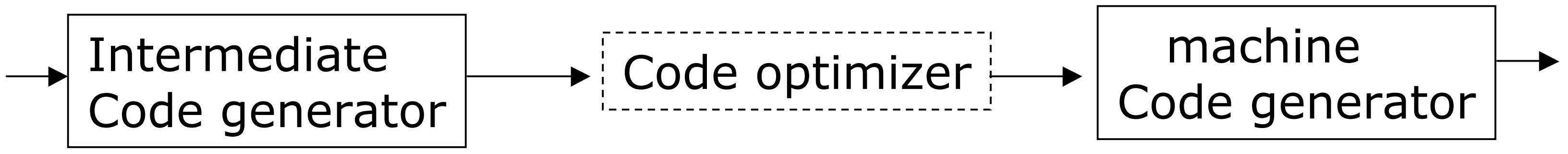
Instruction Selection and Scheduling

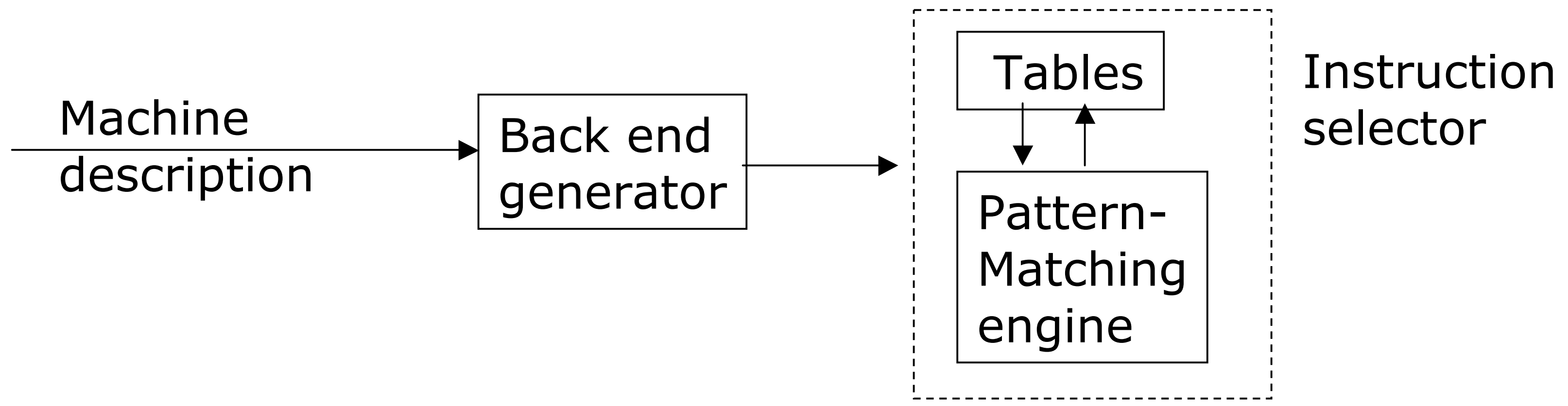
Machine code generation

Machine code generation



- Input: intermediate code + symbol tables
  - All variables have values that machines can directly manipulate
  - Each operation has at most two operands
  - Assume program is free of errors
    - Type checking has taken place, type conversion done
- Output:
  - Absolute/relocatable machine (assembly) code
  - Architectures
    - RISC machines, CISC processors, stack machines
- Issues:
  - Instruction selection
  - Instruction scheduling
  - Register allocation and memory management

Retargetable back-end



- Build retargetable compilers
  - Compilers on different machines share a common IR
    - Can have common front and mid ends
  - Isolate machine dependent information
    - Table-based back ends share common algorithms
- Table-based instruction selector
  - Create a description of target machine, use back-end generator

Instruction Selection

$$\begin{tikzcd} & * \arrow[dl] \arrow[dr] & \\ \text{ID(a,ARP,4)} & & \text{ID(b,ARP,8)} \end{tikzcd}$$

```
loadI 4 => r5
loadA0 rarp, r5 => r6
LoadI 8 => r7
loadA0 rarp, r7 => r8
Mult r6, r8 => r9
```

vs.

```
loadAI rarp, 4 => r5
loadAI rarp, 8 => r6
Mult r5, r6
```

$$\begin{tikzcd} & * \arrow[dl] \arrow[dr] & \\ \text{ID(a,ARP,4)} & & \text{NUM(2)} \end{tikzcd}$$

```
loadI 4 => r5
loadA0 rarp, r5 => r6
loadI 2 => r7
Mult r6, r7 => r8
```

vs.

