RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Code Generation - Part 2

Intermediate Representations

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Roadmap for the remainder of the course

- Project #2 – Bottom-up parser and compiler
  New due date: Wednesday April 20

- Homework #5 has been posted

- Midterm #1 – Grade challenge deadline is Friday, April 15.
  Please pick up your exams in recitation

- Final exam on May 10 , 1:00pm, (60 minutes in class)

- Grading Scheme
  → Exams: 2 x 30% (best two exams count)
  → Projects: 3 x 10%
  → Homeworks: 5 x 2% (best five homeworks count)

Code Generation

EaC Chapter 7

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 and ISA

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	*becomes*	cmp_LT $r_x, r_y \Rightarrow r_1$
if (x < y) then $stmt_1$ else $stmt_2$	*becomes*	cmp_LT $r_x, r_y \Rightarrow r_1$ cbr $r_1 \Rightarrow$ _$stmt_1$, _$stmt_2$

Boolean & Relational Values

What if the ISA uses a condition code?

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

Example: // r_2 should contain boolean value of "x<y" evaluation

			cmp	$r_x, r_y \Rightarrow cc_1$
			cbr_LT	$cc_1 \rightarrow L_T, L_F$
x <y	*becomes*	L_T:	loadI	$1 \Rightarrow r_2$
			br	$\rightarrow L_E$
		L_F:	loadI	$0 \Rightarrow r_2$
		L_E:	...other stmts...	

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 ← c + d
else a ← e + f

Variations on the ILOC Branch Structure					
Straight Condition Codes			*Boolean Compares*		
	comp	$r_x, r_y \Rightarrow cc_1$		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	$r_c, r_d \Rightarrow r_a$
	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

Conditional move & predication both simplify this code

Example
if (x < y)
then a ← c + d
else a ← e + f

OTHER ARCHITECTURAL VARIATIONS

Conditional Move			*Predicated Execution*	
comp	$r_x, r_y \Rightarrow cc_1$		cmp_LT	$r_x, r_y \Rightarrow r_1$
add	$r_c, r_d \Rightarrow r_1$	(r_1)?	add	$r_c, r_d \Rightarrow r_a$
add	$r_e, r_f \Rightarrow r_2$	$(\neg r_1)$?	add	$r_e, r_f \Rightarrow r_a$
i2i_<	$cc_1, r_1, r_2 \Rightarrow r_a$			

Both versions avoid the branches

Both are shorter than CCs or Boolean-valued compare

Are they better? What about power?

Boolean & Relational Values

Consider the assignment $x \leftarrow a < b \wedge c < d$ (short circuiting?)

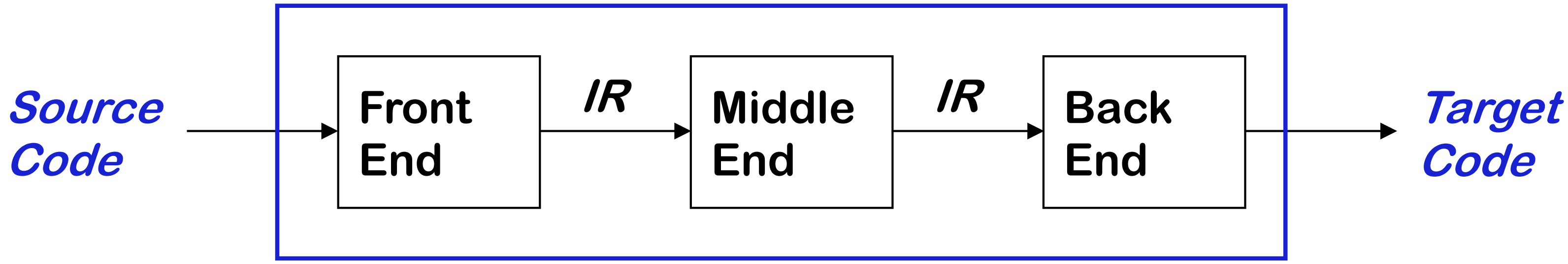
Variations on the ILOC Branch Structure					
Straight Condition Codes			*Boolean Compare*		
	comp	$r_a, r_b \Rightarrow cc_1$	cmp_LT	$r_a, r_b \Rightarrow r_1$	
	cbr_LT	$cc_1 \rightarrow L_1, L_2$	cmp_LT	$r_c, r_d \Rightarrow r_2$	
L_1:	comp	$r_c, r_d \Rightarrow cc_2$	and	$r_1, r_2 \Rightarrow r_x$	
	cbr_LT	$cc_2 \rightarrow L_3, L_2$			
L_2:	loadI	$0 \Rightarrow r_x$			
	br	$\rightarrow L_{OUT}$			
L_3:	loadI	$1 \Rightarrow r_x$			
	br	$\rightarrow L_{OUT}$			
L_{OUT}:	nop				

Here, the boolean compare produces much better code.

