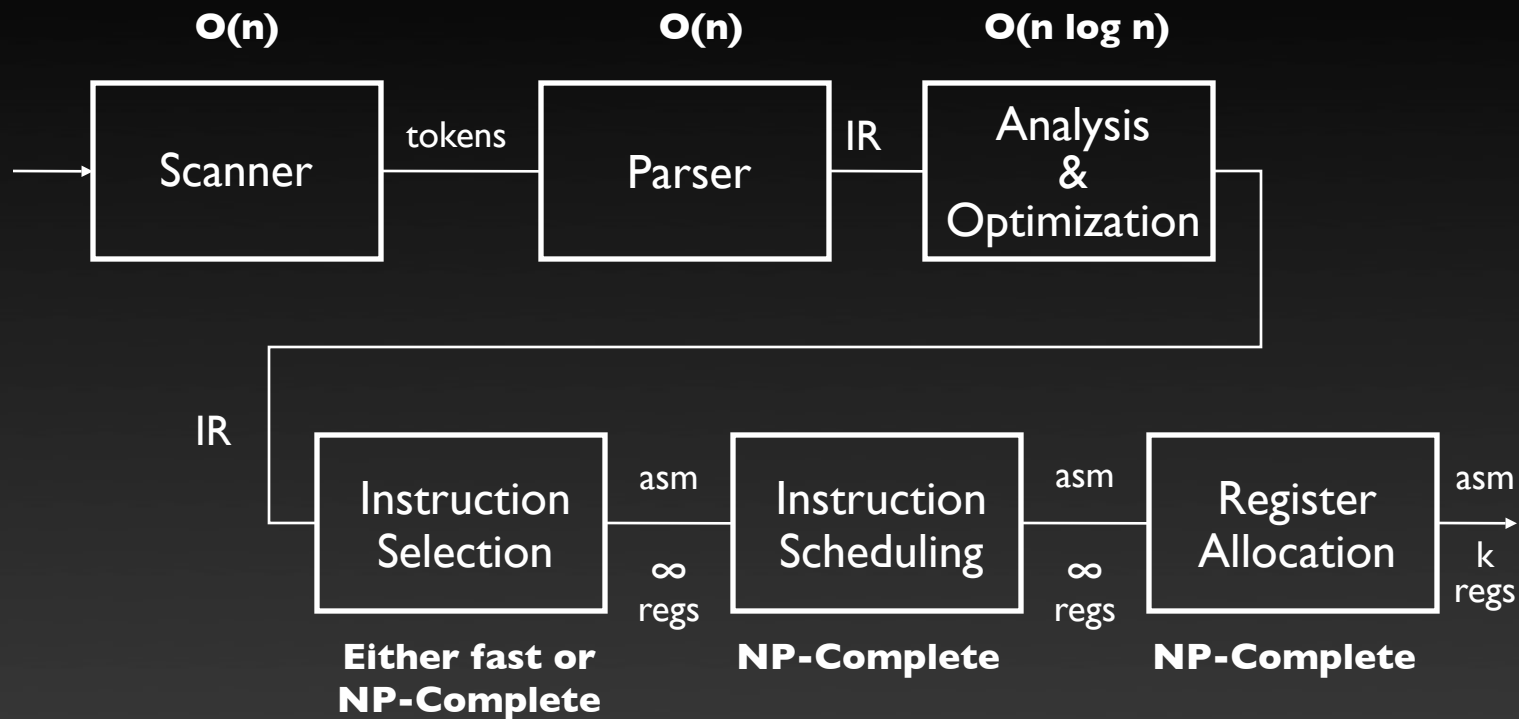


Code Generation

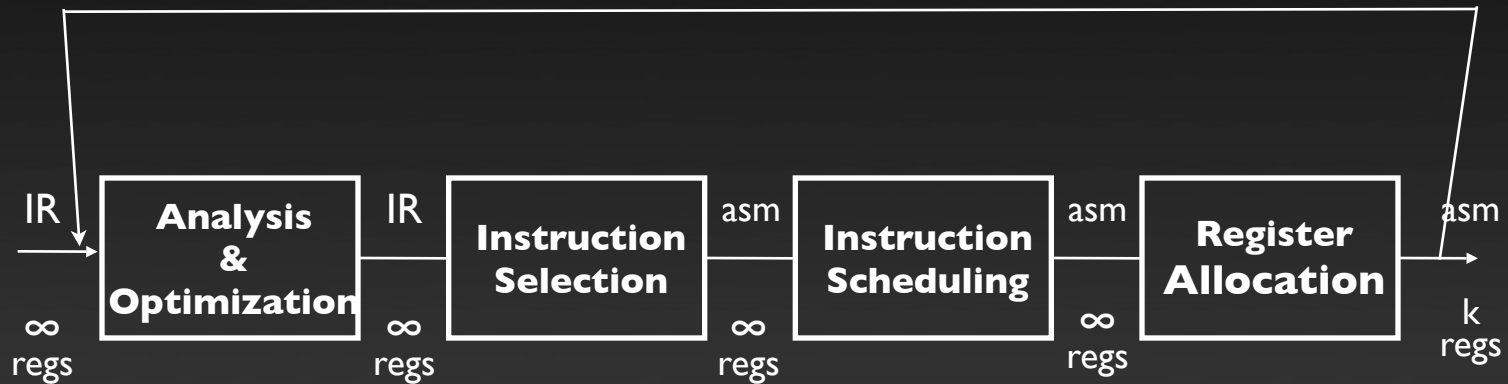
Structure of a Compiler



- A compiler is several quick stages followed by some hard problems
- The harder parts are mostly in *code generation* and *optimization*

Structure of a Compiler

For the rest of the semester, we assume the following model:



- Selection is fairly simple (“solved” in the 1980s)
- Allocation & scheduling are complex
- May “loop” indefinitely ...

Definitions

Instruction selection

- Mapping IR into assembly code
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

Instruction scheduling

- Reordering operations to hide latencies
- Assumes a fixed program
- Changes demand for registers

Register allocation

- Deciding which values will reside in registers
- Changes the storage mapping
- Concerns about placement of data & memory operations

The Big Picture

How hard are these problems?

Instruction selection

- Can make locally optimal choices, with automated tool
- Global optimality is NP-Complete

Instruction scheduling

- Single basic block \Rightarrow heuristics work quickly
