Compilers

Instruction selection

Yannis Smaragdakis, U. Athens
(original slides by Sam Guyer@Tufts)

Back end

Essential tasks:

- Register allocation
  - Low-level IR assumes unlimited registers
  - Map to actual resources of machines
  - Goal: maximize use of registers
- Instruction selection
  - Map low-level IR to actual machine instructions
  - Not necessarily 1-1 mapping
  - CISC architectures, addressing modes

Instruction Selection

- Low-level IR different from machine ISA
  - Why?
  - Allow different back ends
  - Abstraction – to make optimization easier

- Differences between IR and ISA
  - IR: simple, uniform set of operations
  - ISA: many specialized instructions

- Often a single instruction does work of several operations in the IR

Instruction Selection

- Easy solution
  - Map each IR operation to a single instruction
  - May need to include memory operations

```
x = y + z;
```

```
mov y, r1
mov z, r2
add r2, r1
mov r1, x
```

- Problem: inefficient use of ISA

Instruction Selection

- Instruction sets
  - ISA often has many ways to do the same thing
  - ***Idiom***:

    A single instruction that represents a common pattern or sequence of operations

- Consider a machine with the following instructions:

`add r2, r1`	**r1 ← r1 + r2**
`muli c, r1`	**r1 ← r1 * c**
`load r2, r1`	**r1 ← *r2**
`store r2, r1`	***r1 ← r2**
`movem r2, r1`	***r1 ← *r2**
`movex r3, r2, r1`	***r1 ← *(r2 + r3)**

Sometimes **(r2)**

Example

- Generate code for:

```
a[i+1] = b[j]
```

- Simplifying assumptions
  - All variables are globals

    (No stack offset computation)
  - All variables are in registers

    (Ignore load/store of variables)

LIR

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

Possible Translation

	IR	*Assembly*
• Address of b[j]:	`t1 = j*4` `t2 = b+t1`	`muli 4, rj` `add rj, rb`
• Load value b[j]:	`t3 = *t2`	`load rb, r1`

• Address of a[i+1]:	`t4 = i+1`
	`t5 = t4*4`
	`t6 = a+t5`
		`addi 1, ri`
		`muli 4, ri`
		`add ri, ra`

• Store into a[i+1]:	`*t6 = t3`	`store r1, ra`

Another Translation

	IR	*Assembly*
Address of b[j]:	`t1 = j*4`	`muli 4, rj`
	`t2 = b+t1`	`add rj, rb`
(no load)	`t3 = *t2`	
Address of a[i+1]:	`t4 = i+1`	`addi 1, ri`
	`t5 = t4*4`	`muli 4, ri`
	`t6 = a+t5`	`add ri, ra`
Store into a[i+1]:	`*t6 = t3`	`movem rb, ra`

Direct memory-to-memory operation

Yet Another Translation

	IR	Assembly
• Index of b[j]:	`t1 = j*4`	`muli 4, rj`
	`t2 = b+t1`	
• (no load)	`t3 = *t2`	
• Address of a[i+1]:	`t4 = i+1`	`addi 1, ri`
	`t5 = t4*4`	`muli 4, ri`
	`t6 = a+t5`	`add ri, ra`
• Store into a[i+1]:	`*t6 = t3`	`movex rj,rb,ra`

Compute the address of b[j] in the memory move operation

`movex rj, rb, ra` **$*ra \leftarrow *(rj + rb)$**

Different translations

- Why is last translation preferable?
  - Fewer instructions
  - Instructions have different costs
    - Space cost: size of each instruction
    - Time cost: number of cycles to complete
- Example

`add r2, r1`	**cost = 1 cycle**
`muli c, r1`	**cost = 10 cycles**
`load r2, r1`	**cost = 3 cycles**
`store r2, r1`	**cost = 3 cycles**
`movem r2, r1`	**cost = 4 cycles**
`movex r3, r2, r1`	**cost = 5 cycles**

Idioms are cheaper than constituent parts

Wacky x86 idioms

- What does this do?

```
xor %eax, %eax
```

- Why not use this?

```
mov $0, %eax
```

- Answer:
  - Immediate operands are encoded in the instruction, making it bigger and therefore more costly to fetch and execute

More wacky x86 idioms

- What does this do?

```
xor        %ebx, %eax
xor        %eax, %ebx
xor        %ebx, %eax
```

eax = b ⊕ a
ebx = (b ⊕ a) ⊕ b = ?
eax = a ⊕ (b ⊕ a) = ?

