Chapter 7

Syntax-Directed Compilation (AST & Target Code)

The Choices of Program Execution

The choices are:

1) little preprocessing that generates the abstract syntax tree (AST) or the intermediate representation (IR) such as Java byte code, followed by execution of an interpreter, and

2) much preprocessing that generates target (assembly) code, followed by execution by CPU.

Practical Implementation Concerns

- OK, now you know YACC.
- An obvious choice – you can write code in the production actions to generate x86 code and the code takes into account of x86 architecture.
- Is it good?

Of course NOT

- Yes, some guys in earlier time may choose that
- but he/she should learn the painful experience when his/her boss told him to port the compiler to ARM, MIPS….
- It is common to add an additional layer of abstraction to increase maintainability, extensibility, and scalability. (A typical software engineering problem)
- This is when your straightforward/brute-force approach seldom works in practice.

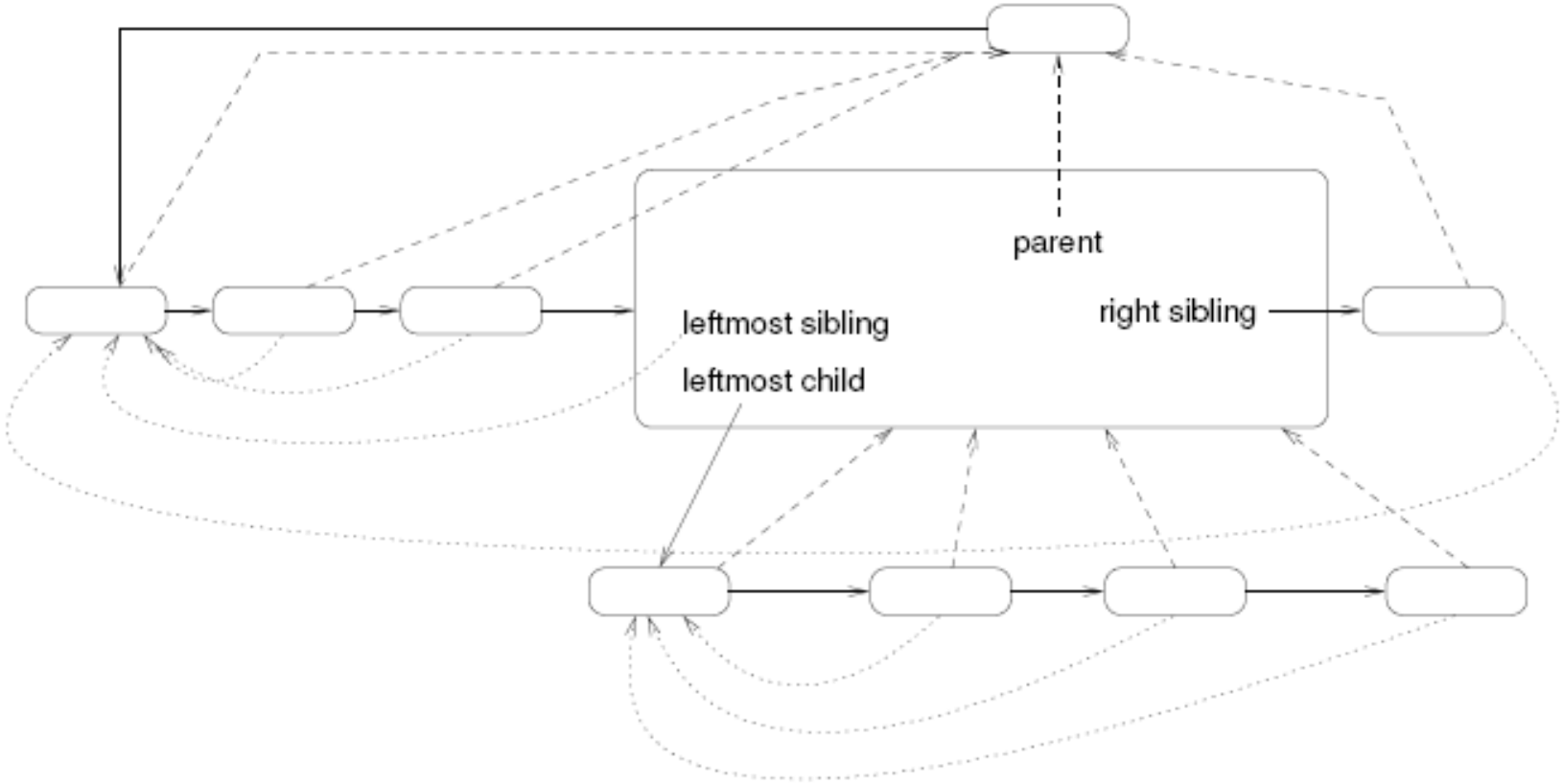


Figure 7.12: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

Interpretation

- The simplest way to have the actions expressed by the source program performed is to process the abstract syntax tree (AST) using an ***interpreter***.
- An interpreter considers the nodes of the AST in the correct order and performs the action prescribed for those nodes by the semantics of the language.

NOTE

- do not mix orange and apple, the interpreter is not the interpreter like java-script interpreter or basic interpreter. It is a program which traverse the AST and generate code So, CPUs with different instruction set must have different interpreter to traverse the AST)
- An interpreter is like a pre/in/post order traversal program in data structure.

Interpretation

- Interpreters come in two forms:

  recursive and iterative.

- A recursive interpreter works directly on AST and requires less preprocessing than an iterative interpreter, which works on threaded AST.

Recursive Interpretation

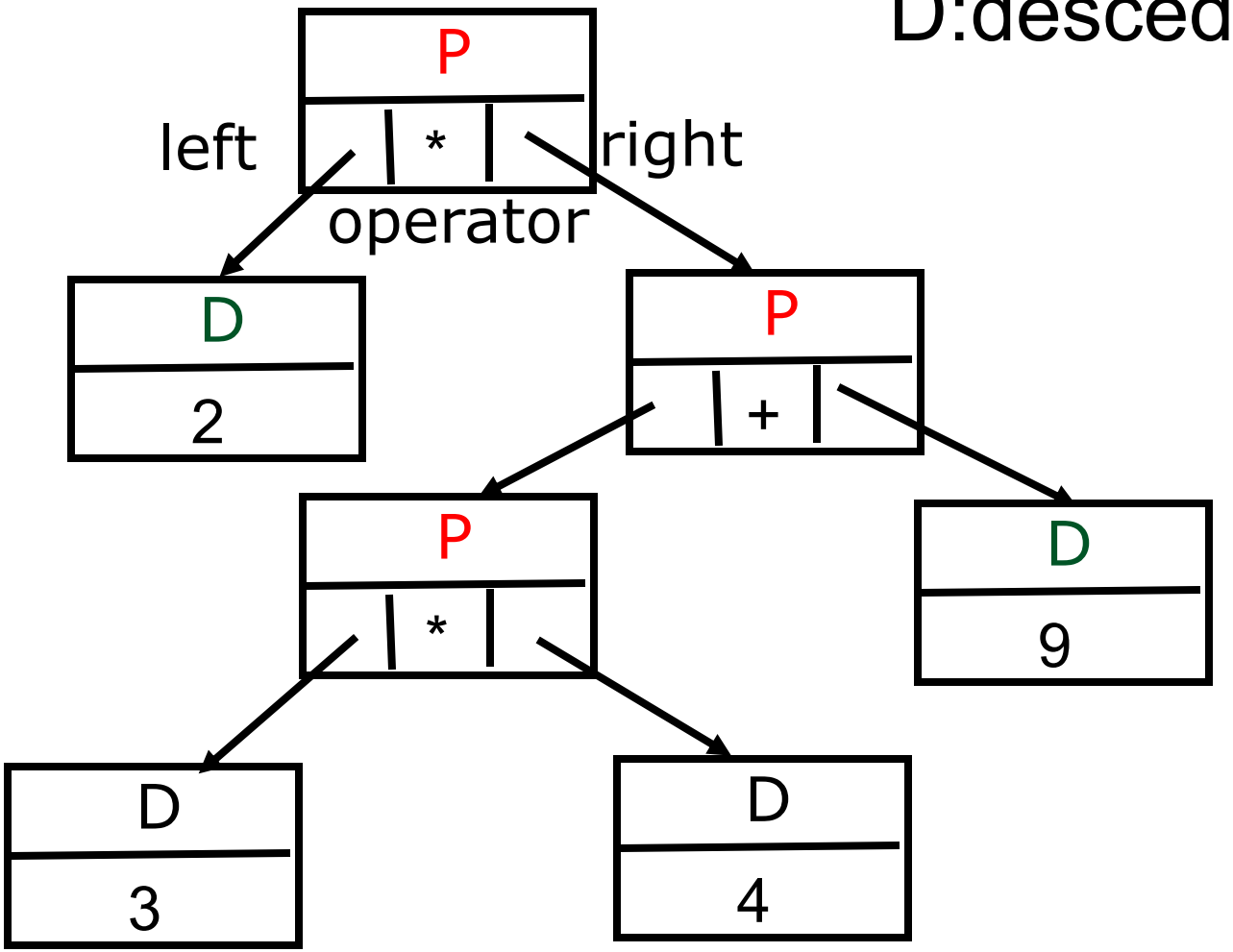
- A recursive interpreter has an interpreting routine for ***each node type*** in the AST.

- Such an interpreting routine calls other similar routines, depending on its children.

Let's give a simple example first

AST for (2*((3*4)+9)) result 42 P:parent

D:descedent



Recursive Interpretation (Cont.)