- General problem, with control flow \Rightarrow NP-Complete

Register allocation

- Single basic block, no “spills”, & 1 register size \Rightarrow linear time
- Whole procedure is also NP-Complete

The Big Picture

These problems are usually solved independently and sub-optimally

Instruction selection

- Use some form of pattern matching
- Assume enough registers or target “important” values

Instruction scheduling

- Within a block, “list scheduling” is “close” to optimal
- Across blocks, build framework and apply “list scheduling”

Register allocation

- Start from virtual registers & map “enough” into k registers
- With targeting, focus on good priority heuristic

Code Shape

Definition

- All those nebulous properties of the code that impact performance & code “quality”
- Includes code, approach for different constructs, cost, storage requirements & mapping, and choice of operations
- Code shape is the end product of many decisions

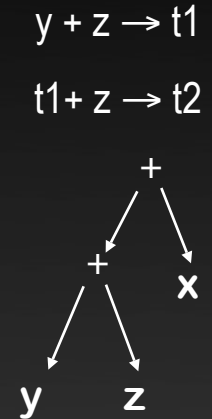
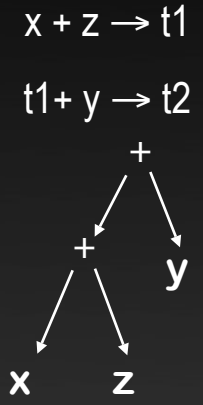
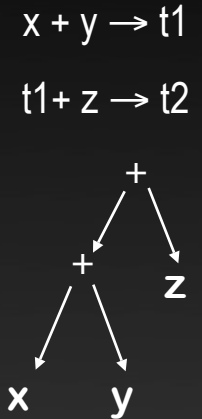
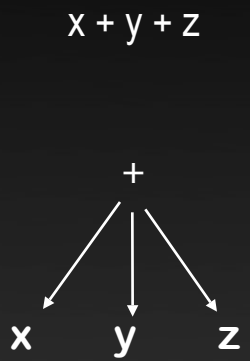
Impact

