

Code Generation

Mooly Sagiv

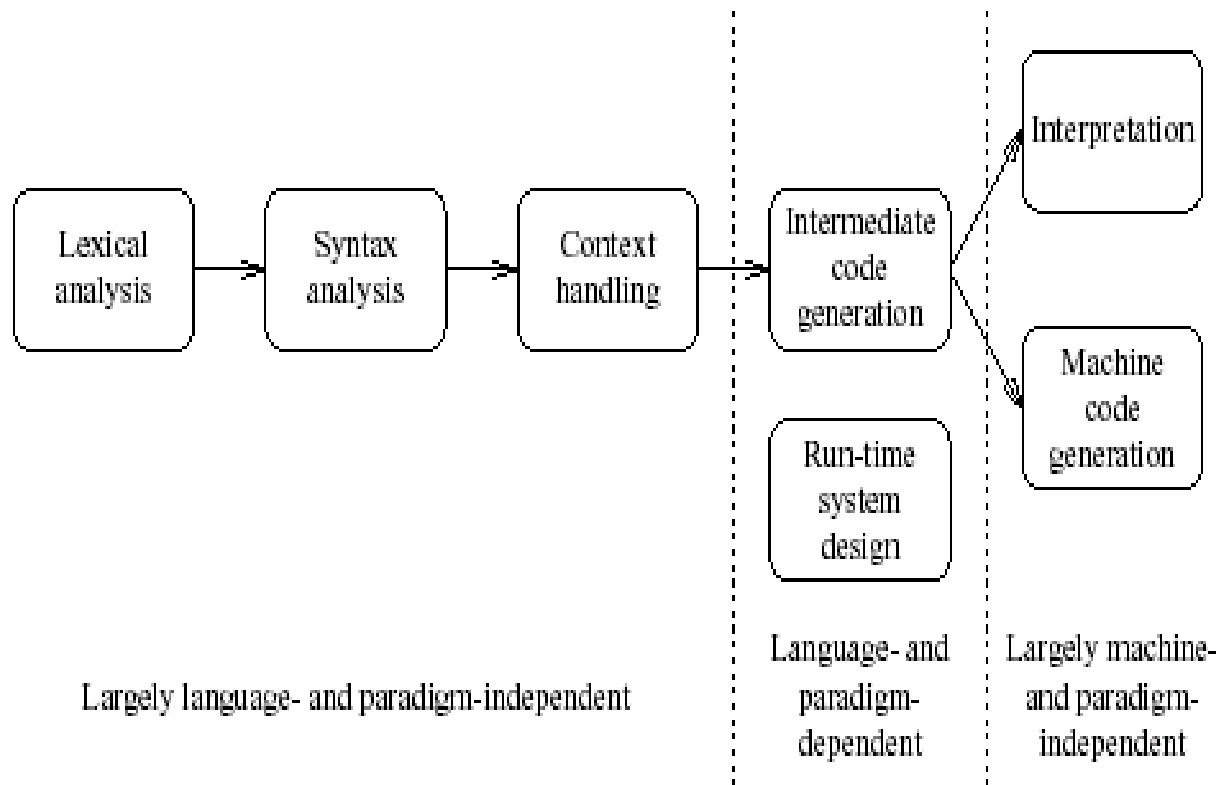
<http://www.cs.tau.ac.il/~msagiv/courses/wcc08.html>

Chapter 4

Tentative Schedule

25/2	Code generation Expression
3/3	Basic Blocks Activation Records
10/3	Program Analysis
{12 13}/3 18-21	Register Allocation
17/3	Control Flow
24/3	Assembler/Linker/Loader OO
[30/3]	Garbage Collection Not included in the course material

Basic Compiler Phases



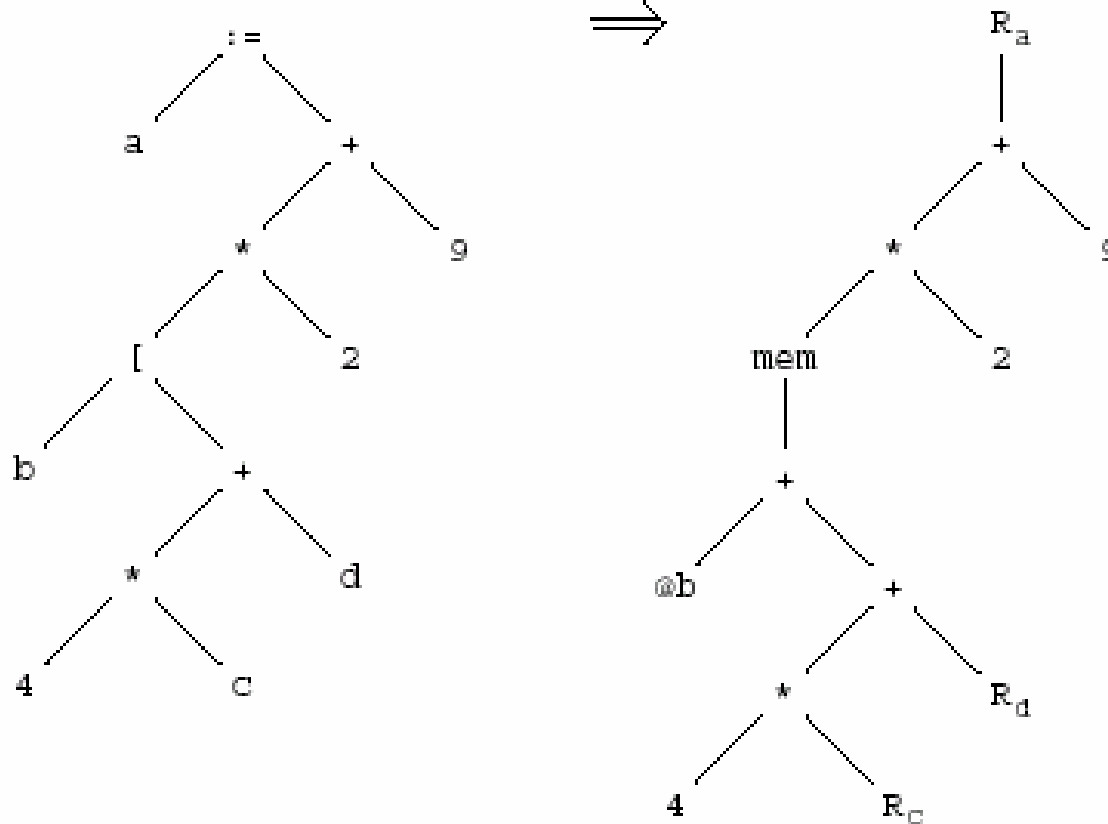
Code Generation

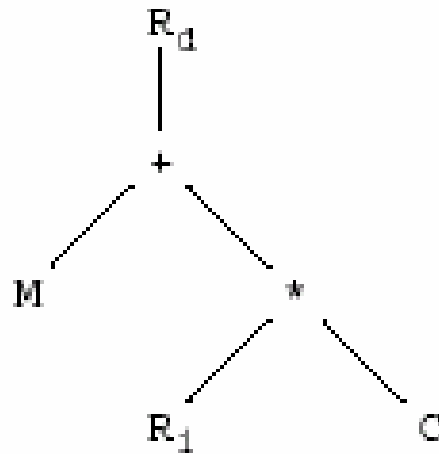
- Transform the AST into machine code
 - Several phases
 - Many IRs exist
- Machine instructions can be described by tree patterns
- Replace tree-nodes by machine instruction
 - Tree rewriting
 - Replace subtrees
- Applicable beyond compilers

$a := (b[4*c+d]*2)+9$

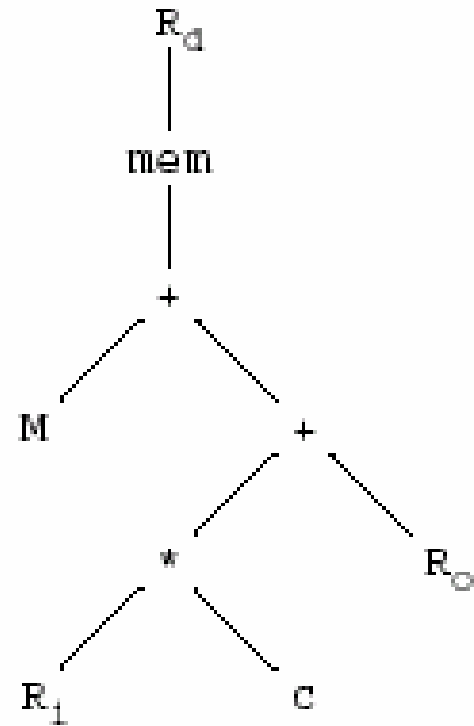
Intermediate Code Generation
and Register Allocation

\Rightarrow





Load_Address $M[R_1], C, R_d$

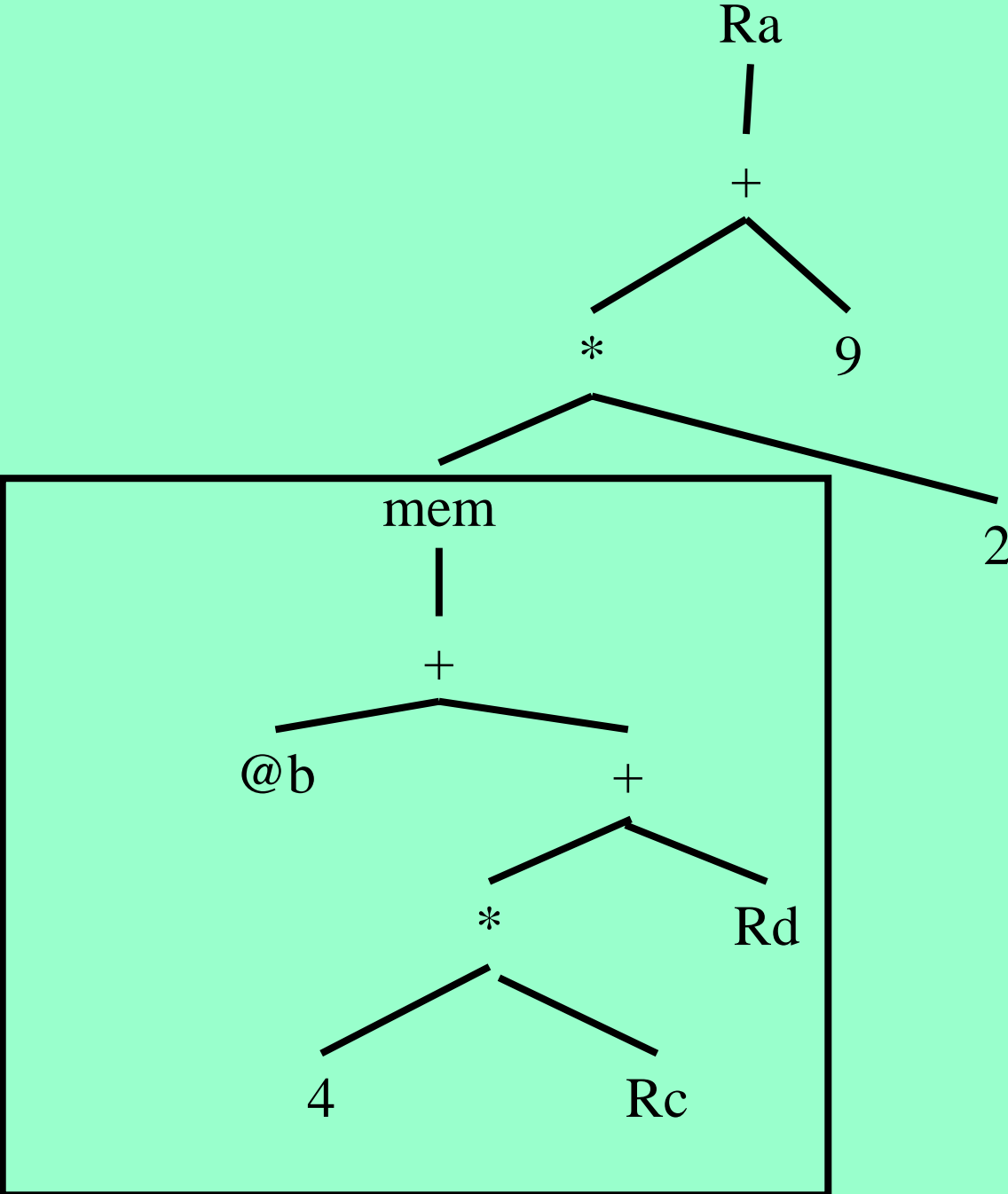


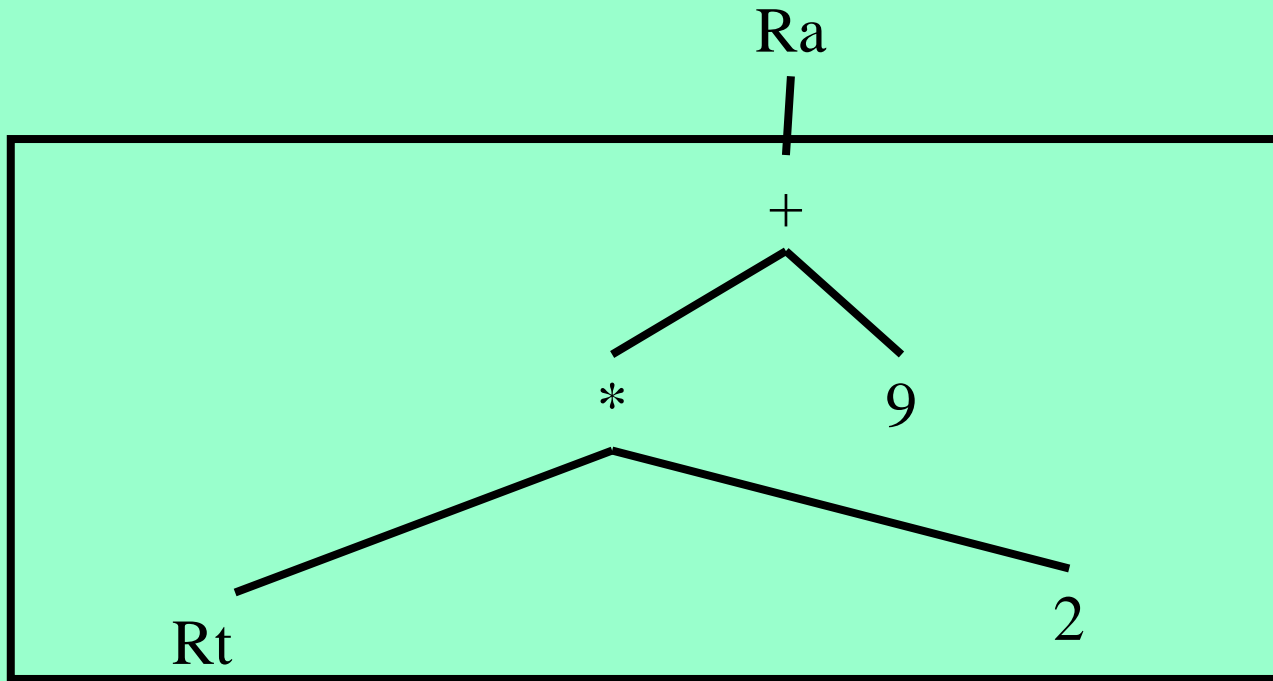
Load_Byte $(M+R_o)[R_1], C, R_d$

Figure 4.10 Two sample instructions with their ASTs.