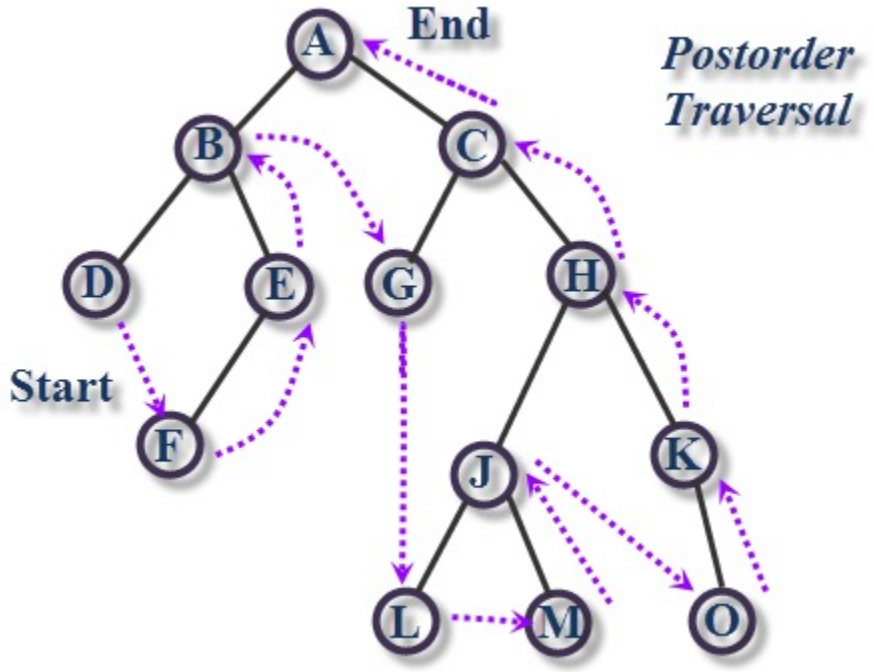
```
#include     "parser.h"        /* for types AST_node and Expression */
#include     "backend.h"       /* for self check */
                                        /* PRIVATE */
static int Interpret_expression(Expression *expr) {
    switch (expr->type) {
    case 'D':
        return expr->value;
        break;
    case 'P': {
        int e_left = Interpret_expression(expr->left);
        int e_right = Interpret_expression(expr->right);
        switch (expr->oper) {
        case '+': return e_left + e_right;
        case '*': return e_left * e_right;
        }}
        break;
    }
}
                                        /* PUBLIC */
void Process(AST_node *icode) {
    printf("%d\n", Interpret_expression(icode));
}
```

Iterative Interpretation

- The structure of an iterative interpreter looks much closer to that of a CPU than a recursive interpreter.
- It consists of ***a flat loop over a case statement*** which contains a code segment for each node type.

- It requires a threaded AST, and maintains an active-node pointer, which points to the node to be interpreted.

Threaded AST with Post-Order



Iterative Interpretation (Cont.)

- The iterative interpreter repeatedly runs the code segment for the node pointed at by the active-node pointer;

  this code sets the active-node pointer to another node, its successor, thus leading the interpreter to that node.

Iterative Interpretation (Cont.)

```
static AST_node *Active_node_pointer;

static void Interpret_iteratively(void) {
    while (Active_node_pointer != 0) {
        /* there is only one node type, Expression: */
        Expression *expr = Active_node_pointer;
        switch (expr->type) {
        case 'D':
            Push(expr->value);
            break;
        case 'P': {
            int e_left = Pop(); int e_right = Pop();
            switch (expr->oper) {
            case '+': Push(e_left + e_right); break;
            case '*': Push(e_left * e_right); break;
            }}
            break;
        }
        Active_node_pointer = Active_node_pointer->successor;
    }
    printf("%d\n", Pop());       /* print the result */
}
                                          /* PUBLIC */
void Process(AST_node *icode) {
    Thread_AST(icode); Active_node_pointer = Thread_start;
    Interpret_iteratively();
}
```

NOTE

- Iterative interpreter uses a stack
- What is that?
- Recursive use system stack to save the states for you. So, to write a program as loop you need to use stack.

Simple Code Generation

- In simple code generation, a fixed translation to the target code is chosen for each node type.

- During code generation, the nodes in the AST are replaced by their translations.

Simple Code Generation (Cont.)

- Simple code generation requires local decisions only.

- With respect to machine types, it is particularly suitable for two similar machines: 1) **pure stack machine** and
  2) **pure register machine**.

Simple Code Generation (Cont.)

- A **pure stack machine** uses a stack to store and manipulate values; it has no registers.
- It has two types of instructions:

1) Instructions that move or copy values between the top of the stack and else where, and

2) Instructions that do operations on the top element or elements of the stack.

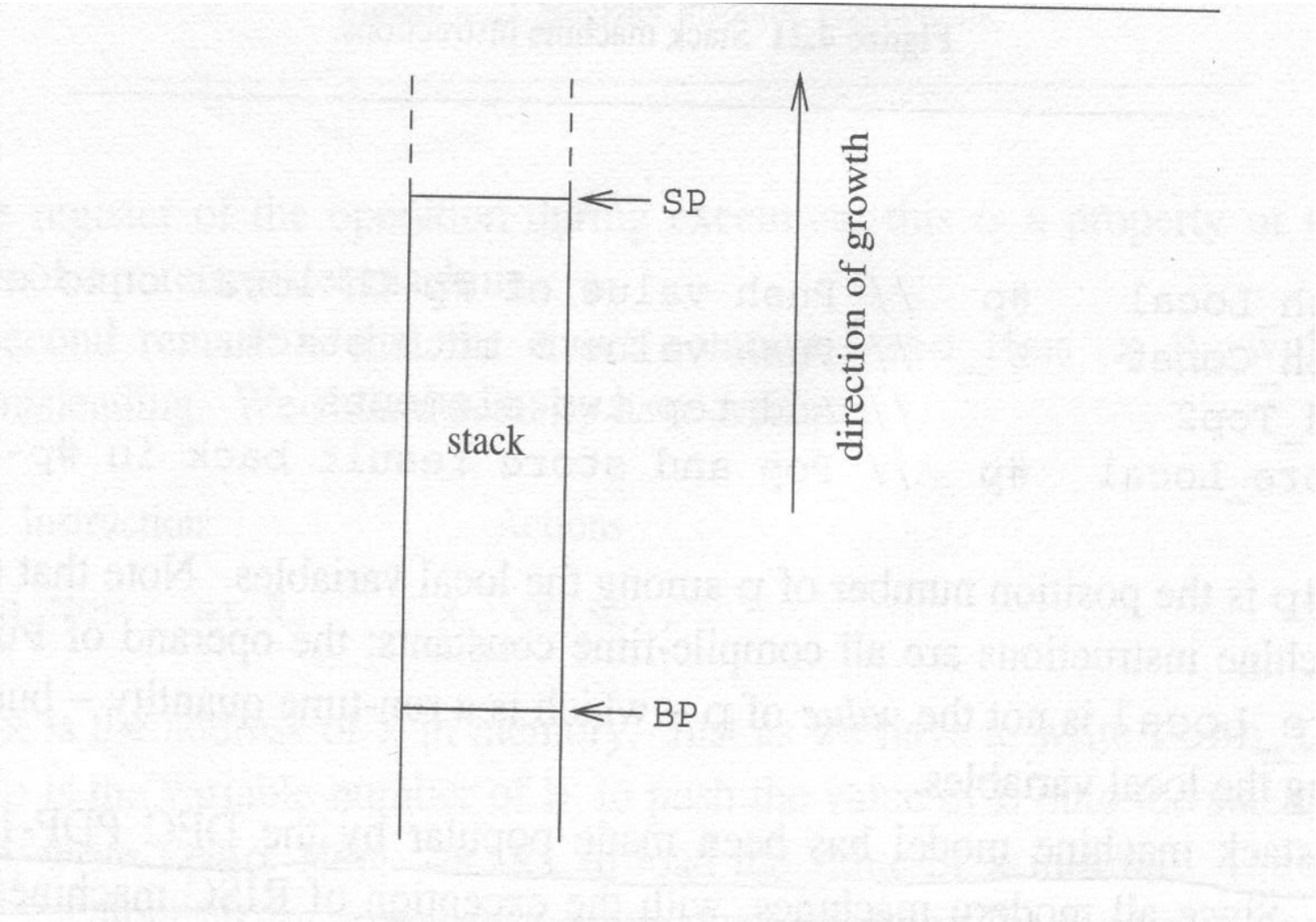
Simple Code Generation (Cont.)

The stack machine has two important data administration pointers:

1) the stack pointer (SP) pointing to the top of the stack.

2) the base pointer (BP) pointing to the beginning of the region on the stack where the local variables are stored.

Simple Code Generation (Cont.)



Simple Code Generation (Cont.)

We assume a very simple stack machine in which:

1) stack entries are of type integer
2) its instructions are summarized next.

Simple Code Generation (Cont.)

Instruction		Actions
Push_Const	*c*	SP:=SP+1; stack[SP]:=*c*;
Push_Local	*i*	SP:=SP+1; stack[SP]:=stack[BP+*i*];
Store_Local	*i*	stack[BP+*i*]:=stack[SP]; SP:=SP-1;
Add_Top2		stack[SP-1]:=stack[SP-1]+stack[SP]; SP:=SP-1;
Subtr_Top2		stack[SP-1]:=stack[SP-1]-stack[SP]; SP:=SP-1;
Mult_Top2		stack[SP-1]:=stack[SP-1]*stack[SP]; SP:=SP-1;

Why stack machine?

- Stack machine instruction are shorter
- Dense machine code was very valuable in the 1960s, when main memory was very expensive and very limited even on mainframes. It became important again on the initially-tiny memories of minicomputers and then microprocessors. Density remains important today, for smartphone applications and for Java applications downloaded into browsers over slow Internet connections.
- Compiler is simpler and code generation is much easier to build.

Practical Stack Machine?

- The **transputer** was a pioneering microprocessora rchitecture of the 1980s, featuring integrated memory andserial communication links, intended for parallel computing.


T414 transputer chip

Simple Code Generation (Cont.)

