Code Generation

Code Generation

- The target machine
- Basic blocks and control flow graphs
- Instruction selector generator
- Register allocation
- Peephole optimization

The Target Machine

- A byte addressable machine with four bytes to a word and *n* general purpose registers
- Two address instructions
  - *op source, destination*
- Six addressing modes
  - absolute M M 1
  - register R R 0
  - indexed c(R) c+content(R) 1
  - ind register *R content(R) 0
  - ind indexed *c(R) content(c+content(R)) 1
  - literal #c c 1

Examples

```
MOV    R0,  M
MOV    4 (R0),  M
MOV    *R0,  M
MOV    *4 (R0),  M
MOV    #1,  R0
```

Instruction Costs

- Cost of an instruction = 1 + costs of source and destination addressing modes
- This cost corresponds to the length (in words) of the instruction
- Minimize instruction length also tend to minimize the instruction execution time

Examples

MOV	R0, R1	1
MOV	R0, M	2
MOV	#1, R0	2
MOV	4 (R0), *12 (R1)	3

An Example

Consider a := b + c

1. ```
   MOV    b,  R0
   ADD    c,  R0
   MOV    R0, a
   ```

2. ```
   MOV    b,  a
   ADD    c,  a
   ```

3. R0, R1, R2 contains the addresses of a, b, c
   ```
   MOV    *R1, *R0
   ADD    *R2, *R0
   ```

4. R1, R2 contains the values of b, c
   ```
   ADD    R2, R1
   MOV    R1, a
   ```

Instruction Selection

- Code skeleton

```
x := y + z          a := b + c          d := a + e
MOV   y, R0         MOV   b, R0         MOV   a, R0
ADD   z, R0         ADD   c, R0         ADD   e, R0
MOV   R0, x         MOV   R0, a         MOV   R0, d
```

- Multiple choices

```
a := a + 1          MOV   a, R0         INC   a

                    ADD   #1, R0
                    MOV   R0, a
```

Register Allocation

- Register allocation: select the set of variables that will reside in registers
- Register assignment: pick the specific register that a variable will reside in
- The problem is NP-complete

An Example

```
t := a + b
t := t * c
t := t / d

MOV a, R1
ADD b, R1
MUL c, R0
DIV  d, R0
MOV R1, t
```

```
t := a + b
t := t + c
t := t / d

MOV   a, R0
ADD   b, R0
ADD   c, R0
SRDA R0, 32
DIV   d, R0
MOV  R1, t
```

Basic Blocks

- A *basic block* is a sequence of consecutive statements in which control enters at the beginning and leaves at the end without halt or possibility of branching except at the end

An Example

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
```

Control Flow Graphs

- A (*control*) *flow graph* is a directed graph
- The *nodes* in the graph are *basic blocks*
- There is an *edge* from B_1 to B_2 iff B_2 immediately follows B_1 in some execution sequence
  - there is a jump from B_1 to B_2
  - B_2 immediately follows B_1 in program text
- B_1 is a *predecessor* of B_2, B_2 is a *successor* of B_1

An Example

