Instruction Selection

Note by Baris Aktemur:
Our slides are adapted from Cooper and Torczon's slides that they prepared for COMP 412 at Rice.

The Problem

Modern computers (still) have many ways to do anything

Consider register-to-register copy in ILOC

- Obvious operation is `i2i` $r_i \Rightarrow r_j$
- Many others exist

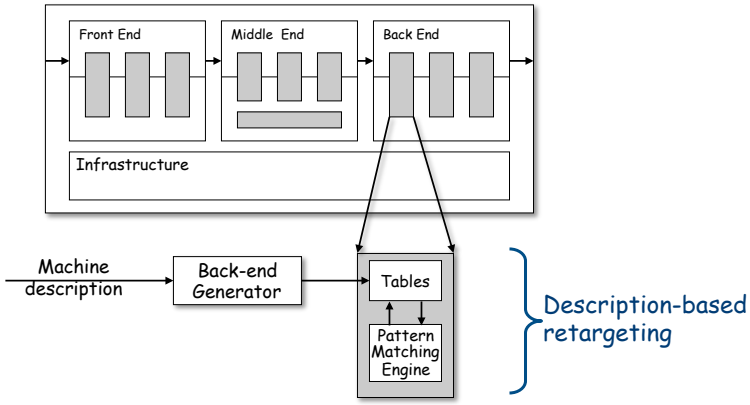
`addI` $r_i,0 \Rightarrow r_j$	`subI` $r_i,0 \Rightarrow r_j$	`lshiftI` $r_i,0 \Rightarrow r_j$
`multI` $r_i,1 \Rightarrow r_j$	`divI` $r_i,1 \Rightarrow r_j$	`rshiftI` $r_i,0 \Rightarrow r_j$
`orI` $r_i,0 \Rightarrow r_j$	`xorI` $r_i,0 \Rightarrow r_j$	*... and others ...*

- Human would ignore all of these
- Algorithm must look at all of them & find low-cost encoding
  - Take context into account (*busy functional unit?*)

And ILOC is an overly-simplified case

The Goal

Want to automate generation of instruction selectors



Machine description should also help with scheduling & allocation

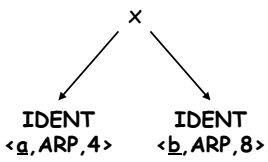
The Big Picture

Need pattern matching techniques

- Must produce good code (*some metric for good*)
- Must run quickly

Our treewalk code generator (Lec. 22) ran quickly
How good was the code?

Tree



Treewalk Code

```
loadI   4      ⇒ r5
loadAO  r0,r5  ⇒ r6
loadI   8      ⇒ r7
loadAO  r0,r7  ⇒ r8
mult    r6,r8  ⇒ r9
```

Desired Code

```
loadAI  r0,4  ⇒ r5
loadAI  r0,8  ⇒ r6
mult    r5,r6 ⇒ r7
```

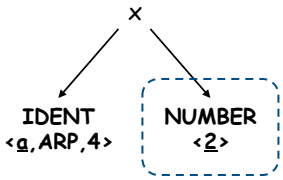
The Big Picture

Need pattern matching techniques

- Must produce good code (*some metric for good*)
- Must run quickly

Our treewalk code generator (Lec. 22) ran quickly
How good was the code?

Tree



Treewalk Code

```
loadI   4      ⇒ r5
loadAO  r0,r5  ⇒ r6
loadI   2      ⇒ r7
mult    r6,r7  ⇒ r8
```

Desired Code

```
loadAI  r0,4  ⇒ r5
multI   r5,2  ⇒ r7
```

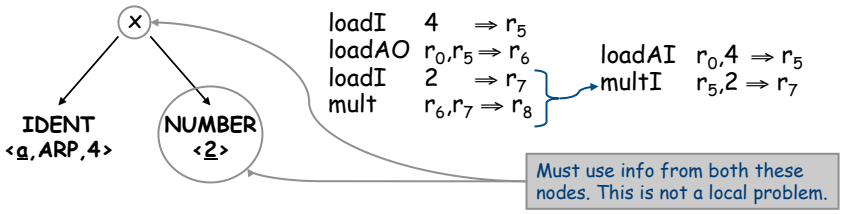

The Big Picture

Need pattern matching techniques

- Must produce good code (*some metric for good*)
- Must run quickly

Our treewalk code generator (Lec. 22) ran quickly
How good was the code?

