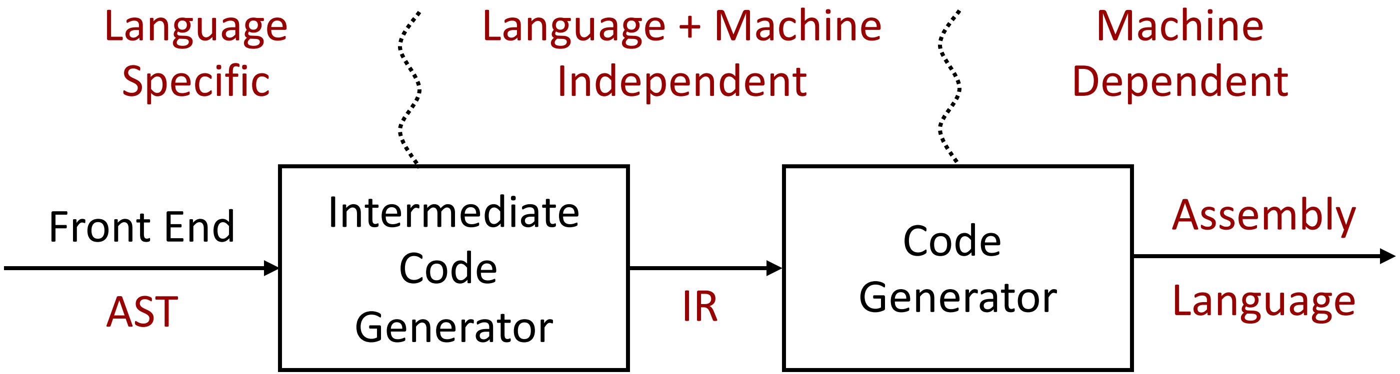
IR

# Intermediate Representation

CMPT 379: Compilers
Instructor: Anoop Sarkar
anoopsarkar.github.io/compilers-class

# Intermediate Representation




Provides an intermediate level of abstraction

- More details than source (programming language)
- Fewer details than target (assembly language)

# IR: 3-Address Code

- High level assembly
- Instructions that operate on named locations and labels
- Locations
  - Each location is some place to store 4 bytes
    - Pretend we can make infinitely many of them
  - Or global variable
    - Referred to by global name
- Labels (you generate as needed)
- 3-address code = **at most** three addresses/locations in each instructions

# IR: 3-Address Code

- Address or locations:
  - Names/Labels
  - Constants
  - Temporaries

# IR: 3-Address Code

For simplicity, in these slides we will omit the type of each address/location. In real IR (like LLVM) the type and alignment for each location has to be defined.

- Instructions:
  - assignments:
    - `x = y op z` (op: binary arithmetic or logical operation)
    - `x = op y` *(op: unary operation)*
  - copy: `x = y`
  - unconditional jump:
    - `goto L` *(L is a symbolic label of a statement)*
  - conditional jumps:
    - `if x goto L`
    - `IfFalse x goto L`
    - `if x relop y goto L` *(relop: relation operator: <,==,<=)*

# IR: 3-Address Code

Instructions:

- Function/procedure calls: p(x1,x2,…,xn)
  - `param x1`
  - `param x2`
  - …
  - `param xn`
  - `y=call p, n`
- Return statement:
  - `return y`
- You can save the return: `y=call p()` or if it is a void return: `call p()`

In LLVM:

```
r = call <return-type> p(<type> x1, <type> x2, …, <type> xn)
```

# IR: 3-Address Code

Instructions:

- Indexed assignments (Arrays):
  - `x = y[i]`
  - `x[i] = y`
- Address assignments:
  - `x = &y` (which sets x to the location of y)
- Pointer assignments:
  - `x = *y` *(y is a pointer, sets x to the value pointed by y)*
  - `*x = y`

# Basic Blocks and Control Flow

# Basic Blocks

- A *basic block* is a sequence of statements that enters at the start and ends with a branch at the end
- Functions transfer control from one place (the caller) to another (the called function)
- Other examples include any place where there are branch instructions
- Code generation should create code for basic blocks and branch them together

# Control Flow

Consider the statement:

```
while (a[i] < v) { i = i+1; }
```

```
L1:
  t1 = i
  t2 = t1 * 8
  t3 = a[ t2 ]
  t4 = t3 < v
  ifFalse t4 goto L2
  t5 = i
  t5 = t5 + 1
  i = t4
  goto L1
L2: ...
```