- Code shape influences algorithm choice & results
- Code shape can encode important facts, or hide them

Rule of thumb: expose as much derived information as possible

- Example: explicit branch targets in LLOC simplify analysis
- Example: hierarchy of memory operations in LLOC

Code Shape



- What if $y+z$ is evaluated earlier?

The “best” shape for $x+y+z$ depends on contextual knowledge

- There may be several (possibly conflicting) options

Code Shape

Another example -- the case statement

- Implement it as cascaded if-then-else statements
 - Cost depends on where your case actually occurs
 - $O(\text{number of cases})$
- Implement it as a binary search
 - Need a dense set of conditions to search
 - Uniform ($\log n$) cost
- Implement it as a jump table
 - Lookup address in a table & jump to it
 - Uniform (constant) cost

Compiler must choose best implementation strategy

No amount of transforming will convert one into another

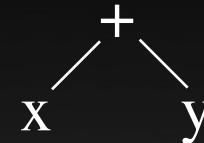
Generating Code for Expressions

- The key code quality issue is holding values in registers
- When can a value be *safely* allocated to a register?
 - When only 1 name can reference its value
 - Pointers, parameters, aggregates & arrays all cause trouble
- When should a value be allocated to a register?
 - When it is both *safe* & *profitable*
- Encoding this knowledge into the IR is important
- Use code shape to make it known to every later phase
- Assign a virtual register to *anything* that can go into one
- Load or store the others at each (de)reference
- ILOC has a hierarchy of loads & stores
- Relies on a strong register allocator being used later

Generating Code for Expressions

```
expr(node) {  
  int result, t1, t2;  
  switch (type(node)) {  
    case x, ÷, +, - :  
      t1 ← expr(left child(node));  
      t2 ← expr(right child(node));  
      result ← NextRegister();  
      emit (op(node), t1, t2, result);  
      break;  
    case IDENTIFIER:  
      t1 ← base(node);  
      t2 ← offset(node);  
      result ← NextRegister();  
      emit (loadAO, t1, t2, result);  
      break;  
    case NUMBER:  
      result ← NextRegister();  
      emit (loadI, val(node), none, result);  
      break;  
  }  
  return result;  
}
```

Example:



Produces:

