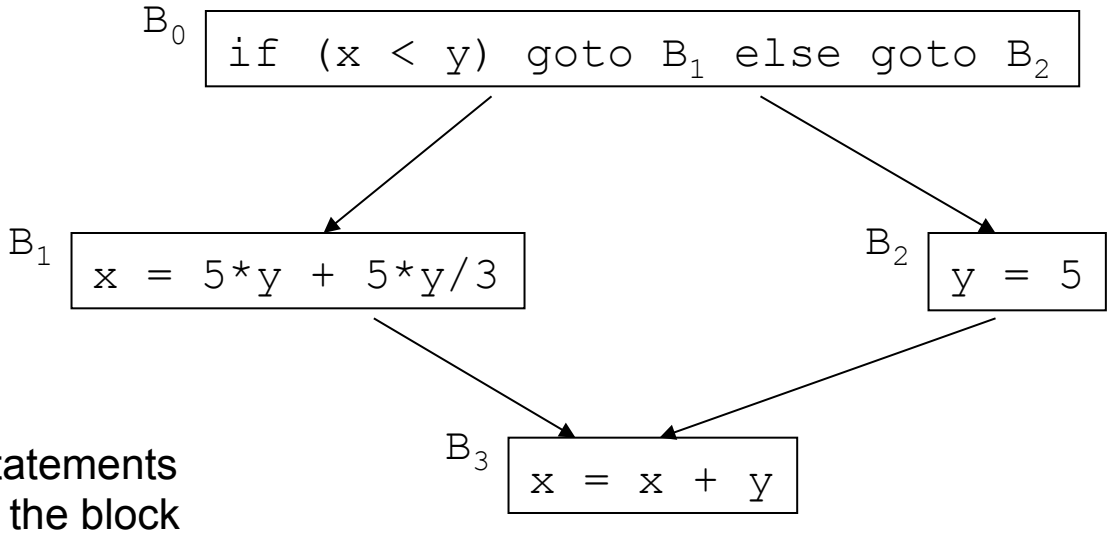

```
if (x < y)
  x = 5*y + 5*y/3;
else
  y = 5;
x = x+y;
```



Control Flow Graphs (CFGs)

- Nodes in the control flow graph are basic blocks
  - A *basic block* is a sequence of statements always entered at the beginning of the block and exited at the end
- Edges in the control flow graph represent the control flow

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



- Each block has a sequence of statements
- No jump from or to the middle of the block
- Once a block starts executing, it will execute till the end