leal

movsbl



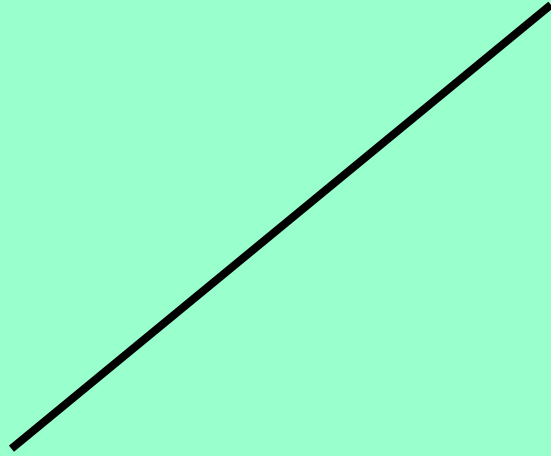


Load_Byte (b+Rd)[Rc], 4, Rt

Ra

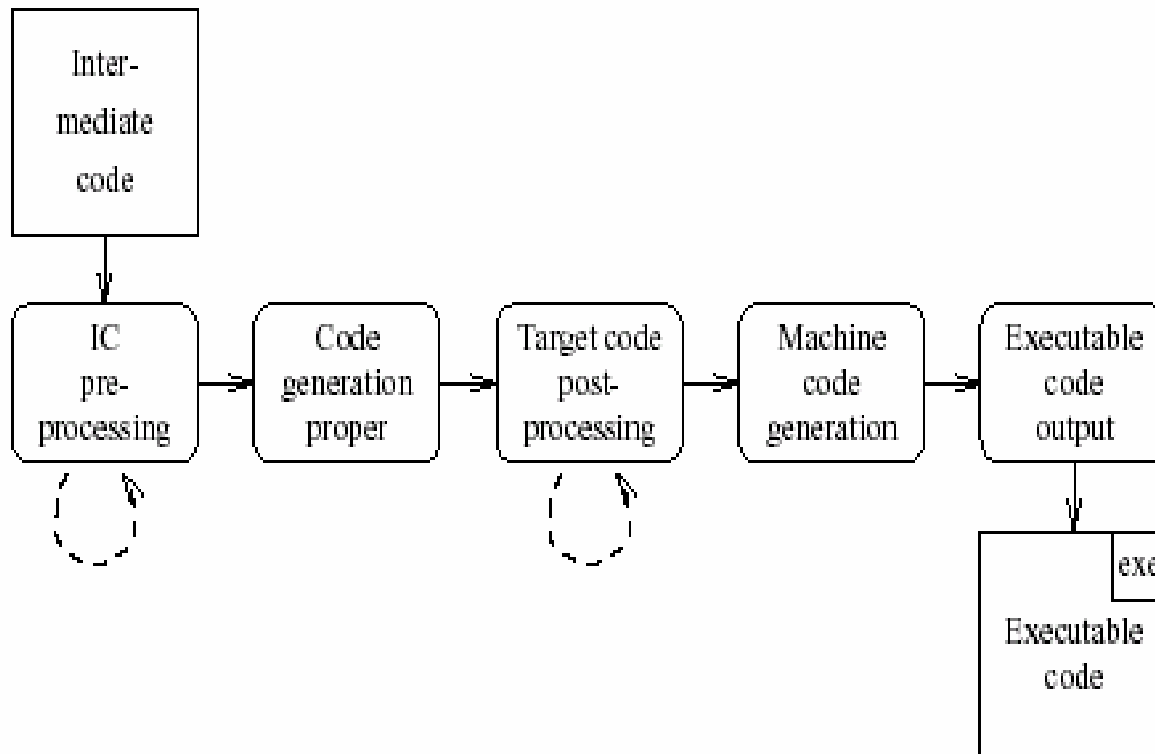


Load_address 9[Rt], 2, Ra



Load_Byte (b+Rd)[Rc], 4, Rt

Overall Structure



Code generation issues

- Code selection
- Register allocation
- Instruction ordering

Simplifications

- Consider small parts of AST at time
- Simplify target machine
- Use simplifying conventions

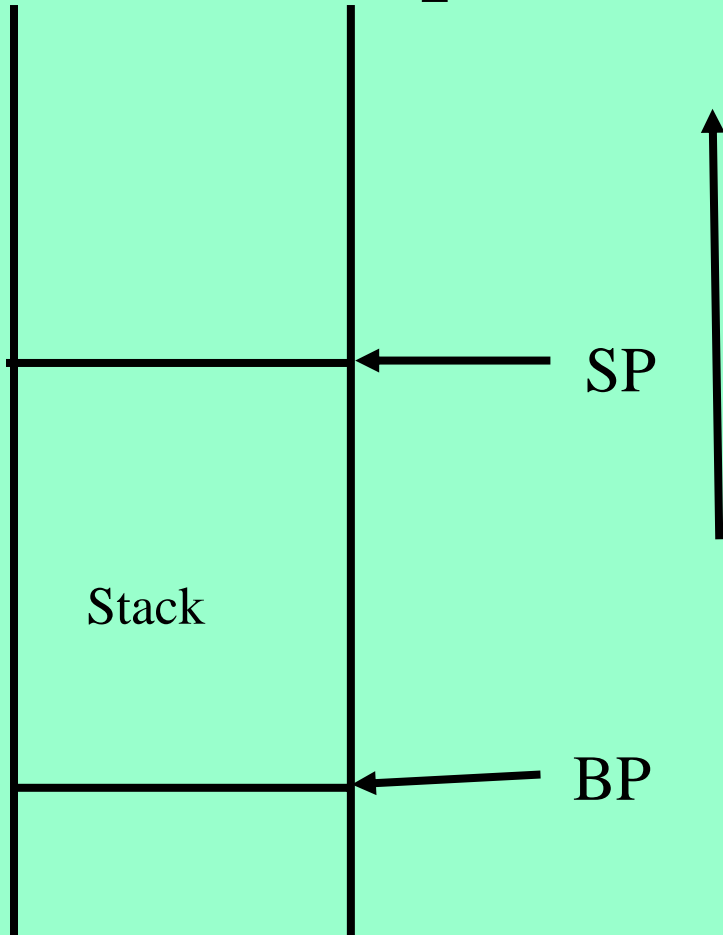
Outline

- Simple code generation for expressions (4.2.4, 4.3)
 - Pure stack machine
 - Pure register machine
- Code generation of basic blocks (4.2.5)
- [Automatic generation of code generators (4.2.6)]
- Later
 - Handling control statements
 - Program Analysis
 - Register Allocation
 - Activation frames

Simple Code Generation

- Fixed translation for each node type
- Translates one expression at the time
- Local decisions only
- Works well for simple machine model
 - Stack machines (PDP 11, VAX)
 - Register machines (IBM 360/370)
- Can be applied to modern machines

Simple Stack Machine



Stack Machine Instructions

Instruction		Actions
Push_Const	<i>c</i>	$SP := SP + 1; \text{stack}[SP] := c;$
Push_Local	<i>i</i>	$SP := SP + 1; \text{stack}[SP] := \text{stack}[BP + i];$
Store_Local	<i>i</i>	$\text{stack}[BP + i] := \text{stack}[SP]; SP := SP - 1;$
Add_Top2		$\text{stack}[SP - 1] := \text{stack}[SP - 1] + \text{stack}[SP]; SP := SP - 1;$
Subtr_Top2		$\text{stack}[SP - 1] := \text{stack}[SP - 1] - \text{stack}[SP]; SP := SP - 1;$
Mult_Top2		$\text{stack}[SP - 1] := \text{stack}[SP - 1] * \text{stack}[SP]; SP := SP - 1;$

