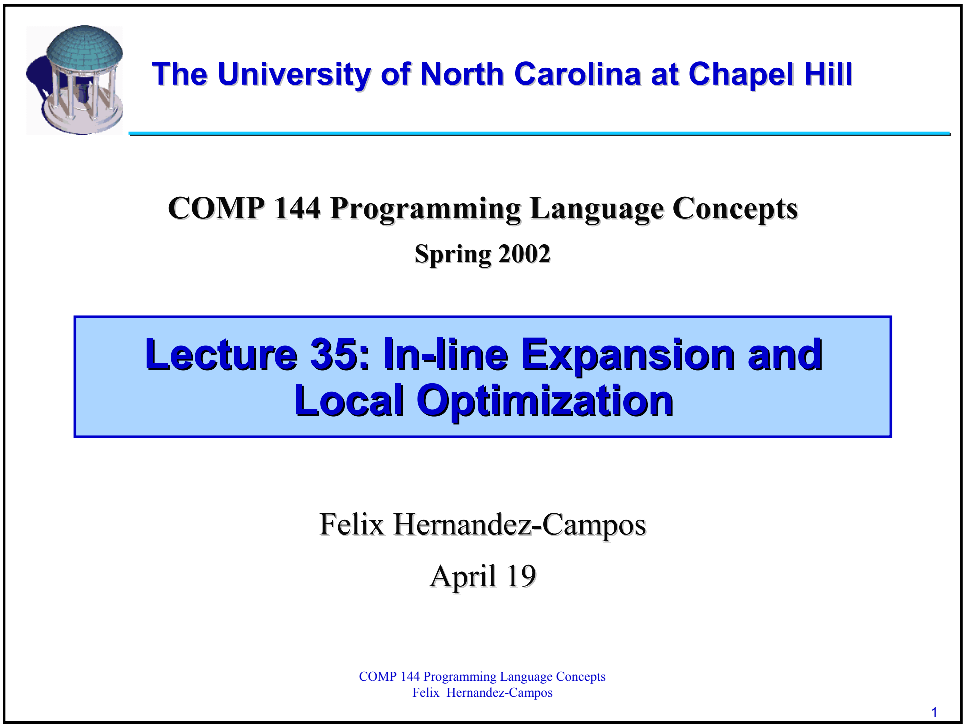


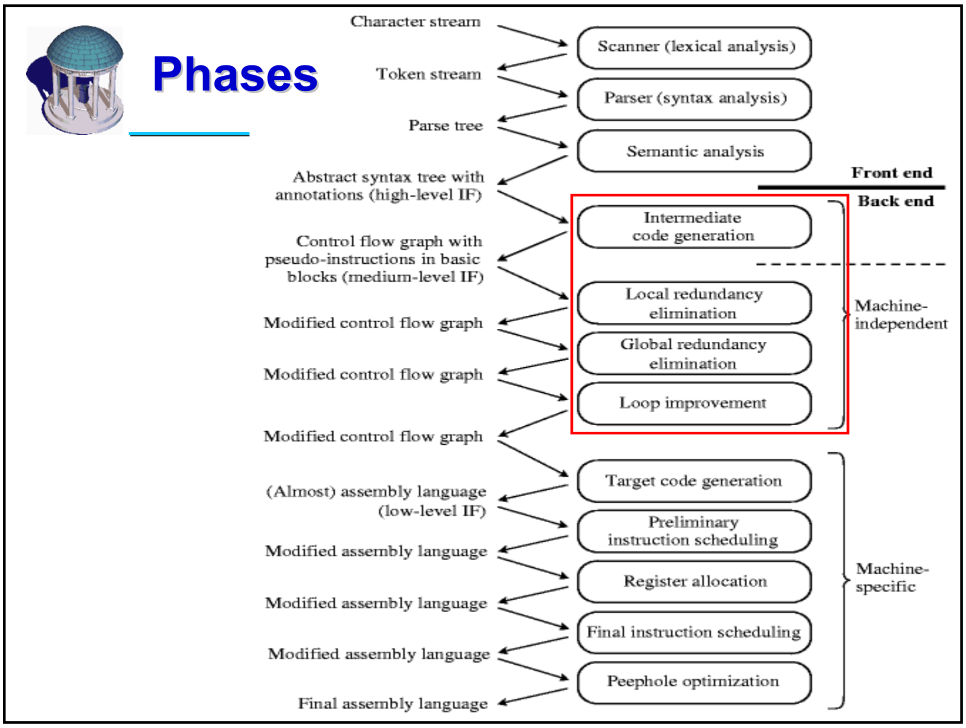
The University of North Carolina at Chapel Hill