expr("x") →

loadI @x ⇒ r1

loadAO r0, r1 ⇒ r2

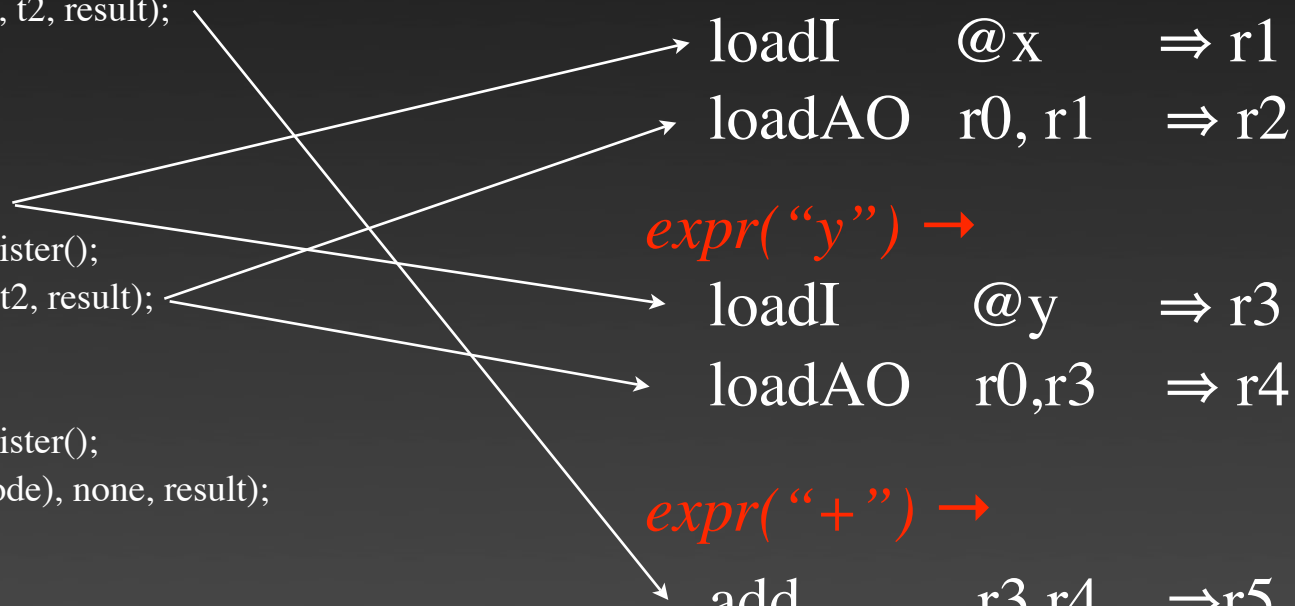
expr("y") →

loadI @y ⇒ r3

loadAO r0, r3 ⇒ r4

expr("+") →

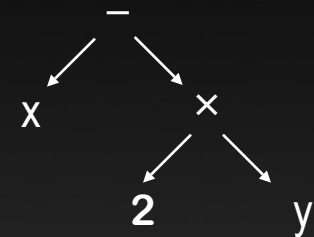
add r3, r4 ⇒ r5



Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
    case ×, ÷, +, - :
      t1 ← expr(left child(node));
      t2 ← expr(right child(node));
      result ← NextRegister();
      emit (op(node), t1, t2, result);
      break;
    case IDENTIFIER:
      t1 ← base(node);
      t2 ← offset(node);
      result ← NextRegister();
      emit (loadAO, t1, t2, result);
      break;
    case NUMBER:
      result ← NextRegister();
      emit (loadI, val(node), none, result);
      break;
  }
  return result;
}
```

Example:



Generates:

loadI	@x	⇒ r1
loadAO	r0, r1	⇒ r2
loadI	2	⇒ r3
loadI	@y	⇒ r4
loadAO	r0, r4	⇒ r5
mult	r3, r5	⇒ r6
sub	r2, r6	⇒ r7

Extending the Simple Treewalk Algorithm

More complex cases for IDENTIFIER

- What about values in registers?
 - Modify the IDENTIFIER case
 - Already in a register \Rightarrow return the register name
 - Not in a register \Rightarrow load it as before *and* record the fact
- What about parameter values?
 - Many linkages pass the first several values in registers
 - Call-by-value \Rightarrow just a local variable with “reverse” offset
 - Call-by-reference \Rightarrow needs an extra indirection
- What about function calls in expressions?
 - Generate the calling sequence & load the return value
 - Severely limits compiler’s ability to reorder operations

Extending the Simple Treewalk Algorithm...

Adding other operators

- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls
- Handle assignment as an operator

Mixed-type expressions

- Insert conversions as needed from conversion table
- Most languages have symmetric & rational conversion tables

Typical
Addition
Table

+	Integer	Real	Double	Complex
Integer	--	Real	Double	Complex
Real	Real	--	Double	Complex
Double	Double	Double	--	Complex
Complex	Complex	Complex	Complex	Complex

Extending the Simple Treewalk Algorithm

Operation evaluation order:

- Can use commutativity & associativity to improve code
- This problem is very hard - often ignored

Operand evaluation:

- 1st operand must be preserved while 2nd is evaluated
- Requires extra register while evaluating 2nd operand
- Should evaluate more demanding operand expression first

Handling Assignment

- really just another operator ...

$lhs \leftarrow rhs$

lvalue \leftarrow *rvalue*

Strategy:

- Evaluate **rhs** to a *value*
- Evaluate **lhs** to a *location*
 - *lvalue* is a register \Rightarrow move rhs
 - *lvalue* is an address \Rightarrow store rhs
- If *rvalue* & *lvalue* have different types
 - Evaluate *rvalue* to its “natural” type
 - Convert that value to the type of *lvalue*

How does the compiler handle $A[i,j]$?

Row-major order

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

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

Vector of pointers to (pointers to ...) values

Takes much more space, trades indirection for arithmetic

Not easily amenable to analysis

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

Column-major order

A

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

Each has a distinct cache behavior

Indirection vectors