Example

$p := p + 5$

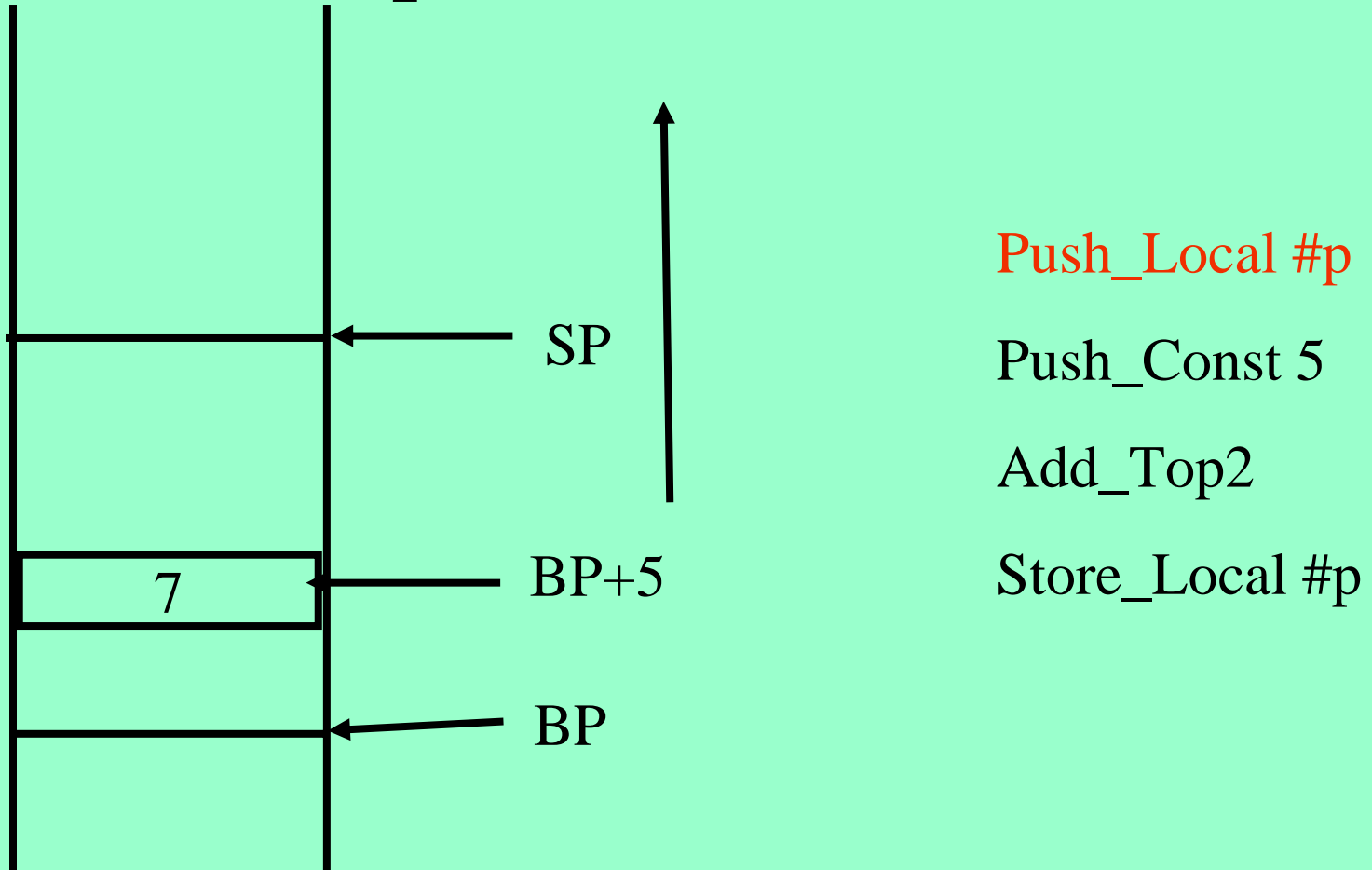
Push_Local #p

Push_Const 5

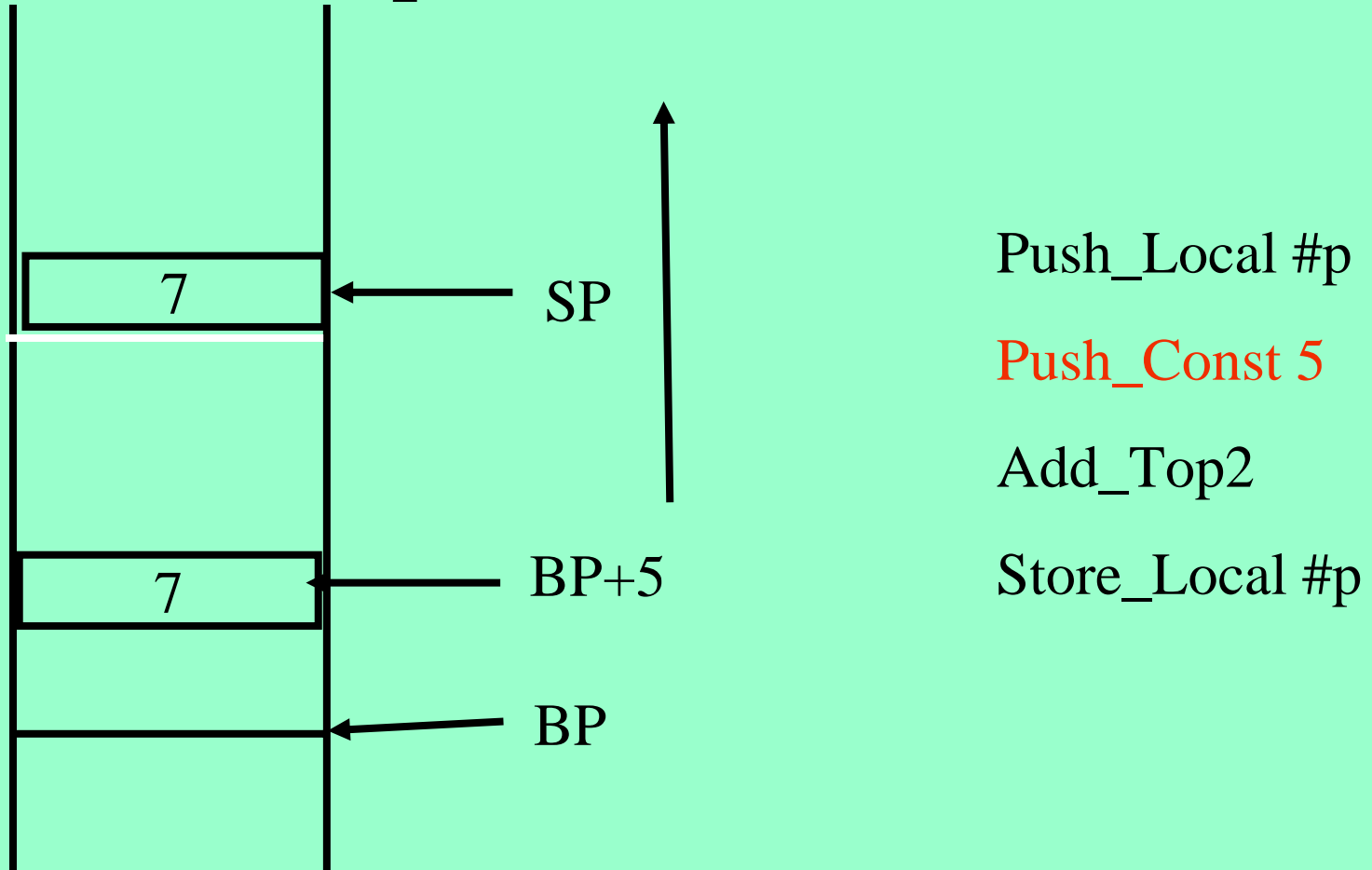
Add_Top2

Store_Local #p

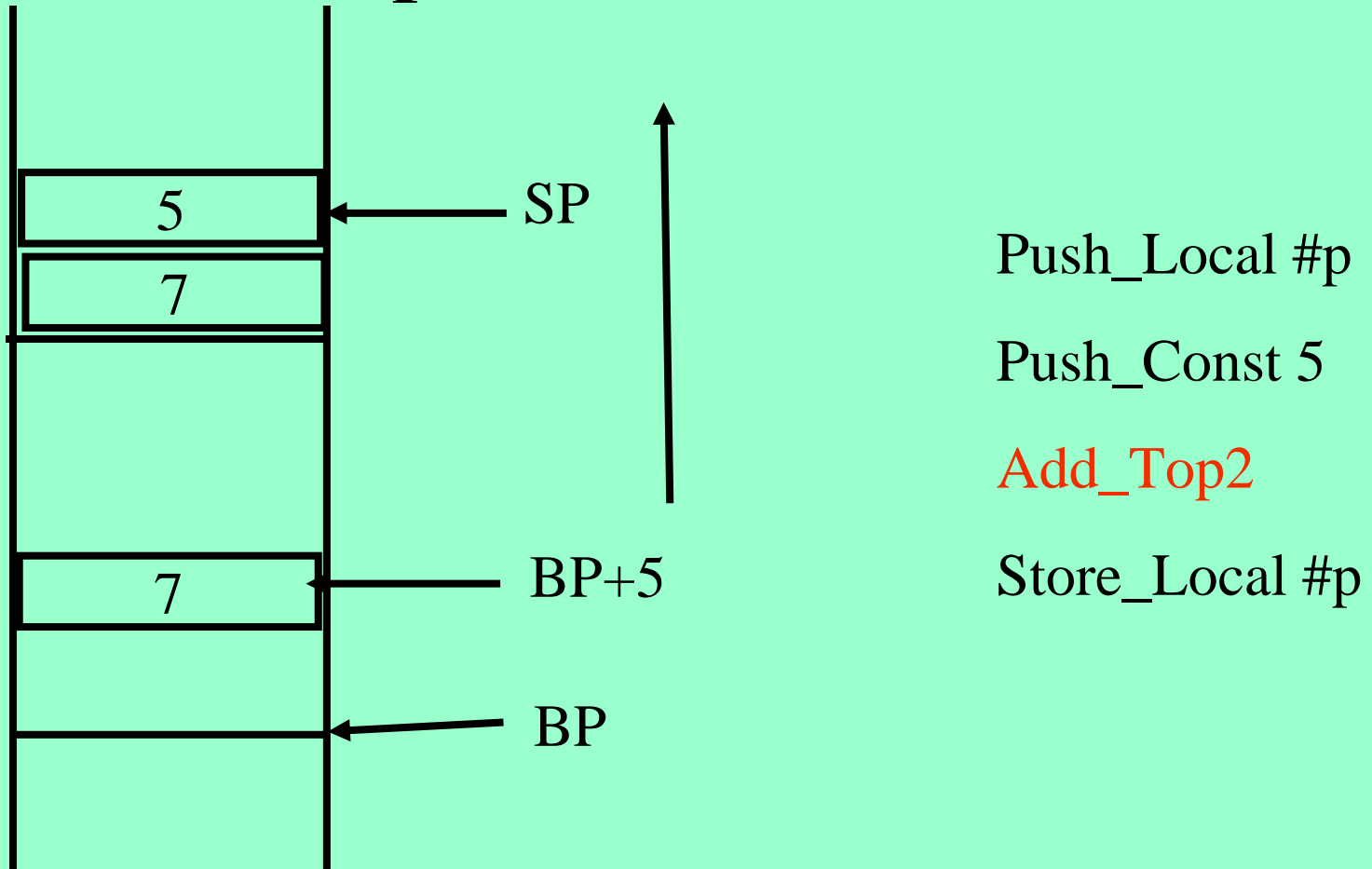
Simple Stack Machine



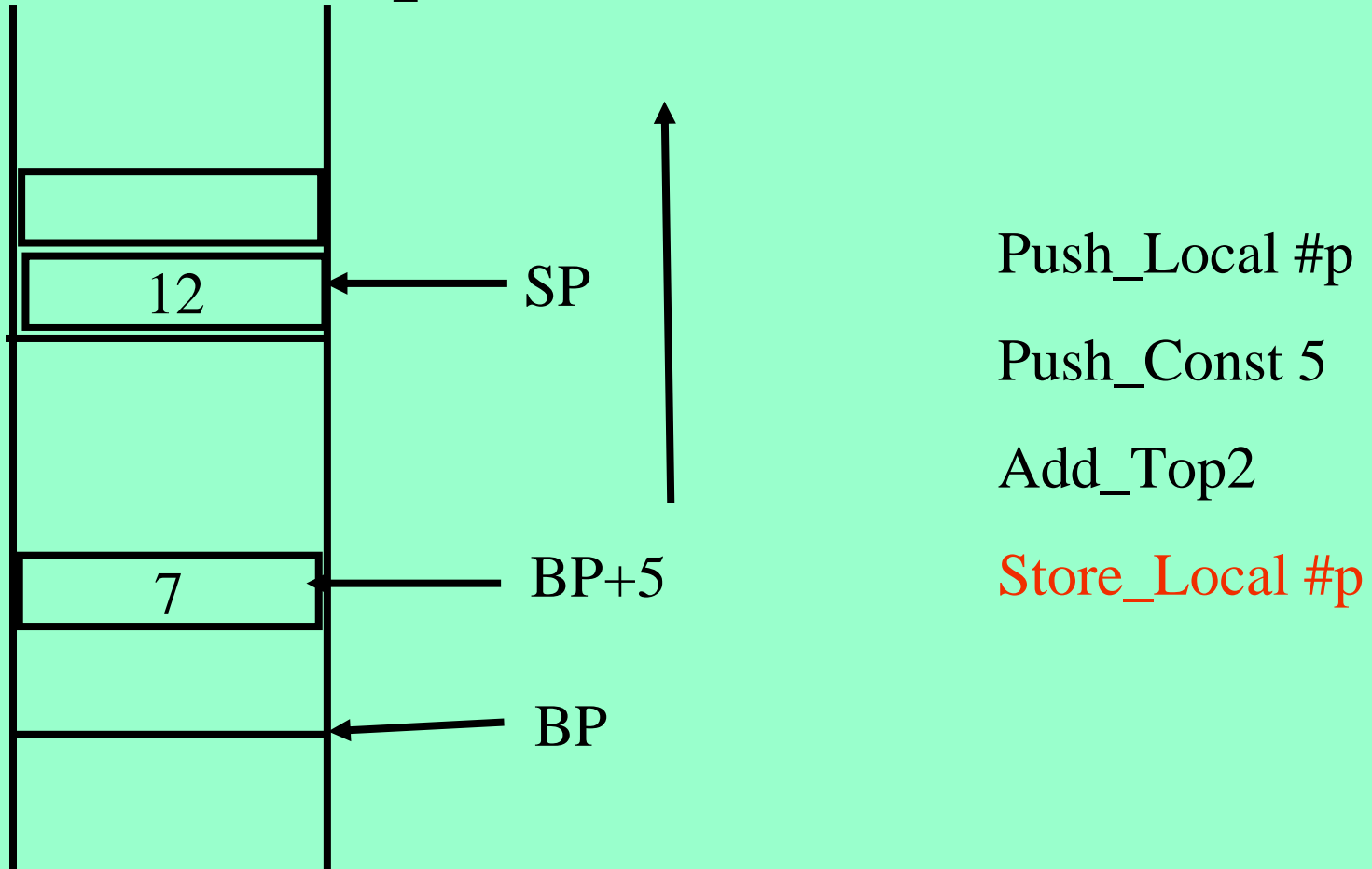
Simple Stack Machine



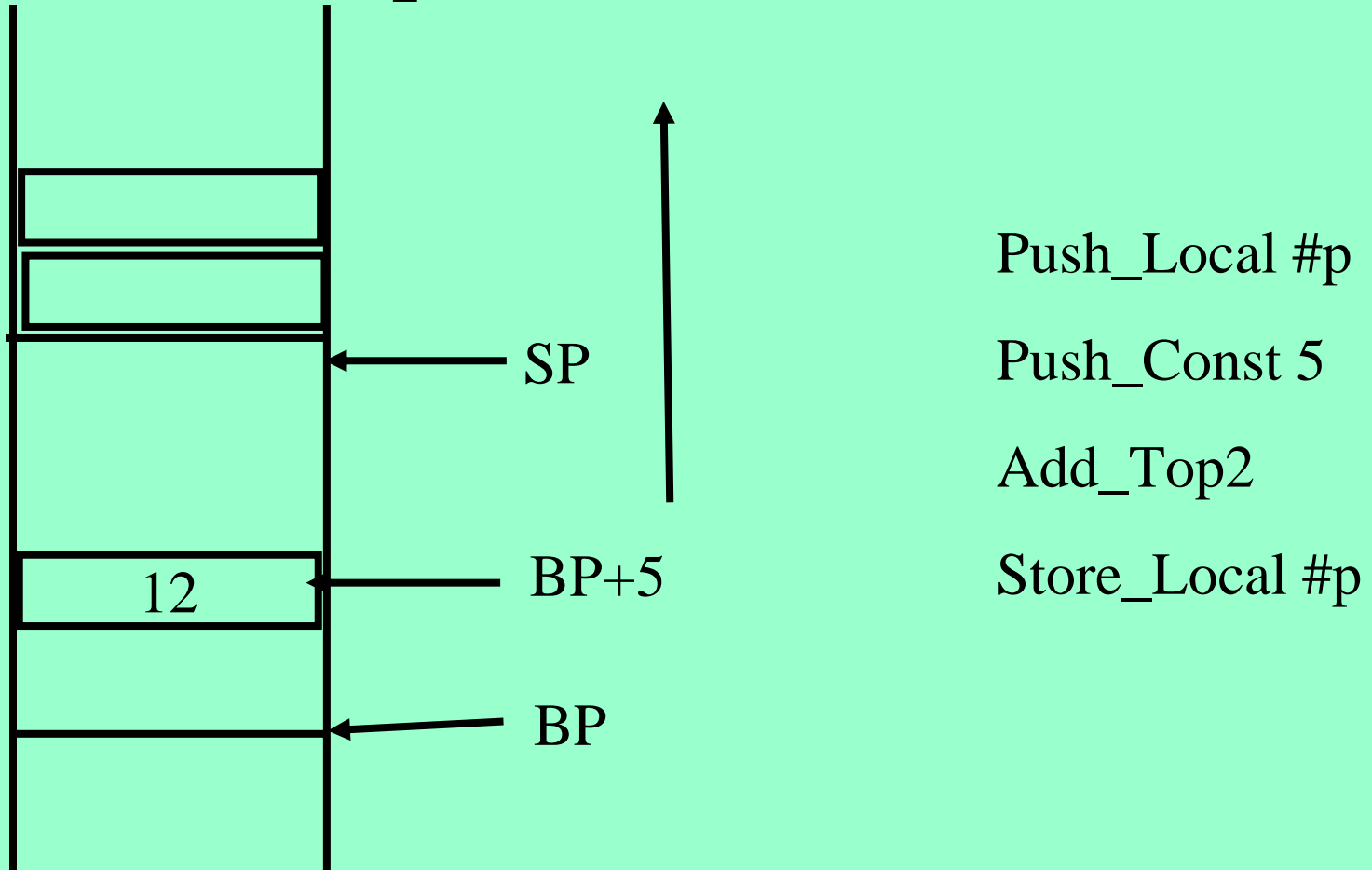
Simple Stack Machine



Simple Stack Machine



Simple Stack Machine



Register Machine

- Fixed set of registers
- Load and store from/to memory
- Arithmetic operations on register only

Register Machine Instructions

Instruction		Actions
Load_Const	c, R_n	$R_n := c;$
Load_Mem	x, R_n	$R_n := x;$
Store_Reg	R_n, x	$x := R_n;$
Add_Reg	R_m, R_n	$R_n := R_n + R_m;$
Subtr_Reg	R_m, R_n	$R_n := R_n - R_m;$
Mult_Reg	R_m, R_n	$R_n := R_n * R_m;$