COMP 144 Programming Language Concepts
Spring 2002

Lecture 35: In-line Expansion and Local Optimization

Felix Hernandez-Campos
April 19

COMP 144 Programming Language Concepts
Felix Hernandez-Campos



Phases

Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis
Abstract syntax tree with annotations (high-level IF)
Front end
Back end
Intermediate code generation
Control flow graph with pseudo-instructions in basic blocks (medium-level IF)
Local redundancy elimination
Modified control flow graph
Global redundancy elimination
Modified control flow graph
Loop improvement
Machine-independent
Modified control flow graph
Target code generation
(Almost) assembly language (low-level IF)
Preliminary instruction scheduling
Modified assembly language
Register allocation
Modified assembly language
Final instruction scheduling
Modified assembly language
Peephole optimization
Final assembly language
Machine-specific

Subroutine In-line Expansion

- Subroutines may be expanded in-line at the point of the call
  - Rather than use a stack-based calling convention
- In-line expansion saves subroutine overheads and help code improvement
- In-line expansion may be decided by the compiler based on some optimization heuristics
  - *E.g.* short, non-recursive subroutines are always in-lined in some languages

Subroutine In-line Expansion

- In-line expansion can also be *suggested* by the programmer
  - *E.g.* C++

```
inline int max (int a, int b) {
  return a > b ? a : b;
}
```

  - *E.g.* Ada

```
function max(a, b : integer) return integer is
begin
  if a > b then return a; else return b; end if;
end max;
pragma inline (max);
```

Macros and In-line Expansion

- What is the difference between a macro and a programmer suggested expansion?
  - Optional in the second case
  - Most importantly, in-line expansion is an implementation technique with no effect in program semantics
- *E.g.*

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

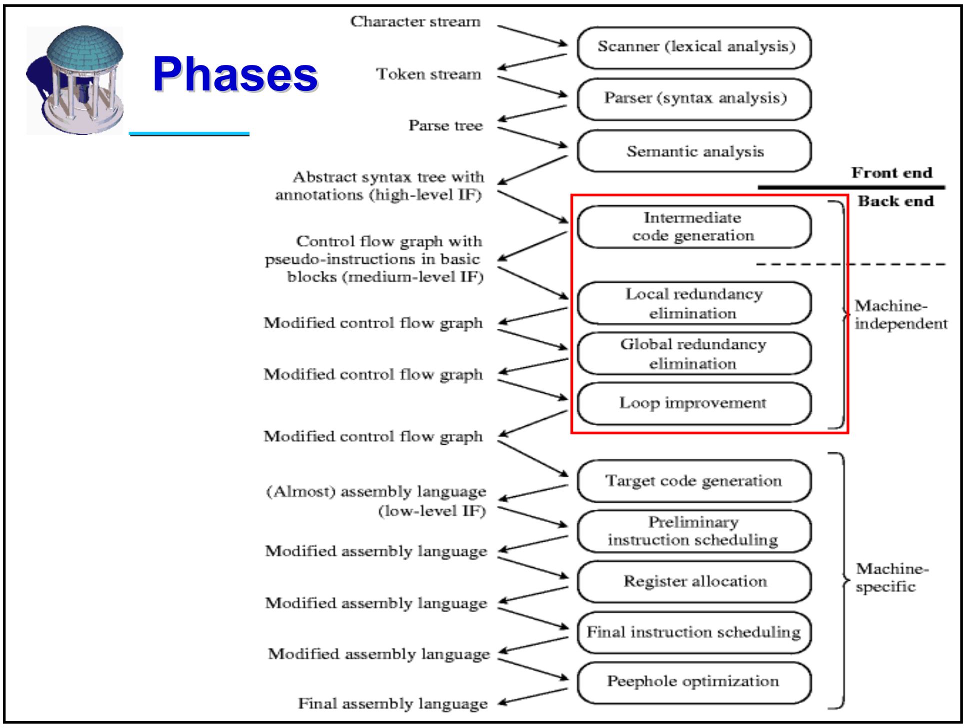
  - No type checking
  - What happens `after MAX(x++, y++)`?
  - The larger argument is incremented twice