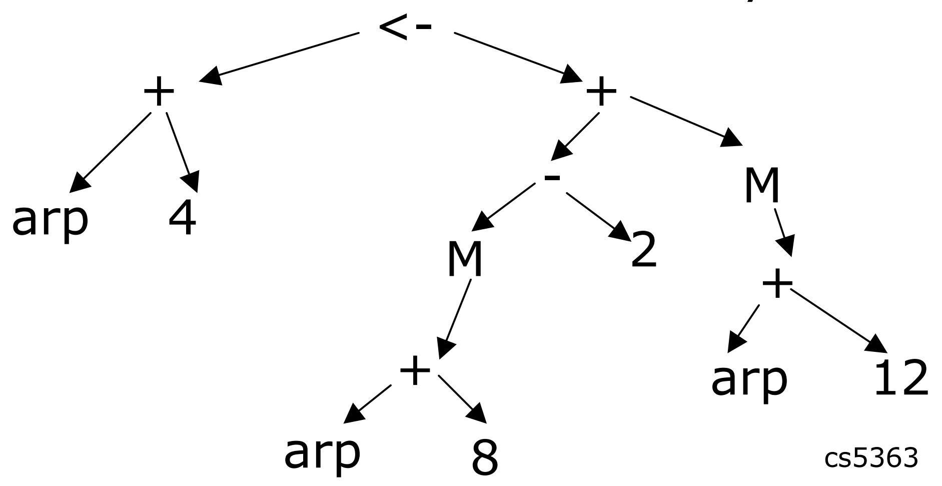
```
loadAI  rarp, 4 => r5
multI r5, 2 => r6
```

- Based on locations of operands, different instructions may be selected
- Two pattern-matching approaches
  - Generate efficient instruction sequences from the AST
  - Generate naïve code, then rewrite inefficient code sequences

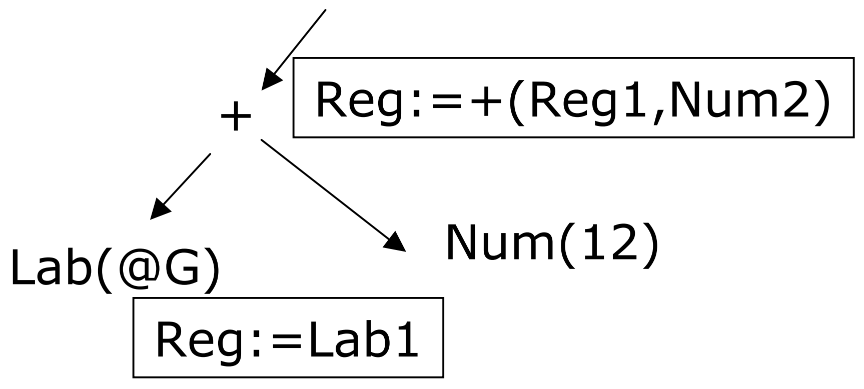
Tree-Pattern Matching

- Tiling the AST
  - Use a low-level AST to expose all the impl. details
  - Define a collection of (operation pattern, code generation template) pairs
  - Match each AST subtree with an operation pattern, then select instructions accordingly
- Given an AST and a collection of operation trees
  - A tiling is a collection of <ASTnode, op-pattern> pairs, each specifying the implementation for a AST node
  - Storage for result of each AST operation must be consistent across different operation trees

low-level AST for w ← x – 2 + y



Tiling an AST for G+12:



Rules Through Tree Grammar

- Use attributed grammar to define code generation rules
  - Summarize structures of AST through context-free grammar
  - Each production defines a tree pattern in prefix-notation
  - Each production is associated with a code generation template (syntax-directed translation) and a cost
  - Each grammar symbol is associated with a synthesized attribute (location of value) to be used in code generation

production	cost	Code template
1: Goal := Assign	0	
2: Assign := <- (Reg1, Reg2)	1	Store r2 => r1
3: Assign := <- (+ (Reg1, Reg2), Reg3)	1	storeA0 r3 => r1, r2
4: Assign := <- (+ (Reg1, num2), Reg3)	1	storeAI r3 => r1, n2
5: Assign := <- (+ (num1, Reg2), Reg3)	1	storeAI r3 => r2, n1
6: Reg:=lab1 (a relocatable symbol)	1	loadI lab1 => rnew
7: Reg:=val1 (value in reg, e.g. rarp)	0	
8: Reg := Num1 (constant integer value)	1	loadI num1 => rnew

Tree Grammar (continued)