Example

$p := p + 5$

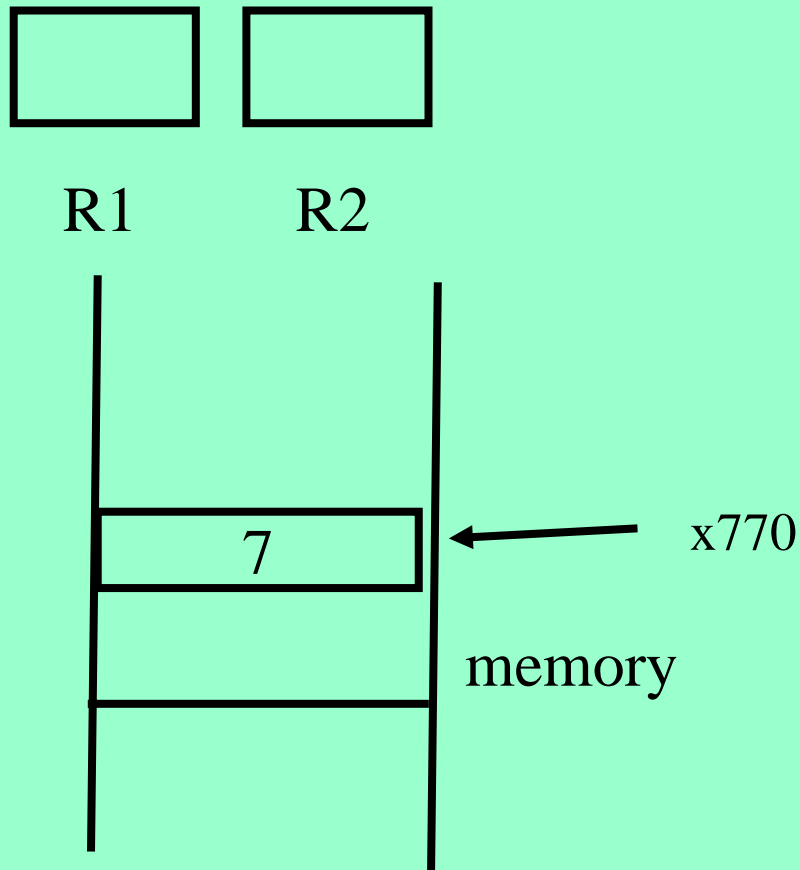
Load_Mem p, R1

Load_Const 5, R2

Add_Reg R2, R1

Store_Reg R1, P

Simple Register Machine



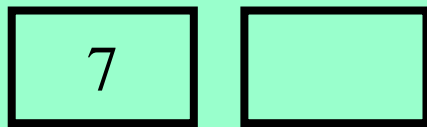
Load_Mem p, R1

Load_Const 5, R2

Add_Reg R2, R1

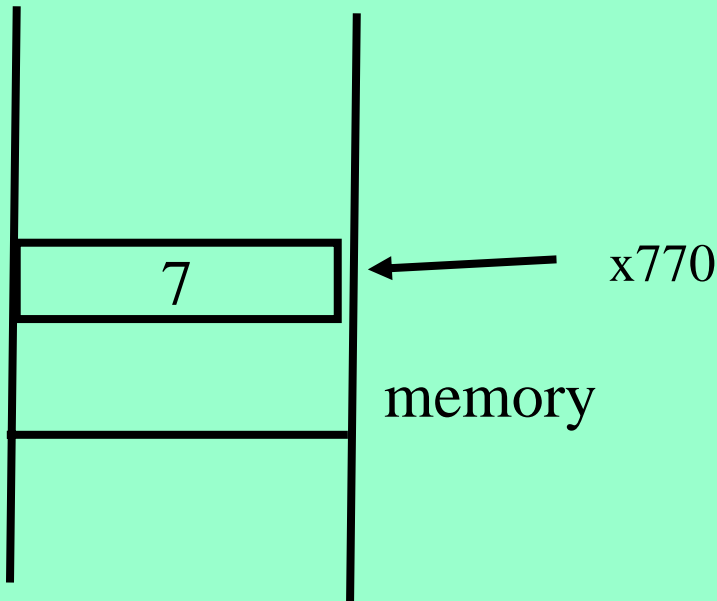
Store_Reg R1, P

Simple Register Machine



R1

R2



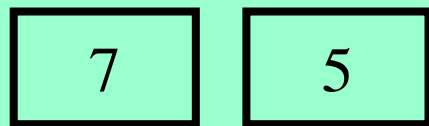
Load_Mem p, R1

Load_Const 5, R2

Add_Reg R2, R1

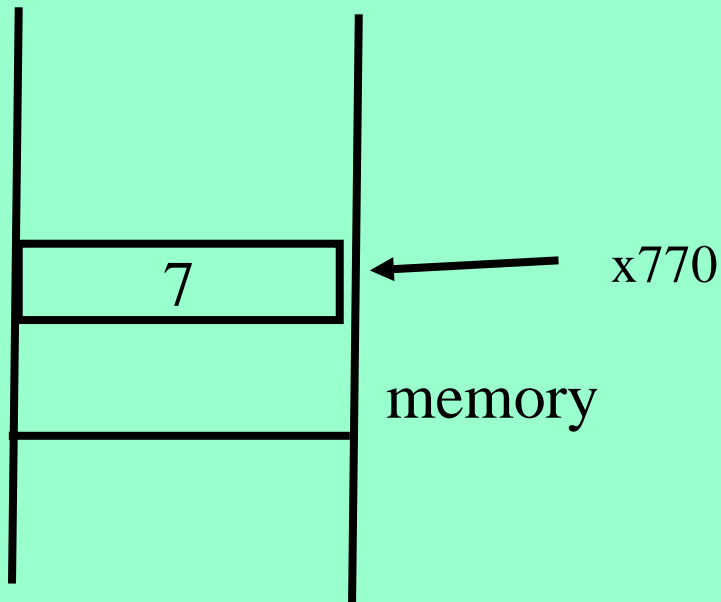
Store_Reg R1, P

Simple Register Machine



R1

R2



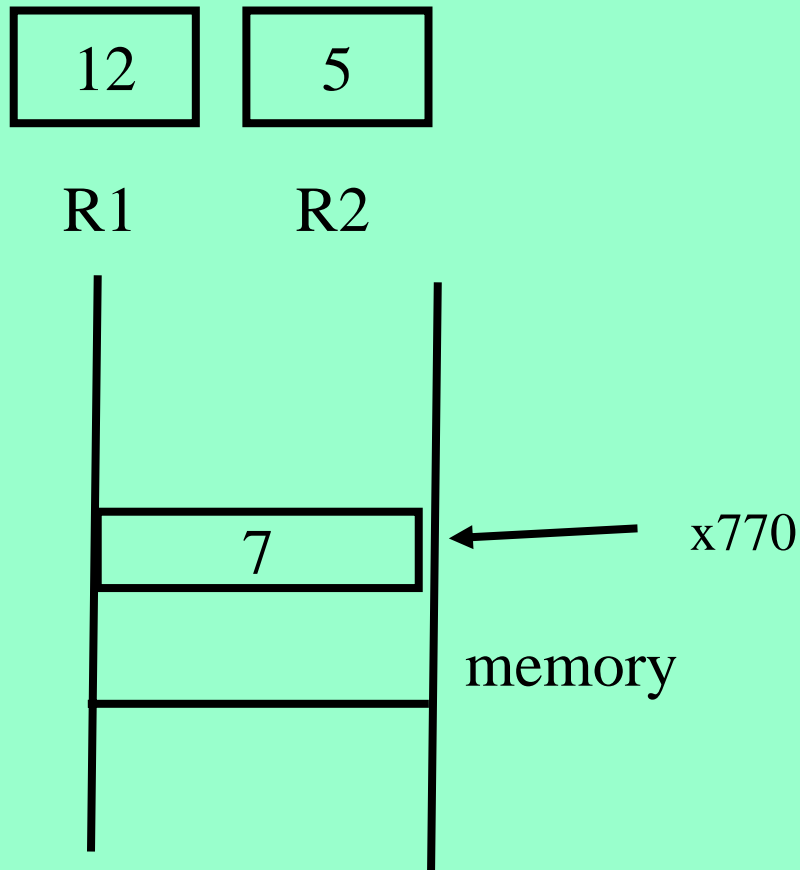
Load_Mem p, R1

Load_Const 5, R2

Add_Reg R2, R1

Store_Reg R1, P

Simple Register Machine



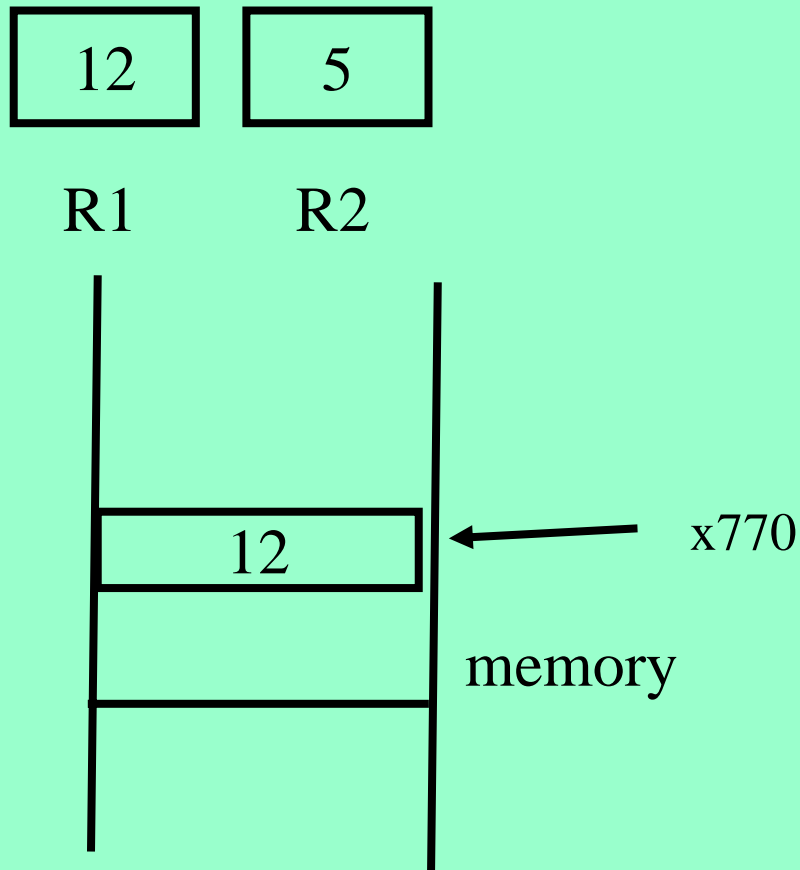
Load_Mem p, R1

Load_Const 5, R2

Add_Reg R2, R1

Store_Reg R1, P

Simple Register Machine



Load_Mem p, R1

Load_Const 5, R2

Add_Reg R2, R1


Store_Reg R1, P

Simple Code Generation for Stack Machine

- Tree rewritings
- Bottom up AST traversal


Abstract Syntax Trees for Stack Machine Instructions

Push_Const c:



```
graph TD; A[Push_Const c:] --- B[c];
```

Push_Local i:



```
graph TD; A[Push_Local i:] --- B[i];
```

Add_Top2:



```
graph TD; A[Add_Top2:] --- B[+]; B --- C[ ]; B --- D[ ];
```

Subtr_Top2:



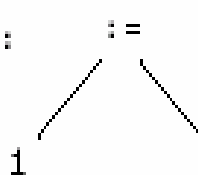
```
graph TD; A[Subtr_Top2:] --- B[-]; B --- C[ ]; B --- D[ ];
```

Mult_Top2:



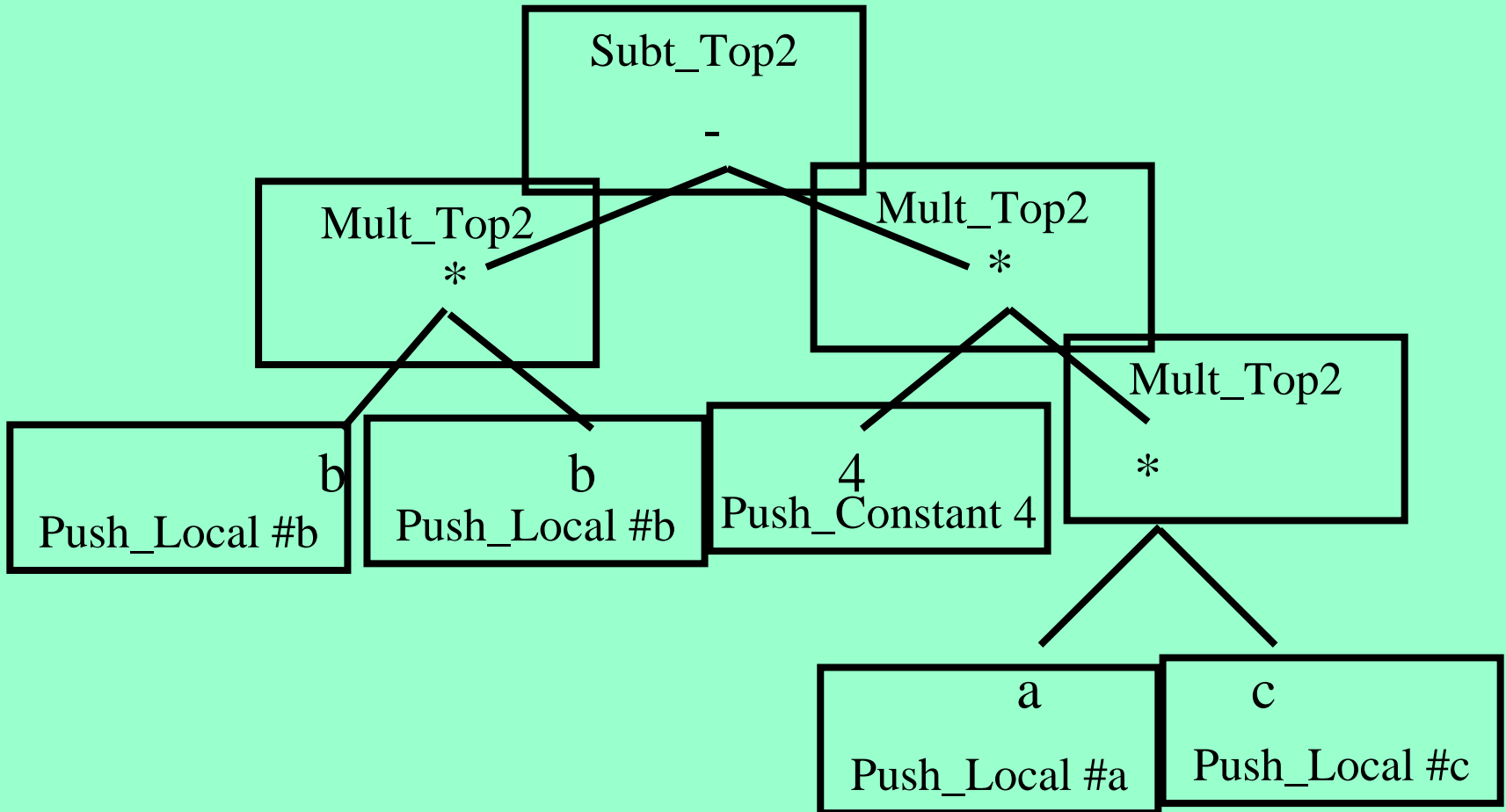
```
graph TD; A[Mult_Top2:] --- B[*]; B --- C[ ]; B --- D[ ];
```

Store_Local i:



```
graph TD; A[Store_Local i:] --- B[:=]; B --- C[i]; B --- D[ ];
```


Example



Bottom-Up Code Generation

```
PROCEDURE Generate code (Node):  
  SELECT Node .type:  
    CASE Constant type:  Emit ("Push_Const" Node .value);  
    CASE LocalVar type:  Emit ("Push_Local" Node .number);  
    CASE StoreLocal type: Emit ("Store_Local" Node .number);  
    CASE Add type:  
      Generate code (Node .left); Generate code (Node .right);  
      Emit ("Add_Top2");  
    CASE Subtract type:  
      Generate code (Node .left); Generate code (Node .right);  
      Emit ("Subtr_Top2");  
    CASE Multiply type:  
      Generate code (Node .left); Generate code (Node .right);  
      Emit ("Mult_Top2");
```

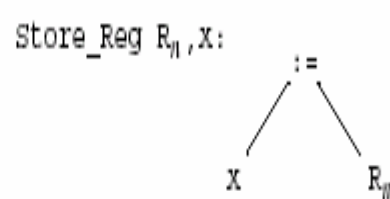
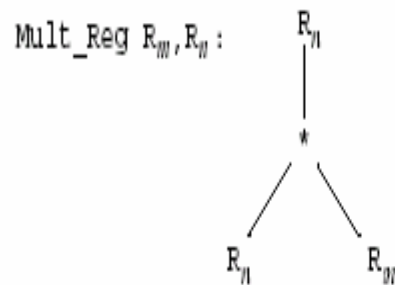
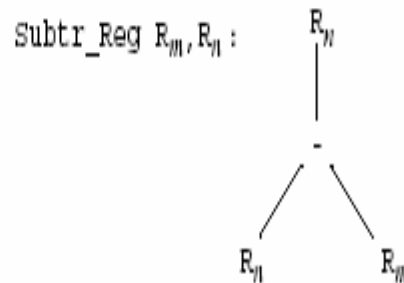
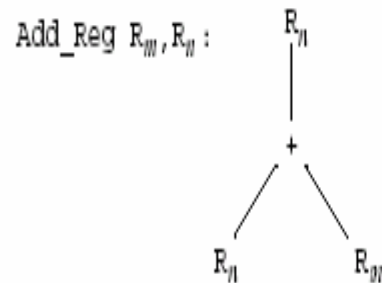
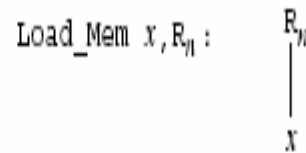
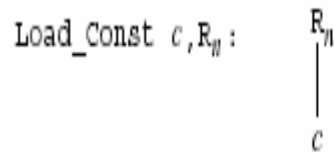
Simple Code Generation for Register Machine

- Need to allocate register for temporary values
 - AST nodes
- The number of machine registers may not suffice
- Simple Algorithm:
 - Bottom up code generation
 - Allocate registers for subtrees

Register Machine Instructions

Instruction		Actions
Load_Const	c, R_n	$R_n := c;$
Load_Mem	x, R_n	$R_n := x;$
Store_Reg	R_n, x	$x := R_n;$
Add_Reg	R_m, R_n	$R_n := R_n + R_m;$
Subtr_Reg	R_m, R_n	$R_n := R_n - R_m;$
Mult_Reg	R_m, R_n	$R_n := R_n * R_m;$

Abstract Syntax Trees for Register Machine Instructions



Simple Code Generation

- Assume enough registers
- Use DFS to:
 - Generate code
 - Assign Registers
 - Target register
 - Auxiliary registers

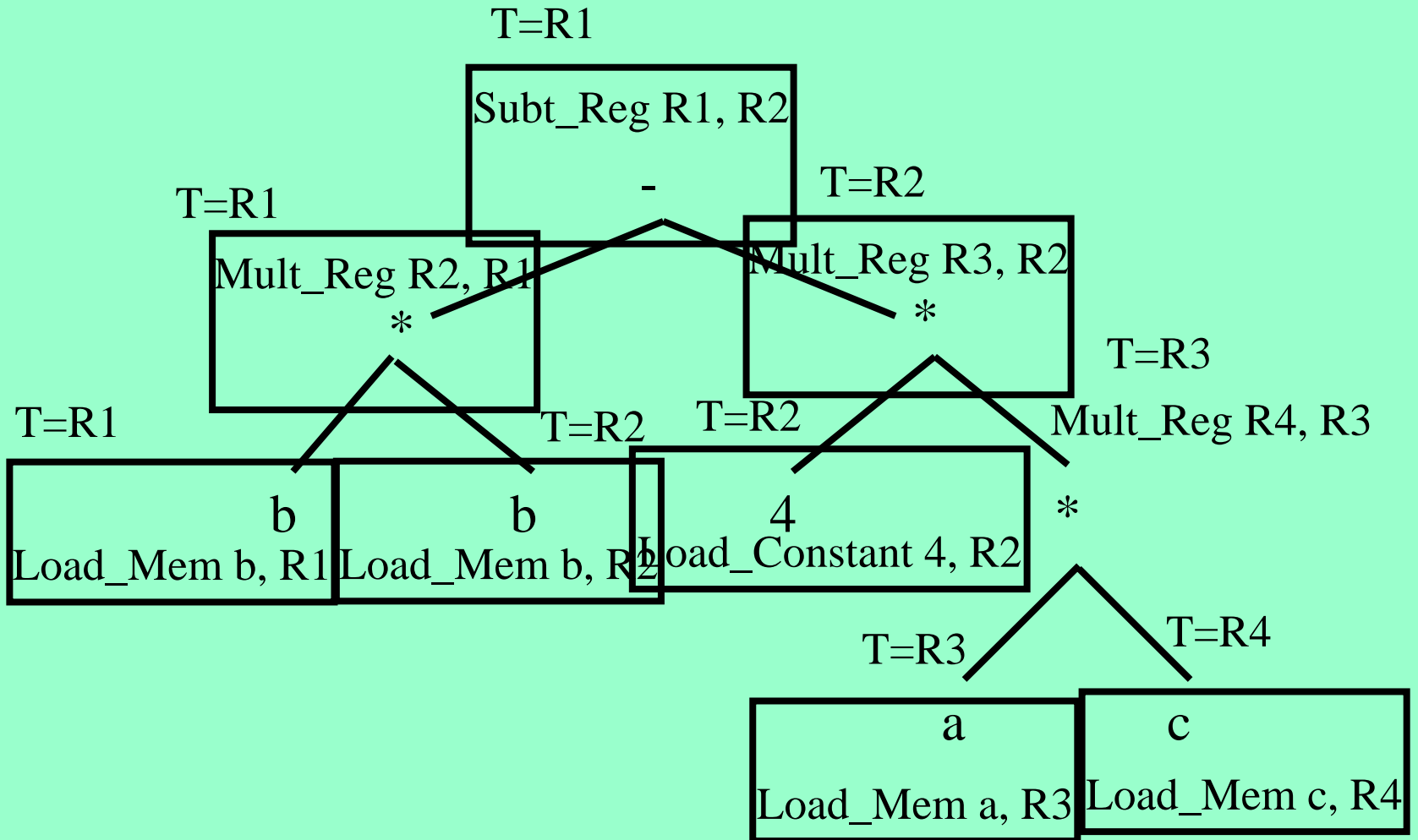
Code Generation with Register Allocation

```
PROCEDURE Generate code (Node, a register Target, a register set Aux):
  SELECT Node .type:
    CASE Constant type:
      Emit ("Load_Const " Node .value ",R" Target);
    CASE Variable type:
      Emit ("Load_Mem " Node .address ",R" Target);
    CASE ...
    CASE Add type:
      Generate code (Node .left, Target, Aux);
      SET Target 2 TO An arbitrary element of Aux;
      SET Aux 2 TO Aux \ Target 2;
      // the \ denotes the set difference operation
      Generate code (Node .right, Target 2, Aux 2);
      Emit ("Add_Reg R" Target 2 ",R" Target);
    CASE ...
```