In-line Expansion

- In-line expansion has some disadvantages
  - Increase in code size
  - It cannot be used with recursive subroutines
- It is sometimes useful to expand the first case in a recursion subroutine
  - *Optimize the common case rule*

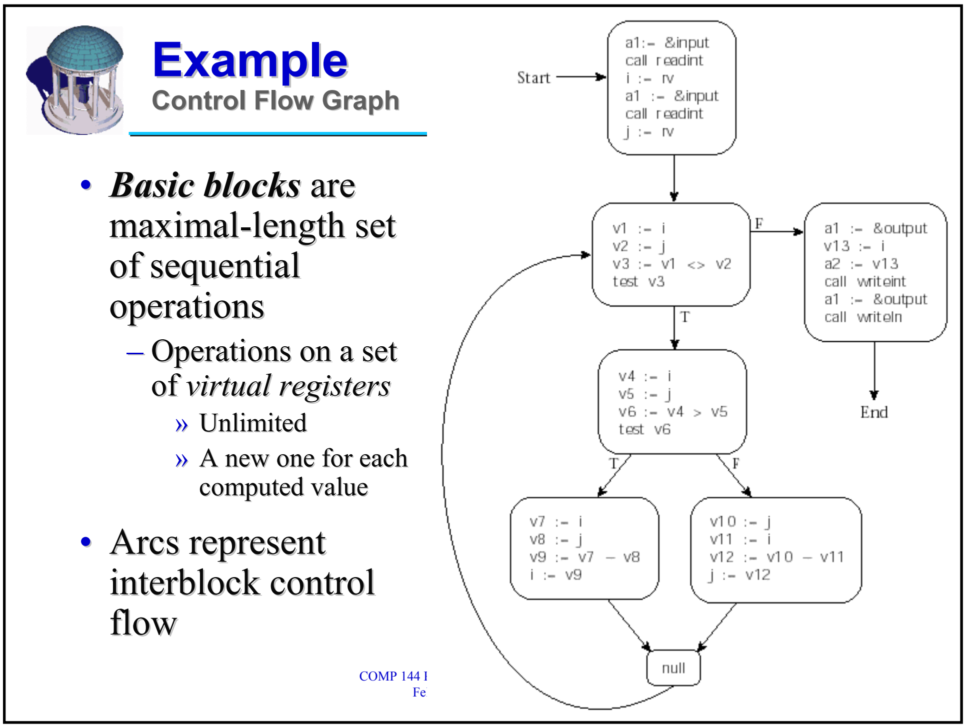
```
range_t bucket_contents (bucket *b, domain_t x)
{
    if (b->key == x)
        return b->val;
    else if (b->next == 0)
        return ERROR;
    else
        return bucket_contents (b->next, x);
}
```