```
(1)   prod := 0
(2)   i := 1
```

B_0

```
(3)   t1 := 4 * i
(4)   t2 := a[t1]
(5)   t3 := 4 * i
(6)   t4 := b[t3]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
```

B_1

Construction of Basic Blocks

- Determine the set of *leaders*
  - the first statement is a leader
  - the target of a jump is a leader
  - any statement immediately following a jump is a leader
- For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

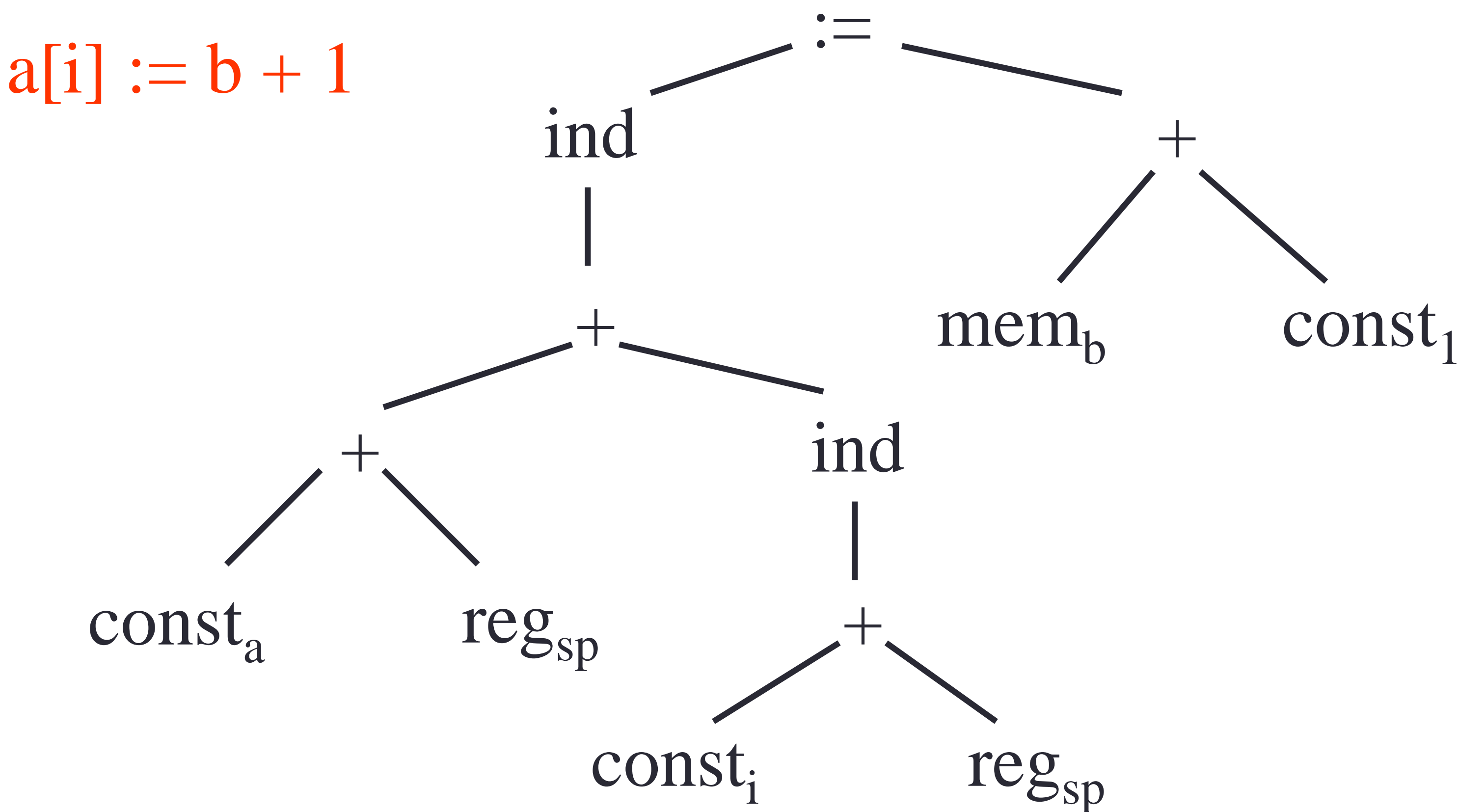
Representation of Basic Blocks

- Each basic block is represented by a record consisting of
  - a count of the number of statements
  - a pointer to the leader
  - a list of predecessors
  - a list of successors

Code Generator Generators

- A tool to automatically construct the *instruction selection* phrase of a code generator