- Swap the values of %eax and %ebx
- Why do it this way?
- No need for extra register!

Minimizing cost

- Goal:
  - Find instructions with low overall cost

- Difficulty
  - How to find these patterns?
  - Machine idioms may subsume IR operations that are not adjacent

- Idea: back to tree representation
  - Convert computation into a tree
  - Match parts of the tree

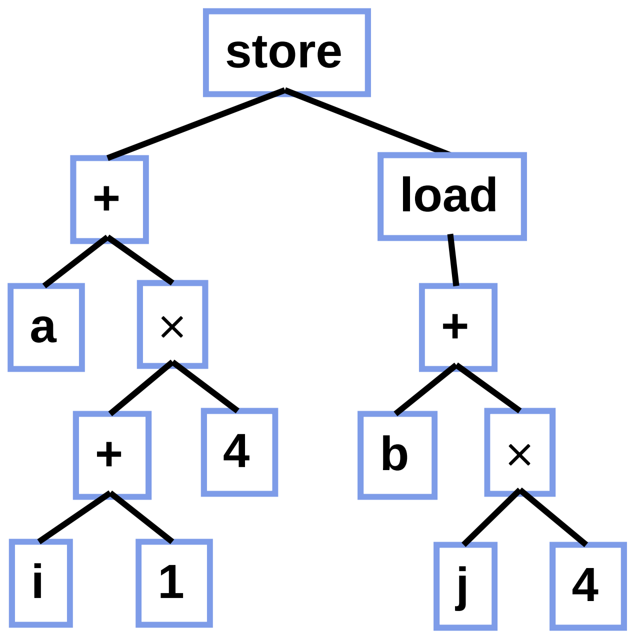
IR

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

```
movem rb, ra
```

Tree Representation

- Build a tree: `a[i+1] = b[j]`



IR

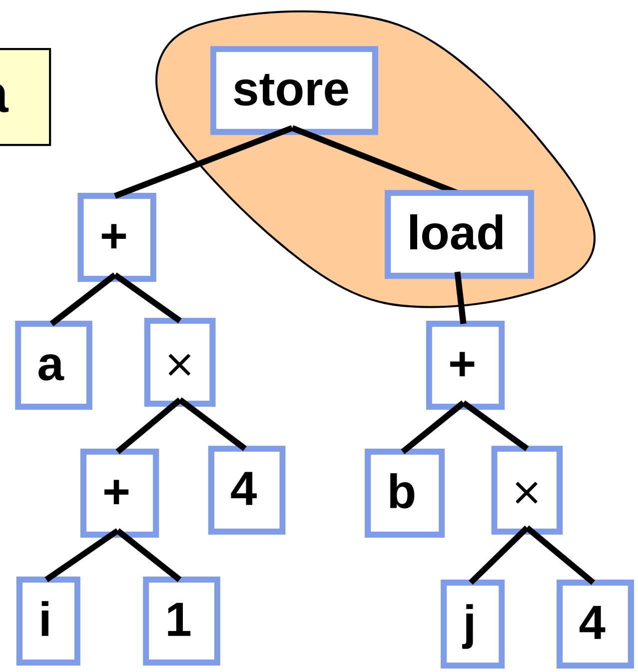
```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

- Goal: find parts of the tree that correspond to machine instructions

Tiles

- Idea: a ***tile*** is contiguous piece of the tree that correponds to a machine instruction