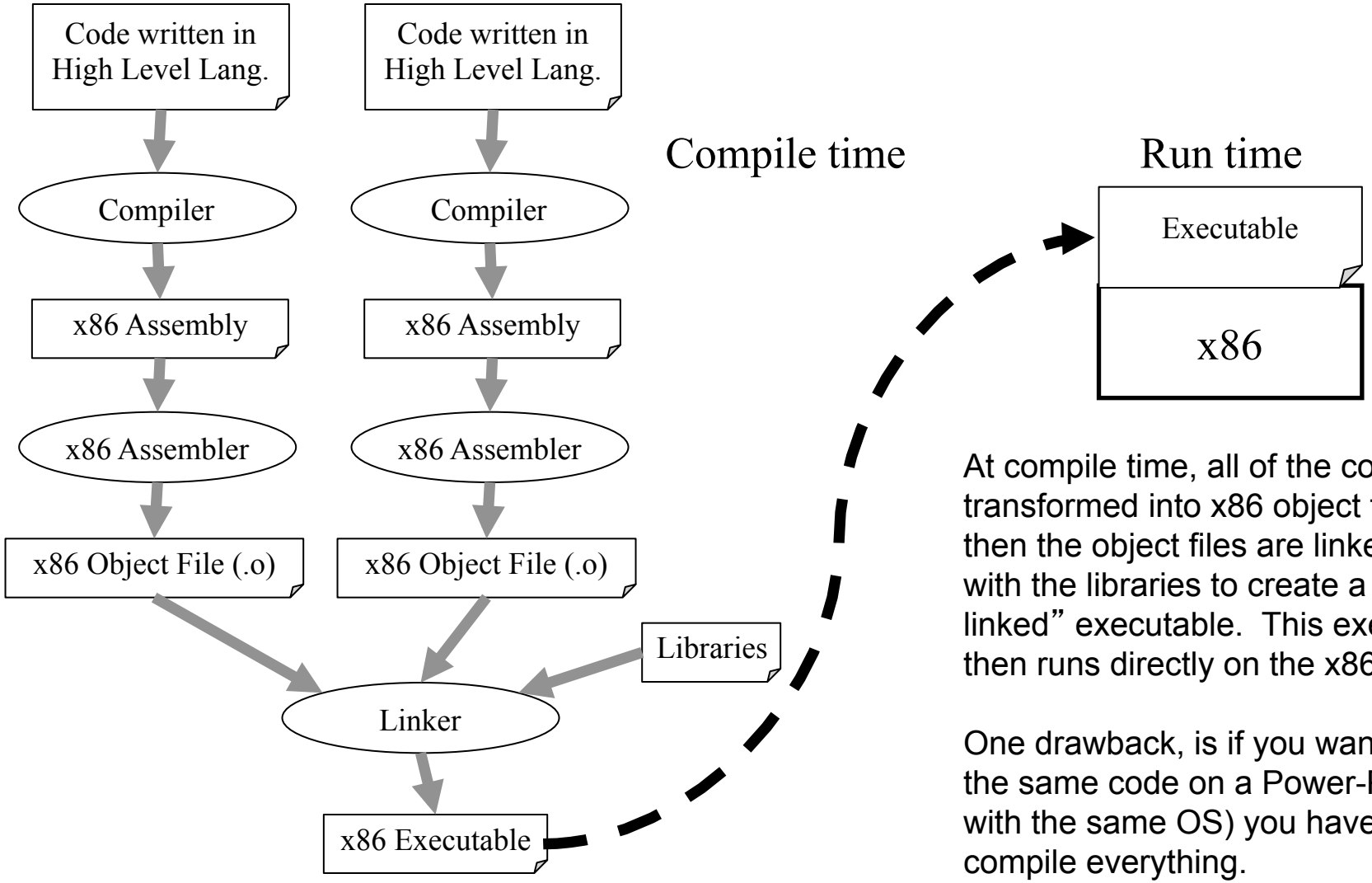
Code Generation

- 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
  - Three-address code: Similar to most processors (x86, Sparc, ARM, …)

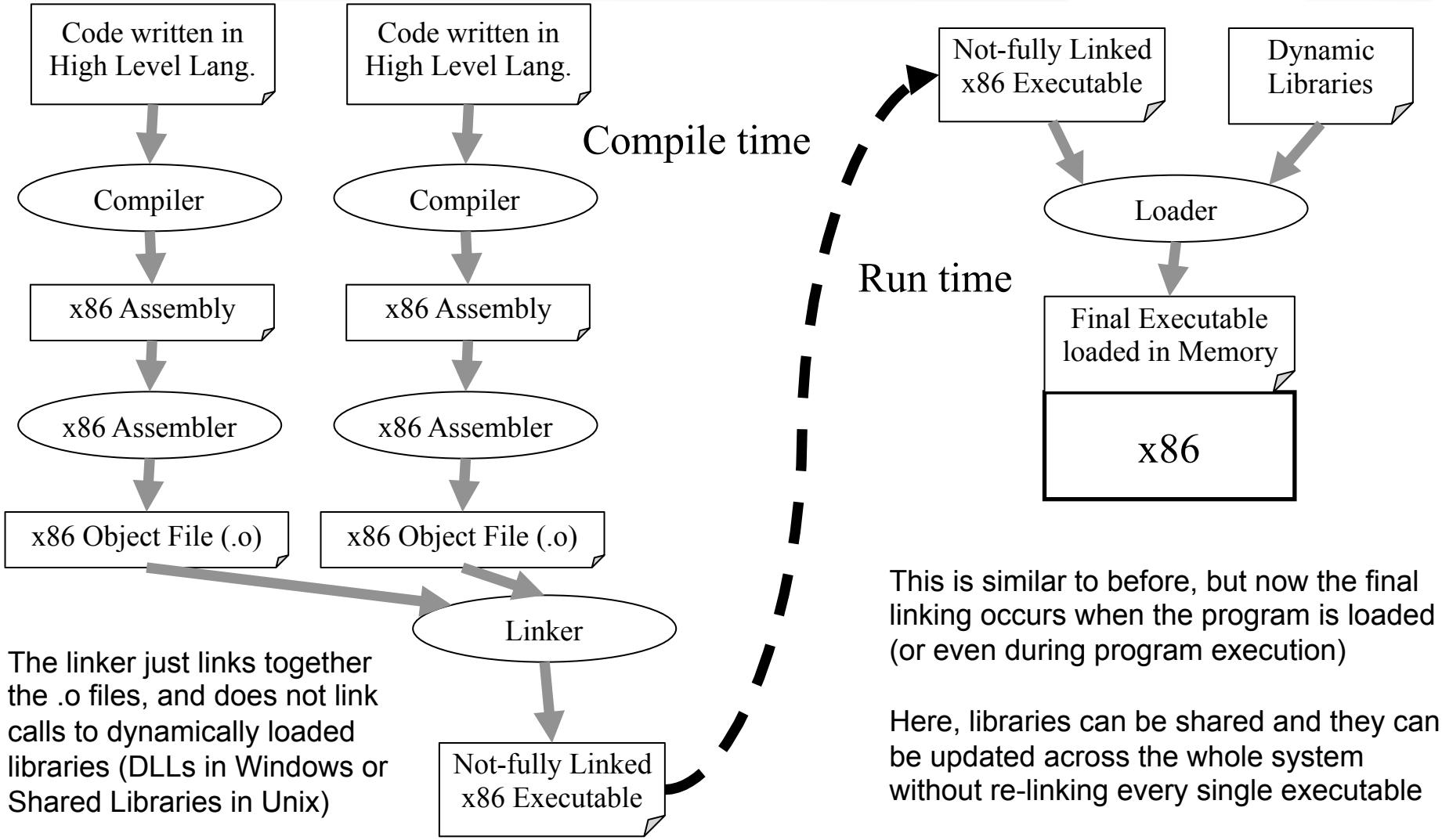
x86 C Compiler with Static Linking



At compile time, all of the code is transformed into x86 object files and then the object files are linked together with the libraries to create a "statically linked" executable. This executable then runs directly on the x86 hardware.