Labels can be implemented using position numbers

```
100: t1 = i
101: t2 = t1 * 8
102: t3 = a[ t2 ]
103: t4 = t3 < v
104: ifFalse t4 goto 109
105: t5 = i
106: t5 = t5 + 1
107: i = t5
108: goto 100
109:
```

# Basic Blocks

Consider the statement:

```
while (a[i] < v) { i = i+1; }
```

**Basic Block**

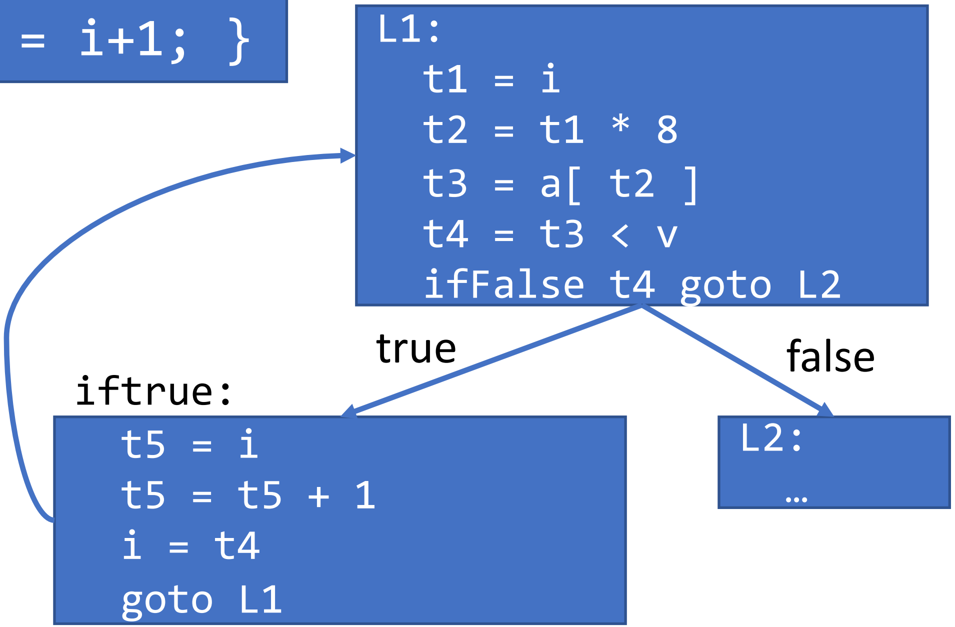
```
L1:
  t1 = i
  t2 = t1 * 8
  t3 = a[ t2 ]
  t4 = t3 < v
  ifFalse t4 goto L2
```

**BB**

```
  t5 = i
  t5 = t5 + 1
  i = t4
  goto L1
```

**BB**

```
L2: ...
```



```
int gcd(int x, int y)
{
    int d;
    d = x - y;
    if (d > 0)
        return gcd(d, y);
    else if (d < 0)
        return gcd(x, -d);
    else
        return x;
}
```

```
gcd:
  t0 = x - y
  d = t0
  t1 = d
  t2 = t1 > 0
  ifFalse t2 goto L0
  t3 = call gcd(d,y)
  return t3
L0:
  t4 = d
  t5 = t4 < 0
  …
```

Avoiding redundant gotos:

```
if t2 goto L1
goto L0
L1: ...
```

# Short-circuiting Booleans

- More complex if statements:
  - if (a or b and not c) { … }
- Typical sequence:

```
t1 = not c
t2 = b and t1
t3 = a or t2
```

- Short-circuit is possible in this case:
  - if (a and b and c) { … }
- Short-circuit sequence:

```
  t1 = a
  if t1 goto L0 /* sckt */
  goto L4
L0:
  t2 = b
  if t2 goto L1
  goto L4
L1:
  t3 = c
…
```

```
void main() {
    int i;
    for (i = 0; i < 10; i = i + 1)
        print(i);
}
```

More Control Flow: for loops

```
main:
  t0 = 0
  i = t0
L0:
  t1 = 10
  t2 = i < t1
  ifFalse t2 goto L1
  call print(i)
  t3 = 1
  t4 = i + t3
  i = t4
  goto L0
L1:
  return
```