Code Generation with Register Allocation(2)

```
PROCEDURE Generate code (Node, a register number Target):
  SELECT Node .type:
    CASE Constant type:
      Emit ("Load_Const " Node .value ",R" Target);
    CASE Variable type:
      Emit ("Load_Mem " Node .address ",R" Target);
    CASE ...
    CASE Add type:
      Generate code (Node .left, Target);
      Generate code (Node .right, Target+1);
      Emit ("Add_Reg R" Target+1 ",R" Target);
    CASE ...
```


Example



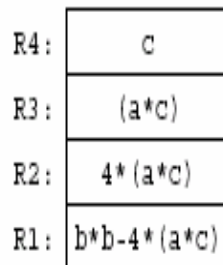
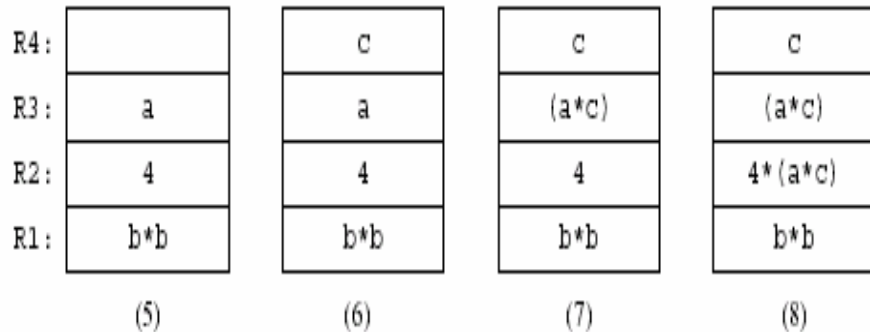
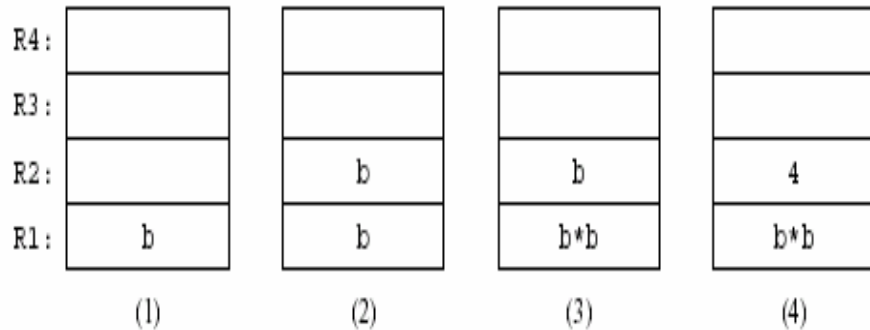
Example

Load_Mem	b, R1
Load_Mem	b, R2
Mult_Reg	R2, R1
Load_Const	4, R2
Load_Mem	a, R3
Load_Mem	c, R4
Mult_Reg	R4, R3
Mult_Reg	R3, R2
Subtr_Reg	R2, R1

Runtime Evaluation

```

Load_Mem  b, R1
Load_Mem  b, R2
Mult_Reg  R2, R1
Load_Const 4, R2
Load_Mem  a, R3
Load_Mem  c, R4
Mult_Reg  R4, R3
Mult_Reg  R3, R2
Subtr_Reg R2, R1
    
```



Optimality

- The generated code is suboptimal
- May consume more registers than necessary
 - May require storing temporary results
- Leads to larger execution time

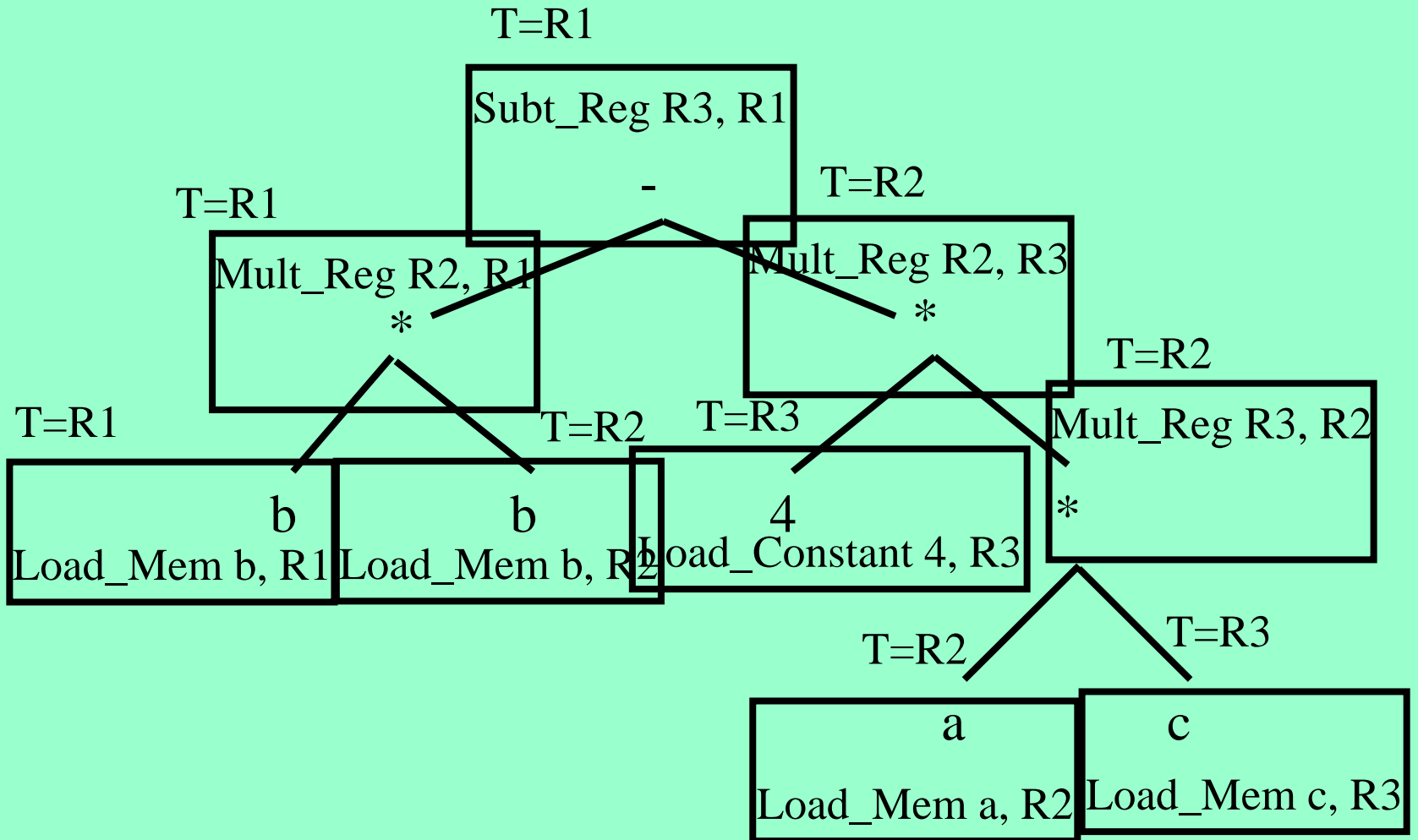
Example

Load_Mem	b, R1
Load_Mem	b, R2
Mult_Reg	R2, R1
Load_Const	4, R2
Load_Mem	a, R3
Load_Mem	c, R4
Mult_Reg	R4, R3
Mult_Reg	R3, R2
Subtr_Reg	R2, R1

Observation (Aho&Sethi)

- The compiler can reorder the computations of sub-expressions
- The code of the right-subtree can appear before the code of the left-subtree
- May lead to faster code

Example



Example

Load_Mem b, R1

Load_Mem b, R2

Mult_Reg R2, R1

Load_Mem a, R2

Load_Mem c, R3

Mult_Reg R3, R2

Load_Constant 4, R3

Mult_Reg R2, R3

Subt_Reg R3, R1

Two Phase Solution

Dynamic Programming

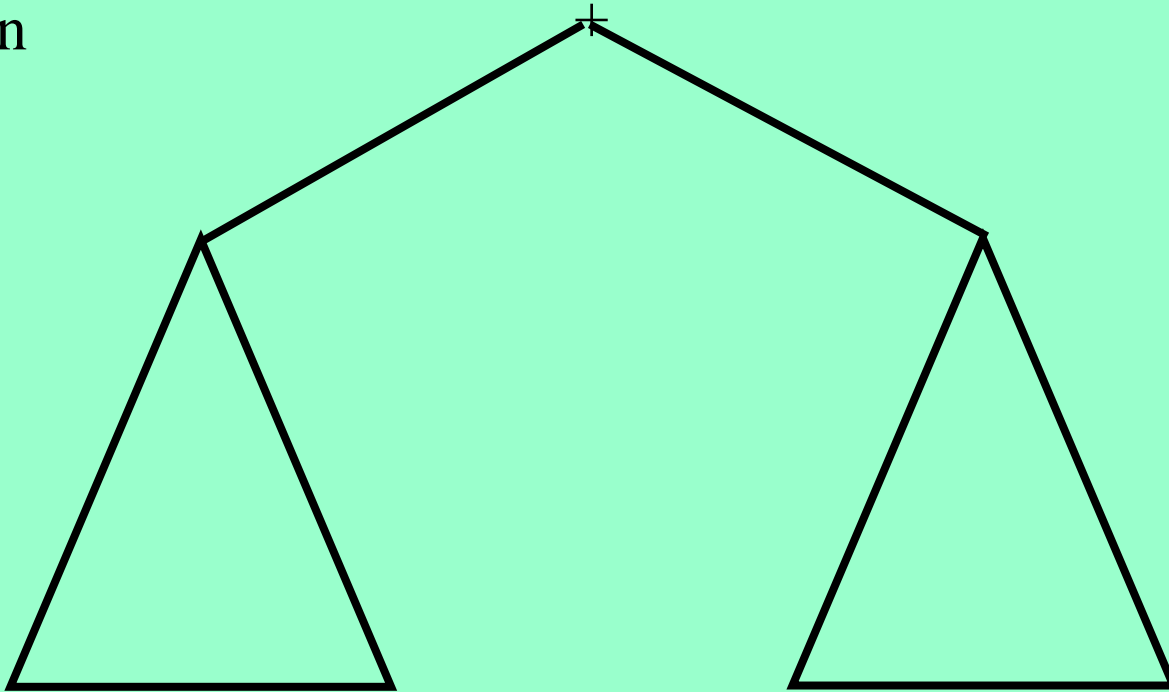
Sethi & Ullman

- Bottom-up (labeling)
 - Compute for every subtree
 - The minimal number of registers needed
 - Weight
- Top-Down
 - Generate the code using labeling by preferring “heavier” subtrees (larger labeling)