- Suppose p is a local variable; then we have

$$p := p + 5$$

the code is:

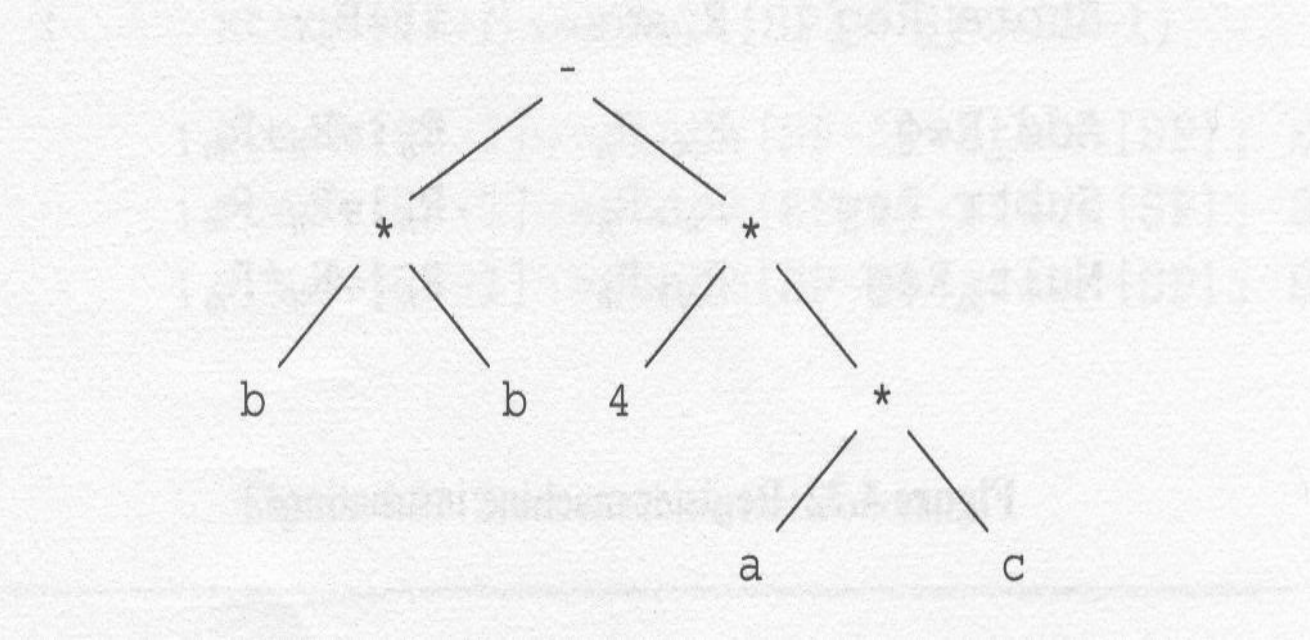
```
Push_Local    #p   // Push value of #p-th local onto stack.
Push_Const    5    // Push value 5 onto stack.
Add_Top2           // Add top two elements.
Store_Local   #p   // Pop and store result back in #p-th local.
```

Simple Code Generation (Cont.)

Another example:

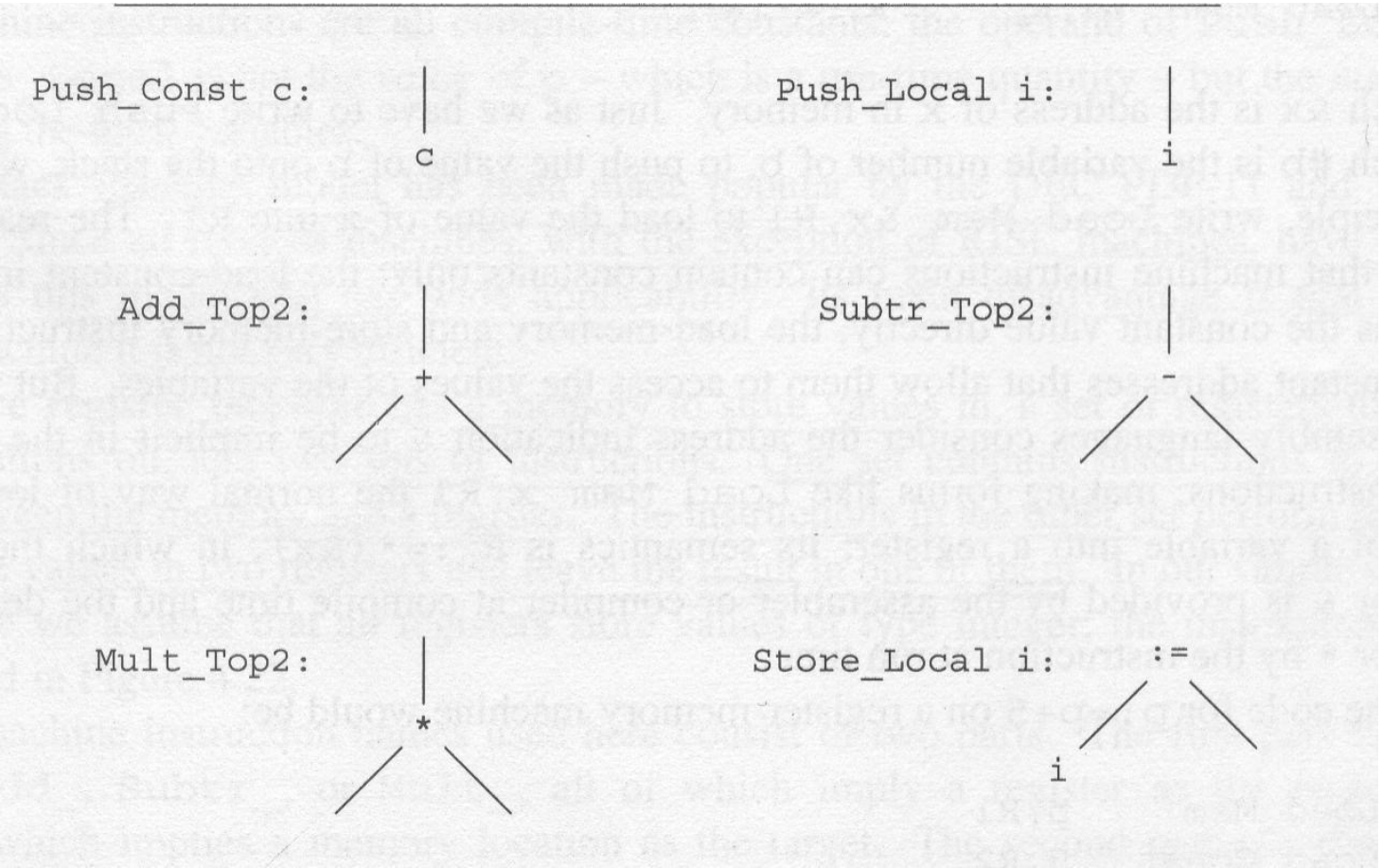
The expression b * b - 4 * (a * c)

its AST is:



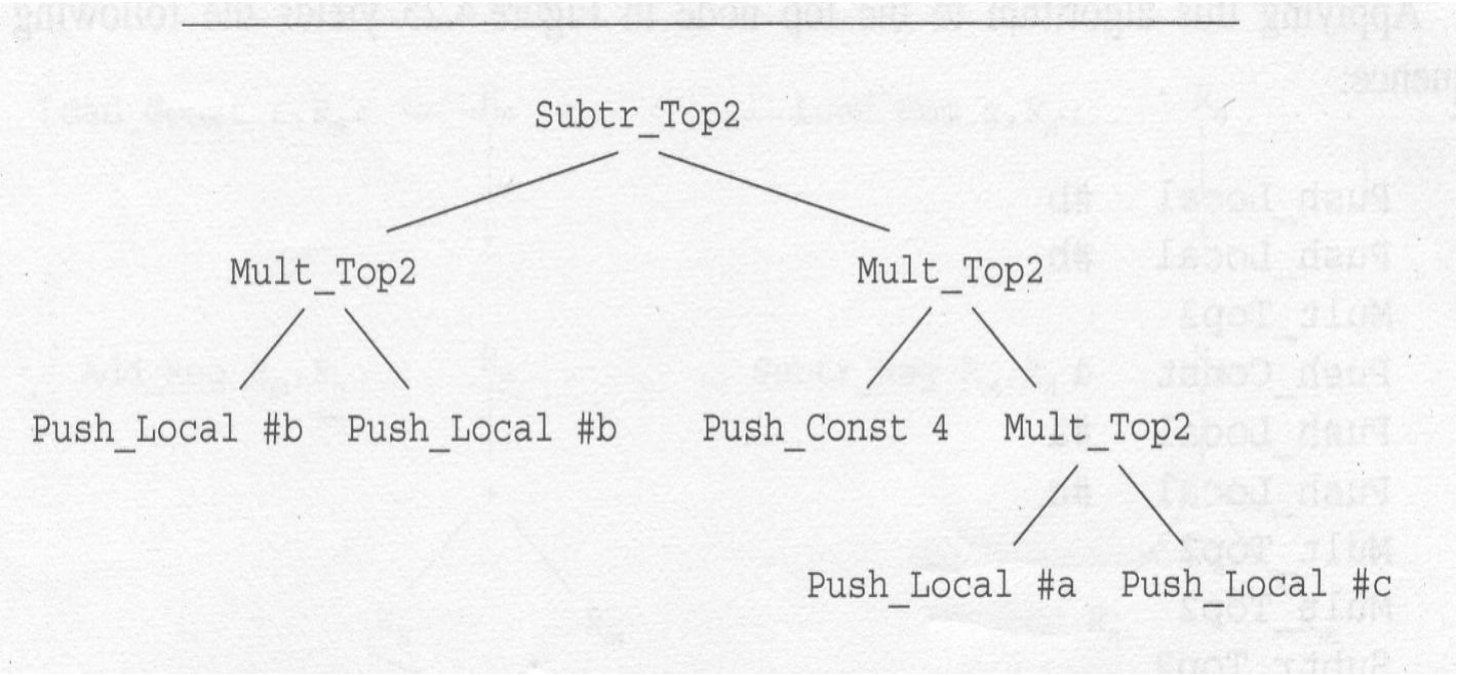
Simple Code Generation (Cont.)

Stack machine instructions for the AST:



Simple Code Generation (Cont.)

Instructions for b * b – 4 * (a * c)



Simple Code Generation (Cont.)