One drawback, is if you wanted to run the same code on a Power-PC (even with the same OS) you have to re-compile everything.

x86 C Compiler with Dynamic Linking

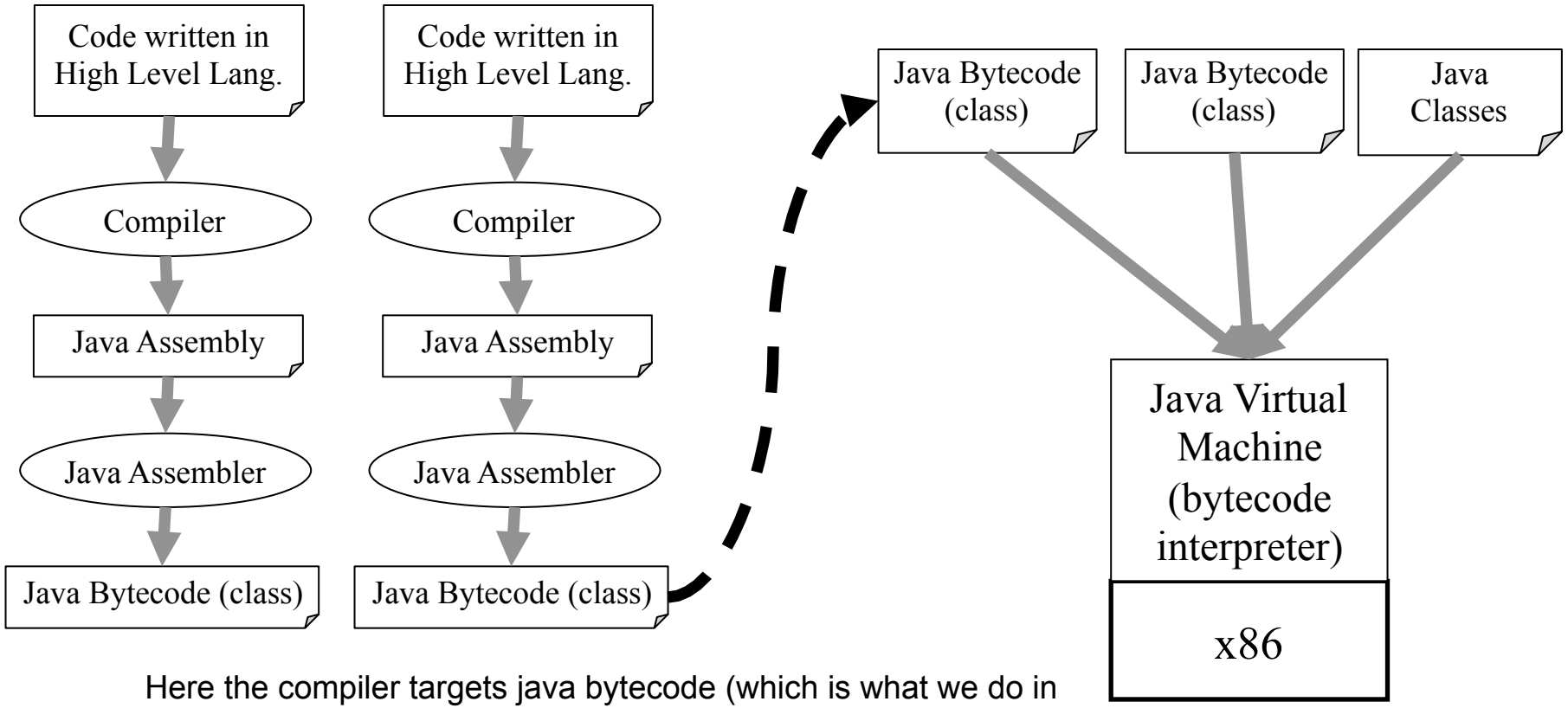


The linker just links together the .o files, and does not link calls to dynamically loaded libraries (DLLs in Windows or Shared Libraries in Unix)

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.

Three-Address Code

- 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

Three-Address Code


```
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 Machine Code

- 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 Machine Code


```
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)

Code

- Three-Address Code:
  - Good - Compact representation
  - Good - Statement is “self contained” in that it has the inputs, outputs, and operation all in one “instruction”
  - Bad - Requires lots of temporary variables
  - Bad - Temporary variables have to be handled explicitly

