

Control Flow

Language Mechanisms

- Sequencing: intuition of linear order between back-to-back statements
- Selection: to represent conditions and choices; if/case/switches
- Iteration: repeat some piece of code
- Procedural abstraction: logical aggregation of work (and probably data)
- Recursion: expression or entity defined in (simpler) terms of itself, directly or indirectly; computational model requires a stack on which to save information

Language Mechanisms

- Concurrency: parts of a program to be executed at the same time, either partially or in full
- Exception handling and speculation: program fragment executed with the knowledge and possibility that something could go wrong; mechanisms to handle the wrong part; state back-tracking; different forms of speculation

Symbol Table

- We will implement it as an array of symbols
- Each symbol will have:
 - Identifier name
 - Identifier type
 - Identifier width
 - Value
- Will assume that we have a function: `maketemp (table);`
- Our simple compiler will only support int

Instruction Table

- A list / table / array of instructions
- Each instruction will have some number of fields (for us):
 - operator
 - operand 1
 - operand 2
 - operand 3
- If using 3 operands, it's called 3-address form (standard terminology)
- Will assume we have a function: `gencode (op, a1, a2, a3)`

Program Execution in our Interpreter

- We distinguish between instructions being generated at compile time and instructions that will be executed (at run-time)
- At compile-time, we need to know how many instructions we have created and/or what is the entry or position of the last instruction generated
- At run-time, execution flows mostly linearly:
 - Execution position defined by the Program Counter (PC)
 - Start with "first" instruction: $PC \leftarrow 0$
 - Statement Sequencing is achieved by executing instructions in the table code, one after another, and incrementing the Program Counter (PC): $PC \leftarrow PC + 1$
 - Control-flow constructs (for, while, repeat, do-while, if-then, if-then-else, switch, try, etc): all manipulate the PC by changing it in non-linear fashion

Instruction Semantics

Operator	a1	a2	a3	Description
OP_ADD	reg0	reg1	reg2	$\text{reg0} \leftarrow \text{reg1} + \text{reg2}$
OP_MUL	reg0	reg1	reg2	$\text{reg0} \leftarrow \text{reg1} * \text{reg2}$
OP_UMIN	reg0	N/A	reg2	$\text{reg0} \leftarrow -\text{reg2}$
OP_LT	reg0	reg1	reg2	$\text{reg0} \leftarrow \text{reg1} < \text{reg2}$
OP_JMP	N/A	N/A	instr	$\text{pc} \leftarrow \text{instr}$
OP_JNZ	cond	N/A	instr	If (cond = True) $\text{pc} \leftarrow \text{instr}$
OP_JZ	cond	N/A	instr	If (cond = False) $\text{pc} \leftarrow \text{instr}$
OP_PUSHA	arg			
OP_CALL	new_instr	Ret	Old_instr	$\text{sp} \leftarrow \text{sp} + \text{something};$ $\text{pc} \leftarrow \text{new_instr}$
OP_LOAD	reg		addr	$\text{reg} \leftarrow \text{M}[\text{addr}]$
OP_STORE	addr		reg	$\text{M}[\text{addr}] \leftarrow \text{reg}$

Sequencing

- Essentially, list of statements:
 - $S_1; S_2; S_3; \dots; S_n$
- Effect: S_1 executes first, followed by S_2 , S_3 , and so on
- Compilers will exploit lack of dependencies (information flow) between pairs of statements (e.g. S_i and S_j) to reorder them in a legal fashion → similar to out-of-order (OoO) execution
- Imperative languages would provide dedicated delimiters: $\{/\}$, begin/end
- List of statements surrounded by delimiters = block or compounded statement

Short-Circuit Evaluation

Consider:

- `if (a < b && b < c) { ... }`
- `for (i = 0; i < N && A[i] != NULL; i++)`
- `for (i = 0; A[i] != NULL && i < N; i++)`

Idea: avoid unnecessary work

Problem: some of the leftover work could
desired side effects

Example: `while (a = my_function(x,a) &&
x++) { ... }`

Language dependent

Some languages (I didn't know this) might
have dedicated operators for short-circuit

Ada provides:

- `and` vs `and followed by then`
- `or` vs `or followed by else`

Examples:

- `if (d = 0.0 or else n/d < threshold) then ..`
- `if (p /= null and then p.parent = someval) ...`