```
movem rb, ra
```

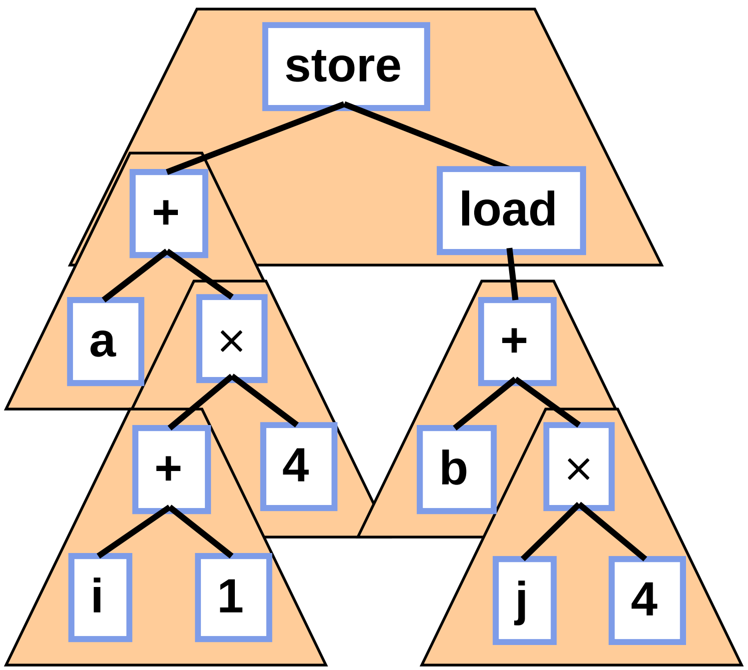


IR

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

Tiling

- *Tiling*: cover the tree with tiles

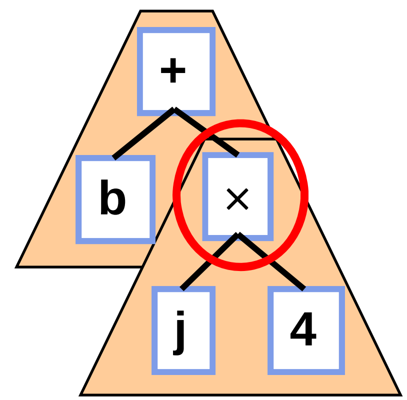


Assembly