# Translation of Expressions

symbol table

```
S → id = E    {$$.code=concat($3.code, $1.lexeme=$3.addr);}
E → E + E     {$$.addr=new Temp(); $$.code=concat($1.code,
              $3.code, $$.addr = $1.addr + $3.addr);}

E → - E       {$$.addr = new Temp(); $$.code = concat($2.code,
              $$.addr = - $2.addr);}

E → ( E )     {$$.addr = $2.addr; $$.code = $2.code;}
E → id        {$$.addr = symtbl($1.lexeme); $$.code = new Code();}
```

# Backpatching in Control-Flow

- Implementing the translations can be done in one or two passes
- The difficulty with code generation in one pass is that we may not know the target label for jump statements
- *Backpatching* allows one pass code generation
  - Generate jump statements with the empty targets (temporarily unspecified)
  - Put each of these statements into a list
  - When the target is known, fill the proper labels in the jump statements (backpatching)
  - In some IR libraries (like LLVM) we can create multiple locations and set them as insert points

# Control Flow using an IR that supports multiple insertion points

New symbol table scope for control flow statements

# Control flow: if statements

Q: What would be the goto target for a break statement in either true: or else: basic block?

```
if (a < b) then i = i+1; else j = i+1;
```

```
func = Builder.GetInsertBlock()->getParent();
BasicBlock::Create(C, "if", func);
CreateBr(IfBB);
```

```
if:
  t0 = a < b
  if t0 goto ???
```

true

```
IfTrueBB = BasicBlock::Create(C, "true", func);
syms.enter_symtbl("true", IfTrueBB);
Builder.SetInsertPoint(IfTrueBB);
```

```
true:
  t1 = 1
  t2 = i + t1
  i = t2
  goto ???
```

goto end

false

```
IfFalseBB = BasicBlock::Create(C, "else", func);
syms.enter_symtbl("else", IfFalseBB);
Builder.SetInsertPoint(IfFalseBB);
```

```
else:
  t1 = 1
  t2 = i+t1
  j = t2
  goto ???
```

```
EndBB = BasicBlock::Create(C, "end", func);
syms.enter_symtbl("end", EndBB);
Builder.SetInsertPoint(EndBB);
```

```
end:
```

IR

# Control flow: while statements

Q: What would be the goto target for a continue statement in the body: basic block?

break;

```
while (i < n) { i = i+1; }
```

```
func = Builder.GetInsertBlock()->getParent();
LoopBB = BasicBlock::Create(C, "loop", func);
CreateBr(LoopBB);
```

i2 = phi(i1,i3)
t0 = i2 < n

```
loop:
  t0 = i < n
  if t0 goto ???
```

true

goto body

```
BodyBB = BasicBlock::Create(C, "body", func);
syms.enter_symtbl("body", BodyBB);
Builder.SetInsertPoint(BodyBB);
```

```
body:
  t1 = 1
  t2 = i + t1
  i = t2
  goto ???
```

goto end

t2 = i2 + t1

i3 = t2

false

```
EndBB = BasicBlock::Create(C, "end", func);
syms.enter_symtbl("end", EndBB);
Builder.SetInsertPoint(EndBB);
```

```
end:
```

# Control flow: for statements

Q: What would be the goto target for a break/continue statement(s) in the body: basic block?

continue;

```
for (i=0; i<n; i=i+1) { c=i; }
```

```
init:
  i = 0
  goto ???
```

```
func = Builder.GetInsertBlock()->getParent();
LoopBB = BasicBlock::Create(C, "loop", func);
CreateBr(LoopBB);
```

```
loop:
  t0 = i < n
  if t0 goto ???
```

true

false

```
EndBB = BasicBlock::Create(C, "end", func);
syms.enter_symtbl("end", EndBB);
Builder.SetInsertPoint(EndBB);
```

```
end:
```

```
BodyBB = BasicBlock::Create(C, "body", func);
syms.enter_symtbl("body", BodyBB);
Builder.SetInsertPoint(BodyBB);
```

```
body:
  c = i
  goto ???
```

goto next

```
NextBB = BasicBlock::Create(C, "next", func);
syms.enter_symtbl("next", NextBB);
Builder.SetInsertPoint(NextBB);
```

```
next:
  t1 = 1
  t2 = i + t1
  i = t2
  goto ???
```

# Backpatching for an IR that only supports line numbers

# Backpatching

If (a < b) then i = i+1; else j = i+1;

```
 99:   t0 = a < b
100:   if t0 goto ???
101:   goto ???
102:   t1 = 1
103:   t2 = i + t1
104:   i = t2
105:   goto ???
106:   t1 = 1
107:   t2 = i+t1
108:   j = t2
109:
```

truelist

falselist

nextlist

backpatch({100}, 102)
backpatch({101}, 106)
backpatch({105}, 109)