Semantically equivalent to nested ifs,
additional logic and control flow

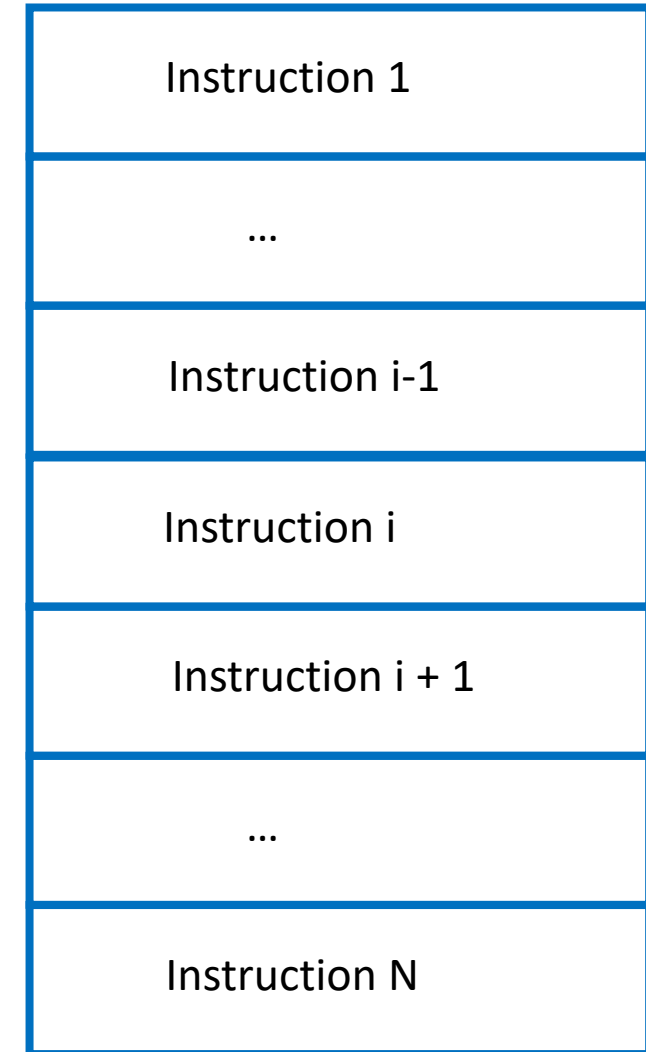
Control-Flow Productions

```
assign ← id ":=" expr ;  
block  ← BEGIN stmt_list END ;  
stmt   ← assign  
        | if  
        | repeat  
        | write  
        | read  
        ;  
stmt_list ← stmt_list ';' stmt  
          | stmt  
          ;
```

```
body ← stmt  
      | block  
      ;  
if   ← IF '(' expr ')'  
      THEN body  
      elsebranch ;  
elsebranch ← ELSE body  
           | ε  
           ;  
repeat ← REPEAT body  
        UNTIL expr ;
```

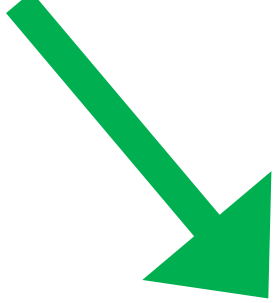
Structured and Unstructured Flow

- In memory, all instructions are stored in contiguous fashion
- *Structured* refers to using control-flow structures (if-then, for, while, do, repeat, case/switch, exceptions, break/exit) → *top-down design*
- Many control-flow constructs were introduced from Algol 60
- *Unstructured* relies on labels and unconditional jumps (as in assembly)
- Label: Identifier serving as placeholder for some instruction position
- Jump: some instruction in the PL that forces continuing execution at the target of the jump
- Some languages do not support unstructured control-flow
- Unstructured flow considered bad (makes code hard to read, maintain, etc)



