THEORY OF COMPILATION

LECTURE 09

OPTIMIZATIONS

+1UP
INSTRUCTION SELECTION
You are here

Compiler

Source text

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Optimization

Code Generation

Executable code
Instruction Selection

• How do we choose which instructions to emit from our IR?
• Greatly depends on how we design our IR. We see two examples in this lecture:
  ‣ Tree-structured IR — Tiling
  ‣ 3AC sequence — Peephole Matching
Canonical IR — Tiling

Example:  
\[ x = x + 1; \]

Diagram: 
```
  MOVE
   /\  
  /   
MEM  +
   |   |
  |   |
FP   @x
```
```
  +
  |
MEM +
   |
  |
1
```
```
  +
  |
FP +
   |
  |
@x
```
Canonical IR — Tiling

Example:  \[ x = x + 1; \]

```
mov t1, [bp+x]
mov t2, t1
add t2, 1
mov [bp+x], t2
```
Tiles

• Capture compiler’s understanding of the instruction set

• Tile = sequence of instructions that update a fresh temporary (may need extra `mov`'s) and associated IR tree

• Each outgoing edge represents a temporary

```
mov t2, t1
add t2, 1
```
CISC vs. RISC Tiling

(e.g. ARM)  RISC  (e.g. x86)

MOVE
MEM
FP
@x

MEM + 1

MEM +

FP
@x

MOVE
MEM
FP
@x

MEM +

FP
@x

add [bp+x], 1
CISC vs. RISC Tiling

\begin{align*}
\text{RISC} & : \quad \text{move} t1, [bp+x] \\
& \quad \text{move} t2, t1 \\
& \quad \text{add} t2, 1 \\
& \quad \text{move} [bp+x], t2 \\
\text{CISC} & : \quad \text{add} [bp+x], 1
\end{align*}
CISC vs. RISC Tiling

(e.g. ARM) RISC

CISC (e.g. x86)

mov t1, [bp+x]
mov t2, t1
add t2, 1
mov [bp+x], t2

add [bp+x], 1

r/m32

CONST(k)
Example Tiles for “+”

Based on Intel Architecture Manual, Vol 2, 3-17
Computing a Tiling

- **Maximal Munch** — a greedy approach
  - Start at statement root
  - Find largest tile covering node, matching all children and outgoing edges
  - Invoke recursively on temporaries dangling from tile
  - Generate code for tile
    - (code for children will have been generated in recursive calls)
Timing Model

• **Idea:** associate **cost** with each tile (proportional to # cycles to execute)
  - sum of costs approximates execution time

\[
\text{Total cost} = 5
\]
Computing Optimum Tiling

- **Goal**: find minimum total cost tiling of tree
- **Algorithm**: for every node, find minimum total cost tiling of that node and sub-tree.
- **Lemma**: once minimum cost tiling for all descendants of a node is known, one can find minimum cost tiling of the node (sub-tree) by trying out all possible tiles matching the top
  ⇒ start from leaves, work upward to top node
Dynamic Programming

Example:

\[ a[i] \]
Dynamic Programming

Example:

\[
a[i]
\]

\[
\text{MEM} + \text{MEM} \times \text{CONST}(4) \text{MEM} + \text{FP} \text{CONST}(@a) + \text{FP} \text{CONST}(@i)
\]
Dynamic Programming

Example:

\[ a[i] \]

```
MEM
+--- MEM
    +--- +
    |    MEM
    |    *--- CONST(4)
    |          +--- MEM
    |              +--- FP
    |                   |--- CONST(@a)
    |                   +--- FP
                   |--- CONST(@i)
```
Dynamic Programming

Example:

\[ a[i] \]
Dynamic Programming

Example:

\[ a[i] \]
Dynamic Programming

Example:

$$a[i]$$
Dynamic Programming

Example:

\[ a[i] \]
Example:
\[ a[i] \]

```
movl a(%ebp), t1
movl i(%ebp), t2
movl (t1, t2, 4), t3
```
Peephole Matching

• **Basic idea**: discover local improvements locally
  ‣ Look at a small set of adjacent operations
  ‣ Move a small sliding window ("peephole") over code and search for improvement

• Classic examples:

  *p := R1
  R15 := *p

  *p := R1
  R15 := R1

  R7 := R2 + 0
  R10 := R4 * R7

  R7 := R2 + 0
  R10 := R4 * R2

  goto L10
  L10: goto L11

  store followed by load  algebraic identities  jump to jump
Peephole Matching

• How to implement it?