```
muli 4, rj
add rj, rb
addi 1, ri
muli 4, ri
add ri, ra
movem rb, ra
```

Generating code

- Given a tiling of a tree
  - A tiling ***implements*** a tree if:
    - It covers all nodes in the tree
    - The overlap between tiles is exactly one node
- Post-order tree walk
  - Emit machine instructions for each tile
  - Tie boundaries together with registers
  - Note: order of children matters



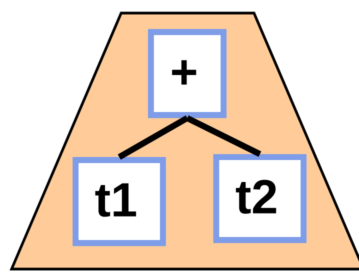
Tiling

- What's hard about this?
  - Define system of tiles in the compiler
  - Finding a tiling that implements the tree

    (Covers all nodes in the tree)
  - Finding a "good" tiling

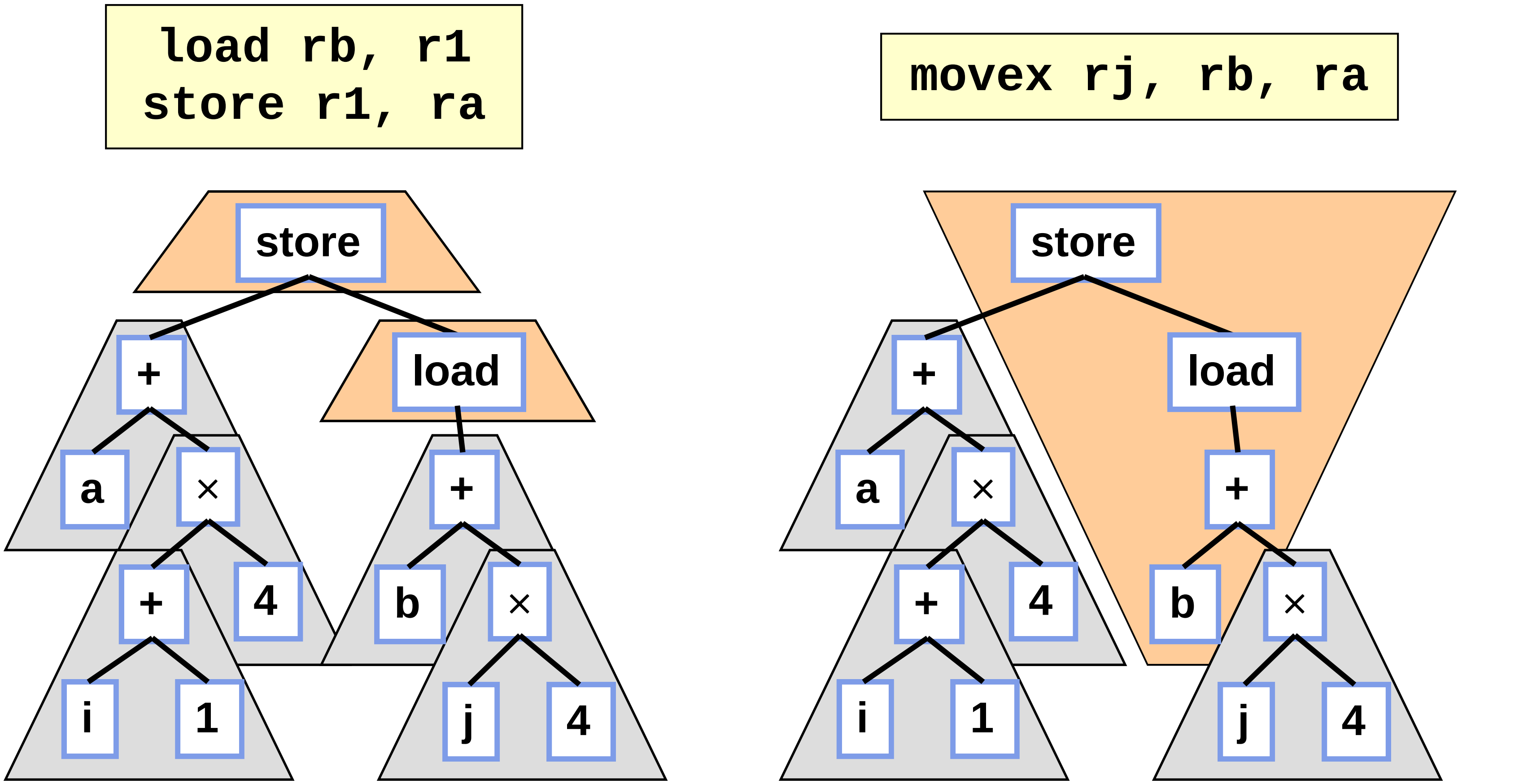
Interesting result (Dias and Ramsey): in general, undecidable

- Different approaches
  - Ad-hoc pattern matching
  - Automated tools




```
mov   t1, t3
add   t2, t3
```

Tiling



load rb, r1
store r1, ra
store
+
load
a
×
+
+
4
b
×
i
1
j
4
movex rj, rb, ra
store
+
load
a
×
+
+
4
b
×
i
1
j
4

Algorithms

- Goal: find a tiling with the fewest tiles
- Ad-hoc top-down algorithm
  - Start at top of the tree
  - Find largest tile matches top node
  - Tile remaining subtrees recursively