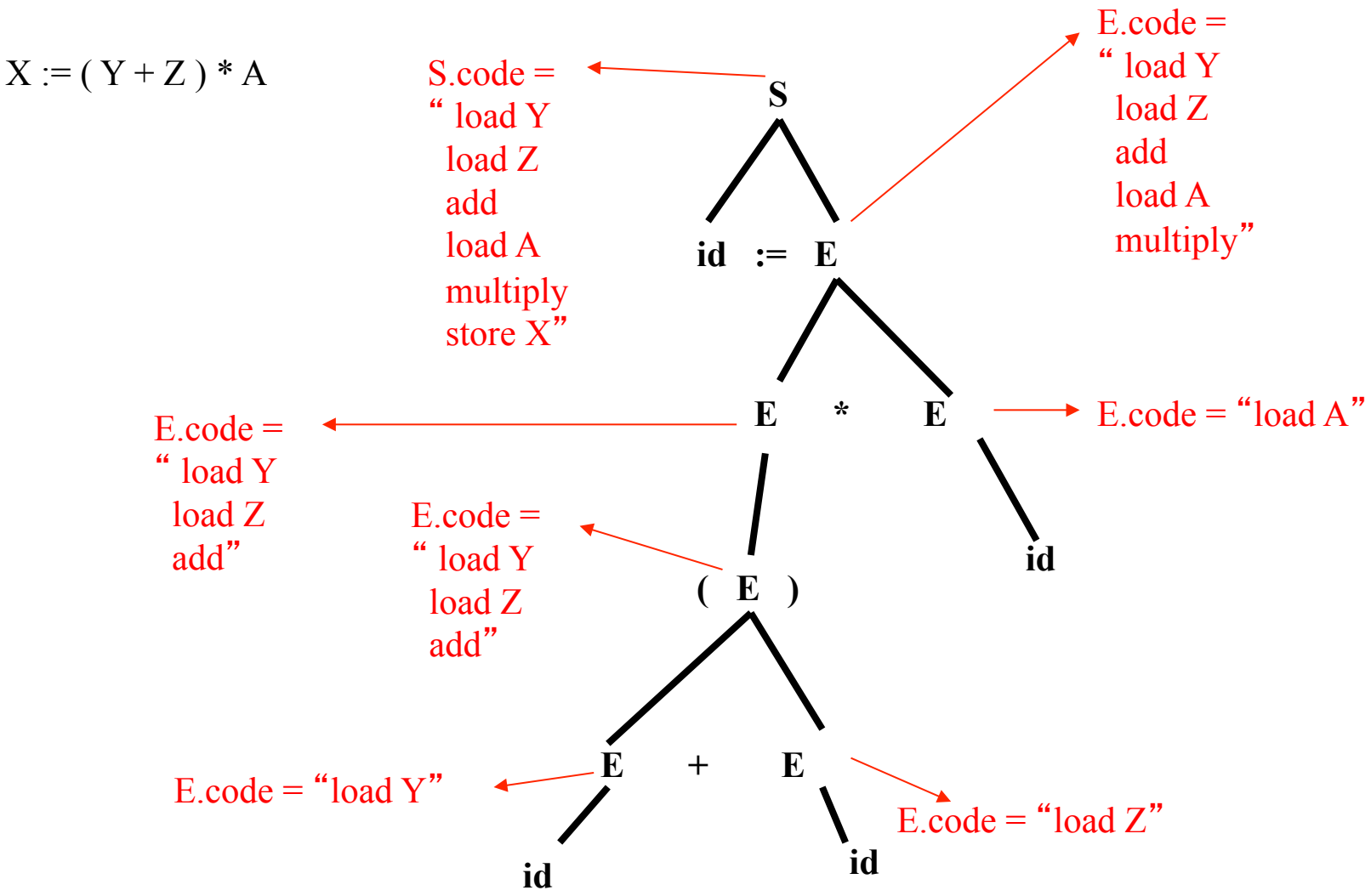
- Stack Based Code:
  - Good – No temporaries, everything is kept on the stack
  - Good – It is easy to generate code for this
  - Bad – Requires more instructions to do the same thing

Stack Based Code Generation

Attributes:	*E.code*: sequence of instructions that are generated for *E* *(no place for an expression is needed since the result of an expression is stored in the operand stack)*
Procedures:	newtemp(): Returns a new temporary each time it is called gen(): Generates instruction (have to call it with appropriate arguments) lookup(id.*name*): Returns the location of id from the symbol table

Productions	Semantic Rules
$S \rightarrow$ id := E	id.*place* ← lookup(id.*name*); *S.code* ← *E.code* || gen('store' id.*place*);
$E \rightarrow E_1 + E_2$	*E.code* ← E_1.*code* || E_2.*code* || gen('add'); *(arguments for the add instruction are in the top of the stack)*
$E \rightarrow E_1 * E_2$	*E.code* ← E_1.*code* || E_2.*code* || gen('multiply');
$E \rightarrow (E_1)$	*E.code* ← E_1.*code*;
$E \rightarrow - E_1$	*E.code* ← E_1.*code* || gen('negate ');
$E \rightarrow$ id	*E.code* ← gen('load' id.*place*)

Example


X := (Y + Z) * A
S.code = " load Y load Z add load A multiply store X"
S
id := E
E.code = " load Y load Z add load A multiply"
E * E
E.code = "load A"
E.code = " load Y load Z add"
E.code = " load Y load Z add"
(E)
id
E + E
E.code = "load Y"
E.code = "load Z"
id
id