• Modern instruction selectors break problem into three tasks:  
  \textit{(Davidson, 1989)}

\begin{center}
\begin{tikzpicture}
\node[draw] (IR) at (0,0) {IR};
\node[draw] (Expander) at (2,-1) {Expander \newline IR$\rightarrow$LLIR};
\node[draw] (LLIR) at (4,-1) {LLIR};
\node[draw] (Simplifier) at (6,-1) {Simplifier \newline LLIR$\rightarrow$LLIR};
\node[draw] (Matcher) at (8,-1) {Matcher \newline LLIR$\rightarrow$ASM};
\node[draw] (ASM) at (10,-1) {ASM};
\draw[->] (IR) -- (Expander);
\draw[->] (Expander) -- (LLIR);
\draw[->] (LLIR) -- (Simplifier);
\draw[->] (Simplifier) -- (Matcher);
\draw[->] (Matcher) -- (ASM);
\end{tikzpicture}
\end{center}
Peephole Matching

**Expander**

- Turns IR code into a low-level IR (LLIR) such as RTL
- Operation-by-operation, template-driven rewriting
- LLIR form includes all direct effects \((e.g.,\) setting cc\)
- Significant, albeit constant, expansion of size

![Diagram](attachment:diagram.png)
Peephole Matching

**Simplifier**

- Looks at LLIR through window and rewrites is
- Uses forward substitution, algebraic simplification, local constant propagation, and dead-effect elimination
- Performs local optimization within window
Peephole Matching

Matcher

- Compares simplified LLIR against a library of patterns
- Picks low-cost pattern that captures effects
- Must preserve LLIR effects, may add new ones (e.g., set cc)
- Generates the assembly code output
## Peephole Matching — Example

### Original IR Code

<table>
<thead>
<tr>
<th>op</th>
<th>arg₁</th>
<th>arg₂</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>2</td>
<td>y</td>
<td>t₁</td>
</tr>
<tr>
<td>−</td>
<td>x</td>
<td>t₁</td>
<td>w</td>
</tr>
</tbody>
</table>

\[ t₁ = r₁₄ \]

\[ w = r₂₀ \]

### LLIR Code

\[ r₁₀ := 2 \]
\[ r₁₁ := @y \]
\[ r₁₂ := bp + r₁₁ \]
\[ r₁₃ := *(r₁₂) \]
\[ r₁₄ := r₁₀ * r₁₃ \]
\[ r₁₅ := @x \]
\[ r₁₆ := bp + r₁₅ \]
\[ r₁₇ := *(r₁₆) \]
\[ r₁₈ := r₁₇ − r₁₄ \]
\[ r₁₉ := @w \]
\[ r₂₀ := bp + r₁₉ \]
\[ *(r₂₀) := r₁₈ \]
Peephole Matching — Example

LLIR Code

\[ \begin{align*}
    r_{10} &:= 2 \\
    r_{11} &:= @y \\
    r_{12} &:= bp + r_{11} \\
    r_{13} &:= *(r_{12}) \\
    r_{14} &:= r_{10} * r_{13} \\
    r_{15} &:= @x \\
    r_{16} &:= bp + r_{15} \\
    r_{17} &:= *(r_{16}) \\
    r_{18} &:= r_{17} - r_{14} \\
    r_{19} &:= @w \\
    r_{20} &:= bp + r_{19} \\
    *(r_{20}) &:= r_{18}
\end{align*} \]

LLIR Code

\[ \begin{align*}
    r_{13} &:= *(bp + @y) \\
    r_{14} &:= 2 * r_{13} \\
    r_{17} &:= *(r_{ar} + @x) \\
    r_{18} &:= r_{17} - r_{14} \\
    *(bp + @w) &:= r_{18}
\end{align*} \]
Peephole Matching — Example

**LLIR Code**

\[
\begin{align*}
  r_{13} & := \ast (bp + @y) \\
  r_{14} & := 2 \ast r_{13} \\
  r_{17} & := \ast (r_{arp} + @x) \\
  r_{18} & := r_{17} - r_{14} \\
  *(bp + @w) & := r_{18}
\end{align*}
\]