Depth-first code generation algorithm for a stack machine

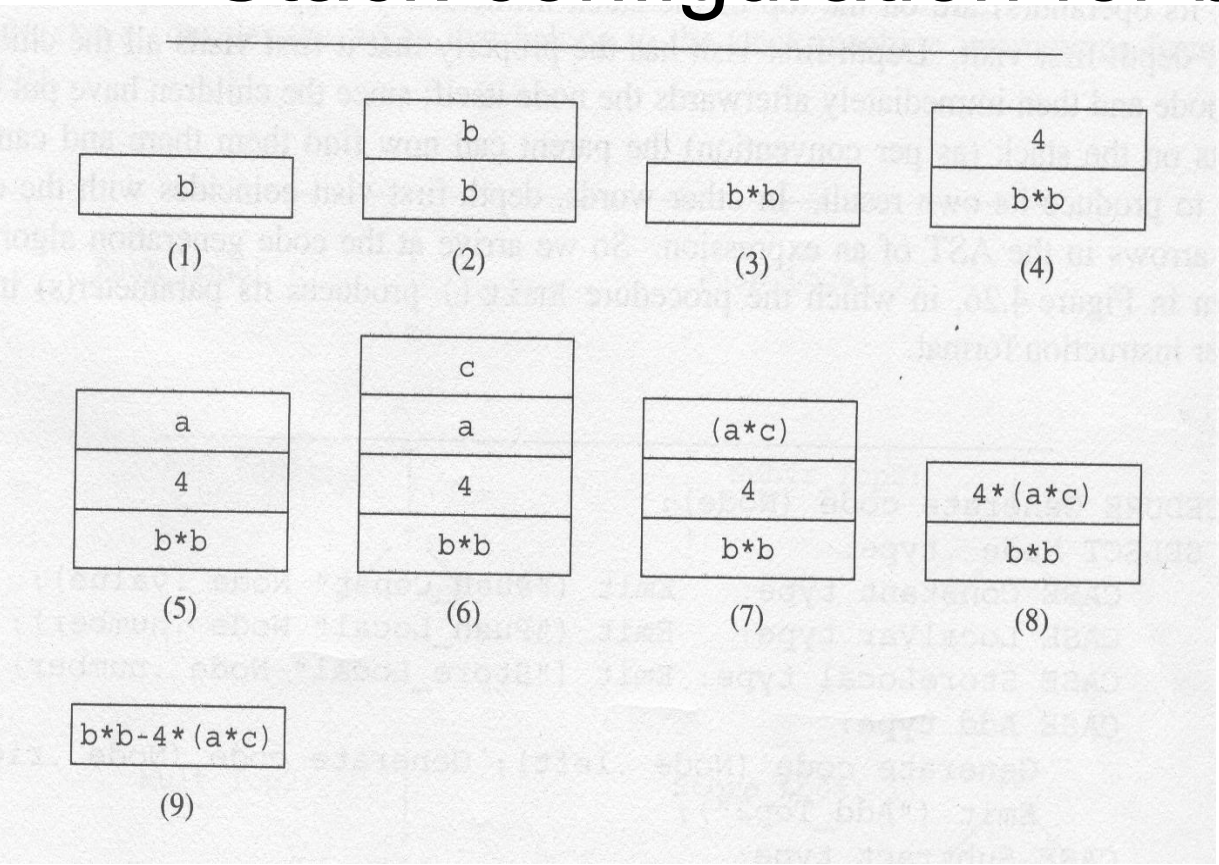
```
PROCEDURE Generate code (Node):
    SELECT Node .type:
        CASE Constant type:   Emit ("Push_Const" Node .value);
        CASE LocalVar type:   Emit ("Push_Local" Node .number);
        CASE StoreLocal type: Emit ("Store_Local" Node .number);
        CASE Add type:
            Generate code (Node .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");
```

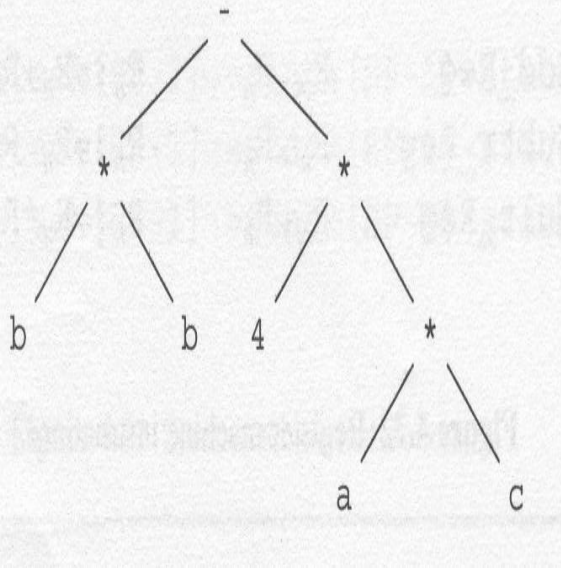
Simple Code Generation (Cont.)

- Applying this algorithm to the top node yields the following code sequence:

```
Push_Local  #b
Push_Local  #b
Mult_Top2
Push_Const  4
Push_Local  #a
Push_Local  #c
Mult_Top2
Mult_Top2
Subtr_Top2
```

Stack configuration for b*b-4*(a * c)






```
Push_Local  #b
Push_Local  #b
Mult_Top2
Push_Const  4
Push_Local  #a
Push_Local  #c
Mult_Top2
Mult_Top2
Subtr_Top2
```

NOTES

- OK, you may never encounter stack machine in your whole life.
- The points of the above are to tell you that if you have a stack machine (simpler instruction set), its architecture and AST's post-order traversal are really 麻吉
- They can sing a simplest code generation rhythm together

Simple Code Generation (Cont.)

- A **pure register machine** stores values in a set of registers to perform operations, and two sets of instructions below:

  1) Instructions that copy values between **the memory and a register**.

  2) Instructions that perform operations on the values in **two registers** and leave the result in one of them.

Simple Code Generation (Cont.)

Instruction		Actions
Load_Const	c, R_n	$R_n := fc;$
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	$R_n := R_n * R_m;$

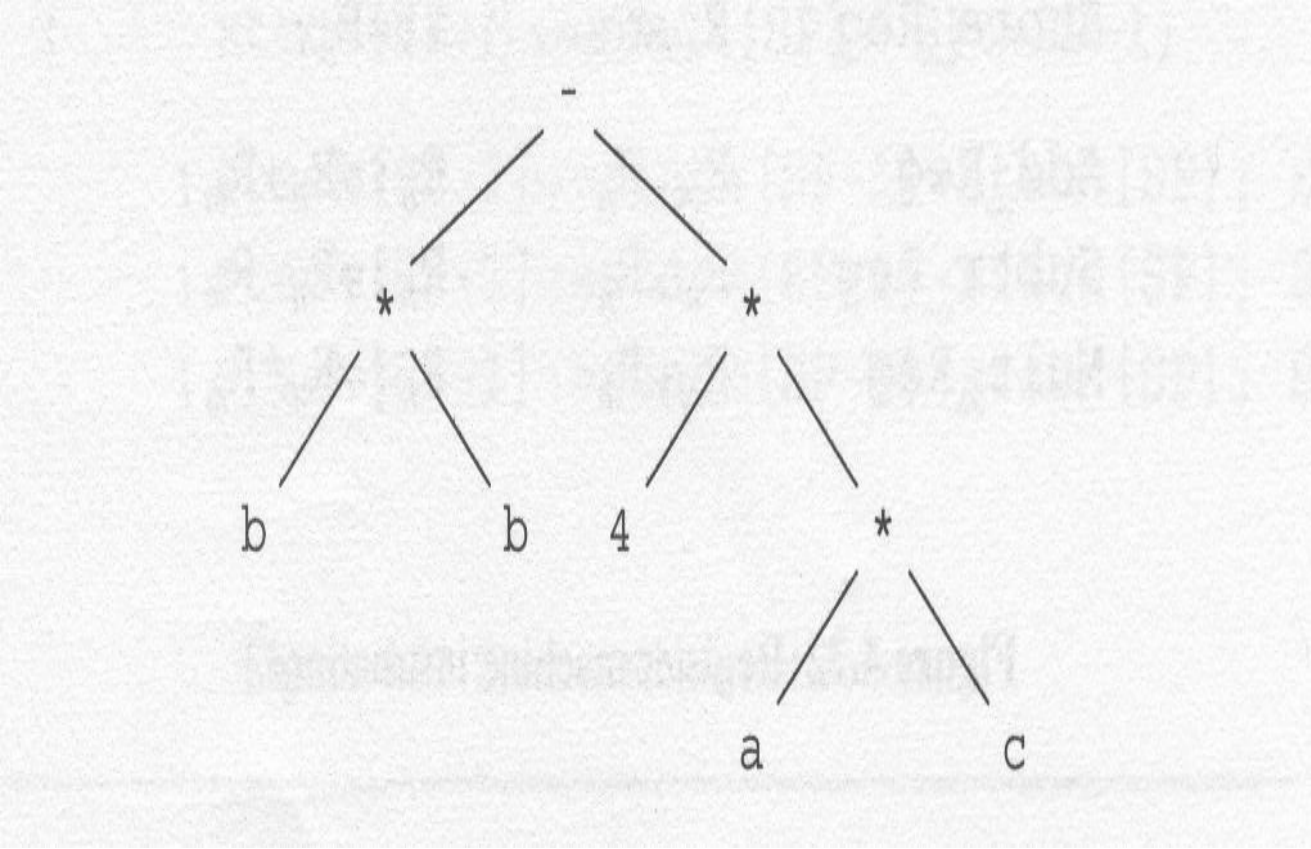
Simple Code Generation (Cont.)