Most hash chains are only one bucket long



Phases

Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis
Abstract syntax tree with annotations (high-level IF)
Front end
Back end
Intermediate code generation
Control flow graph with pseudo-instructions in basic blocks (medium-level IF)
Local redundancy elimination
Modified control flow graph
Global redundancy elimination
Modified control flow graph
Loop improvement
Modified control flow graph
Machine-independent
Target code generation
(Almost) assembly language (low-level IF)
Preliminary instruction scheduling
Modified assembly language
Register allocation
Modified assembly language
Final instruction scheduling
Modified assembly language
Peephole optimization
Final assembly language
Machine-specific



Example

Control Flow Graph

- ***Basic blocks*** are maximal-length set of sequential operations
  - Operations on a set of *virtual registers*
    - » Unlimited
    - » A new one for each computed value
- Arcs represent interblock control flow

Start

```
a1:= &input
call readint
i := rv
a1 := &input
call readint
j := rv
```

```
v1 := i
v2 := j
v3 := v1 <> v2
test v3
```

F

```
a1 := &output
v13 := i
a2 := v13
call writeint
a1 := &output
call writeln
```

End

T

```
v4 := i
v5 := j
v6 := v4 > v5
test v6
```

T

F

```
v7 := i
v8 := j
v9 := v7 - v8
i := v9
```

```
v10 := j
v11 := i
v12 := v10 - v11
j := v12
```

```
null
```

Redundancy Elimination in Basic Blocks

- We will consider the example on the right
- It computes the binomial coefficients

$$\binom{n}{m}$$

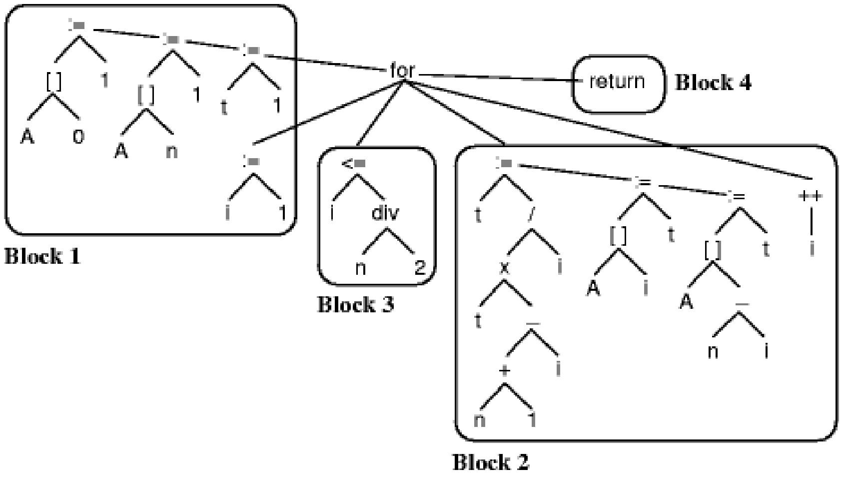
for $0 \leq m \leq n$

- It is based on

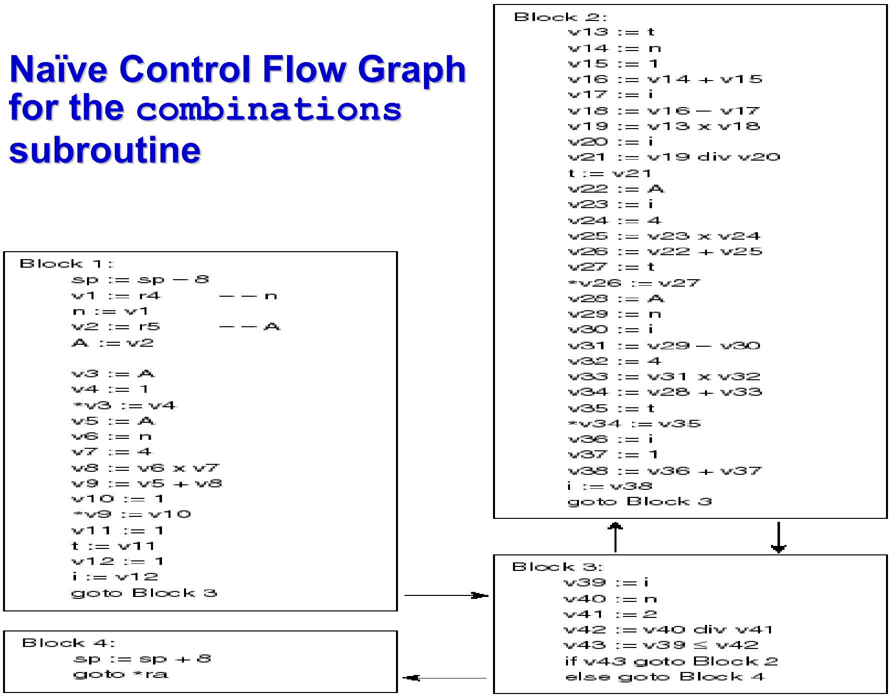
$$\binom{n}{m} = \binom{n}{n-m}$$

```
combinations (int n, int *A) {
    int i, t;
    A[0] = 1;
    A[n] = 1;
    t = 1;
    for (i = 1; i <= n/2; i++) {
        t = (t * (n+1-i)) / i;
        A[i] = t;
        A[n-i] = t;
    }
}
```