**Assembly (ILOC) Code**

\[
\begin{align*}
  \text{loadAI} & \quad r_{arp},@y \Rightarrow r_{13} \\
  \text{multi} & \quad 2, r_{13} \Rightarrow r_{14} \\
  \text{loadAI} & \quad r_{arp},@x \Rightarrow r_{17} \\
  \text{sub} & \quad r_{17}, r_{14} \Rightarrow r_{18} \\
  \text{storeAI} & \quad r_{18} \Rightarrow r_{arp},@w
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
r_{10} & := 2 \\
r_{11} & := @y \\
r_{12} & := bp + r_{11} \\
r_{13} & := *(r_{12}) \\
r_{14} & := r_{10} * r_{13} \\
r_{15} & := @x \\
r_{16} & := bp + r_{15} \\
r_{17} & := *(r_{16}) \\
r_{18} & := r_{17} - r_{14} \\
r_{19} & := @w \\
r_{20} & := bp + r_{19} \\
*(r_{20}) & := r_{18}
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{10} := 2 \]
\[ r_{11} := @y \]
\[ r_{12} := bp + r_{11} \]
\[ r_{13} := *(r_{12}) \]
\[ r_{14} := r_{10} * r_{13} \]
\[ r_{15} := @x \]
\[ r_{16} := bp + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{10} := 2 \]
\[ r_{11} := \@y \]
\[ r_{12} := \text{bp} + r_{11} \]
\[ r_{13} := *(r_{12}) \]
\[ r_{14} := r_{10} \times r_{13} \]
\[ r_{15} := \@x \]
\[ r_{16} := \text{bp} + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := \@w \]
\[ r_{20} := \text{bp} + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{10} := 2 \]
\[ r_{12} := \text{bp} + \text{y} \]
\[ r_{13} := *(r_{12}) \]
\[ r_{14} := r_{10} * r_{13} \]
\[ r_{15} := \text{x} \]
\[ r_{16} := \text{bp} + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := \text{w} \]
\[ r_{20} := \text{bp} + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
r_{10} & := 2 \\
r_{12} & := bp + @y \\
r_{13} & := *(r_{12}) \\
r_{14} & := r_{10} \times r_{13} \\
r_{15} & := @x \\
r_{16} & := bp + r_{15} \\
r_{17} & := *(r_{16}) \\
r_{18} & := r_{17} - r_{14} \\
r_{19} & := @w \\
r_{20} & := bp + r_{19} \\
*(r_{20}) & := r_{18}
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{10} := 2 \]
\[ r_{13} := *(bp + @y) \]
\[ r_{14} := r_{10} * r_{13} \]
\[ r_{15} := @x \]
\[ r_{16} := bp + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
  r_{10} & := 2 \\
  r_{13} & := *(bp + @y) \\
  r_{14} & := r_{10} \ast r_{13} \\
  r_{15} & := @x \\
  r_{16} & := bp + r_{15} \\
  r_{17} & := *(r_{16}) \\
  r_{18} & := r_{17} - r_{14} \\
  r_{19} & := @w \\
  r_{20} & := bp + r_{19} \\
  *(r_{20}) & := r_{18} \\
  r_{12} & := *(bp + @y) \\
  r_{14} & := 2 \ast r_{13}
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{15} := @x \]
\[ r_{16} := bp + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{15} := @x \]
\[ r_{16} := bp + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{15} := @x \]
\[ r_{16} := bp + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{16} := bp + @x \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{16} := bp + @x \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
    r_{13} & := *(bp + @y) \\
    r_{14} & := 2 \times r_{13} \\
    r_{16} & := bp + @x \\
    r_{17} & := *(r_{16}) \\
    r_{18} & := r_{17} - r_{14} \\
    r_{19} & := @w \\
    r_{20} & := bp + r_{19} \\
    *(r_{20}) & := r_{18}
\end{align*}
\]

\[
\begin{align*}
    r_{14} & := 2 \times r_{13} \\
    r_{17} & := *(bp + @x)
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{14} := 2 \times r_{13} \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
    r_{13} &:= *(bp + @y) \\
    r_{14} &:= 2 * r_{13} \\
    r_{17} &:= *(bp + @x) \\
    r_{18} &:= r_{17} - r_{14} \\
    r_{19} &:= @w \\
    r_{20} &:= bp + r_{19} \\
    *(r_{20}) &:= r_{18}
\end{align*}
\]

\[
\begin{align*}
    r_{17} &:= *(bp + @x) \\
    r_{18} &:= r_{17} - r_{13} - r_{13}
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simpler

LLIR Code

\[ \begin{align*}
  r_{13} &:= (bp + @y) \\
r_{17} &:= (bp + @x) \\
r_{18} &:= r_{17} - r_{13} - r_{13} \\
r_{19} &:= @w \\
r_{20} &:= bp + r_{19} \\
*(r_{20}) &:= r_{18}
\end{align*} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ r_{20} := bp + @w \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ r_{20} := bp + @w \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ *(bp + @w) := r_{18} \]
Peephole Matching — Example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r_{10} := 2</td>
<td>r_{13} := *(bp + @y)</td>
</tr>
<tr>
<td>r_{11} := @y</td>
<td>r_{17} := *(r_{arp} + @x)</td>
</tr>
<tr>
<td>r_{12} := bp + r_{11}</td>
<td>r_{18} := r_{17} - r_{13} - r_{13}</td>
</tr>
<tr>
<td>r_{13} := *(r_{12})</td>
<td>*(bp + @w) := r_{18}</td>
</tr>
<tr>
<td>r_{14} := r_{10} * r_{13}</td>
<td></td>
</tr>
<tr>
<td>r_{15} := @x</td>
<td></td>
</tr>
<tr>
<td>r_{16} := bp + r_{15}</td>
<td></td>
</tr>
<tr>
<td>r_{17} := *(r_{16})</td>
<td></td>
</tr>
<tr>
<td>r_{18} := r_{17} - r_{14}</td>
<td></td>
</tr>
<tr>
<td>r_{19} := @w</td>
<td></td>
</tr>
<tr>
<td>r_{20} := bp + r_{19}</td>
<td></td>
</tr>
<tr>
<td>*(r_{20}) := r_{18}</td>
<td></td>
</tr>
</tbody>
</table>
Making Peephole Matching Work

Details.