Computing an Array Address

$A[i]$

- $@A + (i - \text{low}) \times \text{sizeof}(A[1])$
- In general: $\text{base}(A) + (i - \text{low}) \times \text{sizeof}(A[1])$

Computing an Array Address

$A[i]$

- $@A + (i - \text{low}) \times \text{sizeof}(A[1])$
- In general: $\text{base}(A) + (i - \text{low}) \times \text{sizeof}(A[1])$

`int A[1:10]` \Rightarrow low is 1
Make low 0 for faster
access (saves a $-$)

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

Computing an Array Address

$A[i]$

- $@A + (i - \text{low}) \times \text{sizeof}(A[1])$
- In general: $\text{base}(A) + (i - \text{low}) \times \text{sizeof}(A[1])$

This stuff looks expensive!
Lots of implicit +, -, x ops

What about $A[i_1, i_2]$?

Row-major order, two dimensions

$$@A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times \text{sizeof}(A[1])$$

Column-major order, two dimensions

$$@A + ((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{low}_1) \times \text{sizeof}(A[1])$$

Indirection vectors, two dimensions

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

Optimizing Address Calculation for $A_{[i,j]}$

In row-major order

where $w = \text{sizeof}(A[l,l])$

$$@A + (i - \text{low}_1)(\text{high}_2 - \text{low}_2 + 1) \times w + (j - \text{low}_2) \times w$$

Which can be factored into

$$\begin{aligned} & @A + i \times (\text{high}_2 - \text{low}_2 + 1) \times w + j \times w \\ & - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w) + (\text{low}_2 \times w) \end{aligned}$$

If low_1 , high_1 , and w are known, the last term is a constant

Define $@A_0$ as

$$@A - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w + \text{low}_2 \times w)$$

And len_2 as $(\text{high}_2 - \text{low}_2 + 1)$

Then, the address expression becomes

$$@A_0 + (i \times \text{len}_2 + j) \times w$$

Compile-time constants



Array References

What about arrays as actual parameters?



Whole arrays, as call-by-reference parameters

- Need dimension information \Rightarrow build a *dope vector*
- Store the values in the calling sequence
- Pass the address of the dope vector in the parameter slot
- Generate complete address polynomial at each reference

Some improvement is possible

- Save len_i and low_i rather than low_i and $high_i$
- Pre-compute the fixed terms in prologue sequence

What about call-by-value?

- Most c-b-v languages pass arrays by reference
- This is a language design issue

Array Dereferences as Parameters

What about $A[12]$ as an actual parameter?

If corresponding parameter is a scalar, it's easy

- Pass the address (by reference) or value (by value)
- Language definition must force this interpretation

What if corresponding parameter is an array?

- Must know about both formal & actual parameter
- Meaning must be well-defined and understood

⇒ Again, we're treading on language design issues

Example: Array Address Calculations in a Loop

```
DO J = 1, N  
  A[I,J] = A[I,J] + B[I,J]  
END DO
```

- **Naïve:** Perform the address calculation twice

```
DO J = 1, N  
  R1 = @A0 + (J × len1 + 1) × floatsize  
  R2 = @B0 + (J × len1 + 1) × floatsize  
  MEM(R1) = MEM(R1) + MEM(R2)  
END DO
```

Example: Array Address Calculations in a Loop

```
DO J = I, N
  A[I,J] = A[I,J] + B[I,J]
END DO
```

- **Sophisticated:** Move common calculations out of loop

