RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Code Generation

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

Announcements

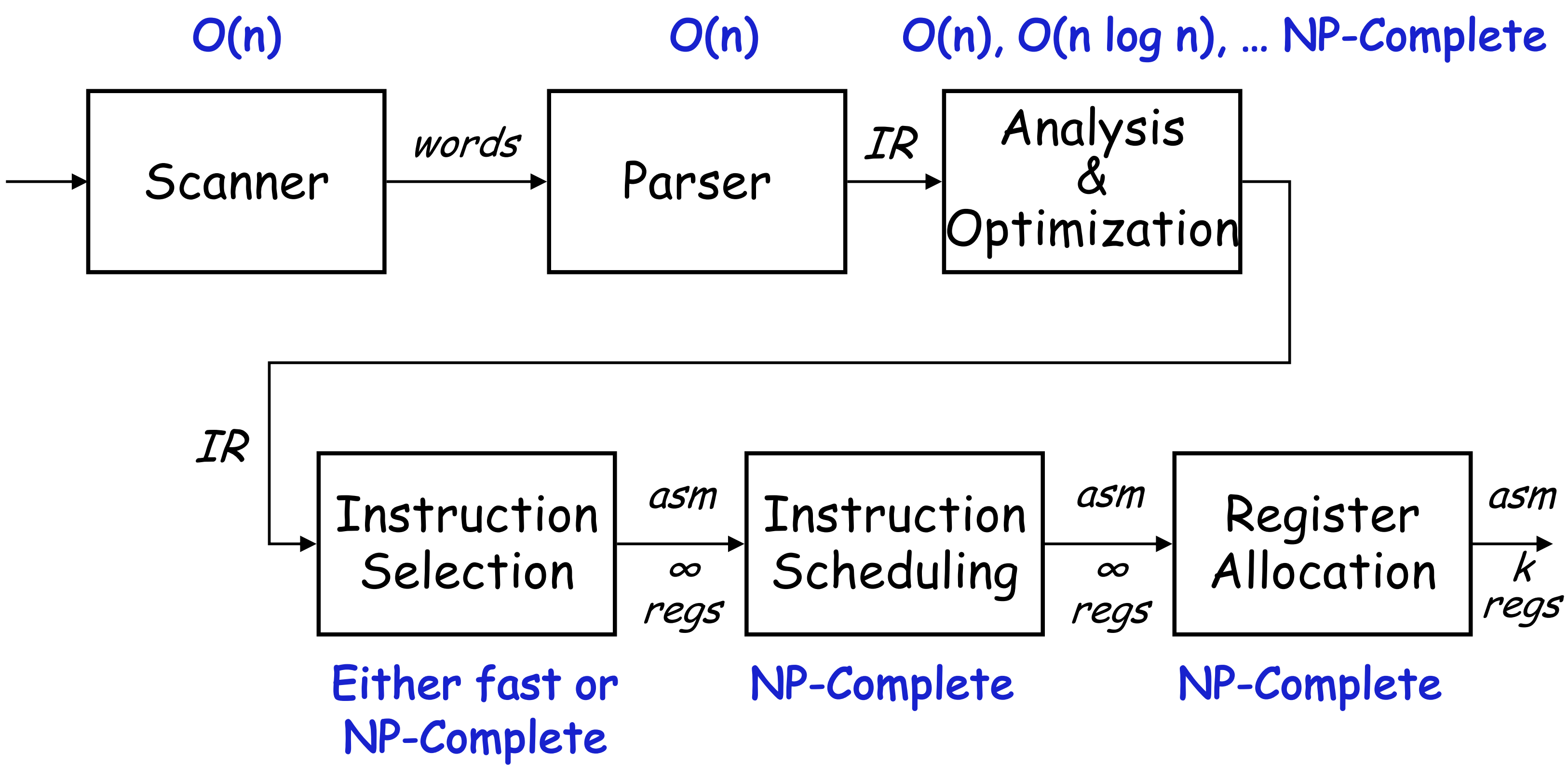
Roadmap for the remainder of the course

- Project #2 – Bottom-up parser and compiler
  Due date Friday April 15
- 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

Review - Structure of a Compiler



A compiler is a lot of fast stuff followed by some hard problems

- The hard stuff is mostly in code generation and optimization
- For superscalars, its allocation & scheduling that is particularly important

Rutgers Review - Generating Code

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 (no aliasing)
  - → 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* (*register-register model*)

- 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 reference

Relies on a strong register allocator

Recursive Treewalk vs. Ad-hoc SDT

Top-down "LL"

```
int 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;
}
```