Repeat Codegen

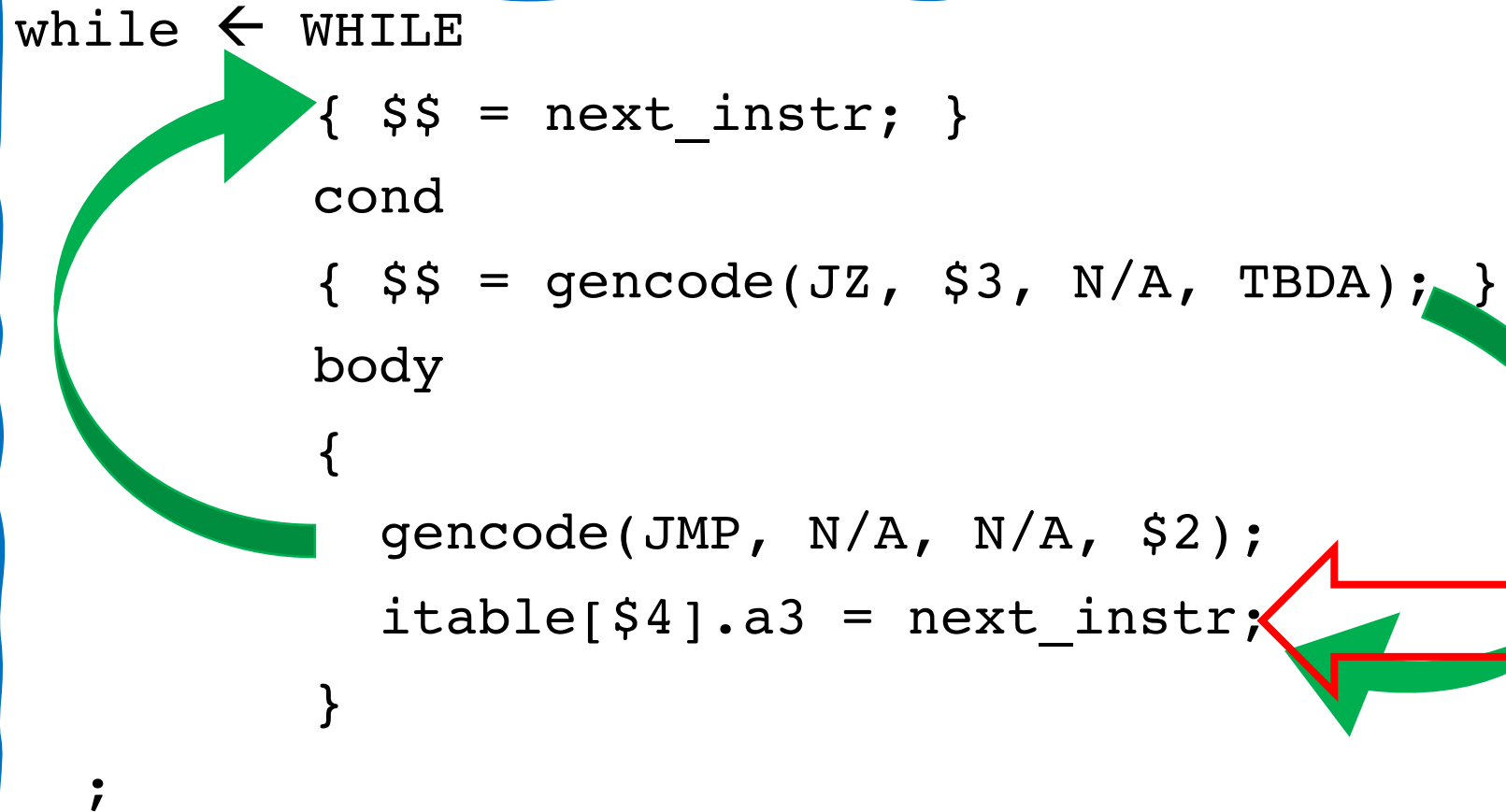
```
repeat ← REPEAT { $$ = next_instr; }  
    body  
    UNTIL  
    expr {  
        $$ = gencode (JZ, expr, N/A, $2);  
    }
```



Target of
jump instruction
is “backward”, so need to
store temporarily the
instruction
destination that will
be generated “later”

While Codegen

```
while ← WHILE
{ $$ = next_instr; }
cond
{ $$ = gencode(JZ, $3, N/A, TBDA); }
body
{
  gencode(JMP, N/A, N/A, $2);
  itable[$4].a3 = next_instr;
}
;
```



- Target of jump instruction is “forward”, so need to store temporarily the instruction entry that will be completed later.
- Unconditional jump to re-evaluate condition