```
R1 = I x floatsize
c = len1 x floatsize // Compile-time constant
R2 = @A0 + R1
R3 = @B0 + R1
DO J = I, N
  a = J x c
  R4 = R2 + a
  R5 = R3 + a
  MEM(R4) = MEM(R4) + MEM(R5)
END DO
```

Example: Array Address Calculations in a Loop

```
DO J = 1, N
  A[I,J] = A[I,J] + B[I,J]
END DO
```

- **Very sophisticated:** Convert multiply to add ...

$R1 = I \times \text{floatsize}$

$c = \text{len}_1 \times \text{floatsize}$! Compile-time constant

$R2 = @A_0 + R1$

$R3 = @B_0 + R1$

```
DO J = 1, N
```

$R2 = R2 + c$

$R3 = R3 + c$

$\text{MEM}(R2) = \text{MEM}(R2) + \text{MEM}(R3)$

```
END DO
```

Boolean & Relational Values

How should the compiler represent them?

- Answer depends on the target machine

Two classic approaches

- Numerical representation
- Positional (implicit) representation

Correct choice depends on both:

- context in source code program
- instruction support in target architecture

Boolean & Relational Values

Numerical representation

- Assign values to TRUE and FALSE
- Use hardware AND, OR, and NOT operations
- Use comparison to get a boolean from a relational expression

Examples

$x < y$

cmp_LT $r_x, r_y \Rightarrow r_1$

if ($x < y$)
then stmt₁
else stmt₂

cmp_LT $r_x, r_y \Rightarrow r_1$
cbr $r_1 \rightarrow \text{stmt}_1, \text{stmt}_2$

Boolean & Relational Values

Most target architecture ISA's use a *condition code* ...

- Must use a conditional branch with result of compare
- Necessitates branches in the evaluation

Example:

	cmp	$r_x, r_y \Rightarrow cc_1$
	cbr_LT	$cc_1 \rightarrow L_T, L_F$
$x < y$	L_T :loadl	$1 \Rightarrow r_1$
	br	L_E
	L_F :loadl	$0 \Rightarrow r_1$
		L_E : <i>next statement</i>