- Such tools may use *tree grammars* or *context free grammars* to describe the *target machines*
- *Instruction selector* assumes that there are infinite symbolic registers
- *Register allocation* will be implemented as a separate mechanism to map symbolic registers to physical registers

Tree Rewriting



Tree Rewriting

- The code is generated by *reducing* the input tree into a single node using a sequence of *tree-rewriting rules*
- Each tree rewriting rule is of the form

  replacement ← *template* { *action* }

  - *replacement* is a single node
  - *template* is a tree
  - *action* is a code fragment
- A set of tree-rewriting rules is called a *tree-translation scheme*

An Example

$reg_i \leftarrow$ + (reg_i, reg_j) { ADD Rj, Ri }

Each tree template represents a computation performed by the sequence of machines instructions emitted by the associated action

Tree Rewriting Rules

	Rule	Action
(1)	$reg_i \leftarrow const_c$	{ MOV #c, R*i* }
(2)	$reg_i \leftarrow mem_a$	{ MOV a, R*i* }
(3)	$mem \leftarrow$:= (mem_a, reg_i)	{ MOV R*i*, a }
(4)	$mem \leftarrow$:= (ind (reg_i), reg_j)	{ MOV R*j*, *R*i* }
(5)	$reg_i \leftarrow$ ind (+ ($const_c$, reg_j))	{ MOV c(R*j*), R*i* }

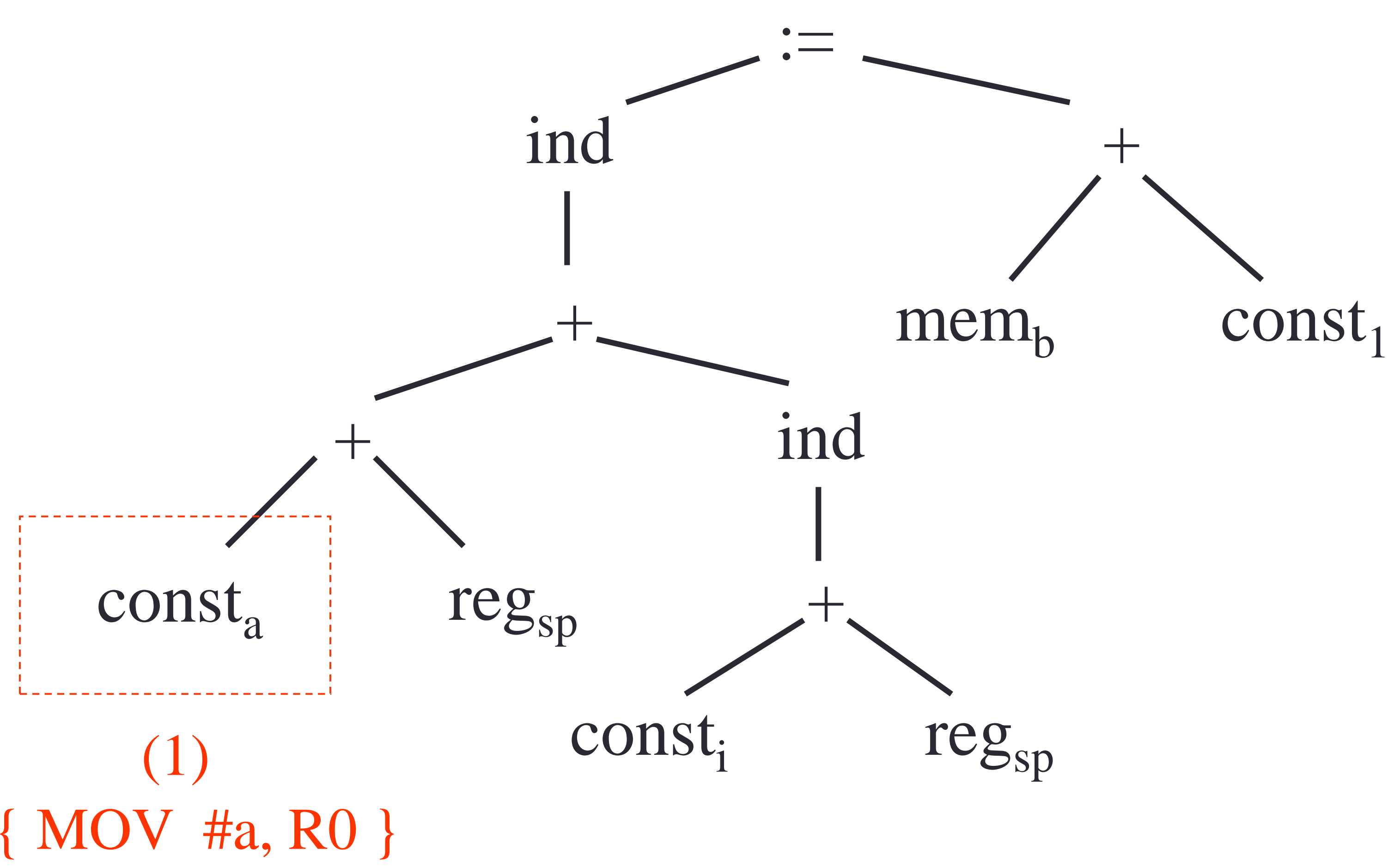
Tree Rewriting Rules

(6) $reg_i \leftarrow$ +(reg_i, ind(+($const_c$, reg_j))) { ADD c(R*j*), R*i* }

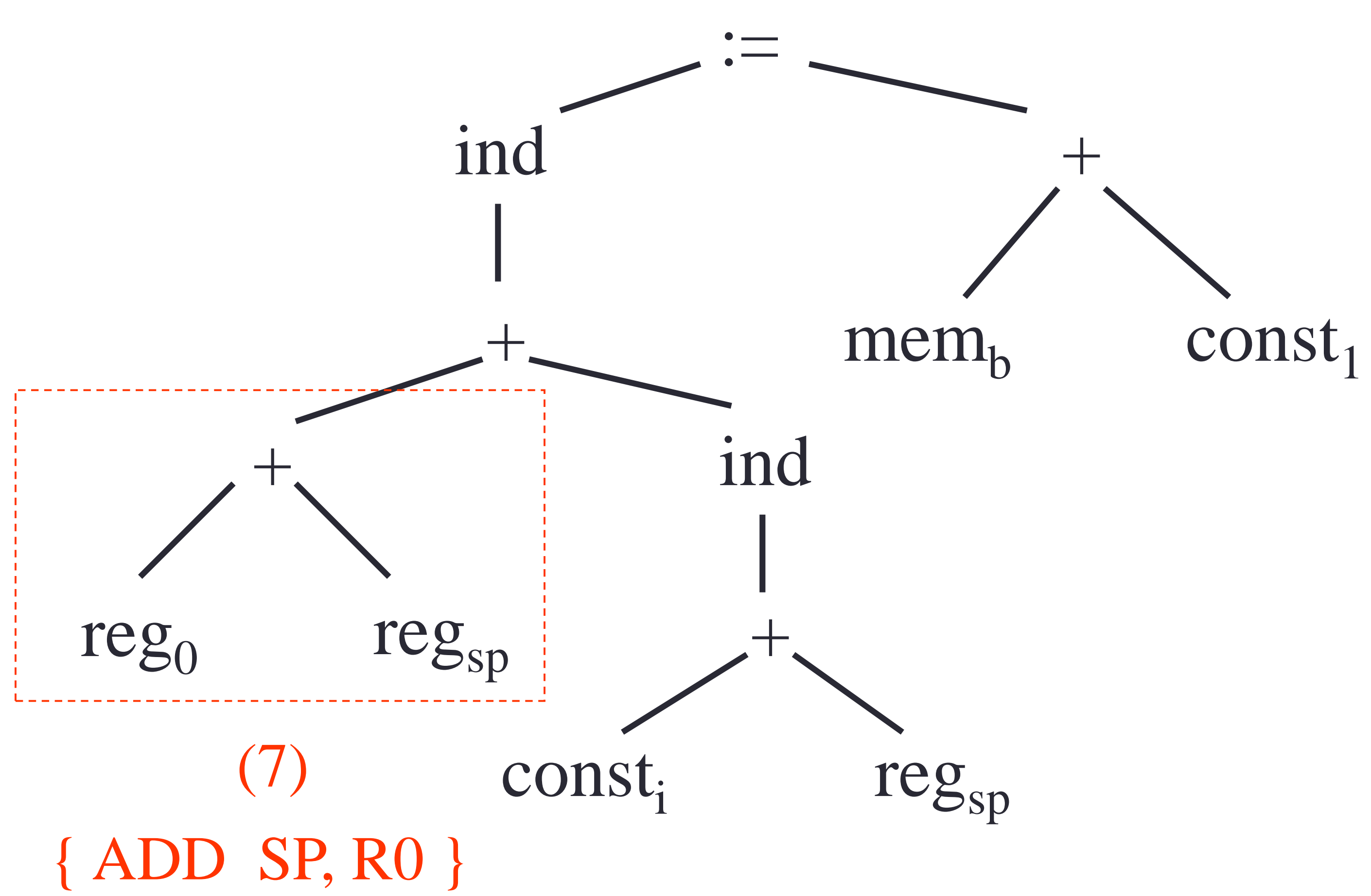
(7) $reg_i \leftarrow$ +(reg_i, reg_j) { ADD R*j*, R*i* }

(8) $reg_i \leftarrow$ +(reg_i, $const_1$) { INC R*i* }

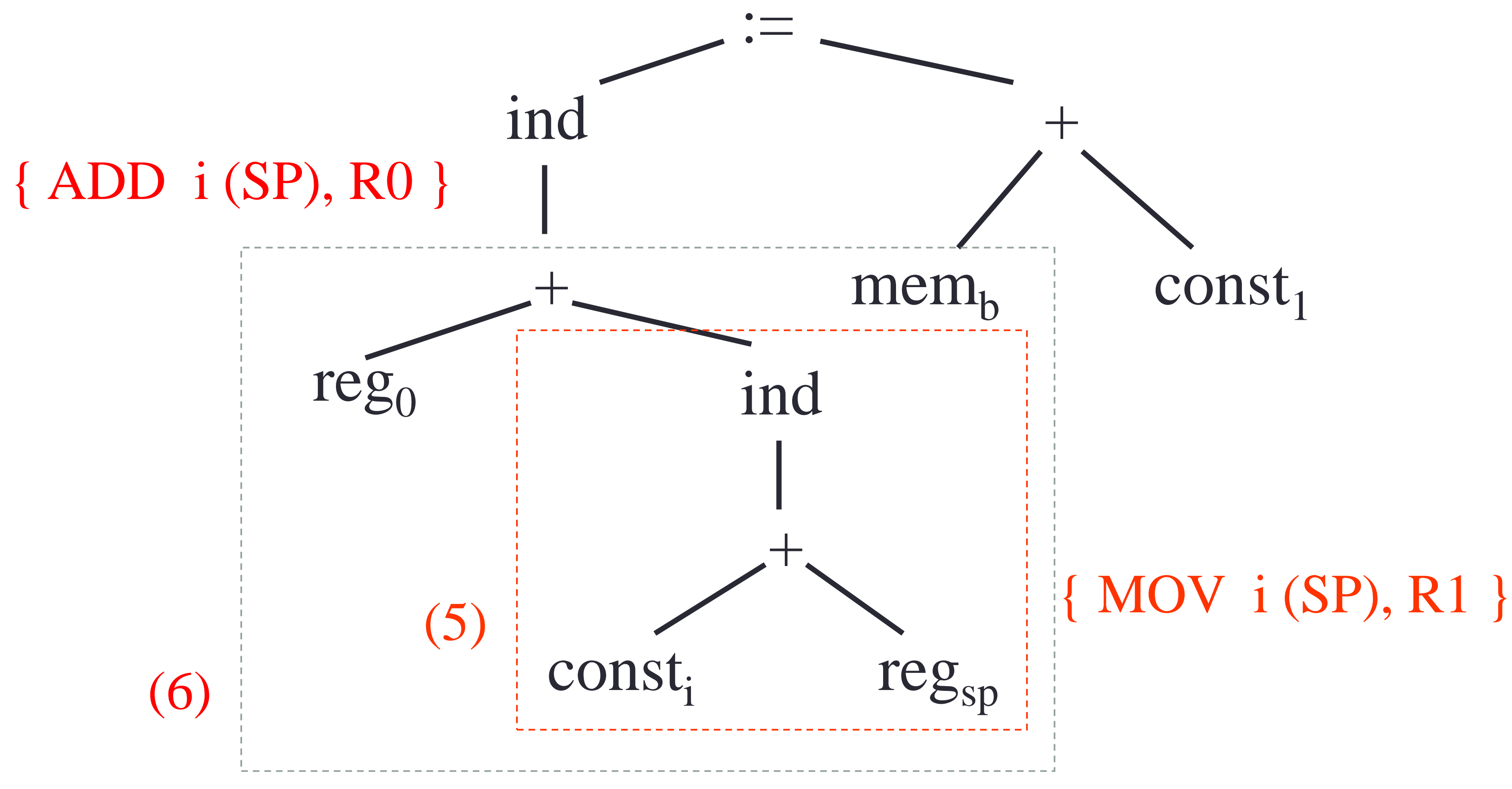