Syntax Tree for the `combinations` subroutine



Naïve Control Flow Graph for the `combinations` subroutine



```
Block 1:
        sp := sp - 8
        v1 := r4        -- n
        n := v1
        v2 := r5        -- A
        A := v2

        v3 := A
        v4 := 1
        *v3 := v4
        v5 := A
        v6 := n
        v7 := 4
        v8 := v6 x v7
        v9 := v5 + v8
        v10 := 1
        *v9 := v10
        v11 := 1
        t := v11
        v12 := 1
        i := v12
        goto Block 3
```

```
Block 2:
        v13 := t
        v14 := n
        v15 := 1
        v16 := v14 + v15
        v17 := i
        v18 := v16 - v17
        v19 := v13 x v18
        v20 := i
        v21 := v19 div v20
        t := v21
        v22 := A
        v23 := i
        v24 := 4
        v25 := v23 x v24
        v26 := v22 + v25
        v27 := t
        *v26 := v27
        v28 := A
        v29 := n
        v30 := i
        v31 := v29 - v30
        v32 := 4
        v33 := v31 x v32
        v34 := v28 + v33
        v35 := t
        *v34 := v35
        v36 := i
        v37 := 1
        v38 := v36 + v37
        i := v38
        goto Block 3
```

```
Block 3:
        v39 := i
        v40 := n
        v41 := 2
        v42 := v40 div v41
        v43 := v39 ≤ v42
        if v43 goto Block 2
        else goto Block 4
```

```
Block 4:
        sp := sp + 8
        goto *ra
```

Naïve Control Flow Graph

- Uses virtual registers
  - – A new register for each new value
- *ra* is the return addres, *fp* is the frame pointer
- *n*, *A*, *I* and *t* perform the appropriate displacement addressing with respect to the stack pointer (*sp*) register
- Parameter passing using *r4* and *r5*

Value Numbering

- How can we eliminate redundant loads and computations?
  - *Expression DAG*
  - *Value numbering*
- In value numbering, the compilers assigns the same name (*i.e., number*) to any two or symbolically equivalent computations (*i.e., values*)
- A dictionary is used to keep track of values that have already been loaded or computed