- LLIR is largely machine independent (RTL)
- Target machine described as LLIR → ASM pattern
- Actual pattern matching
  - Use a hand-coded pattern matcher (gcc)
  - Turn patterns into grammar & use LR parser (VPO)
- Several important compilers use this technology
- It seems to produce good portable instruction selectors

Its key strength appears to be late low-level optimization.
Optimizations

- Improve program performance
- **Must maintain original semantics**
  - The observable behavior of the optimized program should be equivalent to that of the original program
- Typically cannot obtain “optimal program”
- Can optimize various aspects of program execution
  - Memory
  - Time
  - Code size
  - Power
  - ...
Why Optimize?

- Programmer may introduce inefficiencies
- Compiler may introduce inefficiencies
  - Make earlier compiler stages easier to deal with
Reminder: Data-flow Analysis

\[
x := 0 \\
y := 1
\]

\[\text{while (y < n)}\]

\[
x := x + 2 \\
y := y + 1
\]

print x
print y
Reminder: Data-flow Analysis

\[
\begin{align*}
x & := 0 \\
y & := 1
\end{align*}
\]

while \((y < n)\)

\[
\begin{align*}
x & := x + 2 \\
y & := y + 1
\end{align*}
\]

\{x \mapsto T, \ y \mapsto T, \ n \mapsto T\}

print \ x 

print \ y
Reminder: Data-flow Analysis

```
x := 0
y := 1
while (y < n)
x := x + 2
y := y + 1
print x
print y
```
Reminder: Data-flow Analysis

```
x := 0
y := 1
```

```
while (y < n)
  x := x + 2
  y := y + 1
```

```
print x
print y
```
Reminder: Data-flow Analysis

\[
\begin{align*}
x &:= 0 \\
y &:= 1
\end{align*}
\]

\[
\text{while } (y < n)
\]

\[
\begin{align*}
x &:= x + 2 \\
y &:= y + 1
\end{align*}
\]

\[
\text{print } x \\
\text{print } y
\]
Reminder: Data-flow Analysis

\[\begin{align*}
x &:= 0 \\
y &:= 1
\end{align*}\]

while \((y < n)\)

\[\begin{align*}
x &:= x + 2 \\
y &:= y + 1
\end{align*}\]

\{x \mapsto E, y \mapsto O, n \mapsto T\}

print \(x\)

print \(y\)
Reminder: Data-flow Analysis

\[ x := 0 \]
\[ y := 1 \]

\[ \text{while } (y < n) \]
\[ x := x + 2 \]
\[ y := y + 1 \]

\{ x \mapsto \top, y \mapsto \top, n \mapsto \top \}
\{ x \mapsto E, y \mapsto O, n \mapsto T \}
\{ x \mapsto E, y \mapsto O, n \mapsto T \}
\{ x \mapsto E, y \mapsto O, n \mapsto T \}
\{ x \mapsto E, y \mapsto O, n \mapsto T \}

\{ x \mapsto E, y \mapsto O, n \mapsto T \}

print x
print y
Reminder: Data-flow Analysis

\[
\begin{align*}
&x := 0 \\
y := 1
\end{align*}
\]

\[
\text{while (} y < n \text{)}
\]

\[
\begin{align*}
x &:= x + 2 \\
y &:= y + 1
\end{align*}
\]

\[
\text{print } x \\
\text{print } y
\]
Reminder: Data-flow Analysis

\[
\begin{align*}
&\text{while } (y < n) \\
&\quad \text{print } x \\
&\quad \text{print } y
\end{align*}
\]
Reminder: Data-flow Analysis

```
x := 0
y := 1
while (y < n)
x := x + 2
y := y + 1
print x
print y
```
Reminder: Data-flow Analysis

```
x := 0
y := 1

while (y < n)
  x := x + 2
  y := y + 1

print x
print y
```
Reminder: Data-flow Analysis

\[
\begin{align*}
\text{x := 0} & \quad \{x \mapsto \top, \ y \mapsto \top, \ n \mapsto \top\} \\
\text{y := 1} & \quad \{x \mapsto E, \ y \mapsto O, \ n \mapsto \top\} \\
\text{while (y < n)} & \quad \{x \mapsto E, \ y \mapsto \top \cup E, \ n \mapsto \top\} \quad \text{O \cup E = \top} \\
\text{x := x + 2} & \quad \{x \mapsto E, \ y \mapsto \top, \ n \mapsto \top\} \\
\text{y := y + 1} & \quad \{x \mapsto E, \ y \mapsto \top, \ n \mapsto \top\} \\
\text{print x} & \quad \{x \mapsto E, \ y \mapsto O, \ n \mapsto \top\} \\
\text{print y} & \quad \{x \mapsto E, \ y \mapsto \top, \ n \mapsto \top\}
\end{align*}
\]
Reminder: Data-flow Analysis