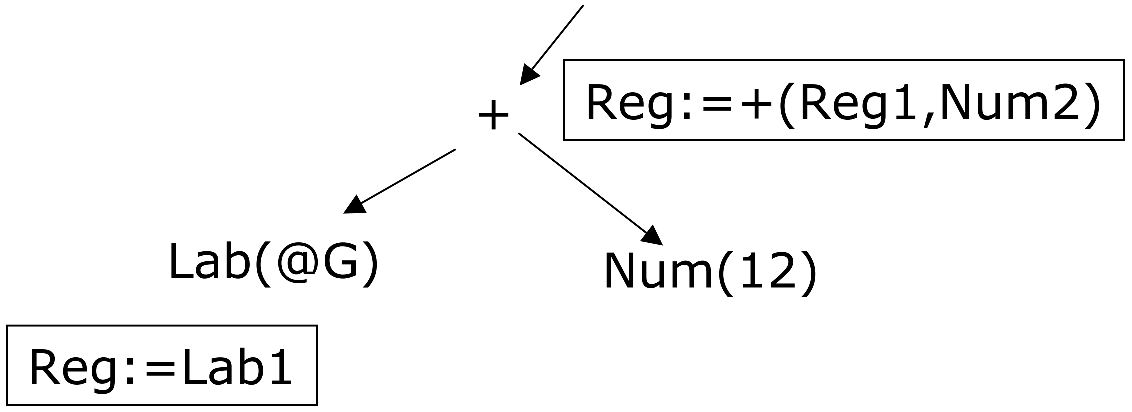
production	cost	Code template
9: Reg := M(Reg1)	1	Load r1 => rnew
10: Reg := M(+ (Reg1,Reg2))	1	loadA0 r1, r2 => rnew
11: Reg := M(+ (Reg1,Num2))	1	loadAI r1, n2 => rnew
12: Reg := M(+ (Num1,Reg2))	1	loadAi r2, n1 => rnew
13: Reg := M(+ (Reg1, Lab2))	1	loadAI r1, l2 => rnew
14: Reg := M(+ (Lab1,Reg2))	1	loadAI r2, l1 => rnew
15: Reg := - (Reg1,Reg2)	1	Sub r1 r2 => rnew
16: Reg := - (Reg1, Num2)	1	subI r1, n2 => rnew
17: Reg := +(Reg1, Reg2)	1	add r1, r2=> rnew
18: Reg := + (Reg1, Num2)	1	addI r1, n2 => rnew
19: Reg := + (Num1, Reg2)	1	addI r2, n1 => rnew
20: Reg := + (Reg1, Lab2)	1	addI r1, l2 => rnew
21: Reg := + (Lab1, Reg2)	1	addI r2, l1 => rnew

Tree Matching Approach

- Need to select lowest-cost instructions in bottom-up traversal of AST
  - Need to determine lowest-cost match for each storage class
- Automatic tools
  - Hand-coding of tree matching
  - Encode the tree-matching problem as a finite automata
  - Use parsing techniques
    - Need to be extended to handle ambiguity
  - Use string-matching techniques
    - Linearize the tree into a prefix string
    - Apply string pattern matching algorithms

Tiling the AST

- Given an AST and a collection of operation trees, tiling the AST maps each AST subtree to an operation tree
- A tiling is a collection of <ASTnode, op-tree> pairs, each specifying the implementation for a AST node
  - Storage for result of each AST operation must be consistent across different operation trees



Finding a tiling

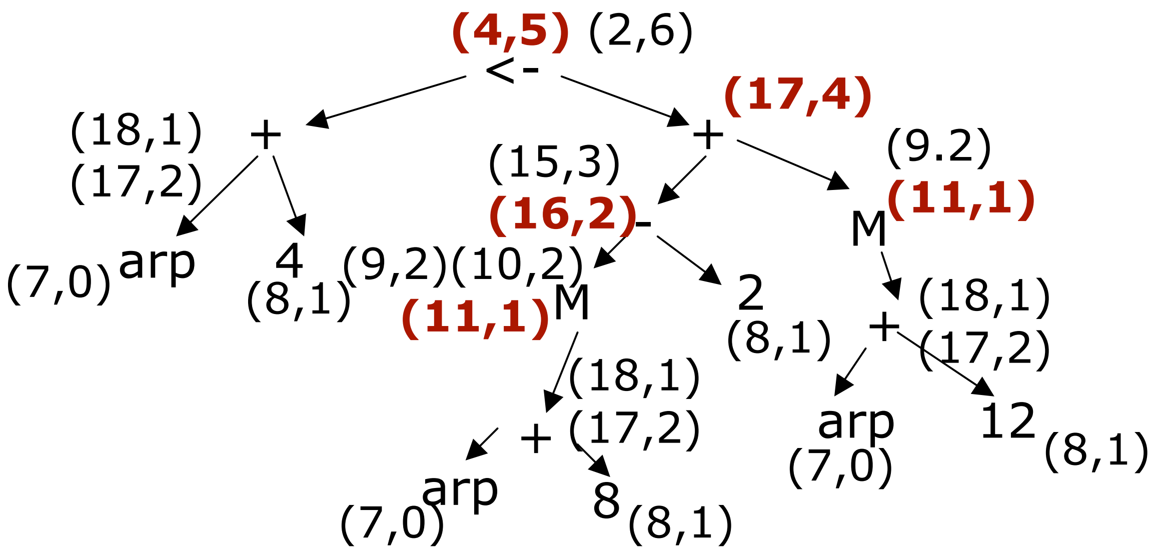
- Bottom-up walk of the AST, for each node n
  - Label(n) contains the set of all applicable tree patterns

```
Tile(n)
  Label(n) := ∅
  if n is a binary node then
      Tile(left(n))
      Tile(right(n))
      for each rule r that matches n’s operation
          if left(r) ∈ Label(left(n)) and right(r) ∈ Label(right(n))
             then Lable(n) := Label(n) ∪ {r}
  else if n is a unary node then
      Tile(left(n))
      for each rule r that matches n’s operation
         if (left(r) ∈ Label(left(n))
             then Label(n) := Label(n) ∪ {r}
  else /* n is a AST leaf */
      Label(n) := {all rules that match the operation in n}
```