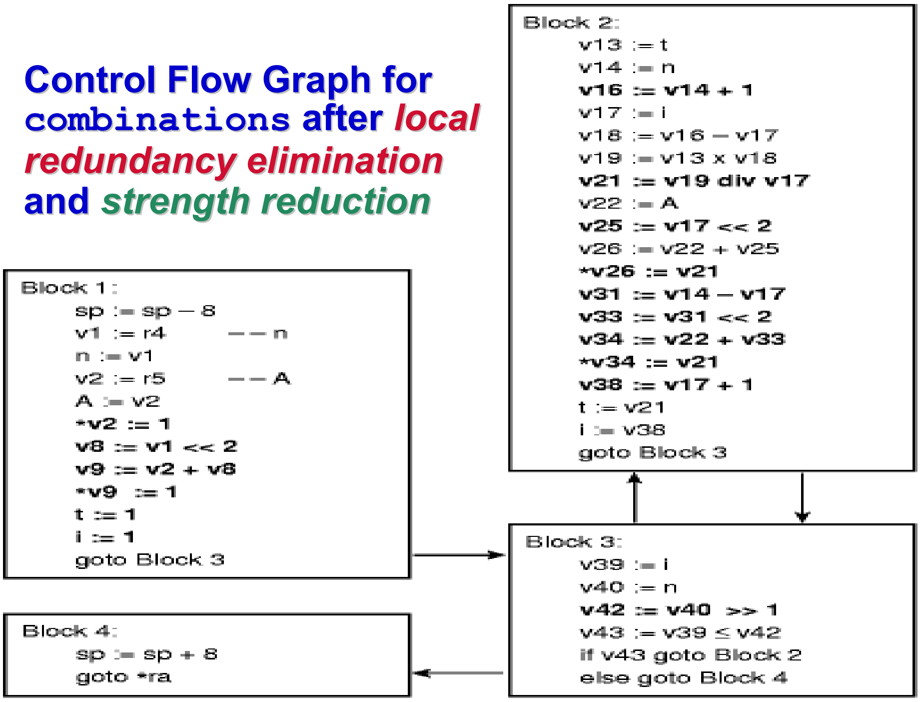
Value Numbering

- If a value is already in a register, reuse that register
  - *E.g.,* the load instruction can be eliminated v*i* := x if the value x is already in register v*j*
    - » *Replace all uses of* vi *by* vj
- Similarly, we can get rid of small constants using immediate value

Value Numbering

- In v*i* := v*j* op v*k*, we can use constant folding if the values in v*j* and v*k* are known to be constants
  - Local constant folding and constant propagation
  - At the same time, strength reduction and useless abstraction elimination
- A key that combine the registers and the operator is used to keep track of the previous operation in the dictionary

Control Flow Graph for `combinations` after *local redundancy elimination* and *strength reduction*



```
Block 1:
        sp := sp – 8
        v1 := r4          – – n
        n := v1
        v2 := r5          – – A
        A := v2
        *v2 := 1
        v8 := v1 << 2
        v9 := v2 + v8
        *v9  := 1
        t := 1
        i := 1
        goto Block 3
```

```
Block 2:
        v13 := t
        v14 := n
        v16 := v14 + 1
        v17 := i
        v18 := v16 – v17
        v19 := v13 x v18
        v21 := v19 div v17
        v22 := A
        v25 := v17 << 2
        v26 := v22 + v25
        *v26 := v21
        v31 := v14 – v17
        v33 := v31 << 2
        v34 := v22 + v33
        *v34 := v21
        v38 := v17 + 1
        t := v21
        i := v38
        goto Block 3
```

```
Block 3:
        v39 := i
        v40 := n
        v42 := v40  >> 1
        v43 := v39 ≤ v42
        if v43 goto Block 2
        else goto Block 4
```

```
Block 4:
        sp := sp + 8
        goto *ra
```