\[
\begin{align*}
x &:= 0 \\
y &:= 1
\end{align*}
\]

\[
\{x \mapsto T, \ y \mapsto T, \ n \mapsto T\}
\]

\[
\{x \mapsto E, \ y \mapsto O, \ n \mapsto T\}
\]

\[
\text{while (y < n)}
\]

\[
\{x \mapsto E, \ y \mapsto O \sqcup E, \ n \mapsto T\}
\]

\[
O \sqcup E = T
\]

\[
\{x \mapsto E, \ y \mapsto T, \ n \mapsto T\}
\]

\[
x := x + 2 \\
y := y + 1
\]

\[
\{x \mapsto E, \ y \mapsto T, \ n \mapsto T\}
\]

\[
\{x \mapsto E, \ y \mapsto T, \ n \mapsto T\}
\]

\[
\text{print } x \\
\text{print } y
\]

\[
\{x \mapsto E, \ y \mapsto T, \ n \mapsto T\}
\]
Reminder: Data-flow Analysis

\begin{align*}
x &:= 0 \\
y &:= 1
\end{align*}

\begin{align*}
\text{while } (y < n) &\Rightarrow \{x \mapsto E, y \mapsto O, n \mapsto T\} \\
x &:= x + 2 \\
y &:= y + 1
\end{align*}

\begin{align*}
\text{print } x &\Rightarrow \{x \mapsto E, y \mapsto T, n \mapsto T\} \\
\text{print } y &\Rightarrow \{x \mapsto E, y \mapsto T, n \mapsto T\}
\end{align*}

\begin{align*}
O \sqcup E &= T \\
\{x \mapsto E, y \mapsto O \sqcup E, n \mapsto T\} &\Rightarrow \{x \mapsto E, y \mapsto T, n \mapsto T\}
\end{align*}
Constant Propagation

• Idea: if on every run, a variable can only take one value, it can be replaced with a constant.
• DFA over the following lattice:
Constant Propagation

- Idea: if on every run, a variable can only take one value, it can be replaced with a constant.
- DFA over the following lattice:

![DFA Diagram]
Basic Optimizations

• Common-subexpression Elimination
  – Eliminate repeated computation inside a basic block (DAG representation)
  – Next time we will see how to do it beyond the boundaries of a single basic block

• Copy-propagation
  – Following an assignment $x = y$, try to use “$y$” whenever “$x$” is used (if possible)
  – Potentially makes “$x$” into a dead variable, saving the assignment $x = y$
Common Subexpression Elimination

- Avoid recomputations
  - ...but be careful:

    \[
    \begin{align*}
    a &= b + c \\
    b &= a - d \\
    c &= b + c \\
    d &= a - d 
    \end{align*}
    \]

    \[
    \begin{align*}
    a &= b + c \\
    b &= a - d \\
    c &= b + c \\
    d &= b 
    \end{align*}
    \]
Static Single-Assignment Form (SSA)

- Every assignment writes to a distinct variable
- Every variable is only assigned once

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```
if (f)
    x = 42;
else
    x = 73;
y = x * a;
SSA

• Branches?

if (f)
  x = 42;
else
  x = 73;
y = x * a;

if (f)
  x1 = 42;
else
  x2 = 73;
x3 = \phi(x1, x2);
y = x3 * a;

• \(\phi\) (phi) function combines different definitions
• \(\phi\) returns the value of \(x1\) if control passes through the true branch and the value of \(x2\) if it passed through the false branch
SSA

• Branches?

  if (f)
  x = 42;
else
  x = 73;
y = x * a;

  if (f)
    x1 = 42;
else
    x2 = 73;
x3 = φ(x1,x2);
y = x3 * a;

• φ (phi) function combines different definitions

• φ returns the value of x1 if control passes through the true branch and the value of x2 if it passed through the false branch
SSA — why should we care?

• Makes it easy to apply many optimizations
  ▸ constant propagation, dead code elimination...