```
Tile(n) {
 if ((op(n) == PLUS) &&
        (left(n).isConst()))
  {
    Code c = Tile(right(n));
    c.append(ADDI left(n) right(n))
  }
}
```

Ad-hoc algorithm

- Problem: what does tile size mean?
  - Not necessarily the best fastest code
    (Example: multiply vs add)
  - How to include cost?

- Idea:
  - Total cost of a tiling is sum of costs of each tile

- Goal: find a minimum cost tiling

Dynamic programming

Including cost:

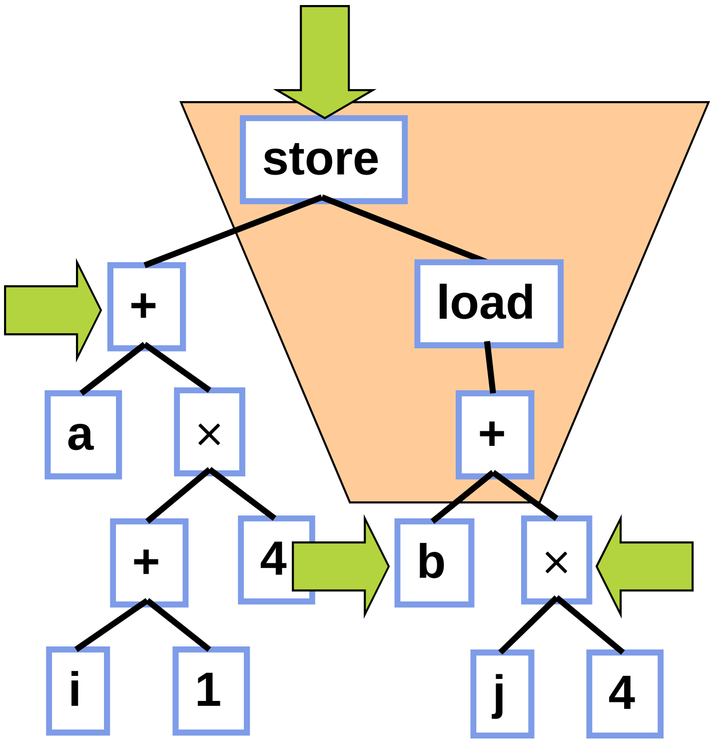
- Idea
  - For problems with ***optimal substructure***
  - Compute optimal solutions to sub-problems
  - Combine into an optimal overall solution
- How does this help?
  - Use ***memoization***:

    Save previously computed solutions to sub-problems
  - Sub-problems recur many times
  - Can work top-down or bottom-up

Recursive algorithm

- Memoization
  - For each subtree, record best tiling in a table
  - (*Note*: need a quick way to find out if we've seen a subtree before – some systems use DAGs instead of trees)
- At each node
  - First check table for optimal tiling for this node
  - If none, try all possible tiles, remember lowest cost
  - Record lowest cost tile in table
  - Greedy, top-down algorithm
- We can emit code from table

Pseudocode