- The code for p := p + 5 on a register machine would be:

```
Load_Mem    p, R1
Load_Const  5, R2
Add_Reg     R2, R1
Store_Reg   R1, p
```

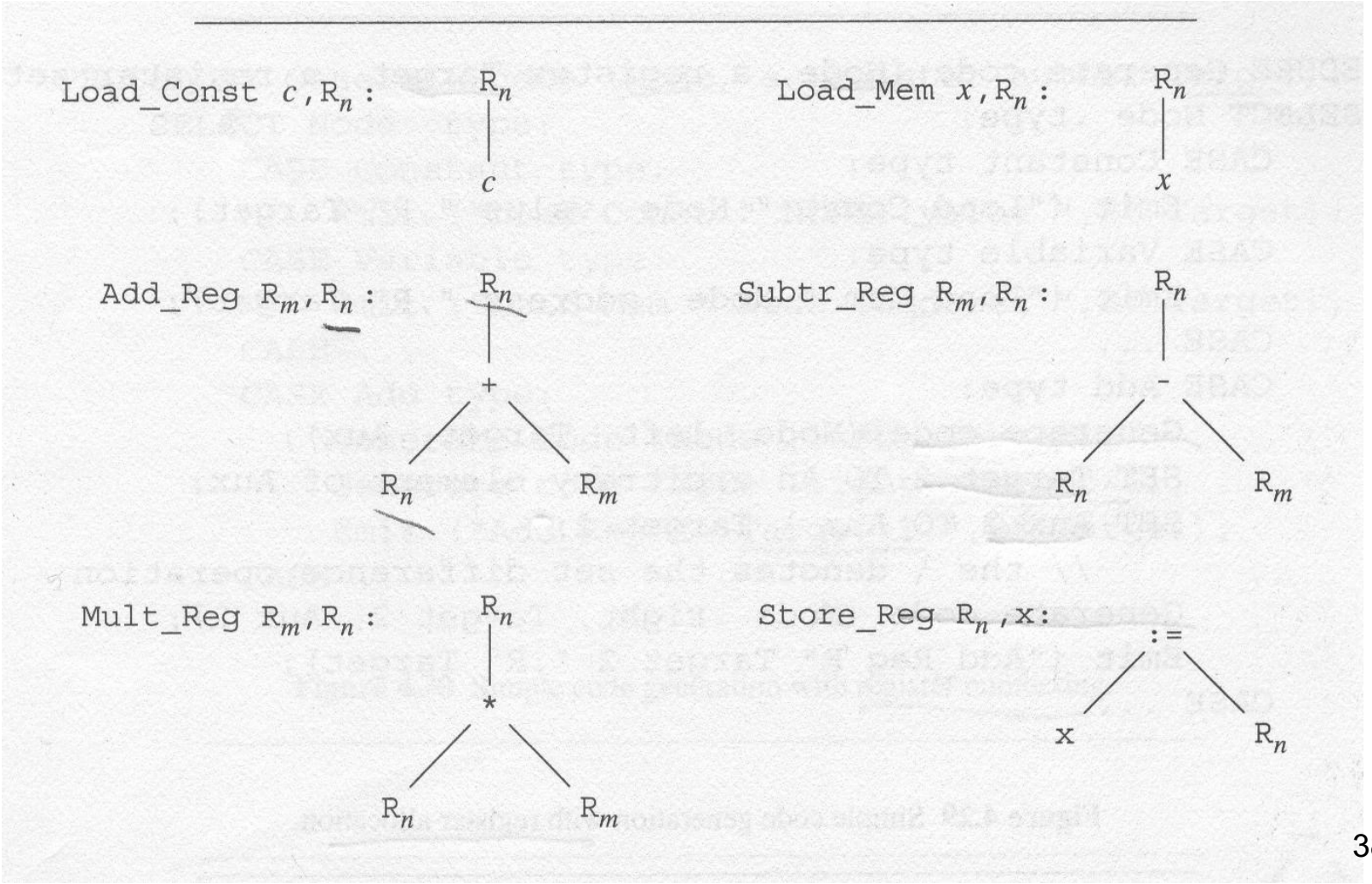
Simple Code Generation (Cont.)

- An example expression: b*b - 4*(a*c)



Simple Code Generation (Cont.)

Register machine instructions for AST



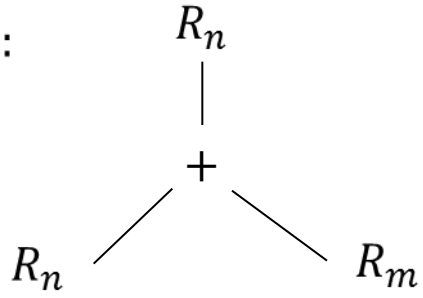
Simple Code Generation (Cont.)

ARM instructions for AST:

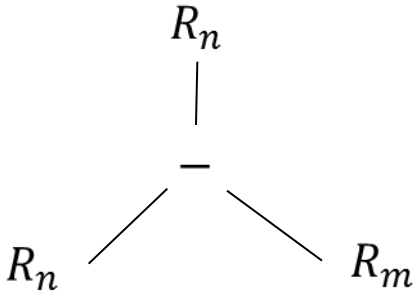
MOV R_n , c :

LDR R_n , [x] :

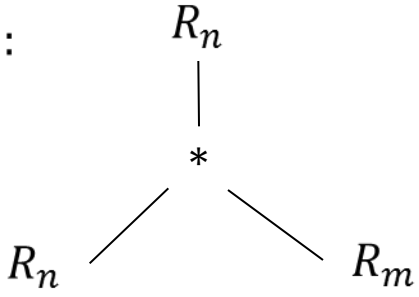
ADD R_n , R_n , R_m :



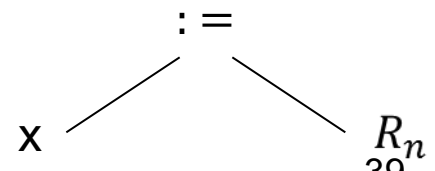
SUB R_n , R_n , R_m :



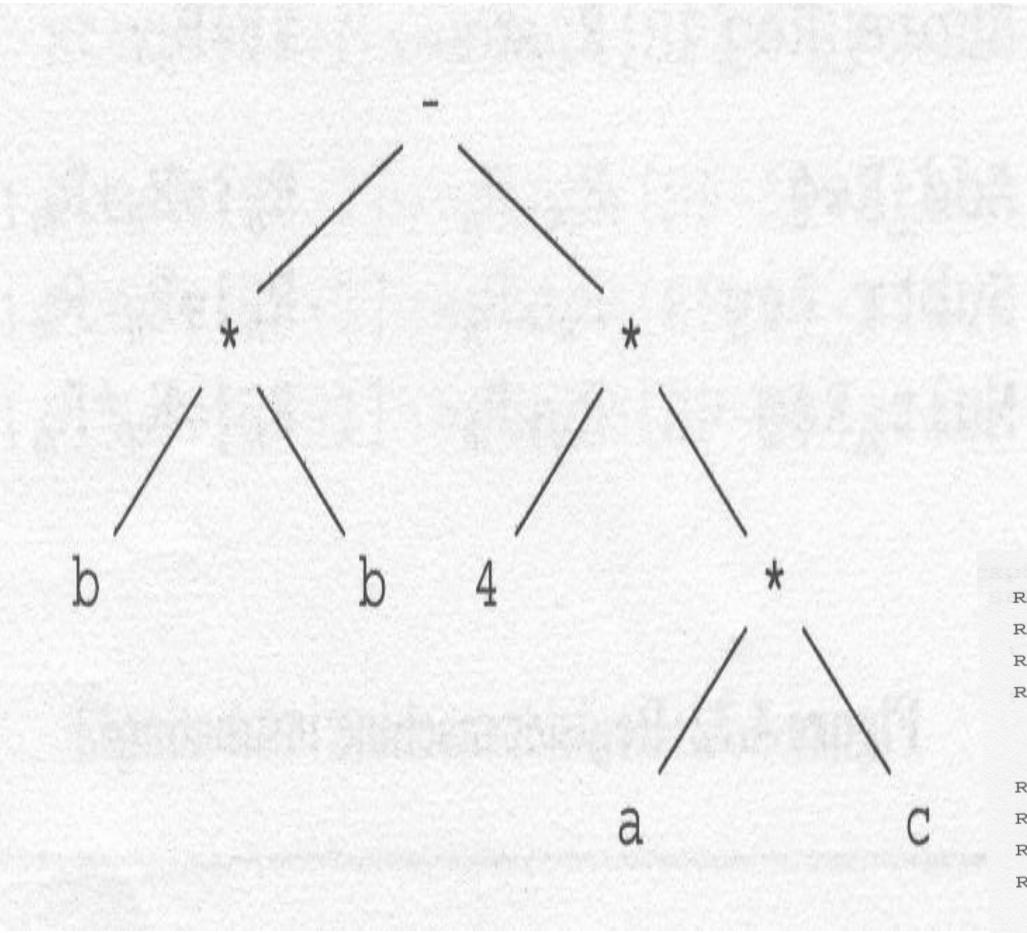
MUL R_n , R_n , R_m :

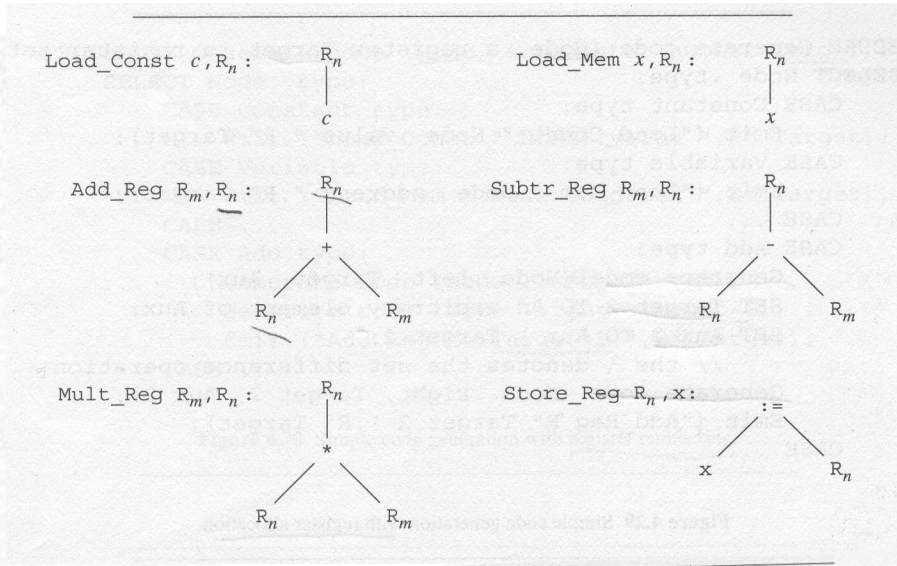


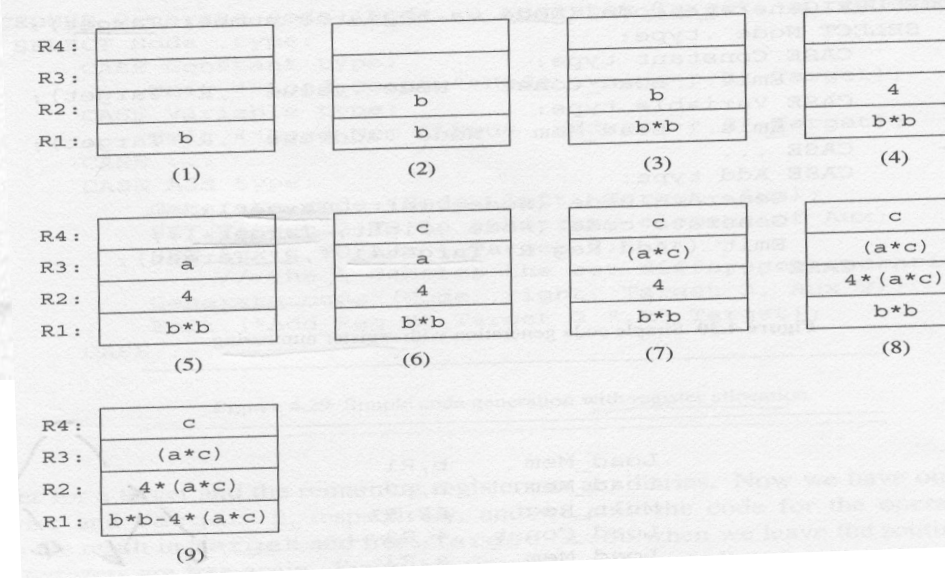
STR R_n , [x] :



Can we do it once by hand?