Finding The Low-cost Tiling

- Tiling can find all the matches in the pattern set
    - Multiple matches exist because grammar is ambiguous
- To find the one with lowest cost, must keep track of the cost in each matched translation

Example: low-level AST for w ← x – 2 + y



```
loadAI rarp,8=>r1
subI r1, 2=> r2
loadAI rarp,12=>r3
Add r2, r3 => r4
storeAI r4=>rarp, 4
```

Peephole optimization

- Use simple scheme to match IR to machine code
  - Discover local improvements by examining short sequences of adjacent operations

```
StoreAI r1 => rarp, 8
loadAI rarp,8 => r15
```

$\Rightarrow$

```
storeAI r1 => rarp 8
I2i     r1 => r15
```

```
addI r2, 0 => r7
Mult r4, r7 => r10
```

$\Rightarrow$

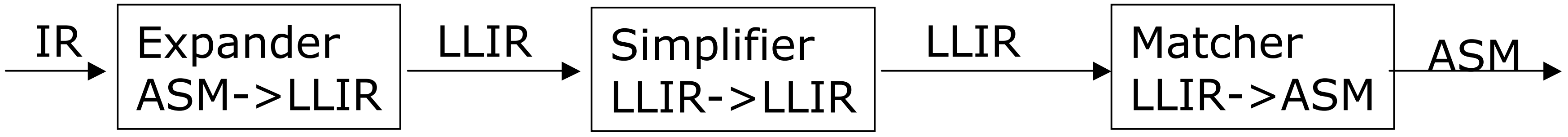
```
Mult r4, r2 => r10
```

```
     jumpI  -> L10
L10: jumpI -> L11
```

$\Rightarrow$

```
     jumpI  -> L11
L10: jumpI -> L11
```

Systematic Peephole Optimization



- Expander
  - Rewrites each assembly instruction to a sequence of low-level IRs that represent all the direct effects of operation
- Simplifier
  - Examine and improve LLIR operations in a small sliding window
    - Forward substitution, algebraic simplification, constant evaluation, eliminating useless effects
- Matcher
  - Match simplified LLIR against pattern library for instructions that best captures the LLIR effects

Peephole optimization example

```
mult   2   y   =>  t1
sub    x   t1  =>  w
```

expand

```
r10 := 2
r11 := @G
r12 := 12
r13 := r11 + r12
r14 := M(r13)
r15 :=r10 * r14
r16 := -16
r17 := rarp + r16
r18 := M(r17)
r19 := M(r18)
r20 := r19 – r15
r21 := 4
r22 := rarp + r21
M(r22) := r20
```

Optimizations:

```
r1 := n1              r1:=r2+n1          r1:=r2+n1
r2 := r3 + r1         r3:=M(r1)          M(r1):=r3

R2:=r3+n1             r3:=M(r2+n1)       M(r2+n1):=r3
```

simplify

```
r10 := 2
r11 := @G
r14 := M(r11+12)
r15 :=r10 * r14
r18 := M(rarp + -16)
r19 := M(r18)
r20 := r19 – r15
M(rarp+4) := r20
```

match

```
loadI 2 => r10
loadI @G => r11
loadAI r11 12=>r14
Mult r10 r14 => r15
loadAI rarp -16=>r18
Load r18 => r19
Sub r19 r15 => r20
storeAI r20 => rarp 4
```

Efficiency of Peephole Optimization

- Design issues
  - Dead values
    - May intervene with valid simplifications
    - Need to be recognized in the expansion process
  - Control flow operations
    - Complicates simplifier
      - Clear window vs. special-case handling
  - Physical vs. logical windows
    - Adjacent operations may be irrelevant
    - Sliding window includes ops that define or use common values
- RISC vs. CISC architectures
  - RISC architectures makes instruction selection easier
- Additional issues
  - Automatic tools to generate large pattern libraries for different architectures
  - Front ends that generate LLIR make compilers more portable