IR

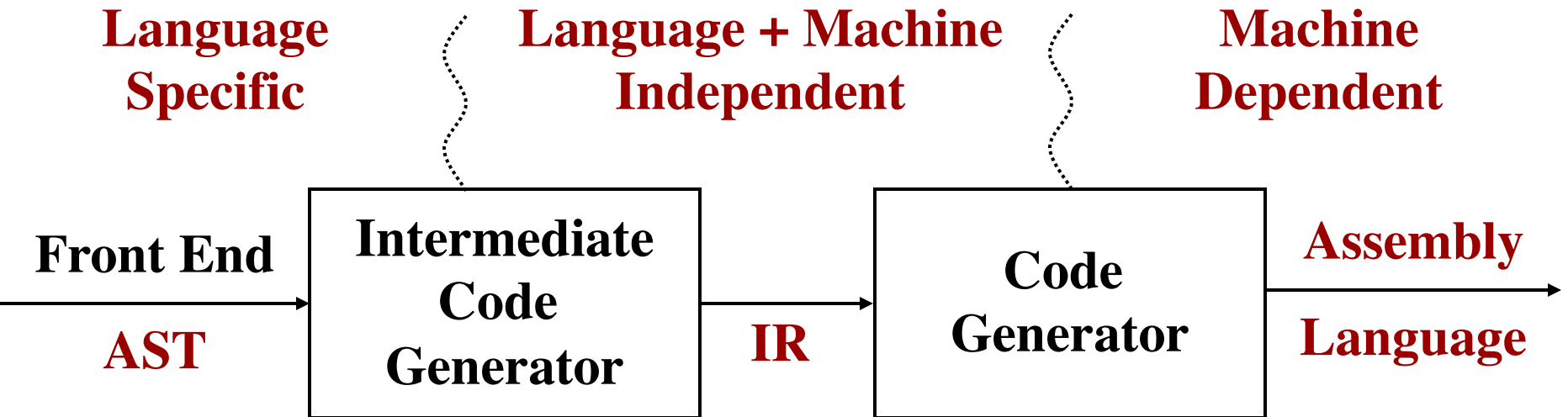# Intermediate Representation

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

Intermediate Representation

Language Specific

Language + Machine Independent

Machine Dependent

Front End

AST

| Intermediate Code Generator |

IR

| Code Generator |

Assembly Language

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 in each instructions

IR: 3-Address Code

- Address or locations:
  - Names/Labels
    - we allow source-program names in TAC (implemented as a pointer to the symbol table)
  - Constants
  - Temporaries

IR: 3-Address Code

- 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:

- Procedure calls: p(x1,x2,…,xn)
  - *param x1*
  - *param x2*
  - *…*
  - *param xn*
  - *call p, n*
- Return statement:
  - *return y*

  You can use it: *y = call p, n*

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)
- Pointers assignments:
  - $x = *y$ (y is a pointer, sets x to the value pointed by y)
  - $*x = y$

Control Flow

- Consider the statement:

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

```
L1:
  t1 = i
  t2 = t1 * 8
  t3 = a[ t2 ]
  ifFalse t3 < v goto L2
  t4 = i
  t4 = t4 + 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: ifFalse t3 < v goto 108
104: t4 = i
105: t4 = t4 + 1
106: i = t4
107: goto 100
108:
```

```
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
    param y
    param d
    t3 = call gcd, 2
    return t3

L0:

    t4 = d
    t5 = t4 < 0
    ...

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
    param i, 1
    call PrintInt, 1
    t3 = 1
    t4 = i + t3
    i = t4
    goto L0
L1:
    return
```

Translation of Expressions

symbol table

- S → id = E

- E → E + E

- E → - E

- E → (E)
- E → id

- $$.code = concat($3.code, $1.lexeme = $3.addr)

- $$.addr = new Temp(); $$.code = concat($1.code, $3.code, $$.addr = $1.addr + $3.addr)

- $$.addr = new Temp(); $$.code = concat($2.code, $$.addr = - $2.addr)

- $$.addr = $2.addr; $$.code = $2.code
- $$.addr = symtbl($1.lexeme); $$.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)