Intermediate Representation

EaC Chapter 5

Intermediate Representations



- Front end - produces an intermediate representation (*IR*)
- Middle end - transforms the *IR* into an equivalent *IR* that runs more efficiently
- Back end - transforms the *IR* into native code

- *IR* encodes the compiler's knowledge of the program
- Middle end usually consists of several passes

Intermediate Representations

- Decisions in *IR* design affect the speed and efficiency of the compiler

- Some important *IR* properties
  - → Ease of generation
  - → Ease of manipulation
  - → Size
  - → Level of abstraction

- The importance of different properties varies between compilers
  - → Selecting an appropriate *IR* for a compiler is critical

Types of Intermediate Representations

Three major categories

- Structural
  - → Graphically oriented
  - → Heavily used in source-to-source translators
  - → Tend to be large

Examples:
Trees, DAGs

- Linear
  - → Pseudo-code for an abstract machine
  - → Level of abstraction varies
  - → Simple, compact data structures
  - → Easier to rearrange

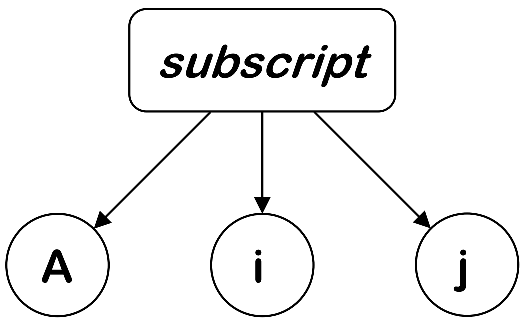
Examples:
3 address code
Stack machine code

- Hybrid
  - → Combination of graphs and linear code

Example:
Control-flow graph

Level of Abstraction

- The level of detail exposed in an *IR* influences the profitability and feasibility of different optimizations.
- Two different representations of an array reference:



loadI	**1**		**=>**	$\mathbf{r_1}$
sub	$\mathbf{r_j}$,	$\mathbf{r_1}$	**=>**	$\mathbf{r_2}$
loadI	**10**		**=>**	$\mathbf{r_3}$
mult	$\mathbf{r_2}$,	$\mathbf{r_3}$	**=>**	$\mathbf{r_4}$
sub	$\mathbf{r_i}$,	$\mathbf{r_1}$	**=>**	$\mathbf{r_5}$
add	$\mathbf{r_4}$,	$\mathbf{r_5}$	**=>**	$\mathbf{r_6}$
loadI	**@A**		**=>**	$\mathbf{r_7}$
Add	$\mathbf{r_7}$,	$\mathbf{r_6}$	**=>**	$\mathbf{r_8}$
load	$\mathbf{r_8}$		**=>**	$\mathbf{r_{Aij}}$