# Backpatching

- We maintain a list of statements that need patching by future statements
- Three lists are maintained:
  - truelist: for targets when evaluation is true
  - falselist: for targets when evaluation is false
  - nextlist: the statement that ends the block (also used for loops)
- These lists can be implemented as a synthesized attribute
  - Using marker non-terminals

- S → if "(" B ")" M block

{ backpatch($3.truelist, $5.instr);

$$.nextlist = merge($3.falselist, $6.nextlist);}

- B → E rel E

next instruction number

{ $$.truelist = makelist(nextinstr+1);

$$.falselist = makelist(nextinstr+2);

Code+="c = $1.addr $2.op $3.addr";

Code+="if c goto –";

Code+="goto –"; }

- B → true

{ $$.truelist=makelist(nextinstr); Code+="goto –";}

- B → false

{ $$.falselist=makelist(nextinstr); Code+="goto –";}

- M → ε {$$.instr = nextinstr;}

If (a < b) {i = i+1;}

- 101: c = a < b
- 102: if c goto – 104
- 103: goto – 107
- 104: t1 = 1
- 105: t2 = i+t1
- 106: i = t2
- 107:

B.truelist={102},
B.falselist={103}
M.instr = 104
backpatch({102}, 104)
S.nextlist={103}
backpatch({103}, 107)

- S → while M "(" B ")" M  block

```
{ backpatch($7.nextlist, $2.instr);
  backpatch($4.truelist, $6.instr);
  backpatch($4.falselist, $7.nextlist);
 $$.nextlist = merge($4.falselist; $7.breaklist);
Code+="goto $2.instr";}
```

- S → "break";

```
{ $$.breaklist=makelist(nextinstr); Code+="goto –";}
```

- S → "continue";

```
{ $$.nextlist=makelist(nextinstr); Code+="goto –"; }
```

- B → E rel E  { /* same as previous slide */ }
- block → "{" S "}"

```
{ $$.breaklist=$2.breaklist;
  $$.nextlist=$2.nextlist; }
```

- M → ε  { $$.instr = nextinstr; }

while (i < n) {continue;}

- 101: c = i < n
- 102:  if c goto –  104
- 103:  goto –  105
- 104:  goto –  101
- 105:

```
M($4).instr = 101
B.truelist={102}, B.falselist={103}
M($6).instr = 104
S($3).nextlist=block($7).nextlist={105}
backpatch({102}, 104)
backpatch({103}, 105)
S.nextlist={105}
```

- S $\rightarrow$ while M "(" B ")" M block

  {backpatch($7.nextlist, $2.instr);

  backpatch($4.truelist, $6.instr);

  $$.nextlist = merge($4.falselist; $7.breaklist); }

  Code+="goto $2.instr";}
- S $\rightarrow$ break ;

  {$$.breaklist=makelist(nextinstr); Code+="goto –";}
- S $\rightarrow$ continue ;

  {$$.nextlist=makelist(nextinstr); Code+="goto –";}
- B $\rightarrow$ E rel E { /* same previous slide */ }
- block $\rightarrow$ "{" S "}"

  {$$.breaklist=$2.breaklist;

  $$.nextlist=$2.nextlist;}
- M $\rightarrow$ $\varepsilon$ {$$.instr = nextinstr;}

while (i < n) {break;}

- 101: c = i < n
- 102: if i < n goto – 104
- 103: goto – 105
- 104: goto – 105
- 105: goto 101

M($2).instr = 101
B.truelist={102}, B.falselist={103}
M($6).instr = 104
S($3).breaklist=block.breaklist={104}
backpatch({102}, 104)
S.nextlist={105}
backpatch({104}, 105)

# Array Elements

- Array elements are numbered `0,…,n-1`
- Let `w` be the width of each array element
- Let `base` be the address of the storage allocated for the array
- Then the `i`th element `A[i]` begins in location `base+i*w`
- The element `A[i][j]` with n elements in the 2nd dimension begins at: `base+(i*n+j)*w`

```
void foo(int[] arr)
        { arr[1] = arr[0] * 2 }
```