Backpatching

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

  99: t0 = a < b

  100: if t0 goto 102

  101: goto ??? — falselist

  102: t1 = 1

  103: t2 = i + t1

  104: i = t2

  105: goto ??? — nextlist

  106: t1 = 1

  107: t2 = i+t1

  108: j = t2

  109:

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

- These lists can be implemented as a synthesized attribute
  - Using marker non-terminals

- $S \rightarrow$ if '(' B ')' M block

  {backpatch(B.truelist, M.instr);

  S.nextlist = merge(B.falselist, block.nextlist);}

- $B \rightarrow E_1$ **rel** E_2

  next instruction number

  {B.truelist = makelist(nextinstr);

  B.falselist = makelist(nextinstr+1);

  print('if' E_1.addr **rel**.op E_2.addr 'goto –');

  print('goto –');

- $B \rightarrow$ **true**

  {B.truelist=makelist(nextinstr);

  print('goto –');}

- $B \rightarrow$ **false**

  {B.falselist=makelist(nextinstr);

  print('goto –');}

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

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

- 101: …
- 102: if a < b goto 104
- 103: goto –
- 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}

- S → while M1 '(' B ')' M2 block

  {backpatch(block.nextlist, M1.instr);

   backpatch(B.truelist, M2.instr);

   S.nextlist = B.falselist; print('goto' M1.instr)}

- B → E1 **rel** E2

  {B.truelist = makelist(nextinstr);

   B.falselist = makelist(nextinstr+1);

   print('if' E1.addr **rel**.op E2.addr 'goto –');

   print('goto –');

- B → true

  {B.truelist=makelist(nextinstr);

   print('goto –');}

- B → false

  {B.falselist=makelist(nextinstr);

   print('goto –');}

- M → ε   {M.instr = nextinstr;}

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

- 101: …
- 102: if i < n goto 104
- 103: goto –
- 104: t1 = 1
- 105: t2 = i+t1
- 106: i = t2
- 107: goto 102
- 108:

M1.instr = 102
B.truelist={102}, B.falselist={103}
M2.instr = 104
backpatch({102}, 104)
S.nextlist={103}

- $S \rightarrow$ while M1 '(' B ')' M2 block

  {backpatch(block.nextlist, M1.instr);

  backpatch(B.truelist, M2.instr);

  S.nextlist = merge(B.falselist; block.breaklist); }

  print('goto' M1.instr)}

- $S_1 \rightarrow$ break ;

  {S1.breaklist=makelist(nextinstr);

  print('goto –');}

- $S_1 \rightarrow$ continue ;

{S1.nextlist=makelist(nextinstr);print('goto –');}

- $B \rightarrow E_1$ **rel** E_2 {…}

- block $\rightarrow$ '{' S_1 '}'

  {block.breaklist=S.breaklist;

  block.nextlist=S.nextlist;}

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

while (i < n){continue;}

- 101: …
- 102: if i < n goto $\cancel{104}$
- 103: goto –
- 104: goto $\cancel{102}$
- 105: goto 102
- 106:

M1.instr = 102
B.truelist={102}, B.falselist={103}
M2.instr = 104
S1.nextlist=block.nextlist={104}
backpatch({104}, 102)
backpatch({102}, 104)
S.nextlist={103}

- S → while M1 '(' B ')' M2 block

  {backpatch(block.nextlist, M1.instr);
   backpatch(B.truelist, M2.instr);
   S.nextlist = merge(B.falselist; block.breaklist); }
   print('goto' M1.instr)}

- S1 → break ;

  {S1.breaklist=makelist(nextinstr);
   print('goto –');}

- S1 → continue ;

{S1.nextlist=makelist(nextinstr);print('goto –');}

- B → E1 **rel** E2 {...}

- block → '{' S1 '}'

  {block.breaklist=S.breaklist;
   block.nextlist=S.nextlist;}

- M → ε   {M.instr = nextinstr;}

while (i < n){break;}

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

M1.instr = 102
B.truelist={102}, B.falselist={103}
M2.instr = 104
S1.breaklist=block.breaklist={104}
backpatch({102}, 104)
S.nextlist={103,104}