If last_instr is used instead, you will get an infinite loop

IF Codegen

```
if ← IF '(' expr ')' THEN
    { $$ = gencode(JZ, expr, N/A, TBDA); }
    body
    { $$ = gencode (JMP, N/A, N/A, TBDA);
      itable[$6].a3 = next_instr;
    }
    elsebranch
    { itable[$8].a3 = next_instr; }
;
elsebranch ← ELSE body
| ε
;
```

Skip the
true-branch
if cond is false

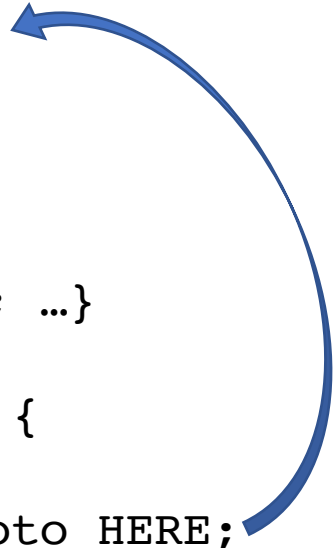
Skip the
false-branch
if cond is True

Targets of both
jump instructions
are “forward”, so need to
store temporarily the
instruction entry that will
be completed later

Structured and Unstructured Flow

- In Basic, the target of the GOTO could also be a line number
- *break*: stops execution of single loop construct in C/C++
- *continue*: skips the remaining instructions in a loop and proceeds with evaluating the condition of the next iteration
- Multi-level returns (MLR): some construct that allows to exit from several function calls
- Unwinding:
 - repair stack, remove corrupted/failed functions, deallocate stack frames
 - A lot of book-keeping: state in registers, fetching access links
- Think how to implement the break and continue constructs in our compiler

```
function f1 () {  
    ...  
    call f2 ();  
    ...  
    label HERE;  
    ...  
}  
  
function f2 ()  
{ ... call f3 (); ...}  
  
function f3 () {  
    ...  
    if (error) goto HERE;  
    ...  
}
```



Structured and Unstructured Flow

- Exception handling:
 - dangerous code + repair code
 - Internally, not very different from a switch or for
 - Several implicit jumps, depending on what happens, where, and when → control transfer
- Similarities between MLR and structured exceptions:
 - Control transfer from inner to outer context
 - Unwinding stack (functions that failed)
- Distinction between (MLR) and structured exceptions:
 - Completion of task: success for MLR, failure for exceptions

```
try {  
    // dangerous code  
} catch (ExceptionType e1) {  
    // do something  
} catch (ExceptionType e2) {  
    // some other error  
}  
[ finally {  
    // optional  
    // code to always executed  
}  
]
```


Expression Evaluation

- An expression is either a simple object (literal or variable) or some function of these
- For the latter, we use the terms *operators* and *operands*
- Languages provide "simple", pre-built math functions via operators
- Operators are applied to operands
- FYI: "simple" is relative; compare C vs. Python
- Some languages could have more than a single name for an operator, and rely on syntactic sugar to simplify writing; examples:
 - Ada: $a + b$ is short for $+(a, b)$
 - C++: $a+b$ is short for $a.operator+(b)$ or $operator+(a, b)$
 - Other (some language): $0 \leq i, j < N$ could be short for $0 \leq i < N$ and $0 \leq j < N$

Expression Evaluation

- Language defines the operator notation: infix, prefix, postfix
 - prefix: **op** a b, or **op**(a,b) or (**op** a b)
 - infix: a **op** b
 - postfix: a b **op**
- Most imperative languages use infix notation for binary operators and prefix notation for unary ones
- Lisp uses prefix notation for all functions, in Cambridge Polish notation:
 - `(* (+ 1 3) 2) == (1 + 3)*2`
 - `(append a b c my_list)`
- ML-family languages avoid parentheses, except for disambiguation:
 - `max (2 + 3) 4;`

Precedence and Associativity

- In what order should operators be evaluated?
- Example with Fortran: $a + b * c ** d ** e / f$
- Choices:
 - i. $((((a + b) * c) ** d) ** e) / f$ or
 - ii. $a + (((b * c) **) ** (e/f))$
 - iii. $a + ((b * (c ** (d ** e))) / f)$
- Fortran opts for the last one, exponentiation being right associative, and having higher precedence than multiplicative operators
- Recall:
 - Precedence: in which order should operators of different categories be evaluated, e.g. $\{+, -\}$ and $\{*, /\}$
 - Associativity: in what order should operations in the same category be evaluated, i.e. left-to-right or right-to-left?