The Labeling Principle

m registers

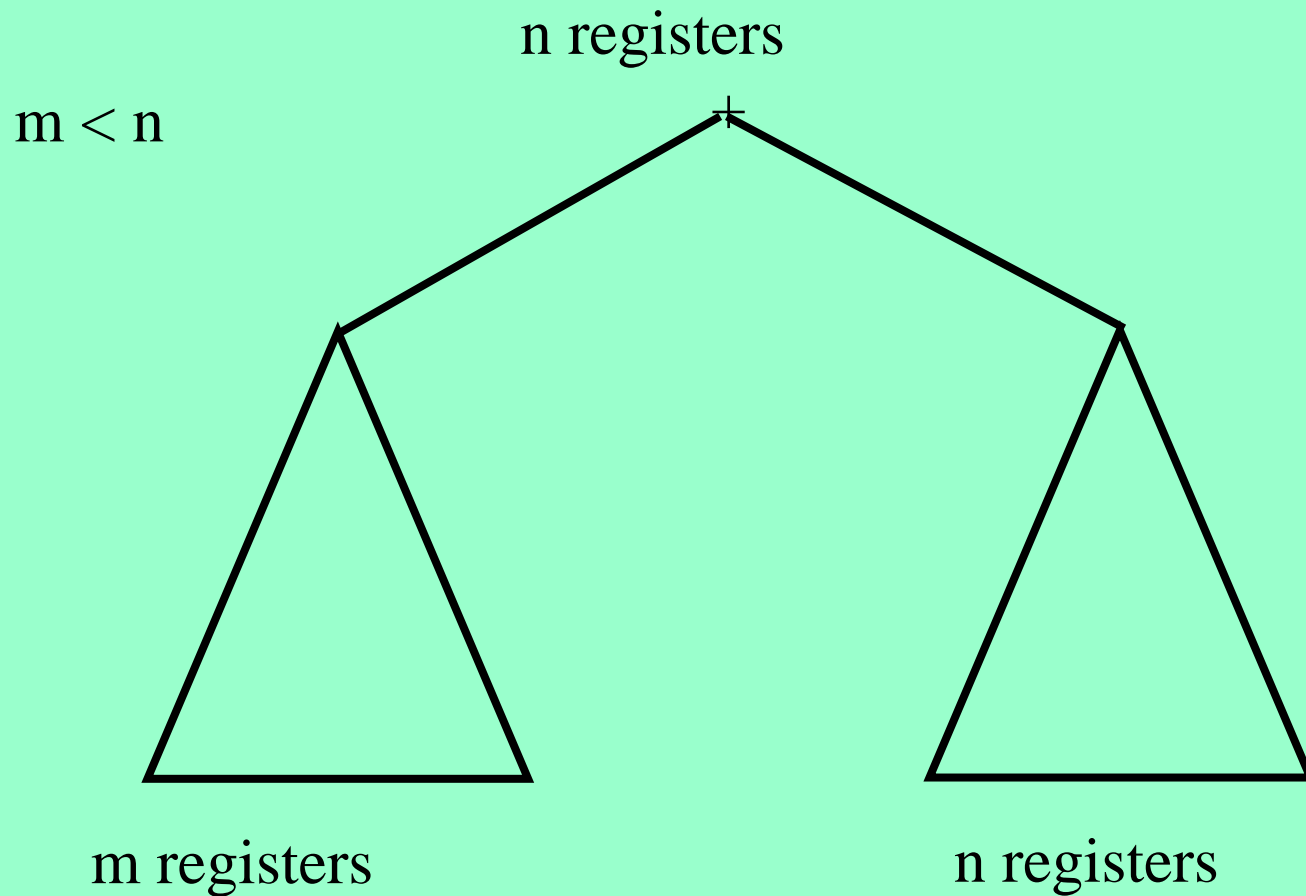
$m > n$



m registers

n registers

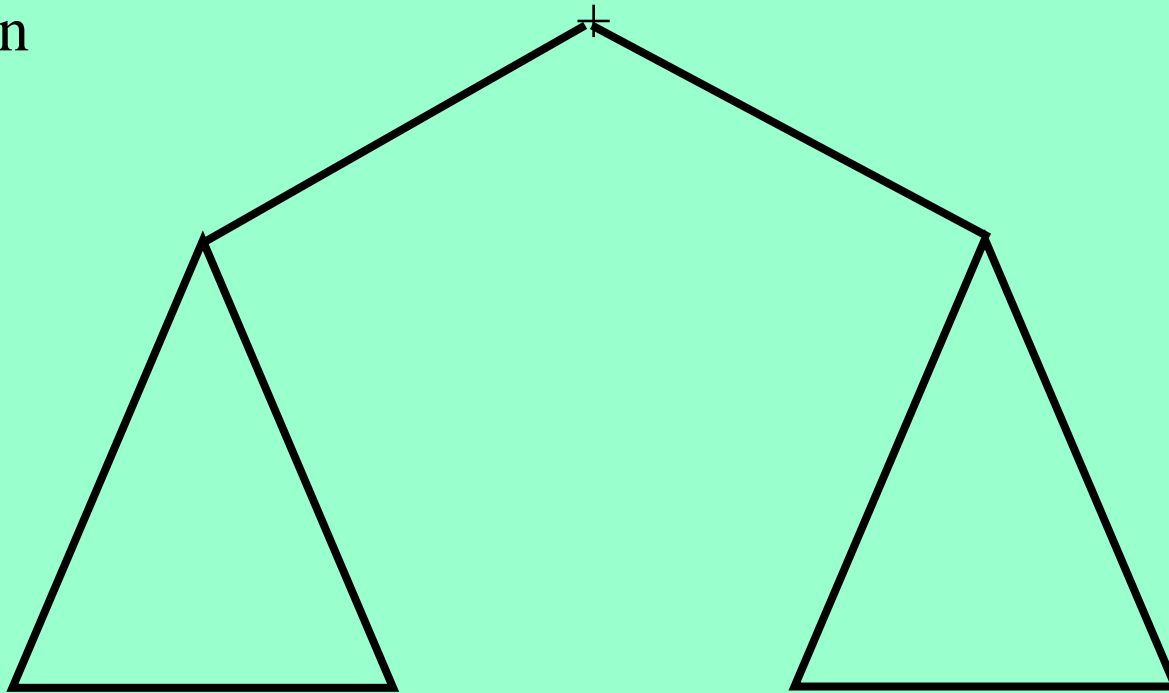
The Labeling Principle



The Labeling Principle

$m+1$ registers

$m = n$



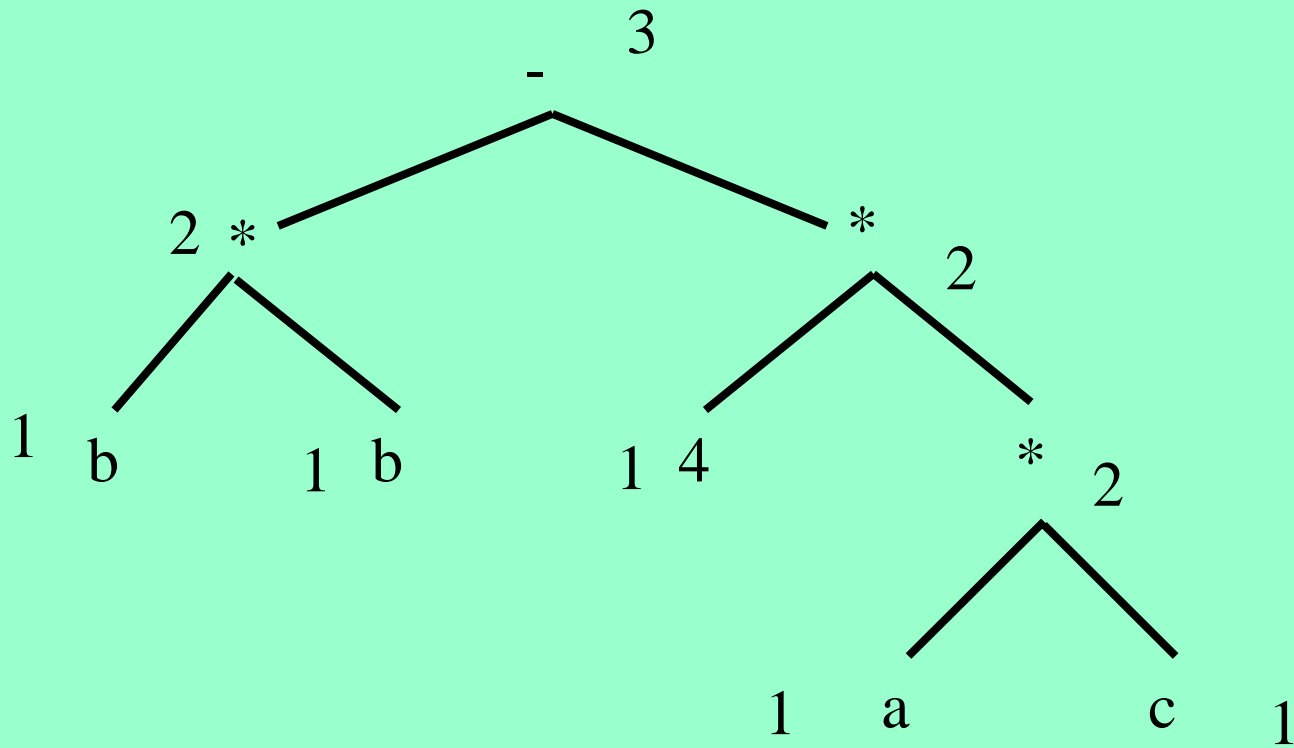
m registers

n registers

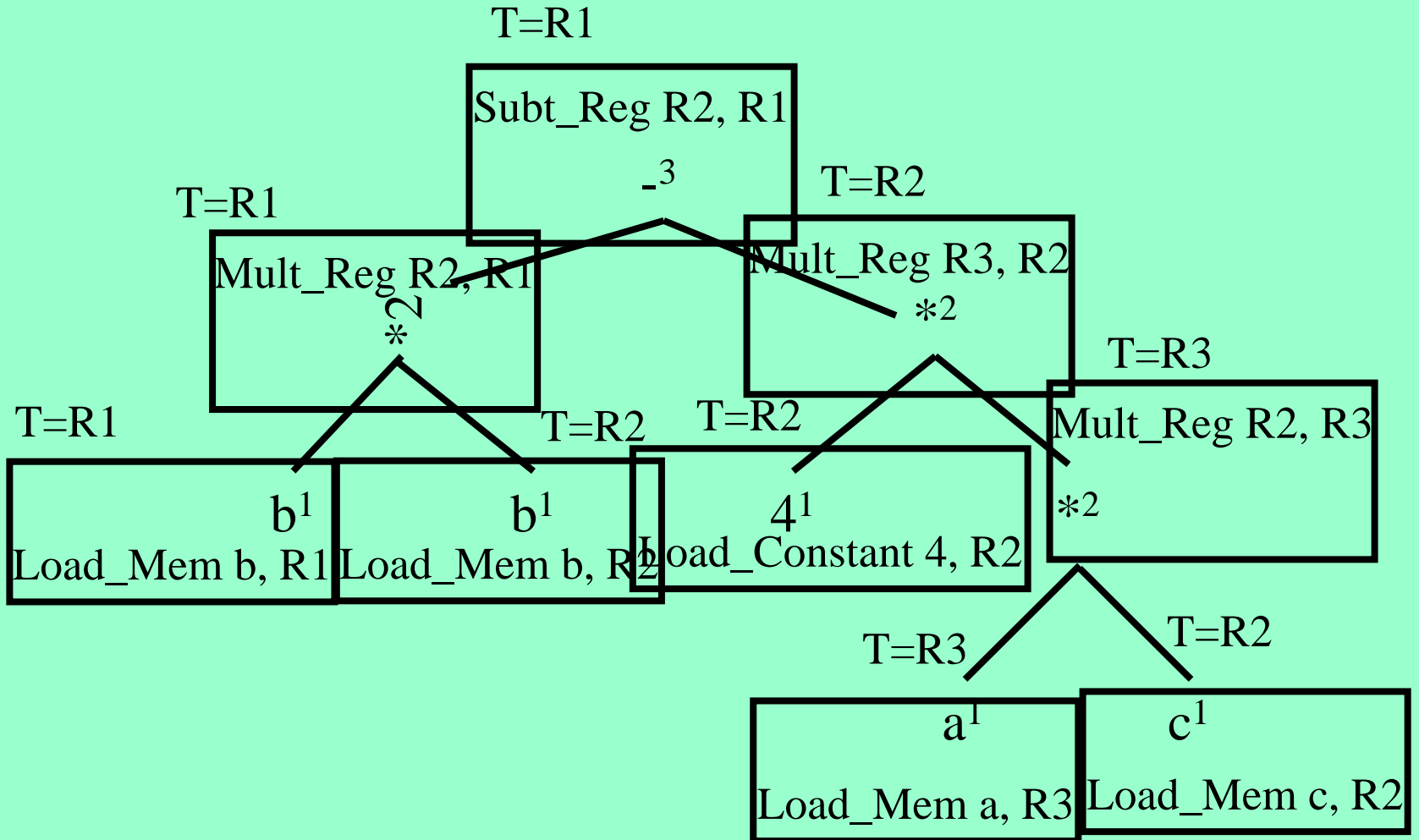
The Labeling Procedure

```
FUNCTION Weight of (Node) RETURNING an integer:
  SELECT Node .type:
    CASE Constant type: RETURN 1;
    CASE Variable type: RETURN 1;
    CASE ...
    CASE Add type:
      SET Required left TO Weight of (Node .left);
      SET Required right TO Weight of (Node .right);
      IF Required left > Required right: RETURN Required left;
      IF Required left < Required right: RETURN Required right;
      // Required left = Required right
      RETURN Required left + 1;
    CASE ...
```

Labeling the example (weight)



Top-Down



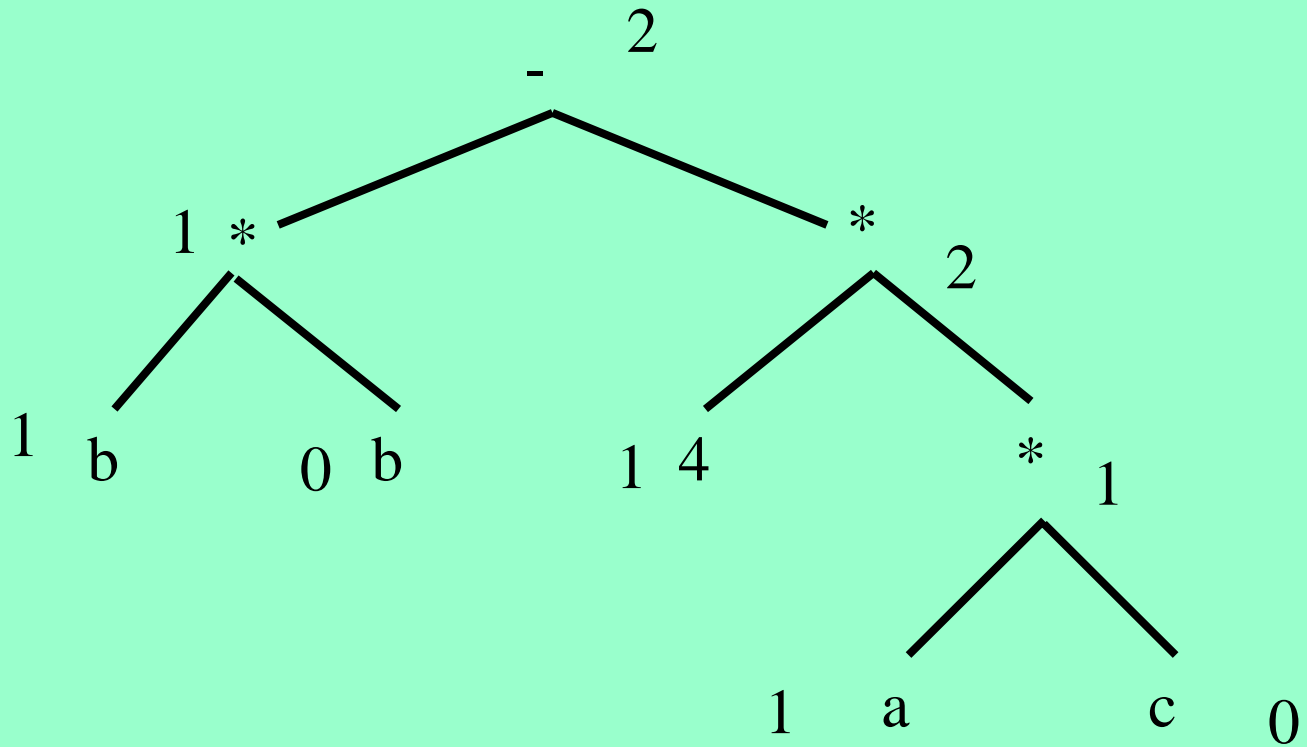
Generalizations

- More than two arguments for operators
 - Function calls
- Register/memory operations
- Multiple effected registers
- Spilling
 - Need more registers than available

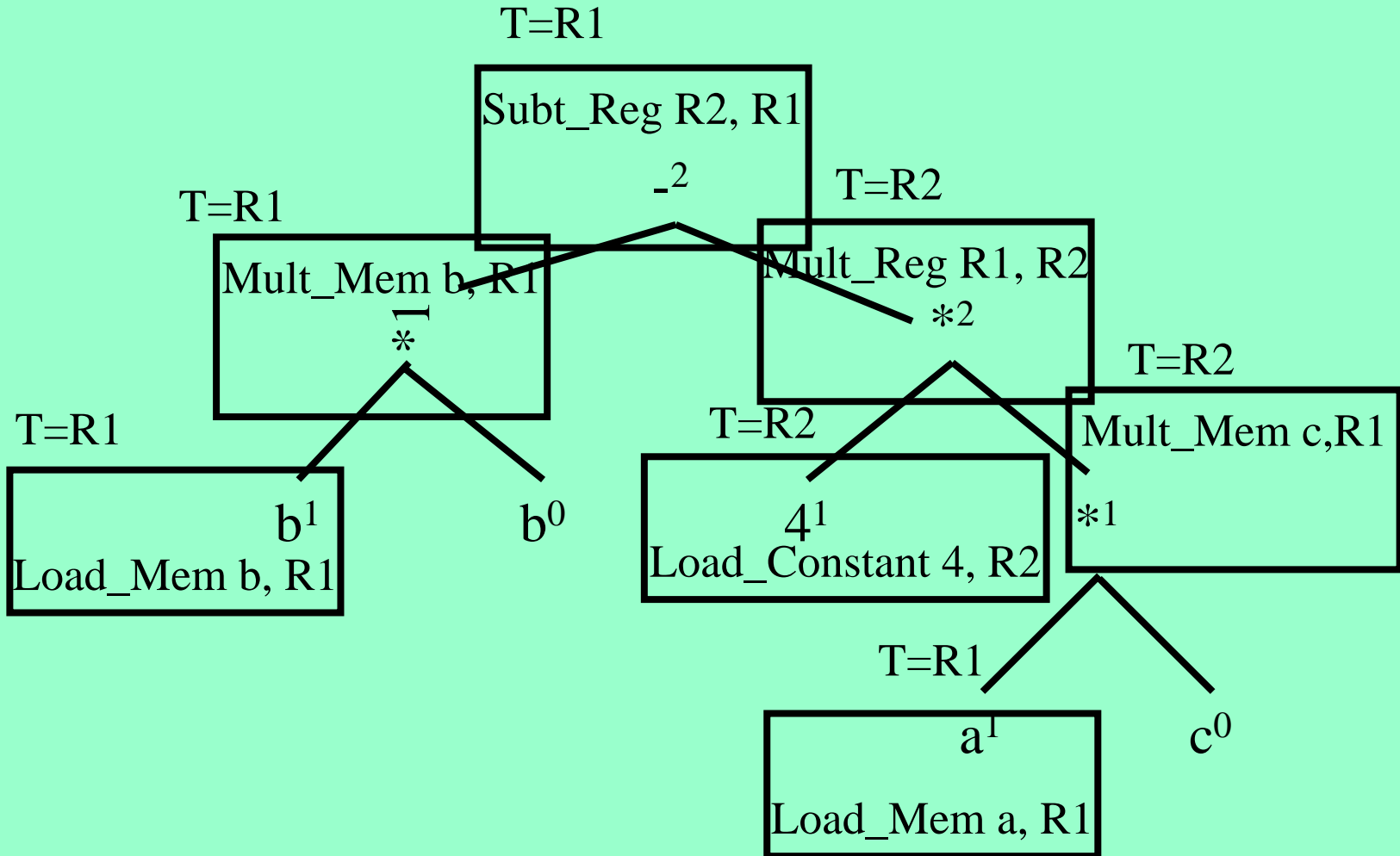
Register Memory Operations

- Add_Mem X, R1
- Mult_Mem X, R1
- No need for registers to store right operands

Labeling the example (weight)



Top-Down



Empirical Results

- Experience shows that for handwritten programs 5 registers suffice (Yuval 1977)
- But program generators may produce arbitrary complex expressions

Spilling

- Even an optimal register allocator can require more registers than available
- Need to generate code for every correct program
- The compiler can save temporary results
 - Spill registers into temporaries
 - Load when needed
- Many heuristics exist

Simple Spilling Method

- Heavy tree – Needs more registers than available
- A `heavy' tree contains a `heavy' subtree whose dependents are `light'
- Generate code for the light tree
- Spill the content into memory and replace subtree by temporary
- Generate code for the resultant tree