- S → if '(' B ')' M block

  {backpatch(B.truelist, M.instr);

  backpatch(B.falselist, block.nextlist);

  S.nextlist = merge(B.falselist, block.nextlist);}

- B → B1 || M B2

  {backpatch(B1.falselist, M.instr);

  B.truelist = merge(B1.truelist, B2.truelist);

  B.falselist = B2.falselist;}

- B → E1 **rel** E2

  {B.truelist = makelist(nextinstr);

  B.falselist = makelist(nextinstr+1);

  print('if' E1.addr **rel**.op

  E2.addr 'goto –');

  print('goto –');

- M → ε   {M.instr = nextinstr;}

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

- 101: …
- 102: if a < b goto –
- 103: goto –104
- 104: if i < n goto –
- 105: goto –

B1.truelist={102} B1.falselist={103}
M.instr = 104
B2.truelist={104} B2.falselist={105}
backpatch({103}, 104)
B.truelist={102,104}, B.falselist={105}

- S → if '(' B ')' M block

  {backpatch(B.truelist, M.instr);

  backpatch(B.falselist, block.nextlist);

  S.nextlist = merge(B.falselist, block.nextlist);}

- B → B1 || M B2

  {backpatch(B1.falselist, M.instr);

  B.truelist = merge(B1.truelist, B2.truelist);

  B.falselist = B2.falselist;}

- B → E1 **rel** E2

  {B.truelist = makelist(nextinstr);

  B.falselist = makelist(nextinstr+1);

  print('if' E1.addr **rel**.op

  E2.addr 'goto –');

  print('goto –');

- M → ε   {M.instr = nextinstr;}

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

- 101: …
- 102: if a < b goto 106
- 103: goto 104
- 104: if i < n goto 106
- 105: goto –
- 106: t1 = 1
- 107: t2 = i+t1
- 108: i = t2
- 109:

B.truelist={102,104}, B.falselist={105}
M.instr = 106
backpatch({102,104}, 106)
S.nextlist={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 ith 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 }
```

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

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

Array References

Wrong

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 L0
    t4 = 1
    return t4
L0:
    t5 = 1
    t6 = n - t5
    param t6
    t7 = call factorial, 1
    t8 = n * t7
    return t8
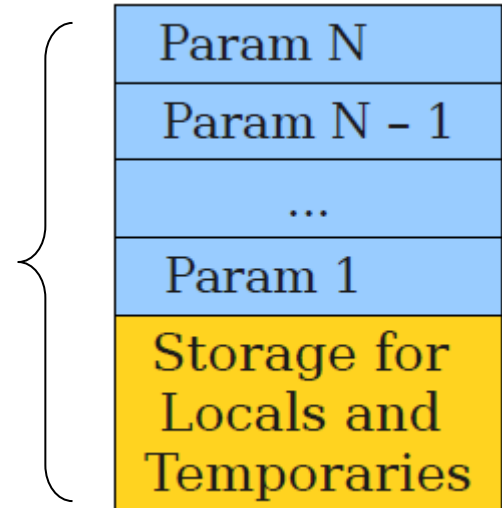
$t3 = n <= 1$

Function arguments

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

| Param N |
| Param N – 1 |
| ... |
| Param 1 |
| Storage for Locals and Temporaries |

Function arguments

Stack frame
for function
$f(a_1,...a_N)$

Param N
Param N – 1
...
Param 1
Storage for Locals and Temporaries
Param M

Function arguments

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

Param N
Param N – 1
...
Param 1
Storage for Locals and Temporaries
Param M
...
Param 1
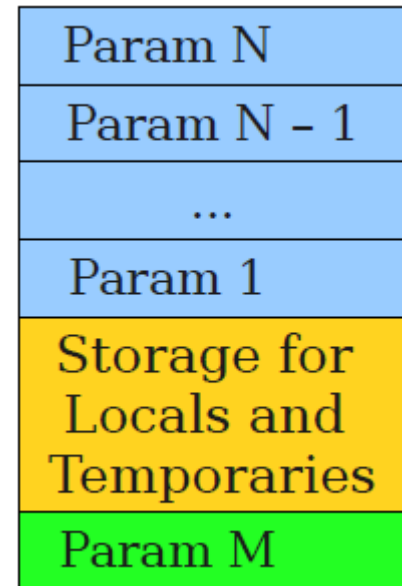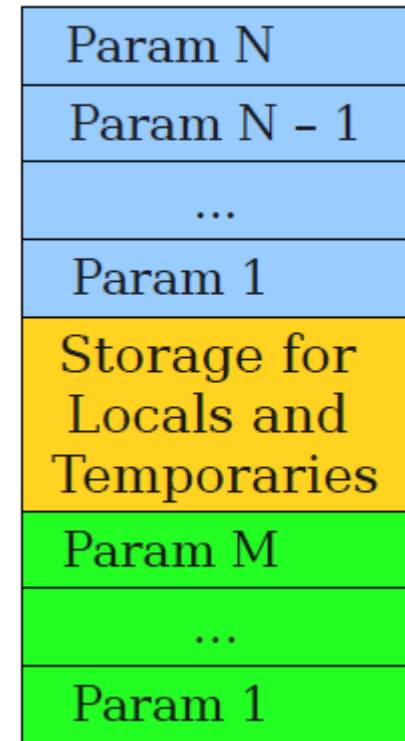
Function arguments

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

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

Param N
Param N – 1
…
Param 1
Storage for Locals and Temporaries

Stack frame for function $g(a_1,\ldots a_M)$

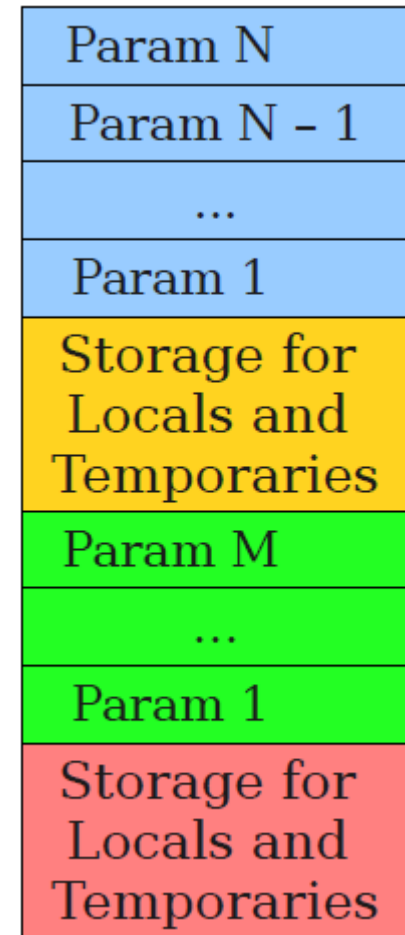| |
|---|
| Param M |
| … |
| Param 1 |
| Storage for Locals and Temporaries |

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, ...

| Param N |
| Param N – 1 |
| ... |
| Param 1 |
| Storage for Locals and Temporaries |

Frame pointer →
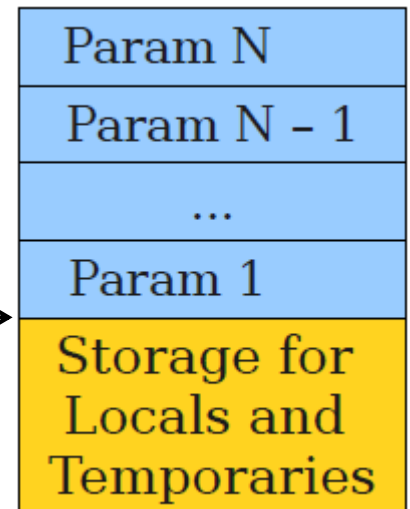
Computing Location Offsets

```
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

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 change the order of instructions

Implementing IR

- Indirect Triples

Instruction List:

		op	arg1	arg2
35	(1)	1. - c	1	minus	c	
36	(2)	2. b * (1)	2	*	b	(1)
37	(3)	3. - c	3	minus	c	
38	(4)	4. b * (3)	4	*	b	(3)
39	(5)	5. (2) + (4)	5	+	(2)	(4)
40	(6)	6. a = (5)	6	=	a	(5)

can be re-ordered by the code optimizer

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

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

Summary

- 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