Tree



Treewalk Code

```
loadI   4      ⇒ r5
loadAO  r0,r5  ⇒ r6
loadI   2      ⇒ r7
mult    r6,r7  ⇒ r8
```

Desired Code

```
loadAI  r0,4  ⇒ r5
multI   r5,2  ⇒ r7
```

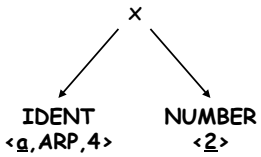
The Big Picture

Need pattern matching techniques

- Must produce good code (*some metric for good*)
- Must run quickly

Our treewalk code generator (Lec. 22) ran quickly
How good was the code?

Tree



Treewalk Code

```
loadI   4     ⇒ r5
loadAO  r0,r5 ⇒ r6
loadI   2     ⇒ r7
mult    r6,r7 ⇒ r8
```

Desired Code

```
loadAI  r0,4  ⇒ r5
add     r5,r5 ⇒ r7
```

Another possibility that might take less time & energy — an algebraic identity

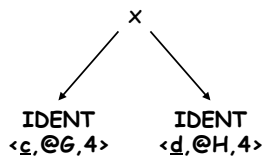
The Big Picture

Need pattern matching techniques

- Must produce good code (*some metric for good*)
- Must run quickly

Our treewalk code generator (Lec. 22) ran quickly
How good was the code?

Tree



Treewalk Code

```
loadI   @G    ⇒ r5
loadI   4     ⇒ r6
loadAO  r5,r6 ⇒ r7
loadI   @H    ⇒ r7
loadI   4     ⇒ r8
loadAO  r8,r9 ⇒ r10
mult    r7,r10⇒ r11
```

Desired Code

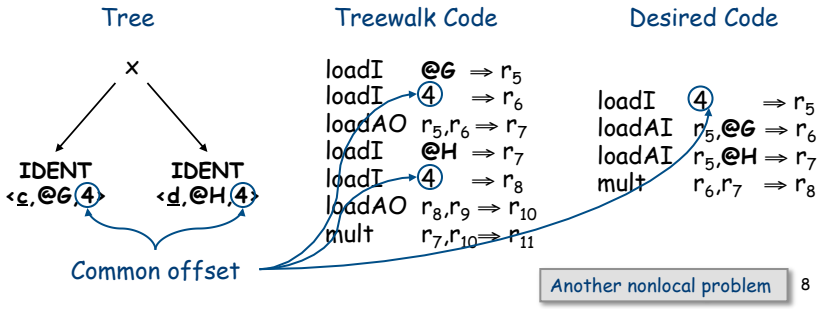
```
loadI   4     ⇒ r5
loadAI  r5,@G ⇒ r6
loadAI  r5,@H ⇒ r7
mult    r6,r7 ⇒ r8
```

The Big Picture

Need pattern matching techniques

- Must produce good code (*some metric for good*)
- Must run quickly

Our treewalk code generator met the second criteria *(lec. 22)*
How did it do on the first ?



8

How do we perform this kind of matching ?

Tree-oriented IR suggests pattern matching on trees

- Process takes tree-patterns as input, matcher as output
- Each pattern maps to a target-machine instruction sequence
- Use dynamic programming or bottom-up rewrite systems

Linear IR suggests using some sort of string matching

- Process takes strings as input, matcher as output
- Each string maps to a target-machine instruction sequence
- Use text matching (Aho-Corasick) or peephole matching

In practice, both work well; matchers are quite different

Peephole Matching

Basic idea

- Compiler can discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a "peephole" over code & search for improvement
- Classic example was store followed by load

Original code	Improved code
storeAI $r_1 \Rightarrow r_0,8$	storeAI $r_1 \Rightarrow r_0,8$
loadAI $r_0,8 \Rightarrow r_{15}$	i2i $r_1 \Rightarrow r_{15}$

Peephole Matching

Basic idea

- Compiler can discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a "peephole" over code & search for improvement
- Classic example was store followed by load
- Simple algebraic identities

Original code	Improved code
addI $r_2,0 \Rightarrow r_7$	mult $r_4,r_2 \Rightarrow r_{10}$
mult $r_4,r_7 \Rightarrow r_{10}$	
multI $r_5,2 \Rightarrow r_7$	add $r_2,r_2 \Rightarrow r_7$

See Table on p 401 of EaC (§8.3)

Peephole Matching

Basic idea

- Compiler can discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a "peephole" over code & search for improvement
- Classic example was store followed by load
- Simple algebraic identities
- Jump to a jump