Reading Assignment

- Read Scott
  - Sect. 8.2.3
  - Ch. 13.3

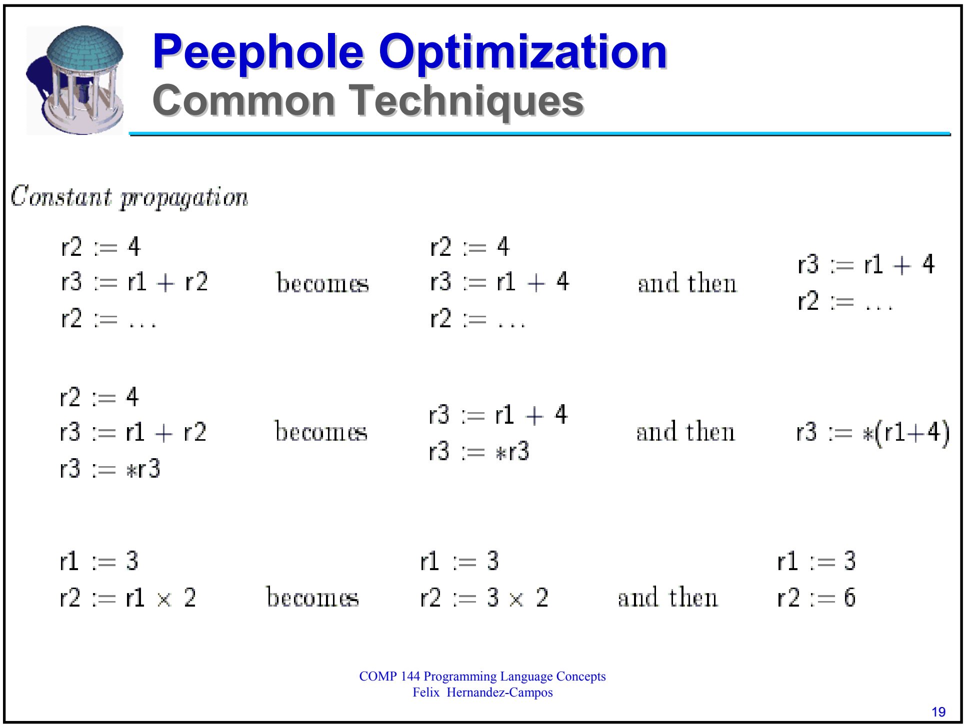
Peephole Optimization
Common Techniques

Elimination of redundant loads and stores

r2 := r1 + 5 i := r2 r3 := i r4 := r3 × 3	becomes	r2 := r1 + 5 i := r2 r4 := r2 × 3