Bottom-up "LR"

```
Goal :    Expr { $$ = $1; } ;
Expr:     Expr PLUS Term
          { t = NextRegister();
            emit(add,$1,$3,t); $$ = t; }
        | Expr MINUS Term {...}
        | Term { $$ = $1; } ;
Term:     Term TIMES Factor
          { t = NextRegister();
            emit(mult,$1,$3,t); $$ = t; };
        | Term DIVIDES Factor {...}
        | Factor { $$ = $1; };
Factor:   NUMBER
          { t = NextRegister();
            emit(loadI,val($1),none, t );
            $$ = t; }
        | ID
          { t1 = base($1);
            t2 = offset($1);
            t = NextRegister();
            emit(loadAO,t1,t2,t);
            $$ = t; }
```

Handling Assignment (just another operator)

lhs ← *rhs*

Strategy

- Evaluate *rhs* to a value *(an rvalue)*
- Evaluate *lhs* to a location (memory address) *(an lvalue)*
  - → *lvalue* is an address ⇒ store rhs
- If *rvalue* & *lvalue* have different types
  - → Evaluate *rvalue* to its "*natural*" type
  - → Convert that value to the type of lhs *value, if possible*

Unambiguous scalars may go into registers (no aliasing)

Ambiguous scalars or aggregates go into memory (possible aliasing)

Example: A(i, j) = 1.42 vs. k = 1.42 ?

Handling Assignment

What if the compiler cannot determine the rhs's type ?

- This is a property of the language & the specific program
- If type-safety is desired, compiler must insert a run-time check
- Add a *tag field* to the data items to hold type information

Code for assignment becomes more complex

```
evaluate rhs
If lhs.type_tag ≠ rhs.type_tag
   then
      convert rhs to type(lhs) or
      signal a run-time error
lhs ← rhs
```

This is much more complex than if it knew the types

Handling Assignment

Compile-time type-checking

- Goal is to eliminate both the runtime check & the tag
- Determine, at compile time, the type of each subexpression
- Use compile-time types to determine if a run-time check is needed

Optimization strategy

- If compiler knows the type, move the check to compile-time
- Unless tags are needed for garbage collection, eliminate them
- If check is needed, try to overlap it with other computation (superscalar or multi-core architectures)

Handling Assignment (with reference counting)

Garbage Collection

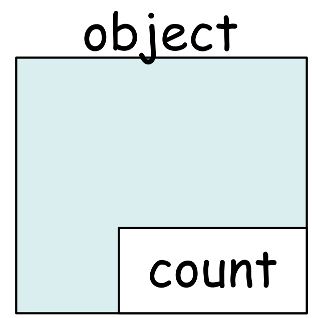
The problem with reference counting

- Must adjust the count on each pointer assignment
- Overhead is significant, relative to assignment

Code for assignment becomes

```
evaluate rhs
lhs→count ← lhs→count - 1
lhs ← addr(rhs)
rhs→count ← rhs→count + 1
```

Plus a check for zero at the end



This adds *1 +, 1 -, 2 loads, & 2 stores*

With extra functional units & large caches, this may become either cheap or free. What about power consumption?

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

First, must agree on a storage scheme

Row-major order (most languages)

- 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 (Fortran)

- 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 (Java)

- Vector of pointers to pointers to … to values
- Takes much more space, trades indirection for arithmetic
- Not easily amenable to (locality) analysis

Laying Out Arrays

The Concept

A

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

These have distinct & different cache behavior

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

Indirection vectors

A

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

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

Computing an Array Address

Declaration: A[low .. high] of …

A[i]

- @A + (i - low) x sizeof(A[1])
- In general: base(A) + (i - low) x sizeof(A[1])

Computing an Array Address

Declaration: A[low .. high] of …

A[i]

- @A + (i - low) x sizeof(A[1])
- In general: base(A) + (i - low) x sizeof(A[1])

int A[1:10] ⇒ low is 1
Make low 0 for faster access (saves a –)

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

Computing an Array Address

Declaration: A[low1 .. high1, low2 .. high2] of …

A[i]

- @A + (i – low) x sizeof(A[1])
- In general: base(A) + (i – low) x sizeof(A[1])

What about $A[i_1,i_2]$?