An Example



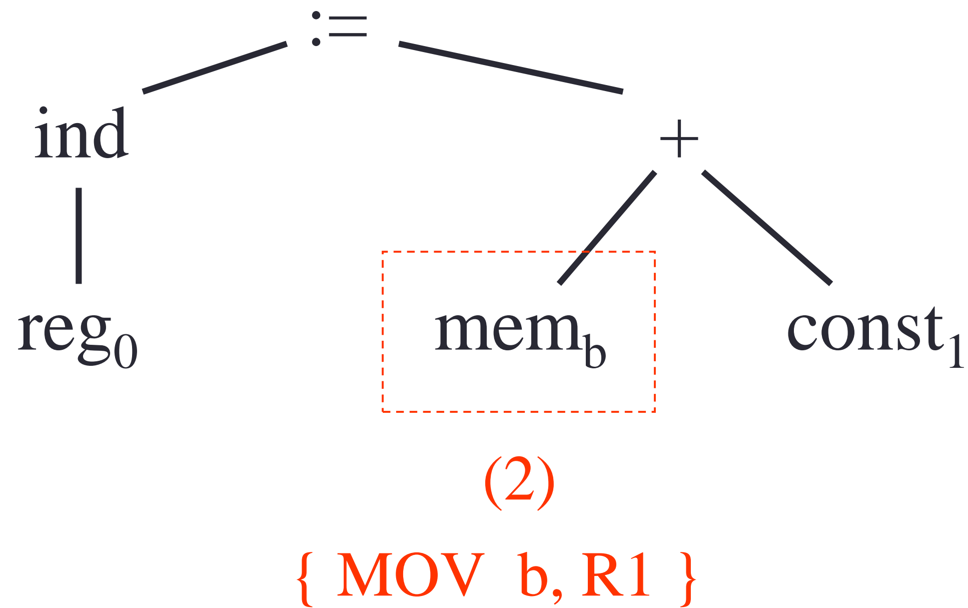
An Example



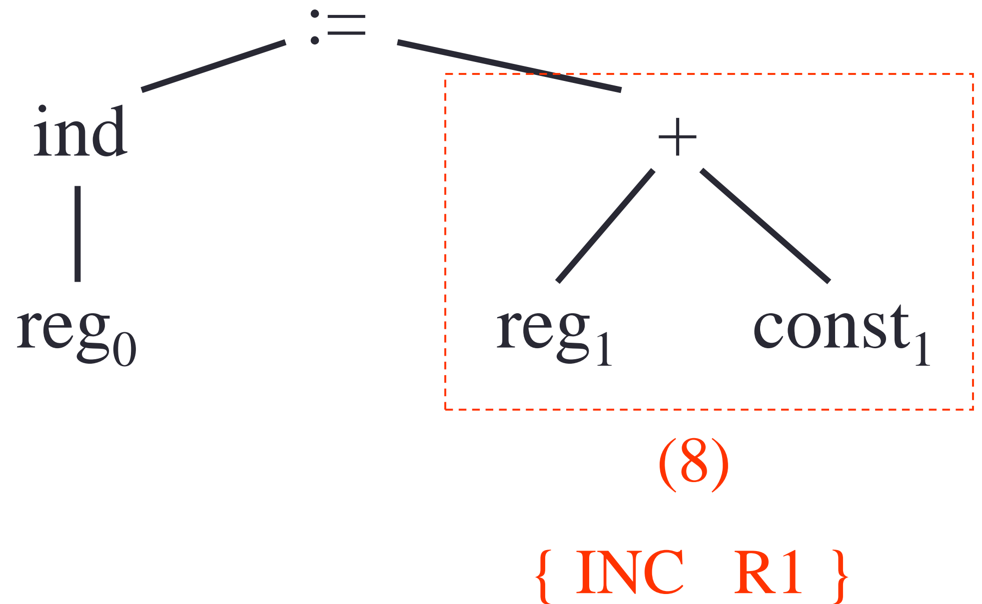
An Example



An Example



An Example



An Example

$$:=$$

ind — reg_1

reg_0

(4)

{ MOV R1, *R0 }

Tree Pattern Matching

- The *tree pattern matching* algorithm can be implemented by extending the *multiple-keyword pattern matching algorithm*
- Each *tree template* is represented by a set of *strings*, each of which represents a *path* from the root to a leave
- Each rule is associated with *cost* information
- The *dynamic programming algorithm* can be used to select an *optimal* sequence of matches

Semantic Predicates

$reg_i \leftarrow$ + (reg_i, $const_c$)

{ **if** c = 1 **then**

INC R*i*

else

ADD #c, R*i* }

The general use of semantic actions and predicates can provide greater flexibility and ease of description than a purely grammatical specification

Graph Coloring

- In the first pass, target machine instructions are *selected* as though there were an *infinite* number of *symbolic* registers
- In the second pass, *physical* registers are assigned to *symbolic* registers using *graph coloring algorithms*
- During the second pass, if a register is needed when all available registers are used, some of the used registers must be *spilled*

Interference Graph

- For each procedure, a *register-interference graph* is constructed
- The nodes in the graph are *symbolic registers*
- An edge connects two nodes if the *life ranges* of the corresponding two symbolic registers *intersect*

An Example