```
Tile(n) {
 if (best(n)) return best(n)
 // -- Check all tiles
 if ((op(n) == STORE) &&
     (op(right(n)) == LOAD) &&
     (op(child(right(n)) == PLUS)) {
   Code c = Tile(left(n))
   c.add(Tile(left(child(right(n)))
   c.add(Tile(right(child(right(n)))
   c.append(MOVEX . . .)
   if (cost(c) < cost(best(n))
     best(n) = c
  }
 // . . . and all other tiles . . .
 return best(n)
}
```

Ad-hoc algorithm

- Problem?
  - Hard-codes the tiles in the code generator

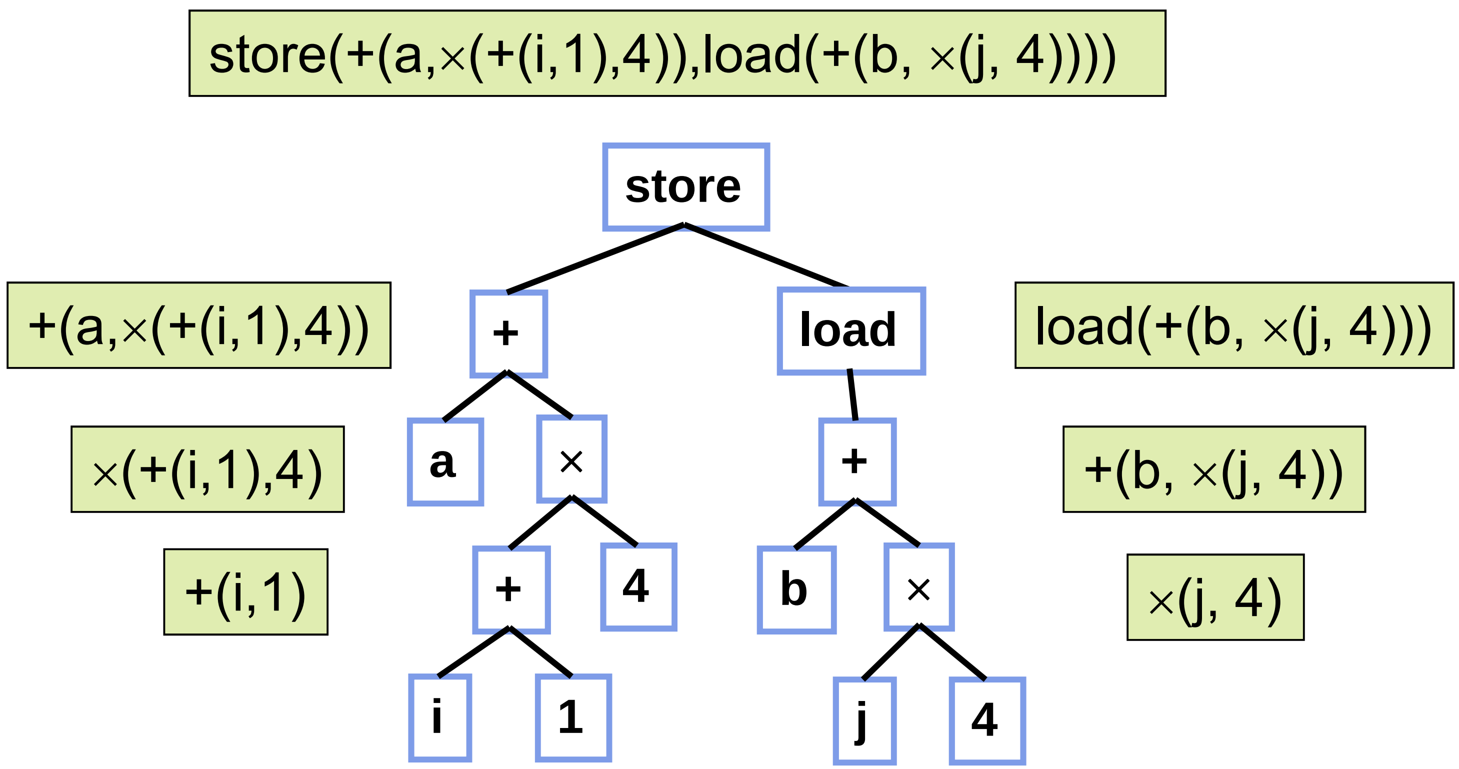
- Alternative:
  - Define tiles in a separate specification
  - Use a generic tree pattern matching algorithm to compute tiling
  - Tools: *code generator generators*
  - Probably overkill for RISC

Code generator generators

- Tree description language
  - Represent IR tree as text
- Specification
  - IR tree patterns
  - Code generation actions
- Generator
  - Takes the specification
  - Produces a code generator

Tree notation

- Use prefix notation to avoid confusion



Rewrite rules

- Rule
  - Pattern to match and replacement
  - Cost
  - Code generation template
  - May include actions – e.g., generate register name

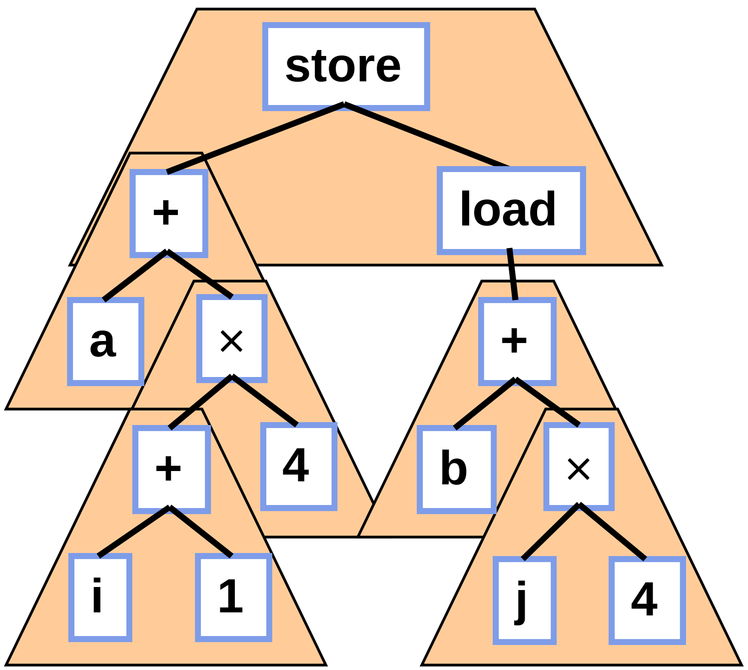
Pattern, replacement	*Cost*	*Template*
$+(reg_1,reg_2) \rightarrow reg_2$	1	add r1, r2
$store(reg_1, load(reg_2)) \rightarrow$ *done*	5	movem r2, r1

Rewrite rules

- Example rules:

#	*Pattern, replacement*	*Cost*	*Template*
1	$+(reg_1,reg_2) \rightarrow reg_2$	1	`add r1, r2`
2	$\times(reg_1,reg_2) \rightarrow reg_2$	10	`mul r1, r2`
3	$+(num,reg_1) \rightarrow reg_2$	1	`addi num, r1`
4	$\times(num,reg_1) \rightarrow reg_2$	10	`muli num, r1`
5	$store(reg_1, load(reg_2)) \rightarrow$ *done*	5	`movem r2, r1`

Example



Assembly