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

Row-major order, two dimensions

$@A + ((i_1 - low_1) \times (high_2 - low_2 + 1) + i_2 - low_2) \times sizeof(A[1])$

Column-major order, two dimensions

$@A + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times sizeof(A[1])$

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 $w = \text{sizeof}(A[1,1])$

$$@A + (i - low_1) \times (high_2 - low_2 + 1) \times w + (j - low_2) \times w$$

Which can be factored into

$$@A + i \times (high_2 - low_2 + 1) \times w + j \times w$$
$$- (low_1 \times (high_2 - low_2 + 1) \times w) + (low_2 \times w)$$

If low_i, $high_i$, and w are known, the last term is a constant

Define $@A_0$ as

$$@A - (low_1 \times (high_2 - low_2 + 1) \times w + low_2 \times w$$

And len_2 as $(high_2 - low_2 + 1)$

Then, the address expression becomes

$$@A_0 + (i \times len_2 + j) \times w$$

Compile-time constants

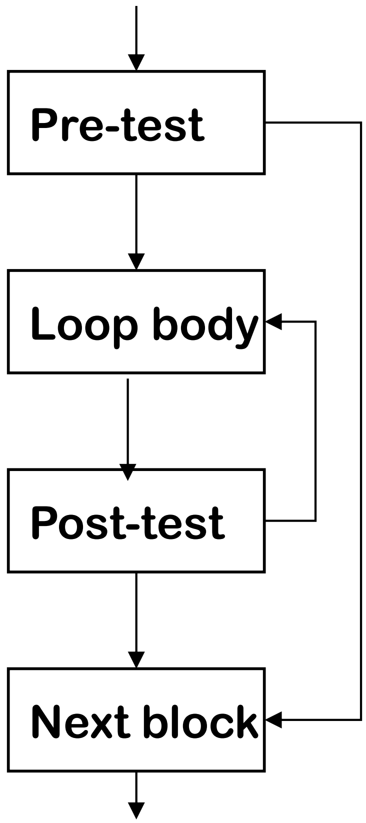
Control Flow

One possible approach for code generation:

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 = 0; i< 100; i++) { *body* }

next statement

```
        loadI    0 ⇒ r1
        loadI    1 ⇒ r2
        loadI    100 ⇒ r3
        cmp_GT   r3, r1 ⇒ r4
        cbr      r4 ⇒ L1, L2
L1:  body
        add      r1, r2 ⇒ r1
        cmp_LT   r1, r3 ⇒ r5
        cbr      r5 ⇒ L1, L2
L2:  next statement
```

Initialization

Pre-test

Post-test

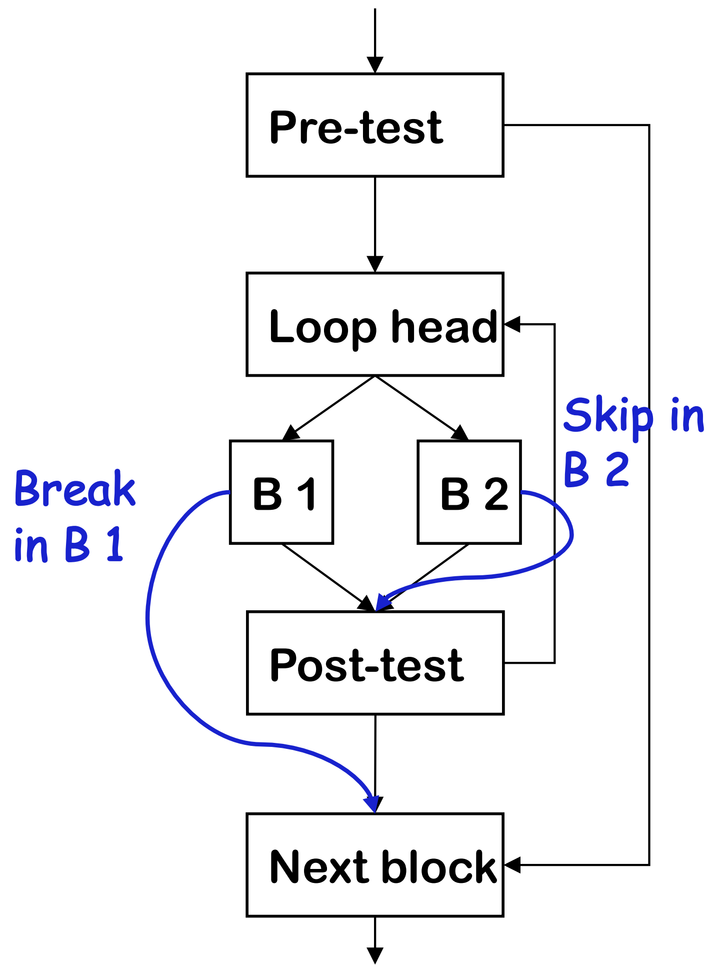
Break statements

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
- skip 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 (*use break*)

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

Surprisingly many compilers do this for all cases!

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: jump table)

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

x <y *becomes*

	cmp	$r_x, r_y \Rightarrow cc_1$
	cbr_LT	$cc_1 \rightarrow L_T, L_F$
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.

Things to do and next class

Work on the project!

Intermediate representations

Read EaC: Chapter 5

Procedure abstraction

Read EaC: Chapter 6.1 – 6.5