Simple Spilling Method

```
PROCEDURE Generate code for large trees (Node, Target register):
  SET Auxiliary register set TO
    Available register set \ Target register;

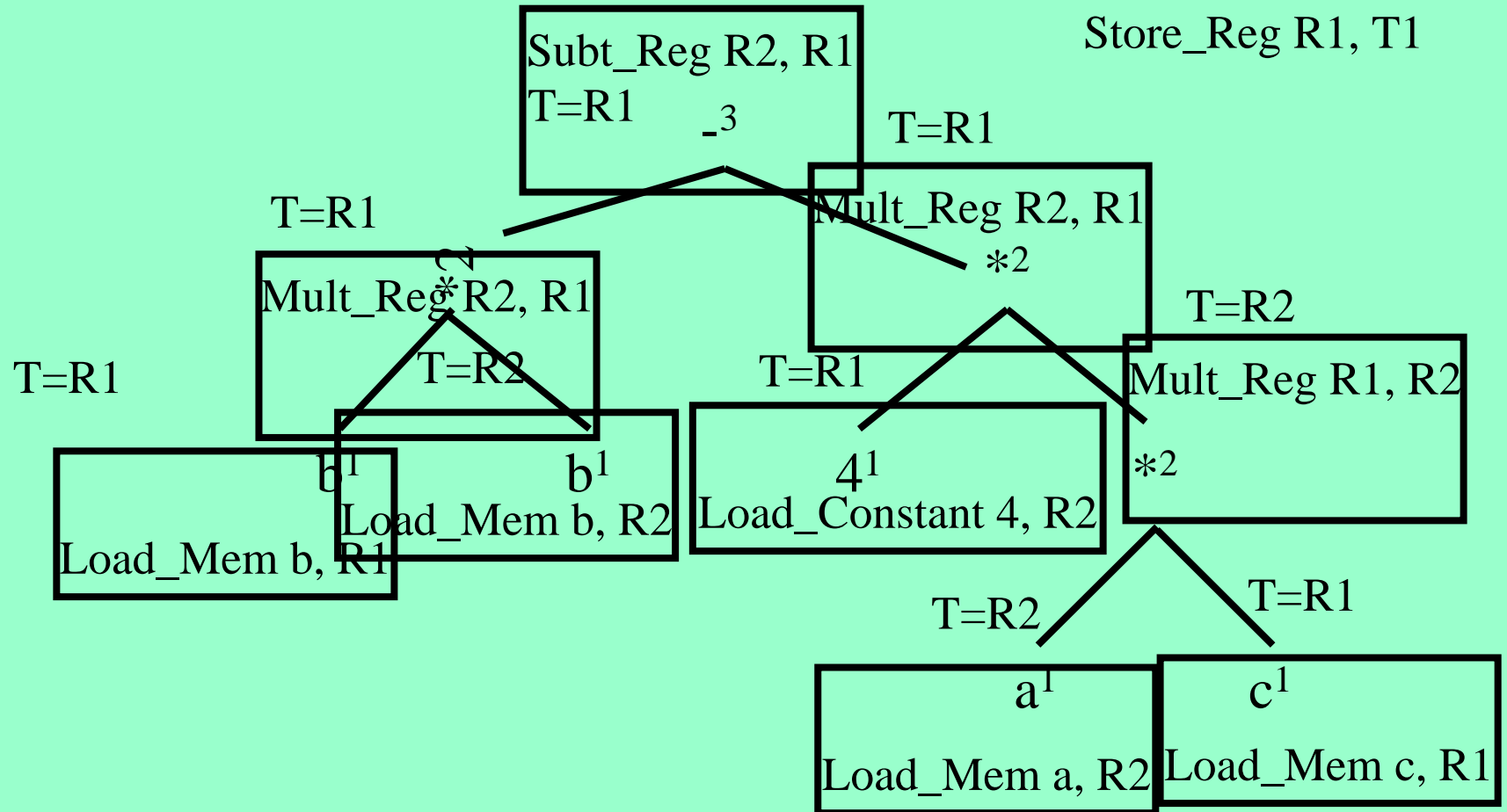
  WHILE Node /= No node:
    Compute the weights of all nodes of the tree of Node;
    SET Tree node TO Maximal non_large tree (Node);
    Generate code
      (Tree node, Target register, Auxiliary register set);

    IF Tree node /= Node:
      SET Temporary location TO Next free temporary location();
      Emit ("Store R" Target register ",T" Temporary location);
      Replace Tree node by a reference to Temporary location;
      Return any temporary locations in the tree of Tree node
        to the pool of free temporary locations;
    ELSE Tree node = Node:
      Return any temporary locations in the tree of Node
        to the pool of free temporary locations;
      SET Node TO No node;

FUNCTION Maximal non_large tree (Node) RETURNING a node:
  IF Node .weight <= Size of Auxiliary register set: RETURN Node;
  IF Node .left .weight > Size of Auxiliary register set:
    RETURN Maximal non_large tree (Node .left);
  ELSE Node .right .weight >= Size of Auxiliary register set:
    RETURN Maximal non_large tree (Node .right);
```

Top-Down (2 registers)

Load_Mem T1, R2



Top-Down (2 registers)

Load_Mem a, R2
Load_Mem c, R1
Mult_Reg R1, R2
Load_Constant 4, R2
Mult_Reg R2, R1
Store_Reg R1, T1
Load_Mem b, R1
Load_Mem b, R2
Mult_Reg R2, R1
Load_Mem T1, R2
Subtr_Reg R2, R1

Summary

- Register allocation of expressions is simple
- Good in practice
- Optimal under certain conditions
 - Uniform instruction cost
 - `Symbolic' trees
- Can handle non-uniform cost
 - Code-Generator Generators exist (BURS)
- Even simpler for 3-address machines
- Simple ways to determine best orders
- But misses opportunities to share registers between different expressions
 - Can employ certain conventions
- Better solutions exist
 - Graph coloring

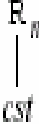
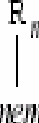
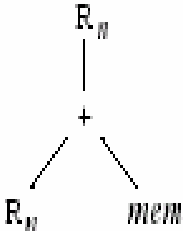
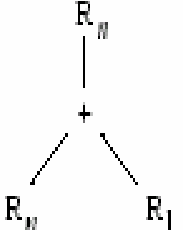
Code Generation for Basic Blocks Introduction

Chapter 4.2.5

The Code Generation Problem

- Given
 - AST
 - Machine description
 - Number of registers
 - Instructions + cost
- Generate code for AST with minimum cost
- NPC [Aho 77]

Example Machine Description

#1		Load_Const <i>cst</i> , <i>R_n</i>	load constant	cost = 1
#2		Load_Mem <i>mem</i> , <i>R_n</i>	load from memory	cost = 3
#3		Add_Mem <i>mem</i> , <i>R_n</i>	add from memory	cost = 3
#4		Add_Reg <i>R₁</i> , <i>R_n</i>	add registers	cost = 1

Simplifications

- Consider small parts of AST at time
 - One expression at the time
- Target machine simplifications
 - Ignore certain instructions
- Use simplifying conventions

Basic Block

- Parts of control graph without split
- A sequence of assignments and expressions which are always executed together
- **Maximal Basic Block** Cannot be extended
 - Start at label or at routine entry
 - Ends just before jump like node, label, procedure call, routine exit

Example

```
void foo()  
{  
    if (x > 8) {  
        z = 9;  
        t = z + 1;  
    }  
    z = z * z;  
    t = t - z;  
    bar();  
    t = t + 1;  
}
```

x > 8

z = 9;
t = z + 1;

z = z * z;
t = t - z;

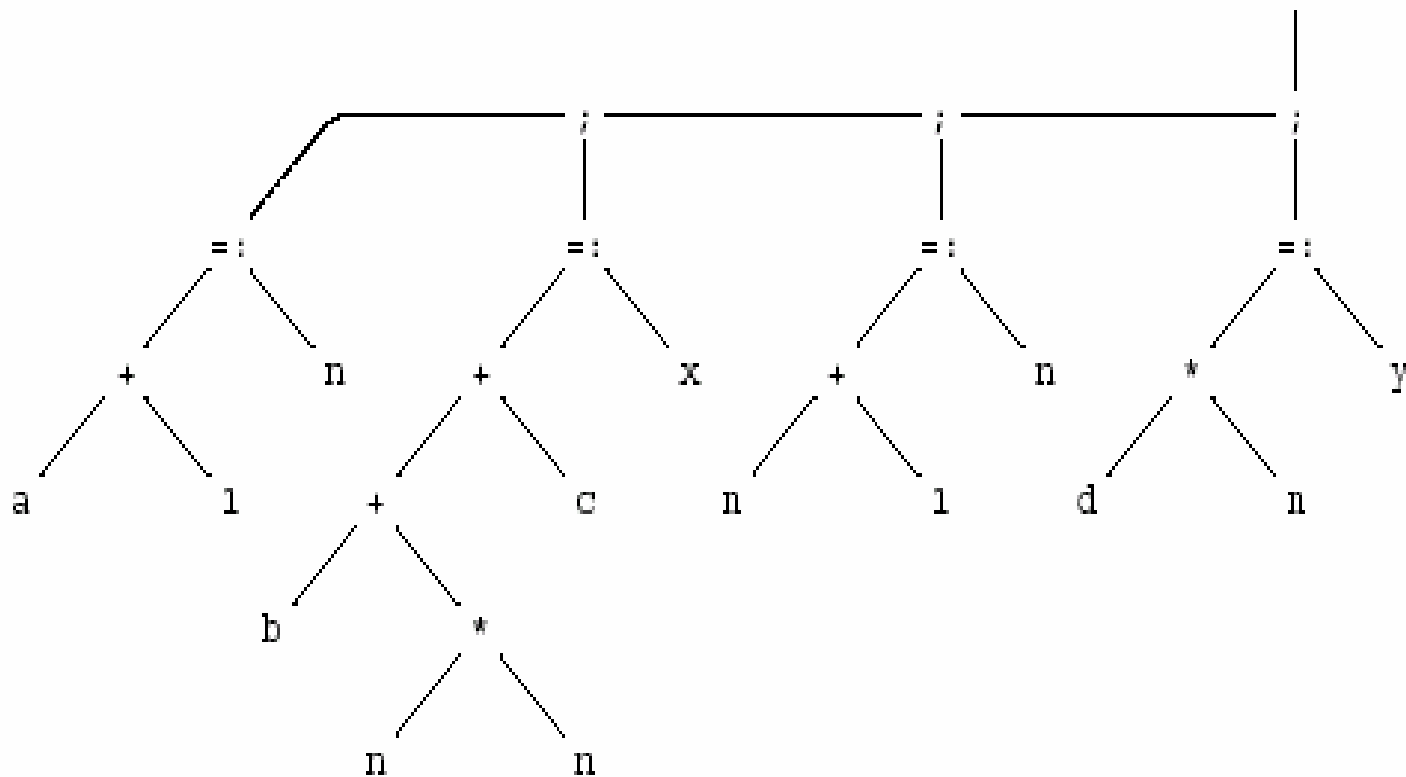
bar()

t = t + 1;

Running Example

```
{   int n;  
  
    n = a + 1;  
    x = b + n*n + c;  
    n = n + 1;  
    y = d * n;  
}
```

Running Example AST



Optimized code(gcc)

```
{  int n;  
  
    n = a + 1;  
    x = b + n*n + c;  
    n = n + 1;  
    y = d * n;  
}
```

```
Load_Mem   a,R1  
Add_Const  1,R1  
Load_Reg   R1,R2  
  
Mult_Reg   R1,R2  
Add_Mem    b,R2  
Add_Mem    c,R2  
Store_Reg  R2,x  
  
Add_Const  1,R1  
Mult_Mem   d,R1  
Store_Reg  R1,y
```

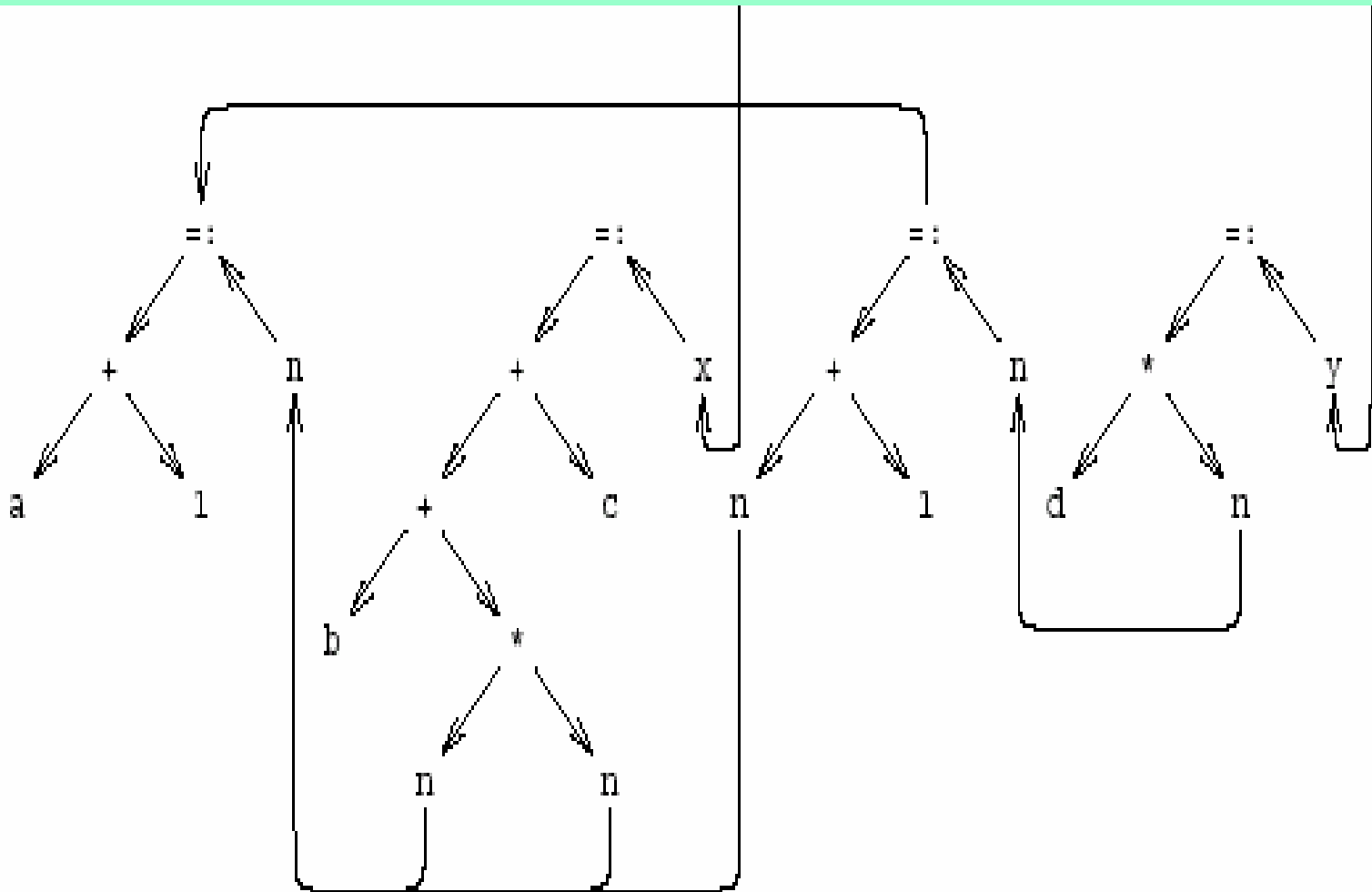
Outline

- Dependency graphs for basic blocks
- Transformations on dependency graphs
- From dependency graphs into code
 - Instruction selection
(linearizations of dependency graphs)
 - Register allocation (the general idea)

Dependency graphs

- Threaded AST imposes an order of execution
- The compiler can reorder assignments as long as the program results are not changed
- Define a partial order on assignments
 - $a < b \Leftrightarrow a$ must be executed before b
- Represented as a directed graph
 - Nodes are assignments
 - Edges represent dependency
- Acyclic for basic blocks

Running Example



Sources of dependency

- Data flow inside expressions
 - Operator depends on operands
 - Assignment depends on assigned expressions
- Data flow between statements
 - From assignments to their use
- Pointers complicate dependencies