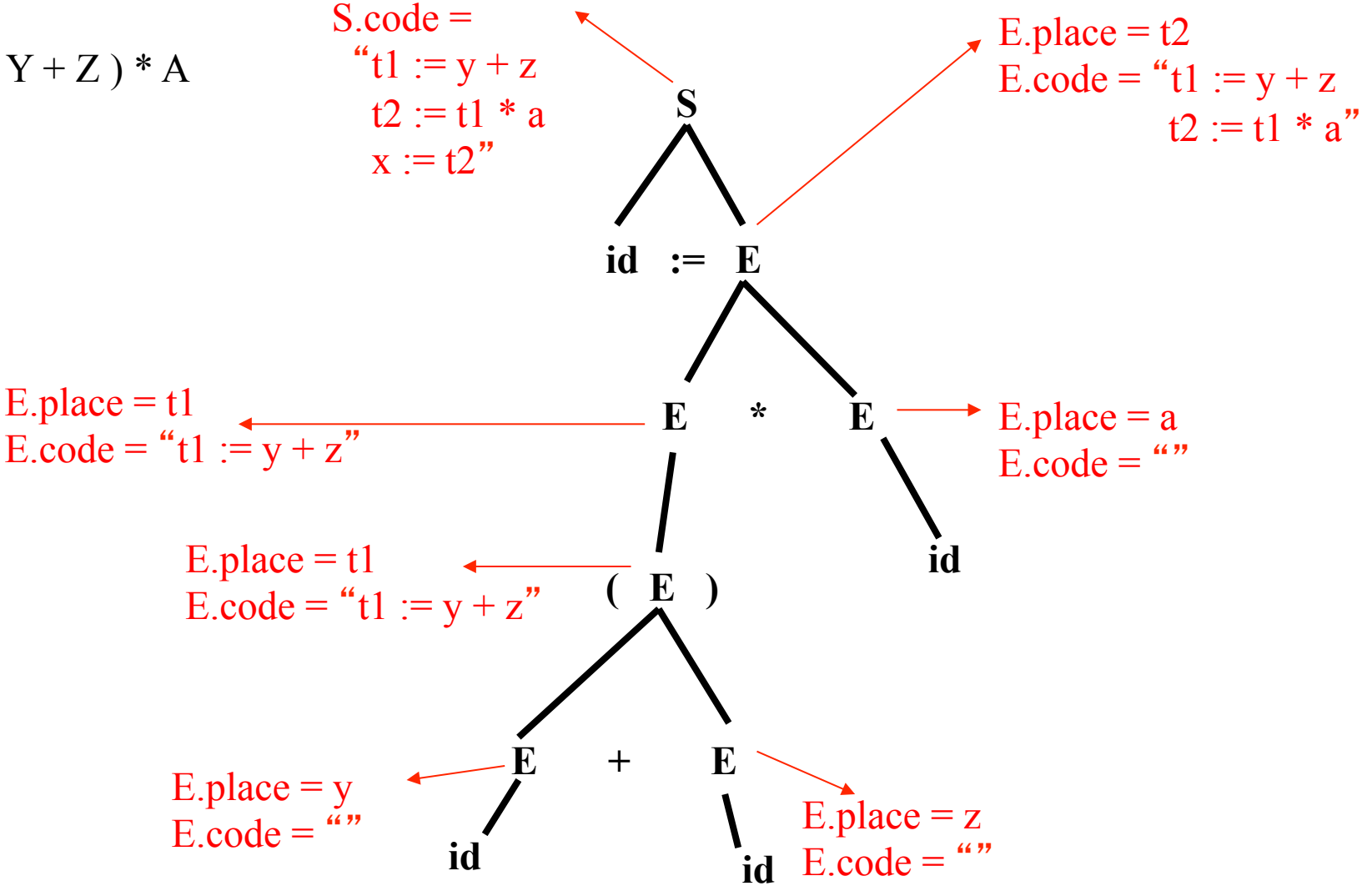
Three-Address Code

Attributes:	*E.place*: location that holds the value of expression *E*
	E.code: sequence of instructions that are generated for *E*
Procedures:	newtemp(): Returns a new temporary each time it is called
	gen(): Generates instruction (have to call it with appropriate arguments)
	lookup(id.*name*): Returns the location of id from the symbol table

Productions	Semantic Rules
$S \rightarrow$ id := E	id.*place* ← lookup(id.*name*);
	S.code ← *E.code* || gen(id.*place* ':=' *E.place*);
$E \rightarrow E_1 + E_2$	*E.place* ← newtemp();
	E.code ← E_1.*code* || E_2.*code* || gen(*E.place* ':=' E_1.*place* '+' E_2.*place*);
$E \rightarrow E_1 * E_2$	*E.place* ← newtemp();
	E.code ← E_1.*code* || E_2.*code* || gen(*E.place* ':=' E_1.*place* '*' E_2.*place*);
$E \rightarrow (E_1)$	*E.code* ← E_1.*code*;
	E.place ← E_1.*place;*
$E \rightarrow - E_1$	*E.place* ← newtemp();
	E.code ← E_1.*code* || gen(*E.place* ':=' 'uminus' E_1.*place*);
$E \rightarrow$ id	*E.place* ← lookup(id.*name*);
	E.code ← '' (*empty string*)