# Array References

**foo:**

- t0 = 1
- t1 = 4
- t2 = t1 * t0
- t3 = arr + t2
- t4 = *(t3)
- t5 = 0
- t6 = 4
- t7 = t6 * t5
- t8 = arr + t7
- t9 = *(t8)
- t10 = 2
- t11 = t9 * t10
- t4 = t11

Wrong

**foo:**

- t0 = 1
- t1 = 4
- t2 = t1 * t0
- t3 = arr + t2
- t4 = 0
- t5 = 4
- t6 = t5 * t4
- t7 = arr + t6
- t8 = *(t7)
- t9 = 2
- t10 = t8 * t9
- *(t3) = t10

Correct

```
int factorial(int n)
{
  if (n<=1) return 1;
  return n*factorial(n-1);
}

void main()
{
    print(factorial(6));
}
```

```
factorial:
    t0 = 1
    t1 = n lt t0
    t2 = n eq t0
    t3 = t1 or t2
    ifFalse t3 goto end
    t4 = 1
    return t4
end:
    t5 = 1
    t6 = n - t5
    t7 = call factorial (t6)
    t8 = n * t7
    return t8
```

t3 = n <= 1

# Implementing IR

- Quadruples:

```
t1 = - c
t2 = b * t1
t3 = - c
t4 = b * t3
t5 = t2 + t4
a = t5
```

| op | arg1 | arg2 | result |
|---|---|---|---|
| minus | c | | t1 |
| * | b | t1 | t2 |
| minus | c | | t3 |
| * | b | t3 | t4 |
| + | t2 | t4 | t5 |
| = | t5 | | a |

# Implementing IR

- Triples

```
1. - c
2. b * (1)
3. - c
4. b * (3)
5. (2) + (4)
6. a = (5)
```

| | op | arg1 | arg2 |
|---|---|---|---|
| (1) | minus | c | |
| (2) | * | b | (1) |
| (3) | minus | c | |
| (4) | * | b | (3) |
| (5) | + | (2) | (4) |
| (6) | = | a | (5) |

We refer to results of an operation x op y by its position

Code optimizer can change the order of instructions

# Implementing IR

Instruction List:

| | |
|---|---|
| 35 | (1) |
| 36 | (2) |
| 37 | (3) |
| 38 | (4) |
| 39 | (5) |
| 40 | (6) |

- Indirect Triples

```
1. - c
2. b * (1)
3. - c
4. b * (3)
5. (2) + (4)
6. a = (5)
```

can be re-ordered by the code optimizer

| | op | arg1 | arg2 |
|---|---|---|---|
| 1 | minus | c | |
| 2 | * | b | (1) |
| 3 | minus | c | |
| 4 | * | b | (3) |
| 5 | + | (2) | (4) |
| 6 | = | a | (5) |

# Implementing IR

- Static Single Assignment (SSA): All assignments are to variables with distinct names

instead of:

```
a = t1
b = a + t1
a = b + t1
```

the SSA form has:

```
a1 = t1
b1 = a1 + t1
a2 = b1 + t1
```

a variable is never reassigned

# Correctness vs. Optimizations

- When writing backend, correctness is paramount
    - Efficiency and optimizations are secondary concerns at this point
- Don’t try optimizations at this stage

# Summary

- 3-address code (TAC) is one example of an intermediate representation (IR)
- An IR should be close enough to existing machine code instructions so that subsequent translation into assembly is trivial
- In an IR we ignore some complexities and differences in computer architectures, such as limited registers, multiple instructions, branch delays, load delays, etc.

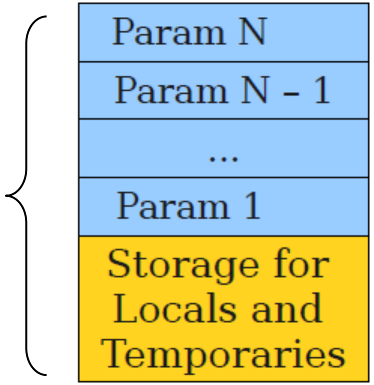
# Extra Slides

# What TAC doesn't give you

- Check bounds (array indexing)
- Two or n-dimensional arrays
- Conditional branches other than **if** or **ifFalse**
- Field names in records/structures
  - Use base+offset load/store
- Object data and method access