Simple Code Generation (Cont.)

We use depth-first code generation again, but this time we use registers.

The result of the expression is expected in a **given register** (**the target register)**, and a given set of **auxiliary registers** is available to help get there.

Simple Code Generation (Cont.)

Target is a register number

```
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 ...
```

Simple Code Generation (Cont.)

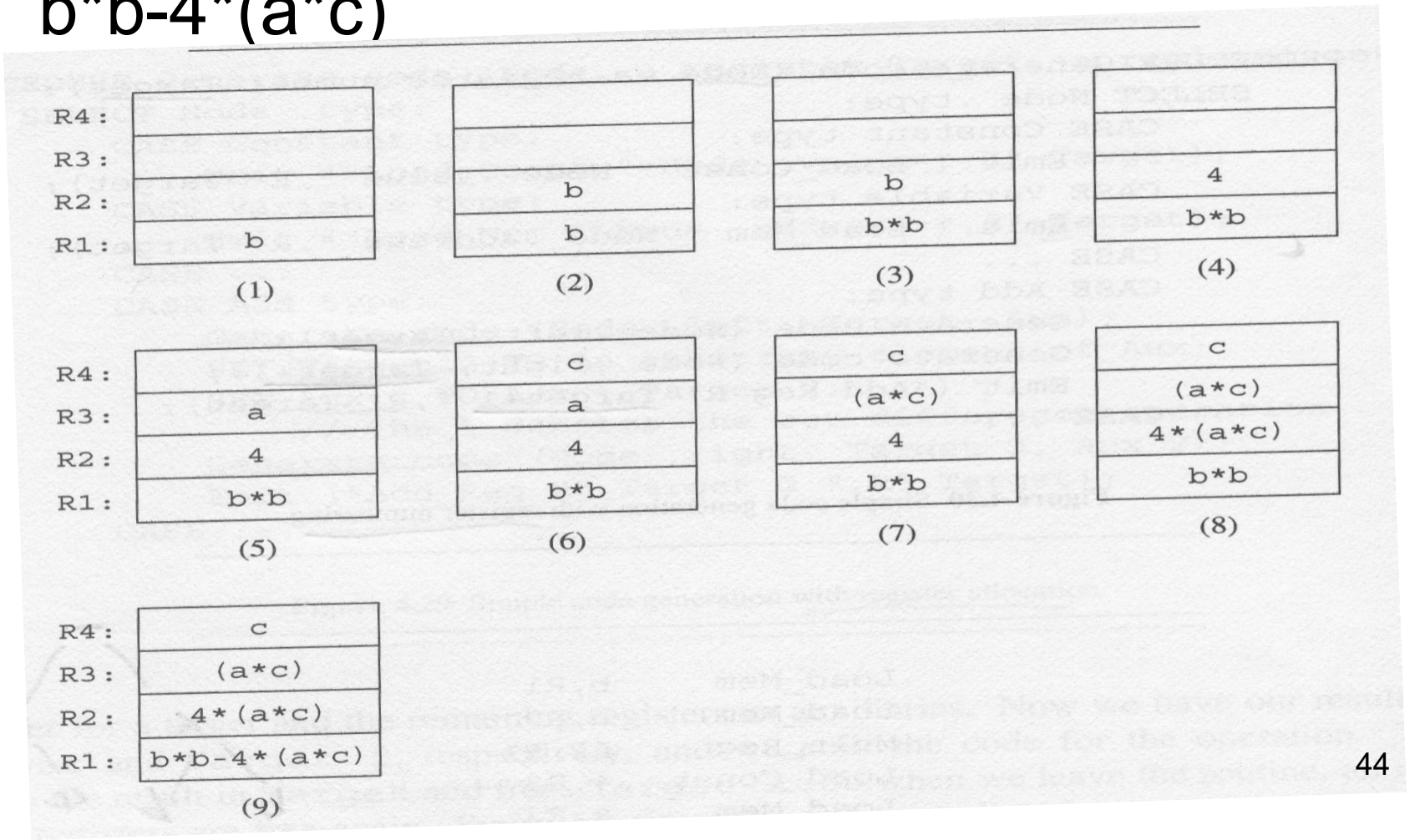
- Register machine code for the expression b*b-4*(a*c)

```
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
```

Simple Code Generation (Cont.)

- Successive register contents for:

  b*b-4*(a*c)



Simple Code Generation- weighted register allocation

- It is disappointing to see that:

  4 registers are required for the expression where **3 would do**.

- The reason is that **one register gets tied up holding the value 4** while the sub-tree a*c is being computed.

- If we had treated the right sub-tree (a*c) first, rather than 4, **3 registers would have sufficed.**

Simple Code Generation- weighted register allocation (Cont.)

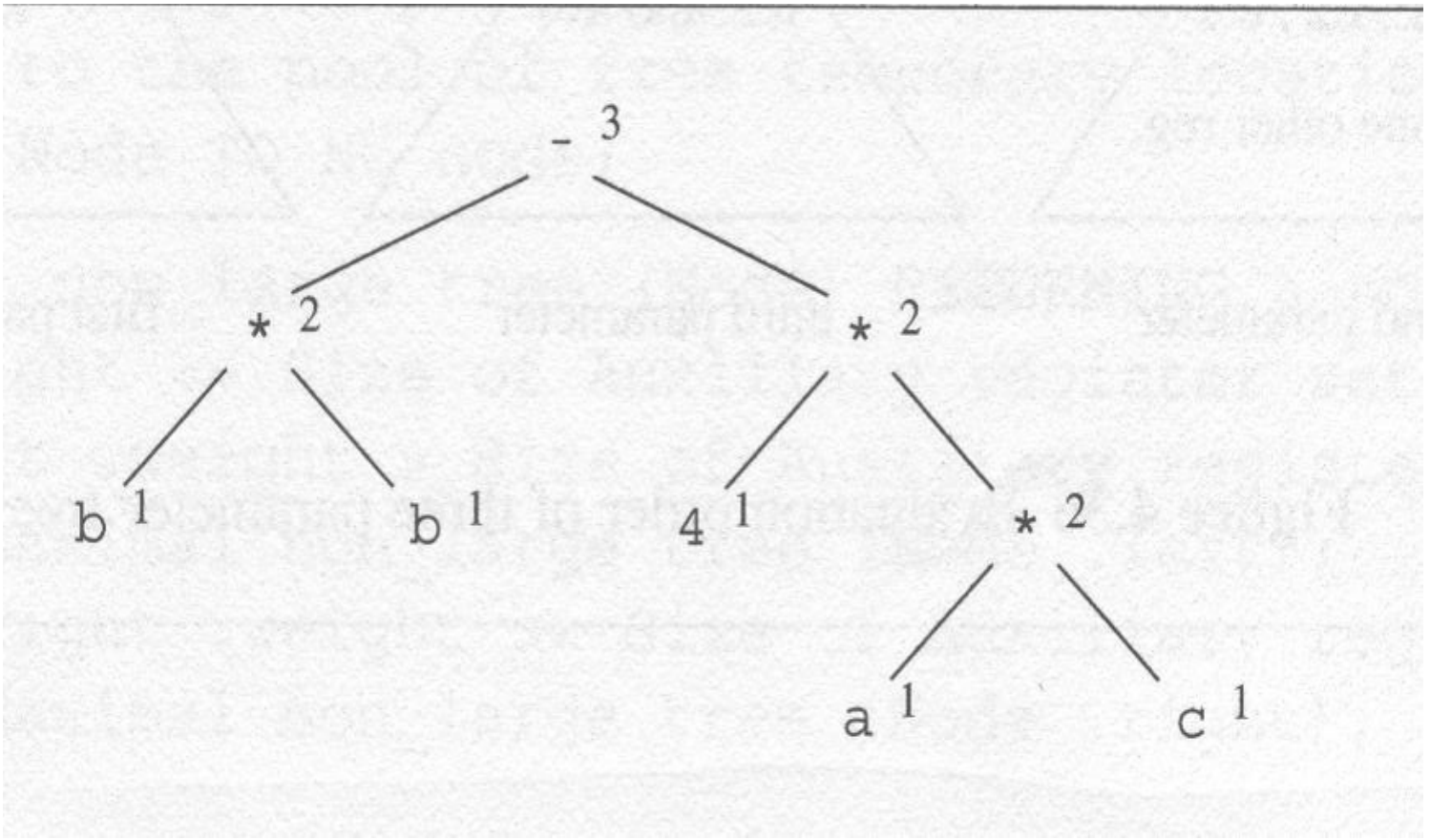
- We will call the number of registers required by a node its **weight**.

- Since the weight of each leaf is known and the weight of a node can be computed from the weights of its children,

  the weight of a sub-tree can be determined simply by a pre-scan.

Simple Code Generation- weighted register allocation (Cont.)