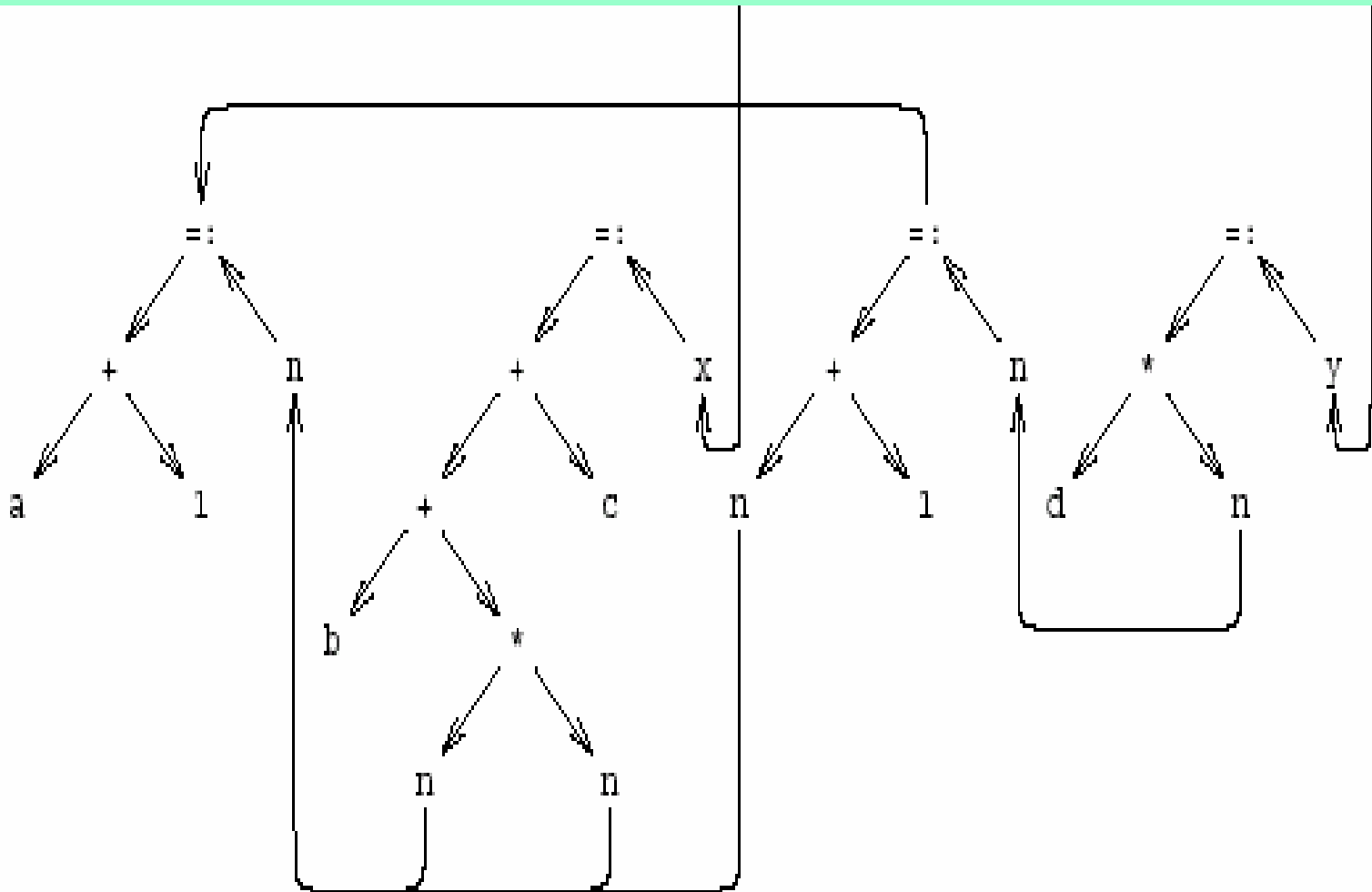
Sources of dependency

- Order of subexpression evaluation is immaterial
 - As long as inside dependencies are respected
- The order of uses of a variable are immaterial as long as:
 - Come between
 - Depending assignment
 - Next assignment

Creating Dependency Graph from AST

1. Nodes AST becomes nodes of the graph
2. Replaces arcs of AST by dependency arrows
 - Operator \rightarrow Operand
3. Create arcs from assignments to uses
4. Create arcs between assignments of the same variable
5. Select output variables (roots)
6. Remove ; nodes and their arrows

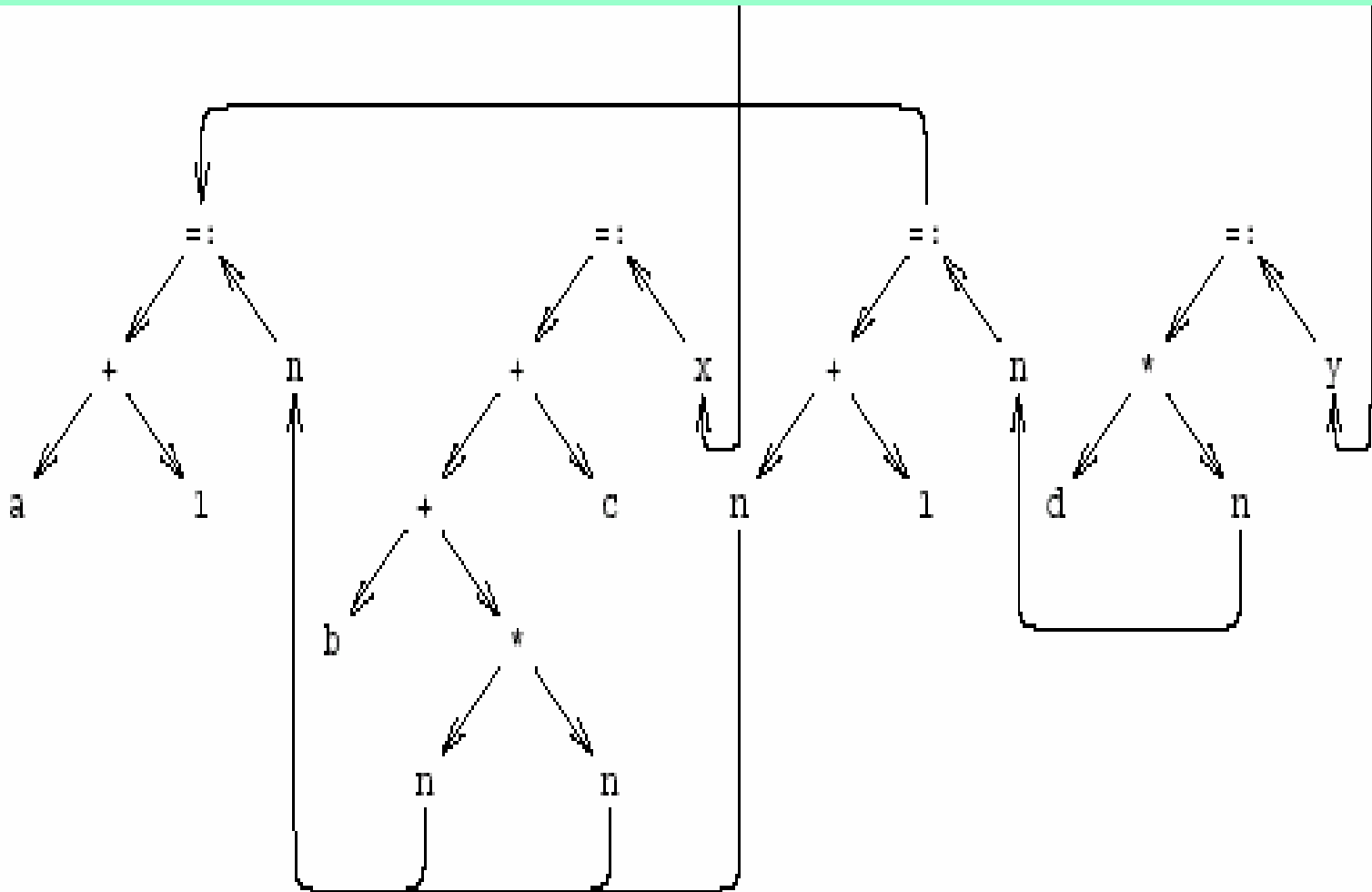
Running Example



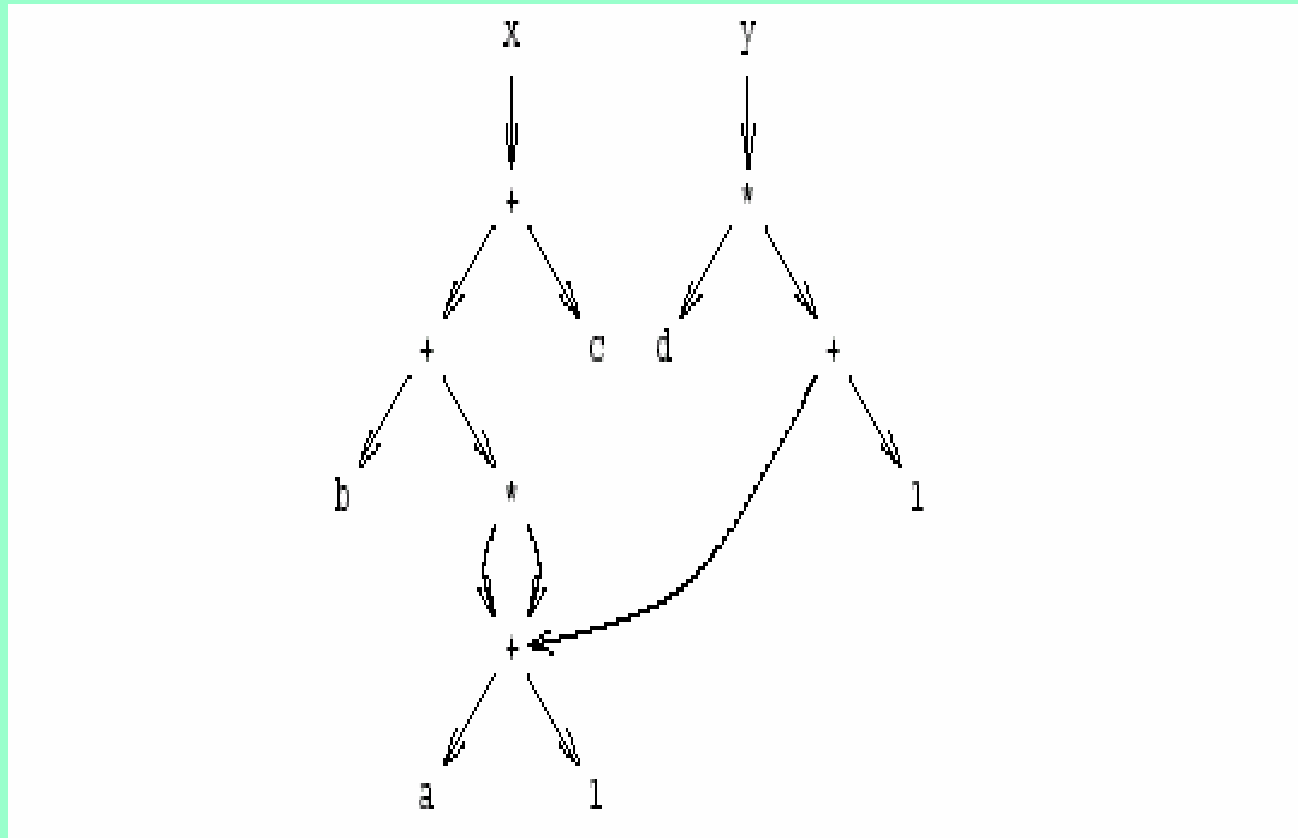
Dependency Graph Simplifications

- Short-circuit assignments
 - Connect variables to assigned expressions
 - Connect expression to uses
- Eliminate nodes not reachable from roots

Running Example



Cleaned-Up Data Dependency Graph



Common Subexpressions

- Repeated subexpressions

- Examples

$$x = a * a + 2 * a * b + b * b;$$

$$y = a * a - 2 * a * b + b * b ;$$

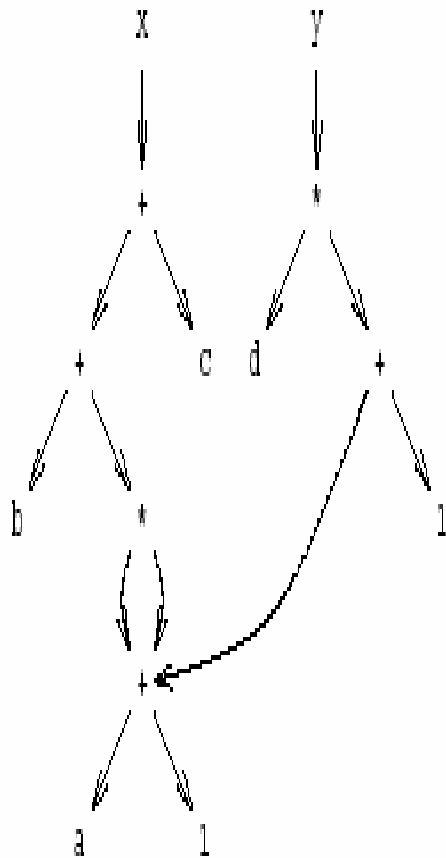
$$a[i] + b [i]$$

- Can be eliminated by the compiler
- In the case of basic blocks rewrite the DAG

From Dependency Graph into Code

- Linearize the dependency graph
 - Instructions must follow dependency
- Many solutions exist
- Select the one with small runtime cost
- Assume infinite number of registers
 - Symbolic registers
 - Assign registers later
 - May need additional spill
 - Possible Heuristics
 - Late evaluation
 - Ladders

Pseudo Register Target Code



```
Load_Mem   a, R1
Add_Const  1, R1
Load_Reg   R1, X1

Load_Reg   X1, R1
Mult_Reg   X1, R1
Add_Mem    b, R1
Add_Mem    c, R1
Store_Reg  R1, x

Load_Reg   X1, R1
Add_Const  1, R1
Mult_Mem   d, R1
Store_Reg  R1, y
```


Register Allocation

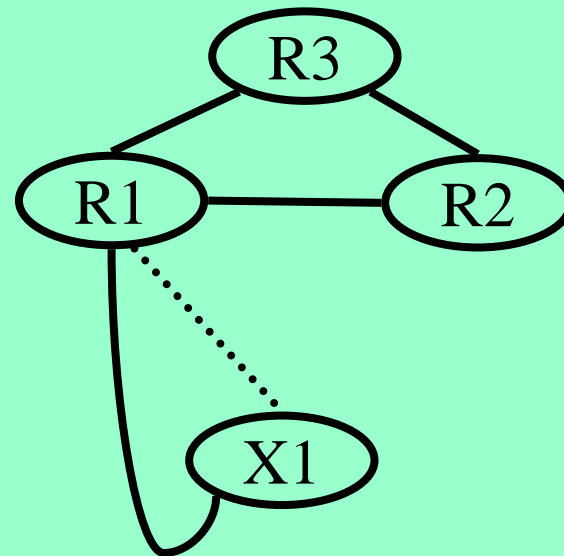
- Maps symbolic registers into physical registers
- Reuse registers as much as possible
- Graph coloring
 - Undirected graph
 - Nodes = Registers (Symbolic and real)
 - Edges = Interference
- May require spilling

Register Allocation (Example)

```
Load_Mem  a, R1
Add_Const  1, R1
Load_Reg   R1, X1

Load_Reg   X1, R1
Mult_Reg   X1, R1
Add_Mem    b, R1
Add_Mem    c, R1
Store_Reg  R1, x

Load_Reg   X1, R1
Add_Const  1, R1
Mult_Mem   d, R1
Store_Reg  R1, y
```



$X1 \rightarrow R2$

Running Example

```
Load_Mem      a, R1
Add_Const     1, R1
Load_Reg      R1, R2

Load_Reg      R2, R1
Mult_Reg      R2, R1
Add_Mem       b, R1
Add_Mem       c, R1
Store_Reg     R1, x

Load_Reg      R2, R1
Add_Const     1, R1
Mult_Mem      d, R1
Store_Reg     R1, y
```

Optimized code(gcc)

```
{  int n;  
  
    n = a + 1;  
    x = b + n*n + c;  
    n = n + 1;  
    y = d * n;  
}
```

```
Load_Mem    a,R1  
Add_Const   1,R1  
Load_Reg    R1,R2  
  
Mult_Reg    R1,R2  
Add_Mem     b,R2  
Add_Mem     c,R2  
Store_Reg   R2,x  
  
Add_Const   1,R1  
Mult_Mem    d,R1  
Store_Reg   R1,y
```

Summary

- Heuristics for code generation of basic blocks
- Works well in practice
- Fits modern machine architecture
- Can be extended to perform other tasks
 - Common subexpression elimination
- But basic blocks are small
- Can be generalized to a procedure

Problem	Technique	Quality
Expression trees, using register-register or memory-register instructions	Weighted trees; Figure 4.30	
with sufficient registers:		Optimal
with insufficient registers:		Optimal
Dependency graphs, using register-register or memory-register instructions	Ladder sequences; Section 4.2.5.2	Heuristic
Expression trees, using any instructions with cost function	Bottom-up tree rewriting; Section 4.2.6	
with sufficient registers:		Optimal
with insufficient registers:		Heuristic
Register allocation when all interferences are known	Graph coloring; Section 4.2.7	Heuristic