$x = 42$
$x = 73$
y = x

$x1 = 42$
x2 = 73
y = x2$
DAG Representation of Basic Blocks

\[
\begin{align*}
  a &= b + c \\
  b &= a - d \\
  c &= b + c \\
  d &= a - d \\
\end{align*}
\]

\[
\begin{align*}
  a &= b_0 + c_0 \\
  b &= a - d_0 \\
  c &= b + c_0 \\
  d &= a - d_0 \\
\end{align*}
\]
DAG Representation of Basic Blocks

\[
\begin{align*}
a &= b + c \\
b &= b - d \\
c &= c + d \\
e &= b + c
\end{align*}
\]

\[
\begin{align*}
a &= b_0 + c_0 \\
b &= b_0 - d_0 \\
c &= c_0 + d_0 \\
e &= b + c
\end{align*}
\]
Aliasing is a problem

- We don’t know whether \( i = j \)
- Same thing happens with pointers
- Have to handle it conservatively
  - Don’t know that it’s safe \( \Rightarrow \) must assume that it’s conflicting
- A major obstacle to effective optimization

\[
x := a[i] \quad x := a[i]
\]
\[
a[j] := y \neq a[j] := y
\]
\[
z := a[i] \quad z := x
\]
Coming Up

MORE OPTIMIZATIONS (w/ LOOPS)
THEORY OF COMPILATION

LECTURE 09

OPTIMIZATIONS
Source text → Compiler → Executable code

Lexical Analysis → Syntax Analysis → Semantic Analysis → IR Optimization → Code Generation
Peephole Optimizations

• Optimizing long code sequences is hard
• A simple and efficient (but sub-optimal) alternative: peephole optimizations
  – Examine a small window (“peep hole”) over the code
  – Identify local optimization opportunities
  – Rewrite code “in the window”
• Example
  – Local (single statement) algebraic simplifications
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
    r_{10} & := 2 \\
    r_{11} & := \@y \\
    r_{12} & := bp + r_{11} \\
    r_{13} & := *(r_{12}) \\
    r_{14} & := r_{10} \times r_{13} \\
    r_{15} & := \@x \\
    r_{16} & := bp + r_{15} \\
    r_{17} & := *(r_{16}) \\
    r_{18} & := r_{17} - r_{14} \\
    r_{19} & := \@w \\
    r_{20} & := bp + r_{19} \\
    *(r_{20}) & := r_{18}
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
r_{10} & := 2 \\
r_{11} & := @y \\
r_{12} & := \text{bp} + r_{11} \\
r_{13} & := \ast(r_{12}) \\
r_{14} & := r_{10} \ast r_{13} \\
r_{15} & := @x \\
r_{16} & := \text{bp} + r_{15} \\
r_{17} & := \ast(r_{16}) \\
r_{18} & := r_{17} - r_{14} \\
r_{19} & := @w \\
r_{20} & := \text{bp} + r_{19} \\
\ast(r_{20}) & := r_{18}
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
    r_{10} & := 2 \\
    r_{11} & := \@y \\
    r_{12} & := \text{bp} + r_{11} \\
    r_{13} & := *(r_{12}) \\
    r_{14} & := r_{10} * r_{13} \\
    r_{15} & := \@x \\
    r_{16} & := \text{bp} + r_{15} \\
    r_{17} & := *(r_{16}) \\
    r_{18} & := r_{17} - r_{14} \\
    r_{19} & := \@w \\
    r_{20} & := \text{bp} + r_{19} \\
    *(r_{20}) & := r_{18}
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

**LLIR Code**

\[
\begin{align*}
  r_{10} & := 2 \\
  r_{12} & := \text{bp} + @y \\
  r_{13} & := *(r_{12}) \\
  r_{14} & := r_{10} \times r_{13} \\
  r_{15} & := @x \\
  r_{16} & := \text{bp} + r_{15} \\
  r_{17} & := *(r_{16}) \\
  r_{18} & := r_{17} - r_{14} \\
  r_{19} & := @w \\
  r_{20} & := \text{bp} + r_{19} \\
  *(r_{20}) & := r_{18}
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
    r_{10} & := 2 \\
    r_{12} & := \text{bp} + \@y \\
    r_{13} & := *(r_{12}) \\
    r_{14} & := r_{10} \cdot r_{13} \\
    r_{15} & := \@x \\
    r_{16} & := \text{bp} + r_{15} \\
    r_{17} & := *(r_{16}) \\
    r_{18} & := r_{17} - r_{14} \\
    r_{19} & := \@w \\
    r_{20} & := \text{bp} + r_{19} \\
    *(r_{20}) & := r_{18}
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

**LLIR Code**

\[ r_{10} := 2 \]
\[ r_{13} := *(bp + @y) \]
\[ r_{14} := r_{10} * r_{13} \]
\[ r_{15} := @x \]
\[ r_{16} := bp + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
  r_{10} & := 2 \\
  r_{13} & := *(bp + @y) \\
  r_{14} & := r_{10} * r_{13} \\
  r_{15} & := @x \\
  r_{16} & := bp + r_{15} \\
  r_{17} & := *(r_{16}) \\
  r_{18} & := r_{17} - r_{14} \\
  r_{19} & := @w \\
  r_{20} & := bp + r_{19} \\
  *(r_{20}) & := r_{18}
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := \times(bp + @y) \]
\[ r_{14} := 2 \times r_{13} \]
\[ r_{15} := @x \]
\[ r_{16} := bp + r_{15} \]
\[ r_{17} := \times(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ \times(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{15} := @x \]
\[ r_{16} := bp + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{15} := @x \]
\[ r_{16} := bp + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{16} := bp + @x \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{14} := 2 \times r_{13} \]
\[ r_{16} := bp + @x \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{16} := bp + @x \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]

\[ r_{14} := 2 * r_{13} \]
\[ r_{17} := *(bp + @x) \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + \@y) \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{17} := *(bp + \@x) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := \@w \]
\[ r_{20} := bp + r_{19} \]
\[ * (r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
\]