Precedence and Associativity

- I like this statement from the book:

“The precedence structure of C (... of its descendants, C++, Java, C#) is substantially richer than that of most other languages ...”
- And this other statement:

“It is probably fair to say that most C programmers do not remember all of their language’s precedence levels.”
- When in doubt: consult the language reference or add parentheses
- Now, compare with Pascal:

if (A < B and C < D) then (* good luck *)

Precedence and Associativity

- Rules for basic arithmetic operators are mostly standard and uniform across languages, i.e. associate left-to-right
- Example 1: $9 - 3 - 2 == 4$, not 8
- Example 2 (Fortran): $4 ** 3 ** 2 == 4 ** 9$ and not $256 ** 2$
- Example 3 (Ada): exponentiation does not associate, so explicit parenthesis are necessary: $(4 ** 3) ** 2$ or $4 ** (3 ** 2)$
- Example 4 (C): multiple assignments as " $a = b = a + c$ ", obviously is right associative

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /=, <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		(bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		(logical or)	
.eqv., .neqv. (logical comparisons)		?: (if...then...else)	
		=, +=, -=, *=, /=, %= >>=, <<=, &=, ^=, = (assignment)	
		, (sequencing)	

Taken from the book
“Programming
Language Pragmatics”
by Michael L. Scoot

Assignments

- Pure functional languages:
 - rely solely on expressions
 - computation = expression evaluation
 - complex computations employ recursion to generate values, expressions and contexts
- Imperative programming:
 - Attempts to model memory state
 - Sequences of changes/updates on/to memory
 - Assignment = abstraction for memory write
 - Assignments = {value, references/address} → value stored to address

Assignments

- Side effect: influence of a program construct on subsequent computations
- Best example, the assignment in imperative languages, two parts:
 - right-hand side evaluation
 - more importantly, writing the result of the rhs expression to the address of the variable on the lhs
- Some PL distinguish between expressions and statements:
 - expressions **may** have side effects, e.g. `a = ++c * 2`
 - statements used to provide side effects
- Purely functional languages: no side effects, value of expression depends only on the referencing environment

References and Values

- Consider the following assignments in C:
 - $d = a;$
 - $a = b + c;$
- RHS: value
- LHS: location / address, where to store something
- Variables are named containers for values
- Distinction between l-values (address) and r-values (proper values, which could be addresses)
- Bunch of subtle rules:
 - not all expressions can be l-values \rightarrow why?
 - Compare two (potential) assignments: $2 + 3 = a$ and $a = 2 + 3$, when will these be valid (if ever)
 - Not all l-values are simple names, consider:
 - $(f(a)+3) \rightarrow b[c] = 2;$

References and Values

- Value model vs reference model
- Value model: values flow from rhs to lhs
- Reference model:
 - All variables are l-values
 - Variables automatically dereferenced when they appear in r-value contexts (equivalent to doing `*variable` in C)
 - Becomes important to distinguish 2 cases:
 - a. variables that refer to the same object
 - b. variables that refer to different objects, but with the same value

```
b := 2;  
c := b;  
a := b + c;
```

Pascal and Clu example

C example: `int a; int * ptr;`



Java `==` and `equals`

References and Values

Reference Model

$a := b + c$; // Apparently as in Clu

a , b and c are essentially addresses

Actions for the above:

1. Get the value stored in the address of b
2. Get the value stored in the address of c
3. Compute new value
4. Store the value in the address of a

Value Model

Consider: $a = b + c$; // in C

Variables in rhs have values associated to them, so fetching their value is automatic

Actions for the above:

1. Get the address associated to b
2. Dereference the address of b and get the value stored there
3. Repeat the above for c
4. Compute new value
5. Store value at the address of a (lhs)

Read page 232, example 6.16: Boxing

Assignment Operators

- Assignment operators modify memory:
 - +=
 - *=
 - op=
- Advantage:
 - address calculation performed just once
 - simplifies code (we write a lot of something = something + somethingelse)
- C provides an assignment operator for each of its binary arithmetic and bit-wise operators, for a total of 10
 - also prefix and postfix [in|de]crements: var++, ++var, var-- and --var
- Prefix form: syntactic sugar for += and -=