Example

X := (Y + Z) * A



Code Generation for Boolean Expressions

- Two approaches
  - Numerical representation
  - Implicit representation
- Numerical representation
  - Use 1 to represent true, use 0 to represent false
  - For three-address code, store this result in a temporary
  - For stack machine code, store this result on the stack
- Implicit representation
  - For the Boolean expressions that are used in flow-of-control statements (such as if-statements, while-statements etc.) Boolean expressions do not have to explicitly compute a value, they just need to branch to the right instruction
  - Generate code for Boolean expressions that branch to the appropriate instruction based on the result of the Boolean expression

Boolean Expressions: Numerical Representation

Attributes :	*E.place*: location that holds the value of expression *E* *E.code*: sequence of instructions that are generated for *E* id.*place*: location for id relop.*func*: the type of relational function

If there are instructions in the architecture that support operations on Boolean data (like "logical and" or "logical or"), then the easiest way to implement Boolean data is to just treat it like normal data

Productions	Semantic Rules
$E \rightarrow \text{id}_1$ relop id_2	*E.place* ← newtemp(); *E.code* ← id_1.*code* || id_2.*code* || gen(*E.place* ':=' id_1.*place* relop.*func* id_1.*place*)
$E \rightarrow E_1$ and E_2	*E.place* ← newtemp(); *E.code* ← E_1.*code* || E_2.*code* || gen(*E.place* ':=' E_1.*place* 'and' E_2.*place*);

Boolean Expressions: Implicit Representation

Attributes :

E.code: sequence of instructions that are generated for *E*
E.false: instruction to branch to if *E* evaluates to false
E.true: instruction to branch to if *E* evaluates to true
(*E.code* is synthesized whereas *E.true* and *E.false* are inherited)
id.*place*: location for id

Productions	Semantic Rules
$E \rightarrow \text{id}_1 \ \text{relop} \ \text{id}_2$	$E.code \leftarrow \text{gen}(\text{'if'} \ \text{id}_1.place \ \text{relop}.op \ \text{id}_2.place \ \text{'goto'} \ E.true) \ \| \ \text{gen}(\text{'goto'} \ E.false)$;
$E_0 \rightarrow E_1 \text{ and } E_2$	$E_1.false \leftarrow E_0.false$; (*short-circuiting*) $E_2.false \leftarrow E_0.false$; $E_1.true \leftarrow \text{newlabel}()$; $E_2.true \leftarrow E_0.true$; $E.code \leftarrow E_1.code \ \| \ \text{gen}(E_1.true \ \text{':'}) \ \| \ E_2.code$;

can be any relational operator:
==, <=, >= !=

This generated label will be inserted to the place for $E_1.true$ in the code generated for E_1

These places will be filled with labels later on when they become available

Example

These are the locations of three-address code instructions, they are not labels

Numerical representation:

```
100     if x < y goto 103
101     t1 := 0
102     goto 104
103     t1 := 1
104     if a = b goto 107
105     t2 := 0
106     goto 108
107     t2 := 1
108     t3 := t1 and t2
```

Input Boolean expression:
x < y and a == b

Implicit representation:

```
        if x < y goto L1
        goto LFalse
L1:     if a = b goto LTrue
        goto LFalse
        ...
```

These labels will be generated later on, and will be inserted to the corresponding places

LTrue:

LFalse:

Flow-of-Control Statements

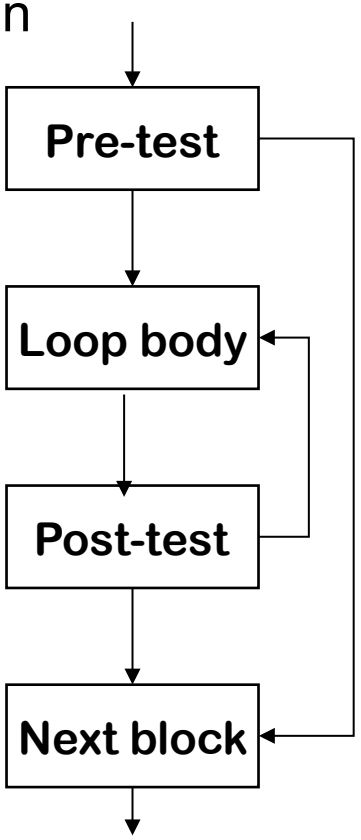
If-then-else

- Branch based on the result of Boolean expression

Loops

- Evaluate condition before loop (if needed)
- Evaluate condition after loop
- Branch back to the top if condition holds