\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[
\begin{align*}
r_{13} & := *(bp + @y) \\
r_{17} & := *(bp + @x) \\
r_{18} & := r_{17} - r_{13} - r_{13} \\
r_{19} & := @w \\
r_{20} & := bp + r_{19} \\
* (r_{20}) & := r_{18}
\end{align*}
\]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]

\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ r_{20} := bp + @w \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ r_{20} := bp + @w \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ r_{20} := bp + @w \]
\[ *(r_{20}) := r_{18} \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ *(bp + @w) := r_{18} \]
Peephole Matching — Example

Steps of the Simplifier

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{17} := *(bp + @x) \]
\[ r_{18} := r_{17} - r_{13} - r_{13} \]
\[ *(bp + @w) := r_{18} \]
Peephole Matching — Example

LLIR Code

\[
\begin{align*}
  r_{10} & := 2 \\
r_{11} & := @y \\
r_{12} & := bp + r_{11} \\
r_{13} & := *(r_{12}) \\
r_{14} & := r_{10} * r_{13} \\
r_{15} & := @x \\
r_{16} & := bp + r_{15} \\
r_{17} & := *(r_{16}) \\
r_{18} & := r_{17} - r_{14} \\
r_{19} & := @w \\
r_{20} & := bp + r_{19} \\
*(r_{20}) & := r_{18}
\end{align*}
\]

Simplify

LLIR Code

\[
\begin{align*}
r_{13} & := *(bp + @y) \\
r_{17} & := *(arp + @x) \\
r_{18} & := r_{17} - r_{13} - r_{13} \\
*(bp + @w) & := r_{18}
\end{align*}
\]
Eliminating Redundant Code

(1) \( a = x \)
(2) \( x = a \)
(3) \( a = \text{someFunction}(a) \)
(4) \( x = \text{someOtherFunction}(a, x) \)
(5) if \( a > x \) goto (2)

(1) \( a = x \)
(2) \( a = \text{someFunction}(a) \)
(3) \( x = \text{someOtherFunction}(a, x) \)
(4) if \( a > x \) goto (2)

- Peephole optimizations are only performed within basic block boundaries
Code Motion

- Identify repeated computation inside a loop that depends only on values that are not modified inside the loop
- Move computation outside of the loop

```c
while (x - 3 < y) {
    // ... instructions that do not change x
}
```

```c
t_1 = x - 3;
while (t_1 < y) {
    // ... instructions that do not change x or t_1
}
```
Induction Variables & Strength Reduction

(1) \( i = 0; \)
(2) \( t_1 = i \times 4 \)
(3) \( t_2 = a[t_1] \)
(4) if \( t_2 > 100 \) goto (19)
(5) …

(17) \( i = i + 1 \)
(18) goto (2)
(19) …

(1) \( i = 0; \)
(2) \( t_1 = t_1 + 4 \)
(3) \( t_2 = a[t_1] \)
(4) if \( t_2 > 100 \) goto (19)
(5) …

(17) \( i = i + 1 \)
(18) goto (2)
(19) …

• Identify loop counters and relationship with other variables
Program Transformations

- Code motion and strength reduction can improve performance, provided that they maintain program semantics.

  - When does an optimization apply to a given program?

  - How can this be done automatically?
Program Transformations

• How can a loop be identified?
  ‣ For loop invariant code motion

• Easy to detect in high-level language

• Hard to detect in low-level language and CFG
  ‣ Java bytecode
  ‣ Control Flow Graph
Loop detection

- Loop in CFG has:
  - Loop header
  - Back edge
  - Set of nodes
Dominators

• Use dominators to identify loops
• Node d dominates node n
  – all path from the entry node to n go through d

• Every node dominates itself
Dominators
Dominators

- 1 dominates 1, 2, 3, 4
Dominators

- 1 dominates 1, 2, 3, 4
- 2 dominates 2
Dominators

• 1 dominates 1, 2, 3, 4
• 2 dominates 2
• 3 dominates 3
Dominators

• 1 dominates 1, 2, 3, 4
• 2 dominates 2
• 3 dominates 3
• 4 dominates 4
Loop detection