```
(1)   MOV  a, R0
(2)   ADD  b, R0
(3)   MOV  c, R1
(4)   ADD  d, R1
(5)   MOV  R0, t1
(6)   MOV  e, R2
(7)   SUB  R1, R2
(8)   MOV  t1, R3
(9)   SUB  R2, R3
(10)  MOV  R3, t4
```

K-Colorable Graphs

- A graph is said to be *k-colorable* if each node can be assigned one of the *k* colors such that no two adjacent nodes have the same color
- A *color* represents a *register*
- The problem of determining whether a graph is *k-colorable* is NP-complete

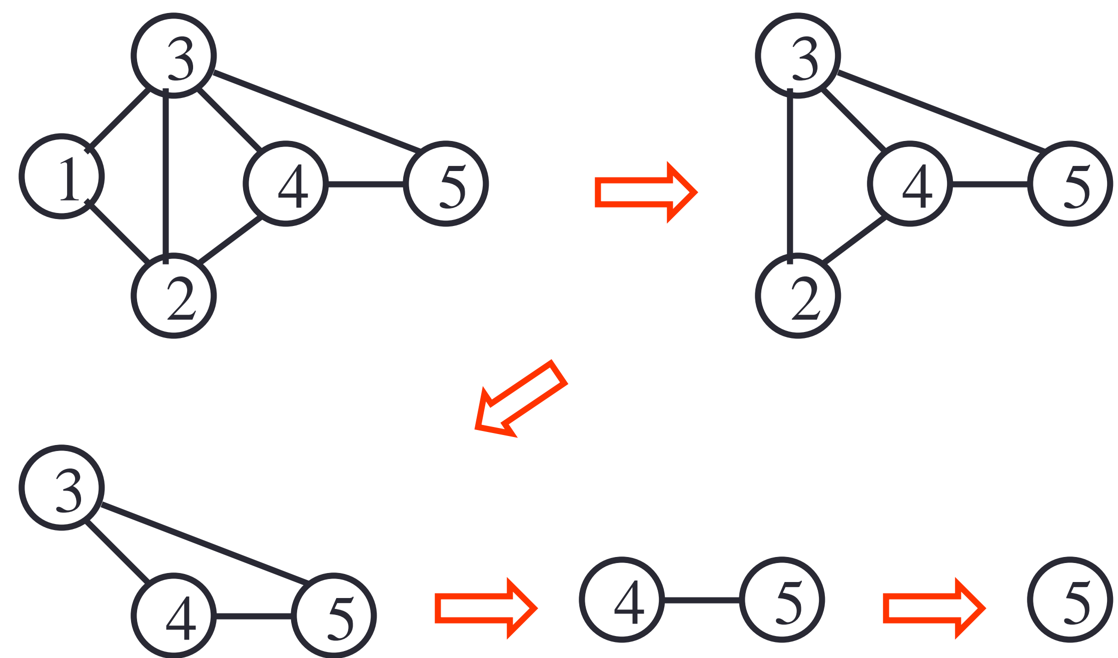
A Graph Coloring Algorithm

- Remove a node *n* and its edges if it has *fewer* than *k* neighbors
- Repeat the removing step above until we end up with the *empty* graph or a graph in which each node has *k* or *more* adjacent nodes
- In the latter case, a node is *selected* and *spilled* by deleting that node and its edges, and the removing step above continues
- Several strategies can be used to select the spilled register: *least-used*, *latest-used*, and so on

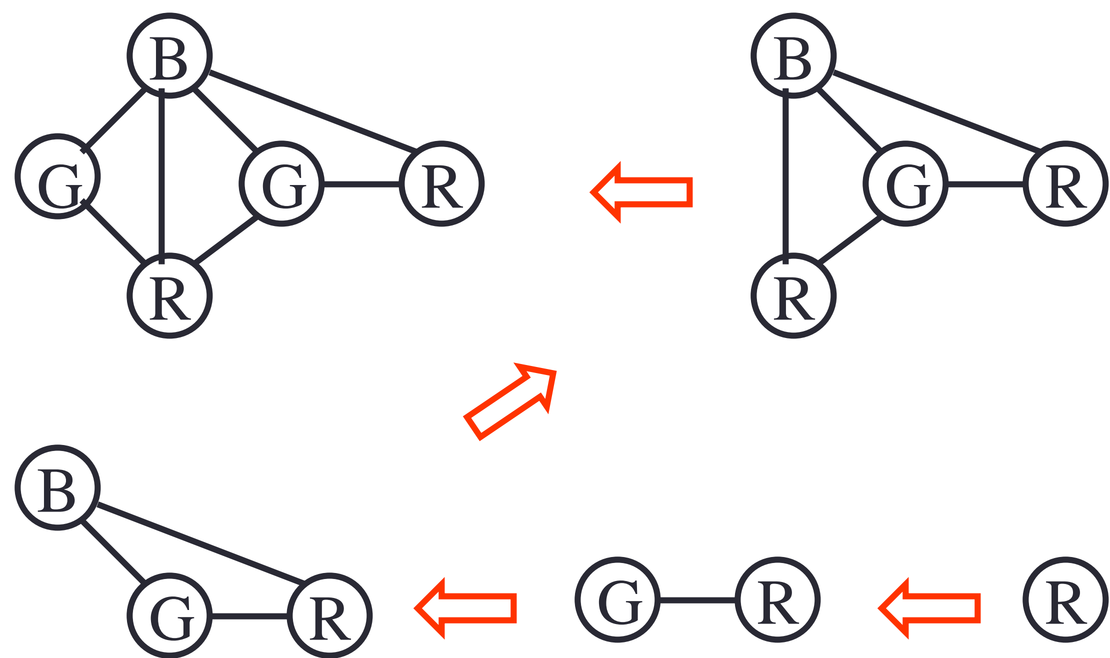
A Graph Coloring Algorithm

- The nodes in the graph can be colored in the *reverse* order in which they are removed
- Each node can be assigned a color *not* assigned to any of its neighbors
- Spilled nodes can be assigned *any* color

An Example



An Example



Peephole Optimization

- Improve the performance of the target program by examining and transforming a short sequence of target instructions