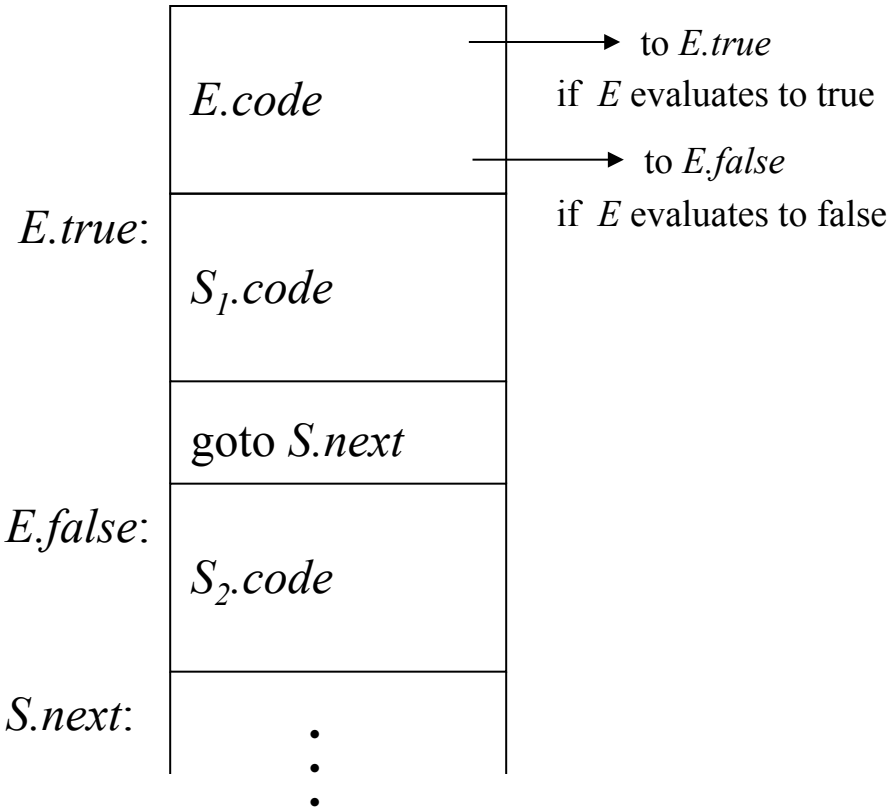
  Merges test with last block of loop body



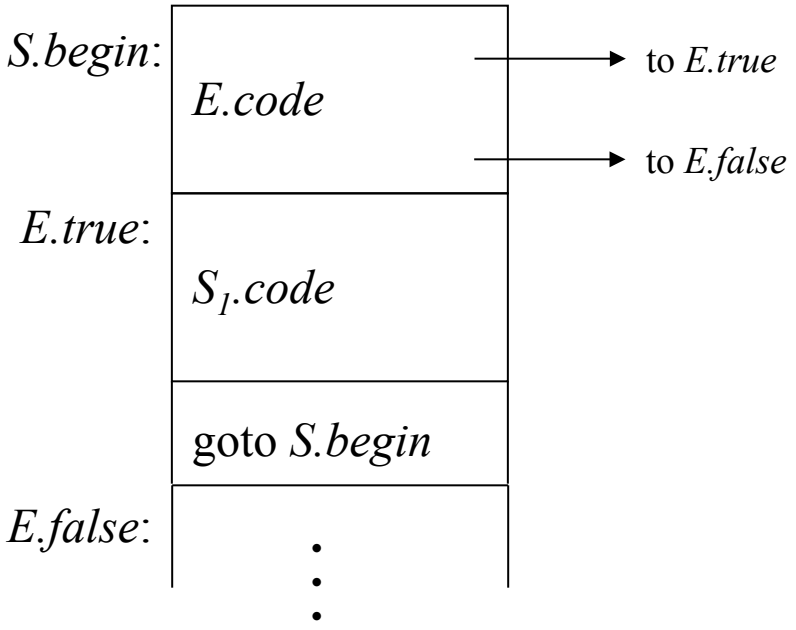
While, for, do, and until all fit this basic model

Flow-of-Control Statements: Code Structure

$S \rightarrow$ if E then S_1 else S_2



$S \rightarrow$ while E do S_1



Another approach is to place *E.code* after S_1*.code*

Flow-of-Control Statements

Attributes : *S.code*: sequence of instructions that are generated for *S*
S.next: label of the instruction that will be executed immediately after *S*
(S.next is an inherited attribute)

Productions	Semantic Rules
$S \rightarrow$ if E then S_1 else S_2	*E.true* ← newlabel(); *E.false* ← newlabel(); S_1*.next* ← *S. next*; S_2*.next* ← *S. next*; *S.code* ← *E.code* || gen(*E.true* ':') || S_1*.code* || gen('goto' *S.next*) || gen(*E.false* ':') || S_2*.code* ;
$S \rightarrow$ while *E* do S_1	*S.begin* ← newlabel(); *E.true* ← newlabel(); *E.false* ← *S. next*; S_1*.next* ← *S. begin*; *S.code* ← gen(*S.begin* ':') || *E.code* || gen(*E.true* ':') || S_1*.code* || gen('goto' *S.begin*);
$S \rightarrow S_1 ; S_2$	S_1*.next* ← newlabel(); S_2*.next* ← *S.next*; *S.code* ← S_1*.code* || gen(S_1*.next* ':') || S_2*.code*