Assignment Operators

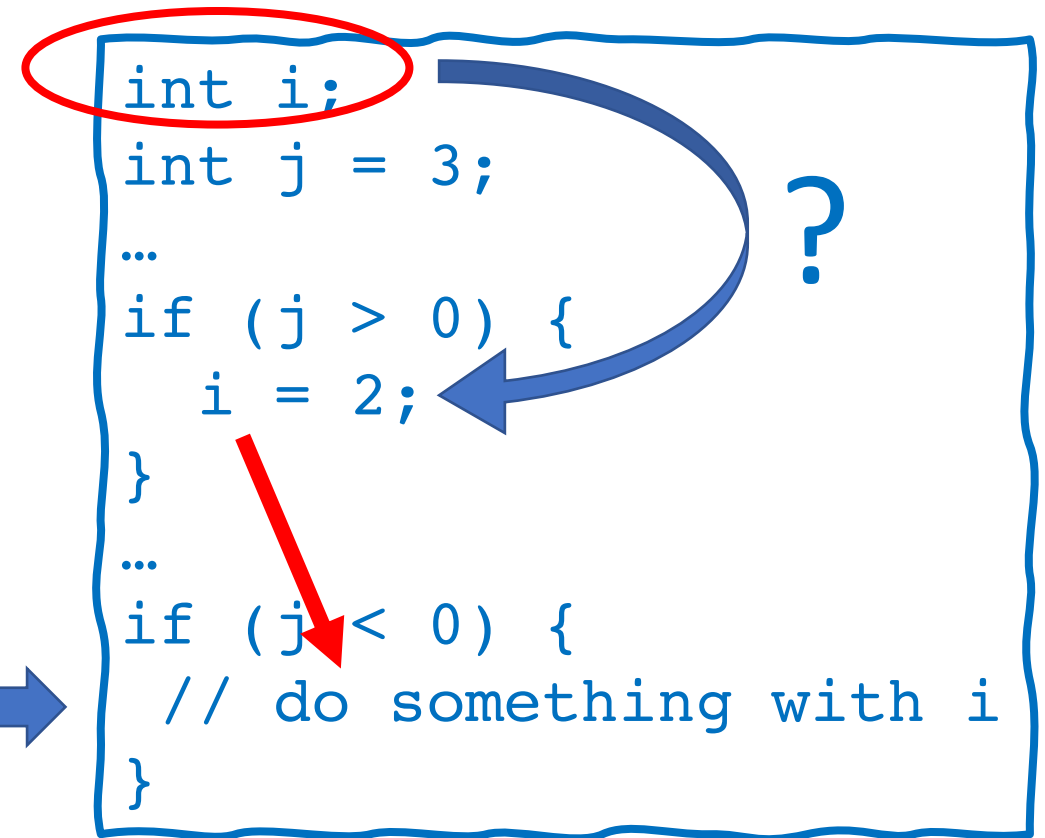
- Postfix form: NOT syntactic sugar, i.e. has different semantics. Consider:
 `*p++ = *q++;`
- The above copies values from q to p, then advances both pointers

Initialization

- Not all (imperative) languages provide mechanisms for declaration + initialization (a la c
→ `int c = (2 * something);`)
- Useful to initialize variables:
 - static variables local to subroutines need an initial value (usually 0)
 - Initialized static variables can use global memory
 - Avoid computational errors
- Most languages will have mechanisms to initialize variables for pre-built datatypes
- Special mechanisms for “aggregate” types, i.e. arrays, structures
- Initialization saves time only for statically allocated variables, not for stack variables nor for heap variables

Definite Assignment

- Uses the (static) control flow of the program
- Conservative analysis
- Considers every possible execution path in the program
- Languages like java use this
- Example:



```
int i;  
int j = 3;  
...  
if (j > 0) {  
    i = 2;  
}  
...  
if (j < 0) {  
    // do something with i  
}
```

Ordering in Expressions

- Precedence and associativity define order in which binary infix operators are evaluated
 - Do not necessarily specify orders in which operands are evaluated
- Example 1: $a - g(b) - c * d$
- Equivalent to : $(a - g(b)) - (c * d)$
- But: which one is evaluated first: $(a - g(b))$ or $(c * d)$
- Example 2: $f(a, g(b), h(c))$
- In which order are the arguments evaluated?
- This is important because of:
 - side effects
 - Code speed
- Usually this is “implementation dependent”
- Java and C# evaluate arguments from left to right