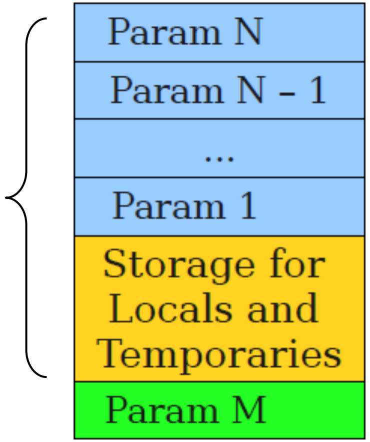
# Function arguments

Stack frame for function $f(a_1,\ldots a_N)$



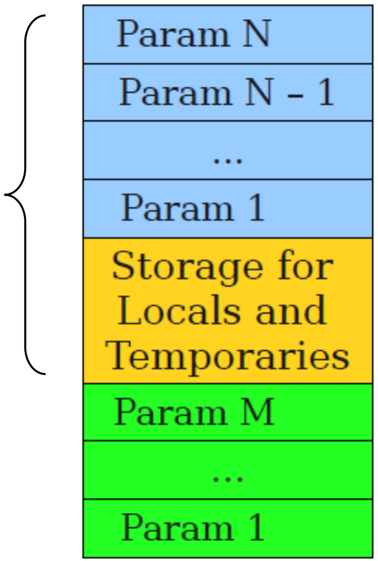
# Function arguments

Stack frame for function $f(a_1,\ldots a_N)$



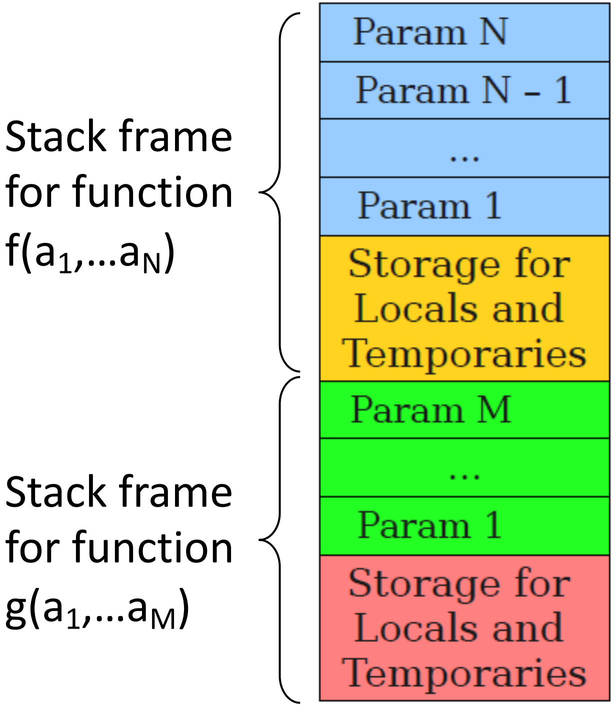
# Function arguments

Stack frame for function $f(a_1,\ldots a_N)$



# Function arguments

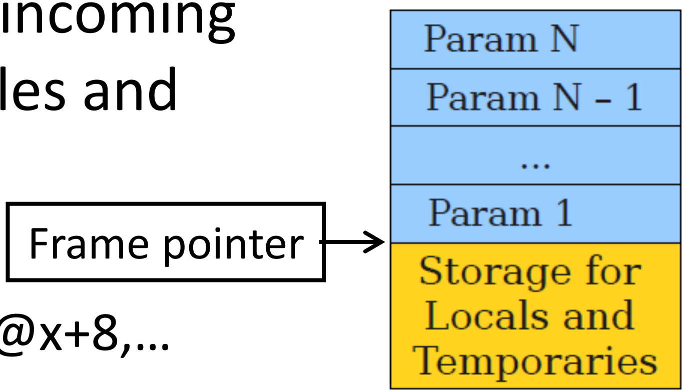
- Usually, stacks start at high memory addresses and grow to low memory addresses.



# Function arguments

- Compute offsets for all incoming arguments, local variables and temporaries
  - Incoming arguments

    are at offset @x, @x+4, @x+8,…
  - Locals+Temps are at @-y-4, @-y-8, …



# Computing Location Offsets

```
class A {
 void f (int a /* @x+4 */,
                int b /* @x+8 */,
                int c /* @x+12 */) {
    int s       // @-y-4
    if (c > 0) {
                int t …         // @-y-8
    } else {
                int u           // @-y-12
                int t …         // @-y-16
    }
 }
}
```

Location offsets for temporaries are ignored on this slide

← You could reuse @-y-8 here, but okay if you don't