Constant folding

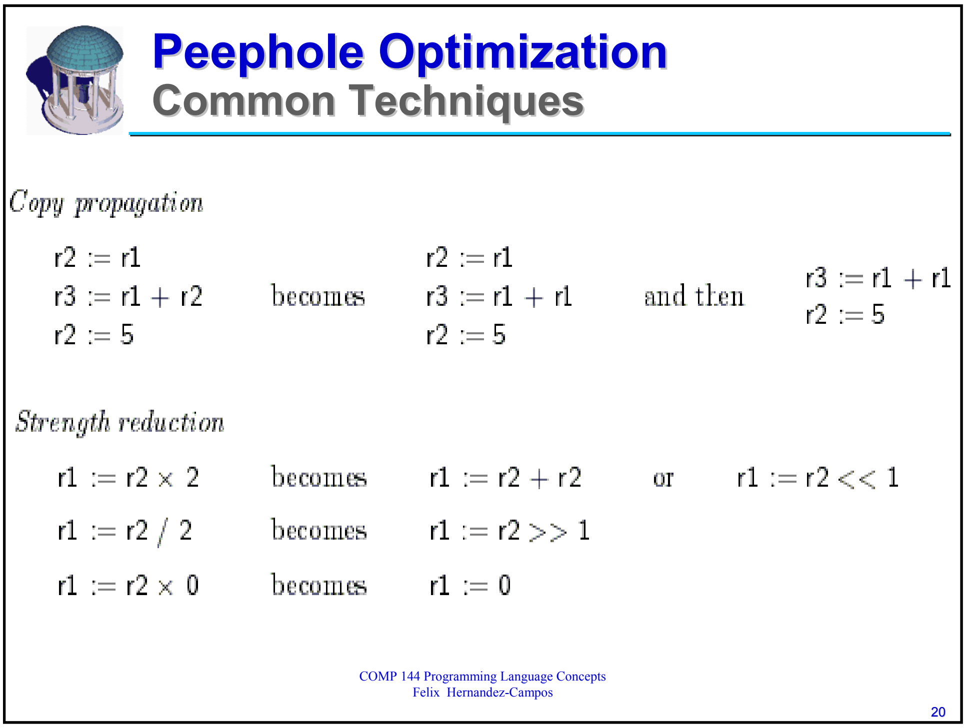
r2 := 3 × 2	becomes	r2 := 6



Peephole Optimization
Common Techniques

Constant propagation

r2 := 4 r3 := r1 + r2 r2 := . . .	becomes	r2 := 4 r3 := r1 + 4 r2 := . . .	and then	r3 := r1 + 4 r2 := . . .
r2 := 4 r3 := r1 + r2 r3 := ∗r3	becomes	r3 := r1 + 4 r3 := ∗r3	and then	r3 := ∗(r1+4)
r1 := 3 r2 := r1 × 2	becomes	r1 := 3 r2 := 3 × 2	and then	r1 := 3 r2 := 6



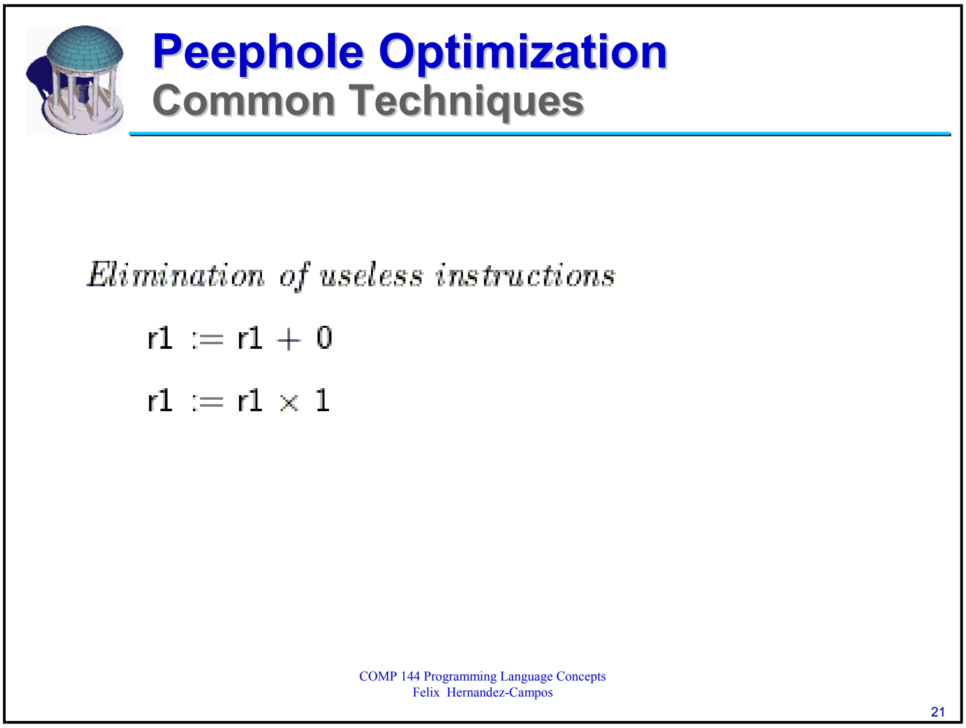
Peephole Optimization
Common Techniques

Copy propagation

r2 := r1 r3 := r1 + r2 r2 := 5	becomes	r2 := r1 r3 := r1 + r1 r2 := 5	and then	r3 := r1 + r1 r2 := 5

Strength reduction

r1 := r2 × 2	becomes	r1 := r2 + r2	or	r1 := r2 << 1
r1 := r2 / 2	becomes	r1 := r2 >> 1		
r1 := r2 × 0	becomes	r1 := 0		



Peephole Optimization
Common Techniques

Elimination of useless instructions

$\mathsf{r1} := \mathsf{r1} + 0$

$\mathsf{r1} := \mathsf{r1} \times 1$