Example

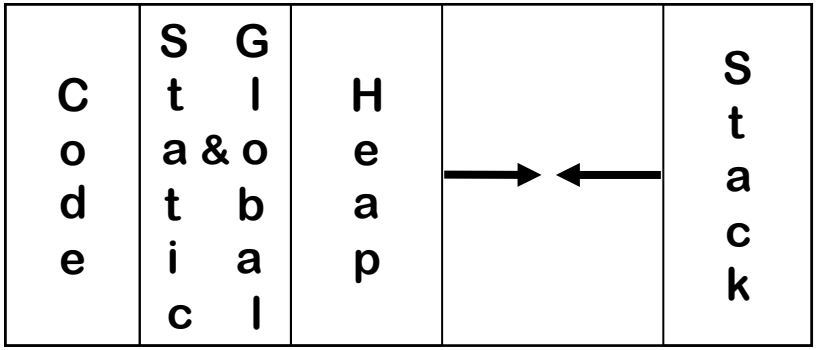
Input code fragment:

```
while (a < b)  {
   if  (c < d)
      x = y + z;
   else
      x = y – z
}
```

```
L1:     if a < b goto L2
        goto LNext
L2:     if c < d goto L3
        goto L4
L3:     t1 := y + z
        x := t1
        goto L1
L4:     t2 := y – z
        x := t2
        goto L1
LNext:     ...
```

Memory Layout

Placing run time data structures



- **Stack and heap share free space**
- **Fixed-size areas together**
- **For compiler, this is the entire picture**

Alignment and padding

- Machines have alignment restrictions
  - 32-bit floating point numbers and integers should begin on a full-word boundary (32-bit boundary)
- Place values with identical restrictions next to each other
- Assign offsets from most restrictive to least
- If needed, insert padding (space left unused due to alignment restrictions) to match restrictions

Memory Allocation

P → D
D → D; D
D → id : T
T → char | int | float | array[num] of T | pointer T

Attributes:	T.type, T.width
Basic types:	char width 4, integer width 4, float width 8
Type constructors:	array(size,type) width is size * (width of type) pointer(type) width is 4

- Enter the variables to the symbol table with their type and memory location
- Set the type attribute T.type
- Layout the storage for variables
  - Calculate the offset for each local variable and enter it to the symbol table
  - Offset can be offset from a static data area or from the beginning of the local data area in the activation record

Translation Scheme for Memory Allocation

P → {offset ← 0;} D
D → D; D
D → id : T {enter(id.name,T.type, offset); offset ← offset + T.width; }
T → char { T.type ← char; T.width ← 4; }
| int { T.type ← integer; T.width ← 4; }
| float { T.type ← float; T.width ← 8; }
| array[num] of T_1 { T.type ← array(num.val, T_1.type);
T.width ← num.val * T_1.width; }
| pointer T { T.type ← pointer(T_1.type); T.width ← 8; }

- Note that if the size of the array is not a constant we cannot compute its width at compile time
- In that case, allocate the memory for the array in the heap and allocate the memory for the pointer to the heap at compile time

Array Accesses

First, must agree on a storage scheme:

Row-major order (most languages)

- Lay out as a sequence of consecutive rows
- Rightmost subscript varies fastest
- A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]

Column-major order (Fortran)

- Lay out as a sequence of columns
- Leftmost subscript varies fastest
- A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

Indirection vectors (Java)

- Vector of pointers to pointers to … to values
- Takes more space
- Locality may not be good

Laying Out Arrays

The Concept

A

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4

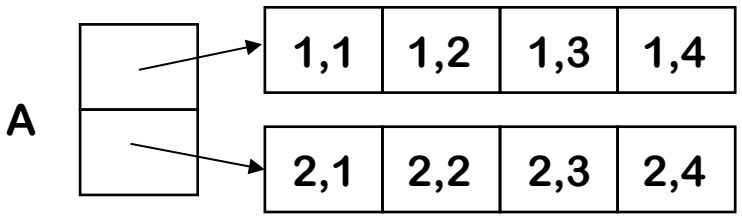
Row-major order

A

1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4

Column-major order

Indirection vectors



These have distinct & different cache behavior

The order of traversal of an array can effect the performance

How do we insert the calculation for arrays

- If I access element A[2,3], what address is storing my variable?

	0	1	2	3	4	5	6	7
A	**1,1**	**1,2**	**1,3**	**1,4**	**2,1**	**2,2**	**2,3**	**2,4**

Need to map i = 2, j = 3 to array element 6

- If i = 2, and we start at 1, we need to skip over one row (Row 1) worth of stuff. In general, we would skip over (*i* – *low)* rows (*low* is the number you start counting at for your arrays - in the example, it is 1)
- Each row is some number of elements in length (*high* – *low* + 1)
  = (4 – 1) + 1 = 4
- Once you get to the correct row, we just add *j* – *low* to get the right index
  = (3 – 1) = 2

Computing an Array Address

1-D array: A[i]

- @A + (i - low) × sizeof(A[1])

Almost always a power of 2, known at compile-time ⇒ use a shift for speed

Base of A (starting address of the array)

int A[1:10] ⇒ low is 1
Make low 0 for faster access (save a subtraction)

Expensive computation!
Lots of +, -, x operations

Two-D array: $A[i_1,i_2]$

Row-major order, two dimensions

$$@A + ((i_1 - low_1) \times (high_2 - low_2 + 1) + i_2 - low_2) \times sizeof(A[1])$$

Column-major order, two dimensions

$$@A + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times sizeof(A[1])$$

Indirection vectors, two dimensions

$*(A[i_1])[i_2]$ — where $A[i_1]$ is, itself, a 1-d array reference