- **Header**
  - dominates loop nodes

- **Back edge**
  - Target dominates source

- **Loop identification**
  - Back edge identification
Post dominators

- Node d post-dominates node n
  - all path from n to the exit node go through d

- d post-dominates n does not imply n dominates d
Dominators
Dominators

- 1 post-dominates 1
Dominators

• 1 post-dominates 1
• 2 post-dominates 2
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
- 4 post-dominates 1, 2, 3, 4
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
- 4 post-dominates 1, 2, 3, 4
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
- 4 post-dominates 1, 2, 3, 4
- 1 dominates 4
Dominators
Dominators

- 1 post-dominates 1
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
- 4 post-dominates 1, 2, 3, 4
Dominators

- 1 post-dominates 1
- 2 post-dominates 2
- 3 post-dominates 3
- 4 post-dominates 1, 2, 3, 4

- 1 does not dominate 4
Computing Dominators

• Using DFA (previous lecture)
• If 1 dominates 2, 3, 4 and 5
  – d dominates 6
• If 1 dominates 6
  – 1 dominates 2, 3, 4 and 5
Computing Dominators

- $\text{Dom}(u)$
  - Set of nodes that dominate $u$
- $\text{Dom}(u_0) = \{u_0\}$
- $\text{Dom}(u) = \bigcap \{\text{Dom}(v) \mid v \in \text{pred}(u)\} \cup \{u\}$
Identify Loops

- **Back edge**
  - Edge $t \xrightarrow{} h$ s.t. $h$ dominates $t$

- **Loop of a back edge** $t \xrightarrow{} h$
  - $h$ is the **loop header**
  - Loop nodes
    - all nodes that can reach $t$ without going through $t$
Identify Loops: Algorithm

• Compute dominator relation
• Identify back edges
• Compute the loop nodes for each back edge
Identify Loops: Algorithm

for each node $h$ in dominator tree
  for each node $n$ for which there exists a back edge $n \leadsto h$
    define the loop with header := $h$
    body := of all nodes reachable from $n$
    by a depth first search backwards from $n$ that stops at $h$
Loop Preheader

- Needed by several optimizations
  - Loop invariant code motion
- Add additional code
Loop Optimization

- After identifying the loops
- We can use this info for loop optimization
- Loop invariant code motion
- Strength reduction of induction variable
- Induction variable elimination
Loop invariant code motion

for (i=0; i <10; ++i)
    a[i] = 10*i + x*x;
Loop invariant code motion

\[
\text{for (i=0; i <10; ++i)} \\
\quad \text{a[i] = 10*i + x*x;}
\]

\[
t = x*x;
\]

\[
\text{for (i=0; i <10; ++i)} \\
\quad \text{a[i] = 10*i + t;}
\]
Loop invariant computation

- $a := b \land c$ is loop invariant if $b$ and $c$:
  - Are constant, or
  - Have only definitions outside the loop, or
  - Have only one definition (each) that are themselves loop invariant

- Reaching definition analysis can be used (DFA)
Loop invariant computation

INV = ∅
repeat
    for each instruction I in loop such that I ∉ INV
        if operands are constants, or operands have no definitions inside the loop, or operands have exactly one definition d ∈ INV
        then INV = INV ∪ {I}
    until no changes in INV
Code motion

• Assume \( a := b \diamond c \) is loop invariant

• When can we hoist it out of the loop?

• Where to?
Valid code motion

Move d: a := b ◇ c to pre header is valid when:

1. d dominates all loop exits where a is live
2. d is the only definition of a in the loop
3. All uses of a in loop can only be reached from d
Valid code motion

1. d dominates all loop exits where a is live

```c
a = 0;
for (i=0; i<10; ++i)
    if (f(i)) a = x*x; else break;
b = a;
```
Valid code motion

1. d dominates all loop exits where a is live

```c
a = 0;
for (i=0; i<10; ++i)
    if (f(i)) a = x*x; else break;

b = a;
```

Use dominator relation to check whether each loop exit is dominated by d
Valid code motion

2. d is the only definition of a in the loop

```c
for (i=0; i<10; ++i)
    if (f(i)) a = x*x;
else a = 0;
```
Valid code motion

2. d is the only definition of a in the loop

```c
for (i=0; i<10; ++i)
    if (f(i)) a = x*x;
    else a = 0;

✗
```

Scan loop body for any other definitions of a
Valid code motion

3. All uses of a in loop can only be reached from d

```cpp
a = 0;
for (i=0; i<10; ++i)
    if (f(i)) a = x*x;
else buf[i] = a;
```
Valid code motion

3. All uses of $a$ in loop can only be reached from $d$

```c
a = 0;
for (i=0; i<10; ++i)
    if (f(i)) a = x*x;
else buf[i] = a;
```

Apply reaching definitions analysis and check each use of $a$ for any definitions of $a$ other than $d$
Summary

• Optimizations
  ✓ Not real “optimal” code
  ✓ But code that is “better” in some respect
  ✓ Often focuses on runtime

• Techniques
  ✓ Local: data-flow DAG representation, peephole
  ✓ Global: (function-level) Code motion, strength reduction — based on running a DFA first
Next