```
muli 4, rj
add rj, rb
addi 1, ri
muli 4, ri
add ri, ra
movem rb, ra
```

Rewriting process

	store(+(ra,×(+(ri,1),4)),load(+(rb, ×(rj, 4))))	
4	store(+(ra,×(+(ri,1),4)),load(+(rb, rj)))	`muli 4, rj`
1	store(+(ra,×(+(ri,1),4)),load(rb))	`add rj, rb`
3	store(+(ra,×(ri,4)),load(rb))	`addi 1, ri`
4	store(+(ra,ri),load(rb))	`muli 4, ri`
1	store(ra,load(rb))	`add ri, ra`
5	*done*	`movem rb, ra`

Implementation

- What does this remind you of?

- Similar to parsing
  - Implement as an automaton
  - Use cost to choose from competing productions

- Provides linear time optimal code generation
  - BURS (bottom-up rewrite system)
  - burg, Twig, BEG

Summary

Ad-hoc pattern matchers	Probably reasonable for RISC machines
Encode matching as automaton	Fast, optimal code generation – requires separate tool
Use parsers	Can lead to highly ambiguous grammars

Modern processors

- Execution time not sum of tile times

- Instruction order matters
  - Pipelining: parts of different instructions overlap
  - Bad ordering stalls the pipeline – e.g., too many operations of one type
  - Superscalar: some operations executed in parallel

- Cost is an approximation

- Instruction scheduling helps