This “positional representation” is much more complex

Boolean & Relational Values

The last example actually encodes result in the PC

If result is used to control an operation, this may be enough

Example
if ($x < y$) then $a \leftarrow c + d$ else $a \leftarrow e + f$

Condition Codes		Boolean Compares	
	cmp $r_x, r_y \Rightarrow cc_1$		cmp_LT $r_x, r_y \Rightarrow r_1$
	cbr_LT $cc_1 \rightarrow L_1, L_2$		cbr $r_1 \rightarrow L_1, L_2$
L ₁ :	add $r_c, r_d \Rightarrow r_a$	L ₁ :	add $r_c, r_d \Rightarrow r_e$
	br $\rightarrow L_{out}$		br $\rightarrow L_{out}$
L ₂ :	add $r_e, r_f \Rightarrow r_a$	L ₂ :	add $r_e, r_f \Rightarrow r_a$
	br $\rightarrow L_{out}$		br $\rightarrow L_{out}$
L _{out} :	nop	L _{out} :	nop

Condition code version does not directly produce ($x < y$)

Boolean version does

Still, there is no significant difference in the code produced

Boolean & Relational Values

Consider the assignment $x \leftarrow a < b \wedge c < d$

Condition Codes	Boolean Compares
cmp $r_a, r_b \Rightarrow cc_1$ cbr_LT $cc_1 \rightarrow L_1, L_2$ L ₁ : cmp $r_c, r_d \Rightarrow cc_2$ cbr_LT $cc_2 \rightarrow L_3, L_2$ L ₂ : loadl $0, r_x$ br $\rightarrow L_{out}$ L ₃ : loadl $1, r_x$ br $\rightarrow L_{out}$ L _{out} : nop	cmp_LT r_a, r_b, r_1 cmp_LT r_c, r_d, r_2 and r_1, r_2, r_x

Here, the boolean compare produces much simpler code

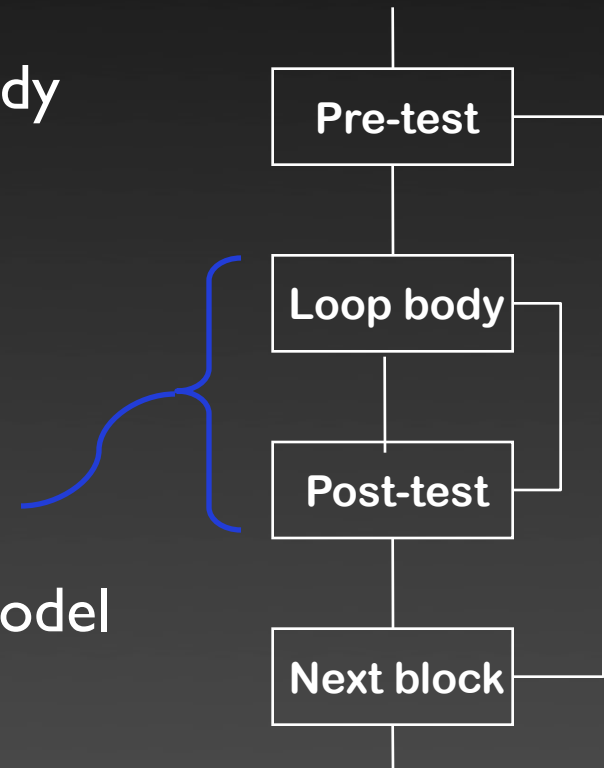
Control Flow

Loops

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

Merges test with last block of loop body

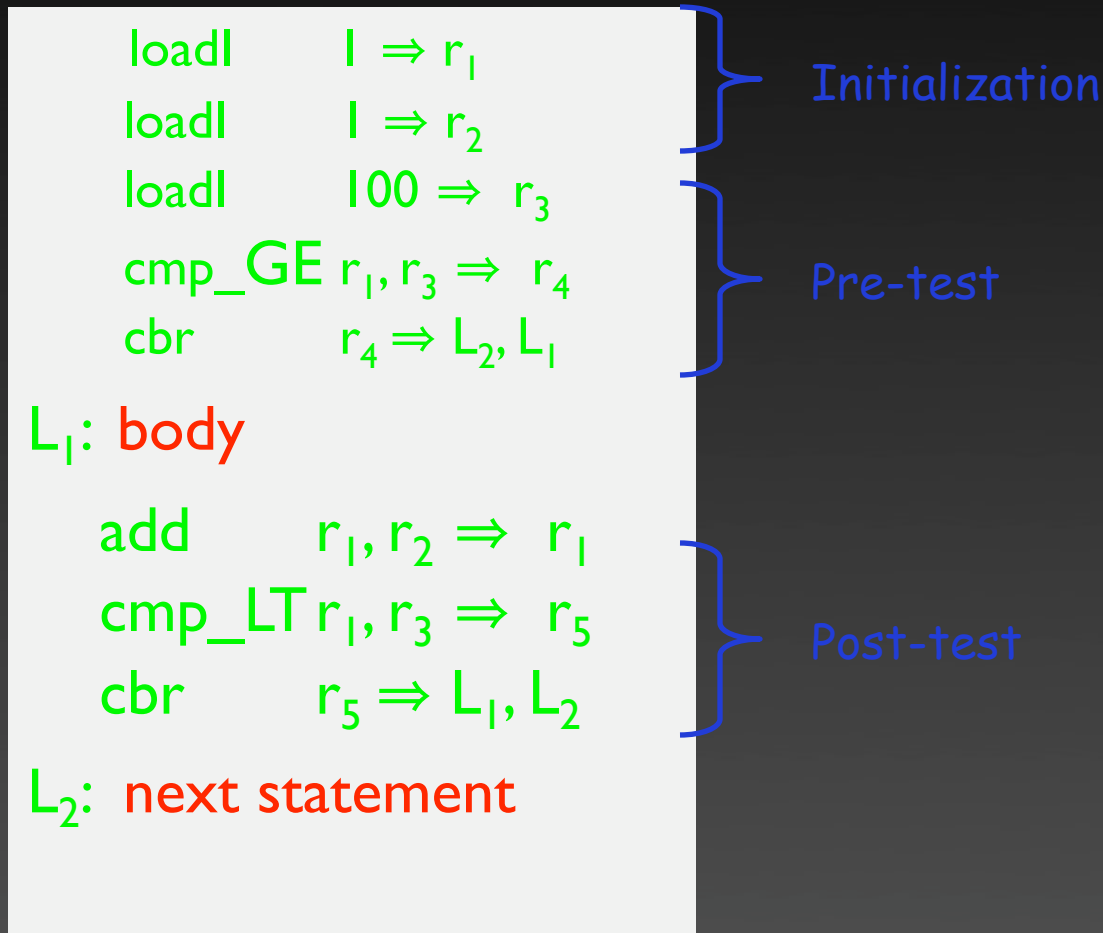
while, for, do, & until all fit this basic model



Loop Implementation Code