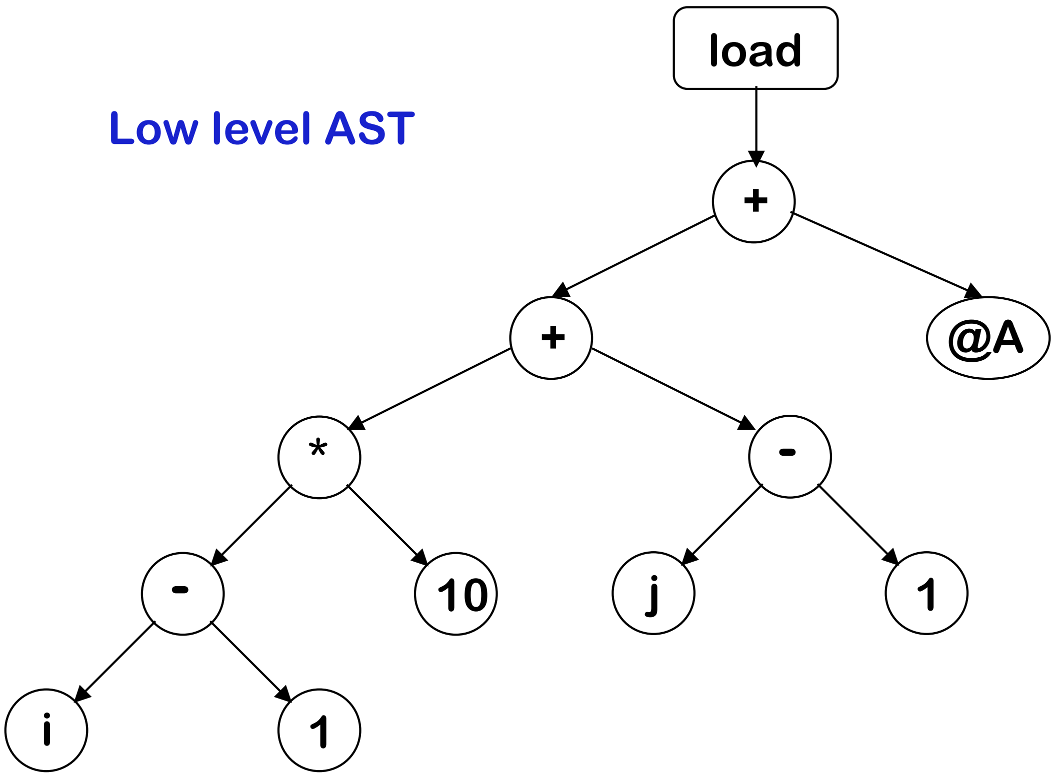
High level AST: Good for memory disambiguation

Low level linear code: Good for address calculation

Level of Abstraction

- Structural *IRs* are usually considered high-level
- Linear *IRs* are usually considered low-level
- Not necessarily true:

Low level AST

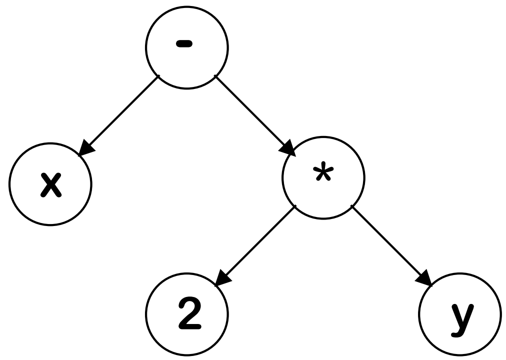



```
loadArray A,i,j
```

High level linear code

Abstract Syntax Tree

An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed



x - 2 * y

- Can use linearized form of the tree

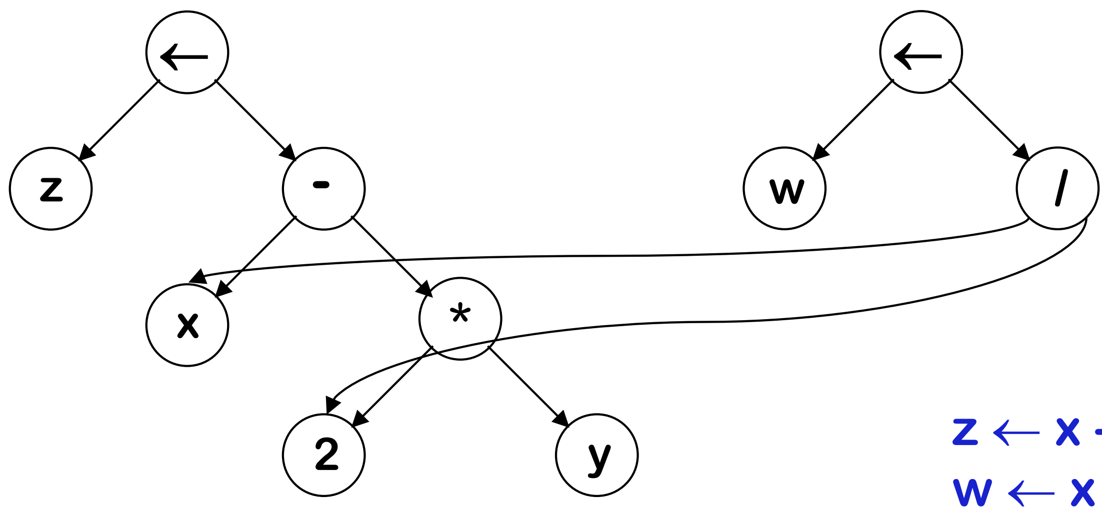
  x 2 y * – in postfix form

  – * 2 y x in prefix form

- S-expressions are (essentially) ASTs (remember functional languages such as Scheme or Lisp!)

Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value



$z \leftarrow x - 2 * y$
$w \leftarrow x / 2$

- Makes sharing explicit
- Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just once!

Stack Machine Code

Originally used for stack-based computers, now Java

- Example:

x - 2 * y   becomes

```
push x
push 2
push y
multiply
subtract
```

Advantages

- Compact form
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code

Implicit names take up no space, where explicit ones do!

Useful where code is transmitted over slow communication links (*the net*)

Three Address Code

Several different representations of three address code

- In general, three address code has statements of the form:

  $x \leftarrow y \ \underline{op} \ z$

  With 1 operator ($\underline{op}$) and, at most, 3 names (x, y, z)

Example:

$z \leftarrow x - \boxed{2 * y}$ becomes

```
t ← 2 * y
z ← x - t
```

Advantages:

- Resembles many machines
- Introduces a new set of names
- Compact form

Three Address Code: Quadruples

Naïve representation of three address code

- Table of k * 4 small integers
- Simple record structure
- Easy to reorder
- Explicit names

The original FORTRAN compiler used "quads"

```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

RISC assembly code (not ILOC)

load	1	y	
loadI	2	2	
mult	3	2	1
load	4	x	
sub	5	4	2

Quadruples

Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

(1)	load	y	
(2)	loadI	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

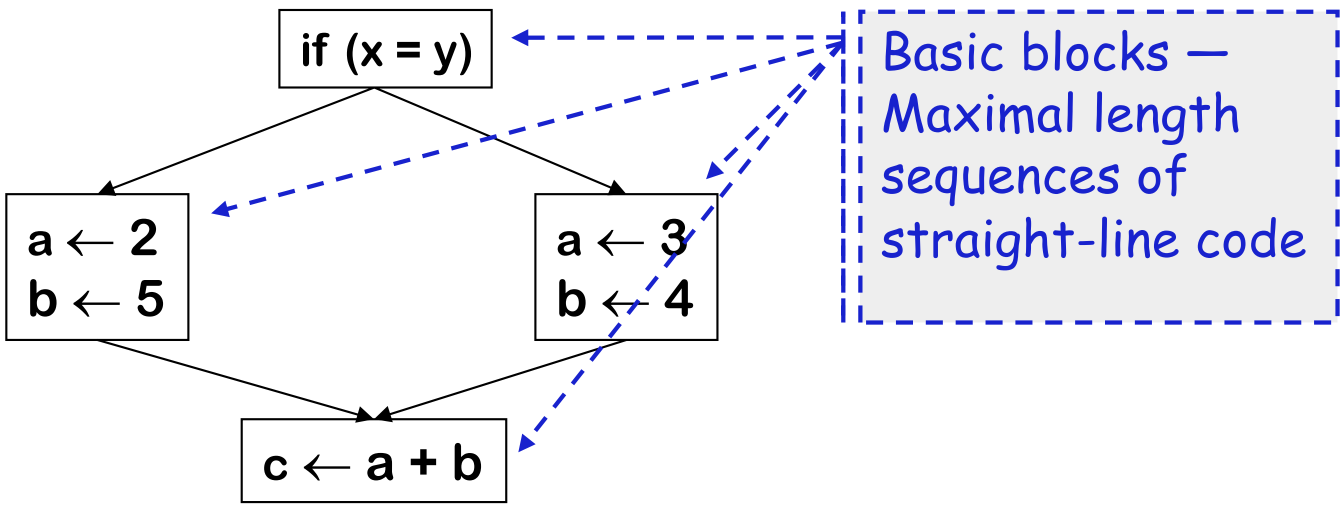
Implicit names take no space!

Control-flow Graph (CFG)

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - → Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example



Static Single Assignment Form (SSA)

- The main idea: each name defined exactly once in program
- Introduce ϕ-functions to make it work

Original

```
x ← ...
y ← ...
while (x < k)
    x ← x + 1
    y ← y + x
```

SSA-form

```
          x0 ← ...
          y0 ← ...
          if (x0 > k) goto next
loop:        x1 ← φ(x0,x2)
             y1 ← φ(y0,y2)
             x2 ← x1 + 1
             y2 ← y1 + x2
             if (x2 < k) goto loop
next:        ...
```

Strengths of SSA-form

- Sharper analysis
- "minimal" ϕ-functions placement is non-trivial
- (sometimes) faster algorithms

Things to do and next class

Work on the project!

Compiler Optimizations

Procedure abstraction

Read EaC: Chapter 6.1 – 6.5