- May need repeated passes over the code
- Can also be applied directly after intermediate code generation

Examples

- Redundant loads and stores

  MOV R0, a

  MOV a, Ro

- Algebraic Simplification

  x := x + 0

  x := x * 1

- Constant folding

  x := 2 + 3 ⇨ x := 5

  y := x + 3 ⇨ y := 8

Examples

- Unreachable code

```
    #define  debug  0
    if  (debug)  (print  debugging  information)

    if  0  <>  1  goto  L1
     print  debugging  information
L1:

    if  1  goto  L1
     print  debugging  information
L1:
```

Examples

- Flow-of-control optimization

```
     goto  L1                      goto  L2
     ...                ⇨          ...
L1: goto  L2                  L2: goto  L2
-----------------------------------------------
     goto  L1                      if  a < b  goto  L2
                                   goto  L3
     ...                ⇨          ...
L1: if  a < b  goto  L2
L3:                           L3:
```

Examples

- Reduction in strength: replace expensive operations by cheaper ones
  - $x^2 \Rightarrow x * x$
  - fixed-point multiplication and division by a power of 2 $\Rightarrow$ shift
  - floating-point division by a constant $\Rightarrow$ floating-point multiplication by a constant

Examples

- Use of machine Idioms: hardware instructions for certain specific operations
  - auto-increment and auto-decrement addressing mode (push or pop stack in parameter passing)