```
for (i = 1; i < 100; i++) { body }
```

next statement



Break statements

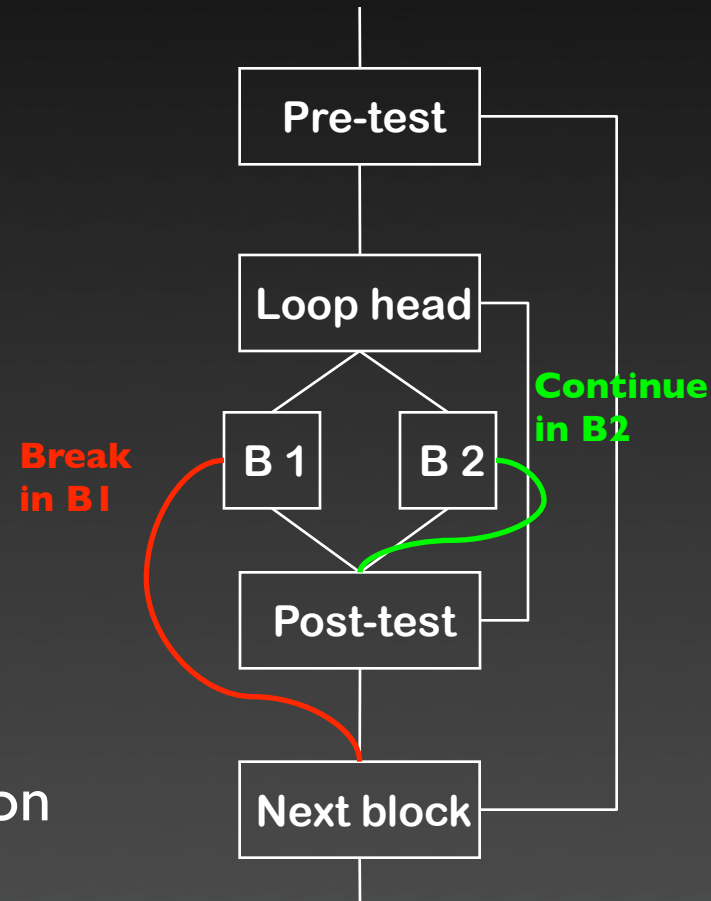
Many modern programming languages include a **break**

- Exits from the innermost control-flow statement
 - Out of the innermost loop
 - Out of a case statement

Translates into a jump

- Targets statement outside control-flow construct
- Creates multiple-exit construct

Continue in loop goes to next iteration



Control Flow

Case Statements

- 1 Evaluate the controlling expression
- 2 Branch to the selected case
- 3 Execute the code for that case
- 4 Branch to the statement after the case

Parts 1, 3, & 4 are well understood, part 2 is the key

Control Flow

Case Statements

- 1 Evaluate the controlling expression
- 2 Branch to the selected case
- 3 Execute the code for that case
- 4 Branch to the statement after the case

Parts 1, 3, & 4 are well understood, part 2 is the key

Strategies

- Linear search (nested if-then-else constructs)
- Build a table of case expressions & binary search it
- Directly compute an address (requires dense case set)