- An example expression b*b-4*(a*c)

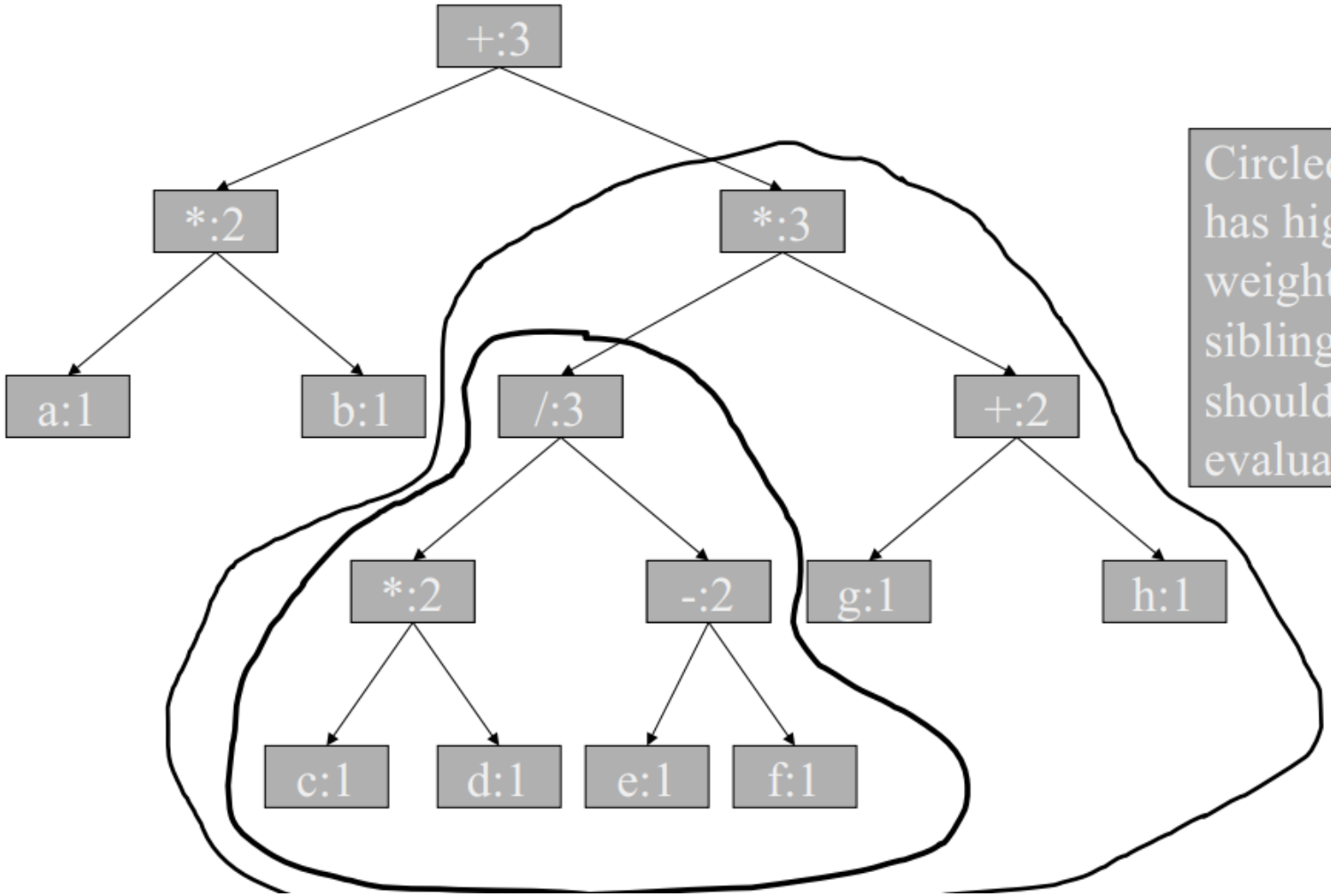


Simple Code Generation- weighted register allocation (Cont.)

```
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 ...
```

Example:

Number shows the weight calculated for each subtree



Circled subtree has higher weight than its sibling, so should be evaluated first.

Code Generation

- Compilation produces target code from the AST through a process called:

  Code Generation.

- The basis of it is the systematic replacement of nodes and sub-trees of the AST by target code segments.
- The replacement process is called:

  Tree Rewriting.

Code Generation (Cont.)

- As a demonstration, suppose we have constructed the AST for the expression

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

  in which a, c, and d are integer variables and b is a byte array in memory.

Code Generation (Cont.)

The compiler has decided that:

1) the variables a, c, and d are in the registers Ra, Rc, and Rd, and that

2) the array indexing operator [for byte arrays has been expanded into:

an addition and

a memory access “mem”.

Code Generation (Cont.)

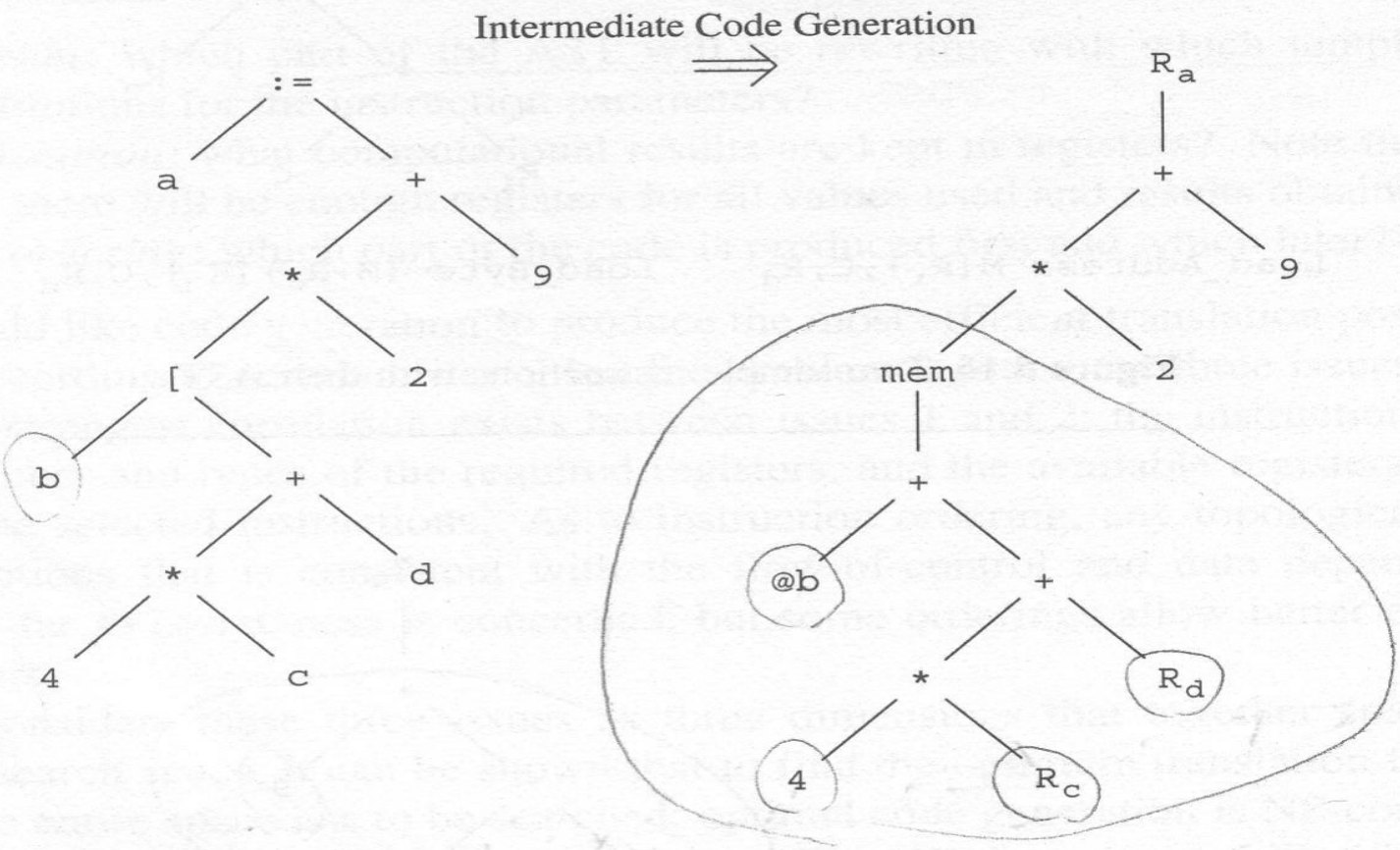


Figure 4.9 Two ASTs for the expression `a := (b[4*c + d] * 2) + 9.`

Code Generation (Cont.)

Let's get more help from the machine instruction sets. We have two machine instructions:

- **Load_Addr M[Ri], C, Rd**

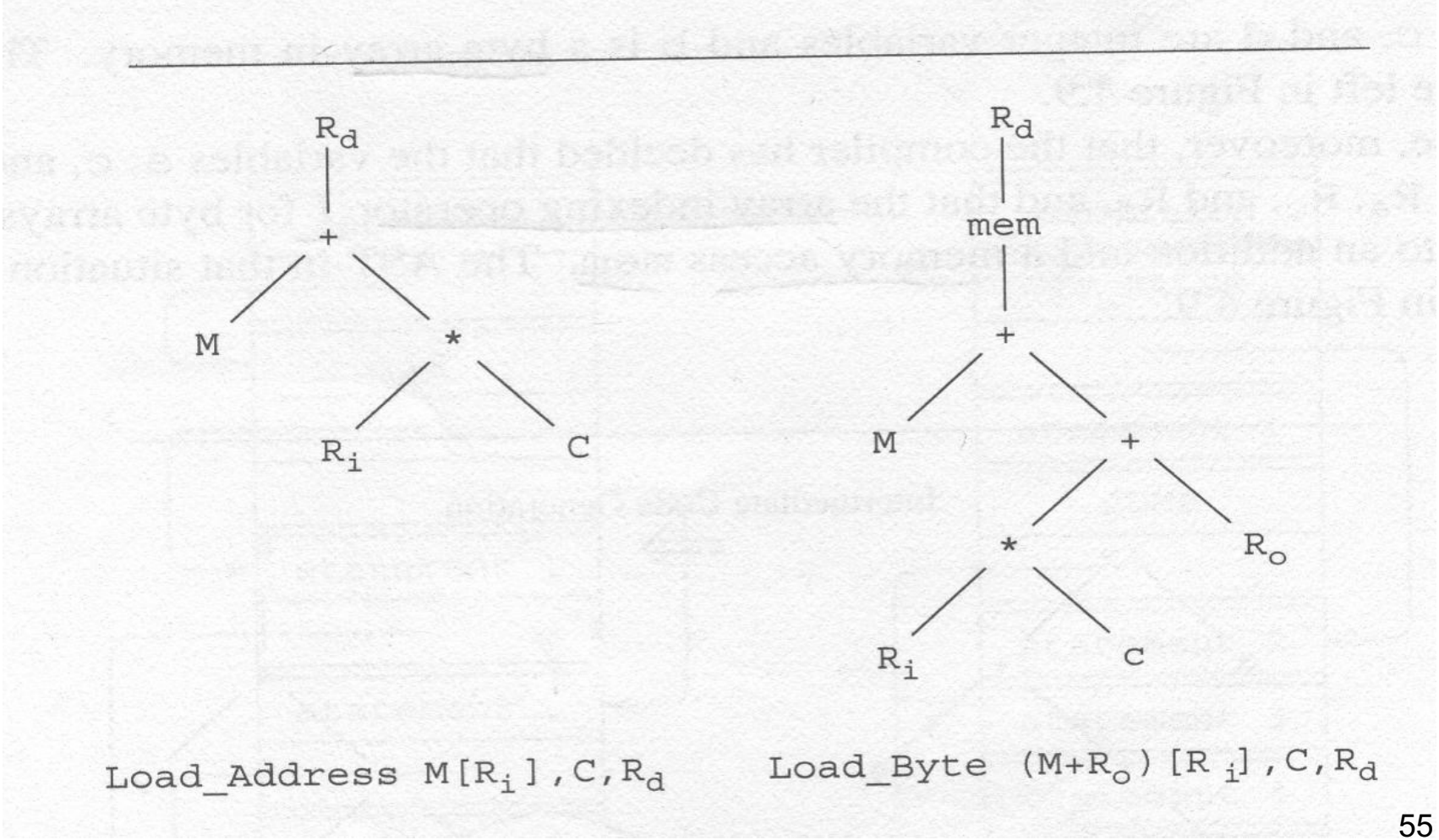
  loads the **address** of the Ri-th element of the array at M into Rd,

  where the size of element of M is C bytes.

- **Load_Byte (M+Ro) [Ri], C, Rd**

  loads the **byte content** of the Ri-th element of the array at M plus offset Ro into Rd.

Code Generation (Cont.)



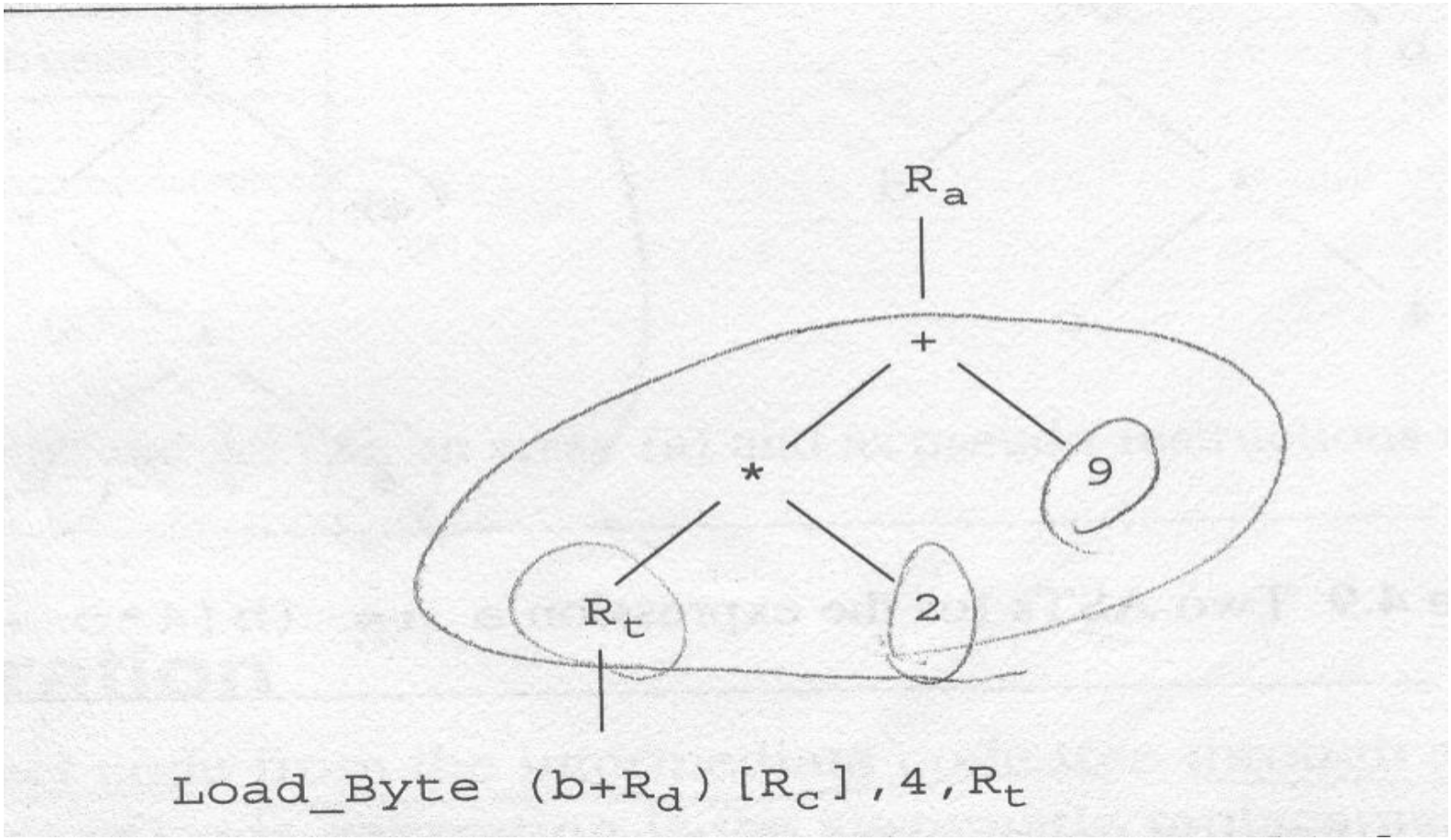
Code Generation (Cont.)

We first replace the bottom right of AST by:

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

by equating M with b, Ro with Rd,
Ri with Rc, C with 4, and
using a temporary register Rt
as the result register.

Code Generation (Cont.)



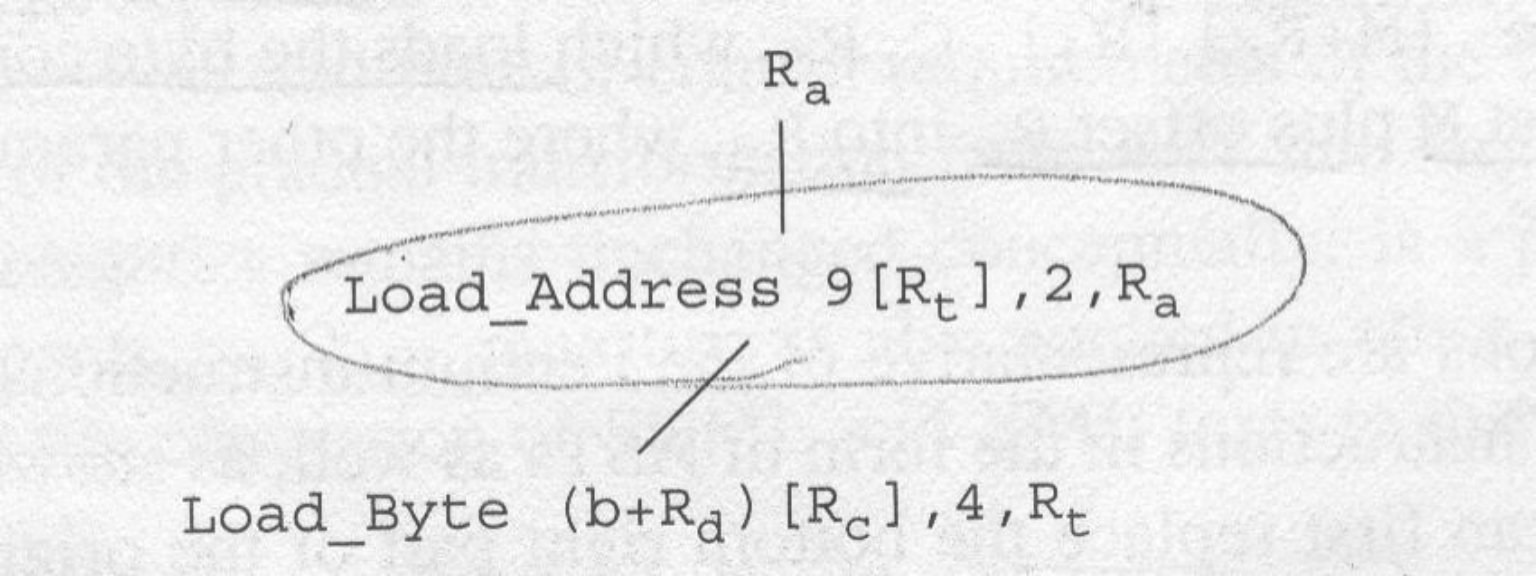
Code Generation (Cont.)

Next we replace the top by:

Load_Addr 9[Rt], 2, Ra

by equating M with 9, Ri with Rt, C with 2,
and using the register Ra which holds “a”
as the result register.

Code Generation (Cont.)



Code Generation (Cont.)

- Note that the fixed address of the array in the “Load_Addr” instruction is specified explicitly as 9.
- This yields the target code:

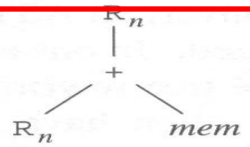
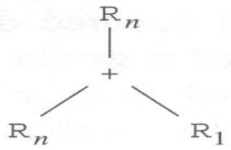
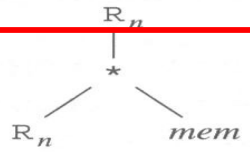
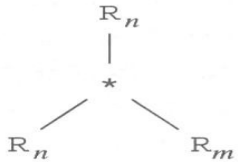
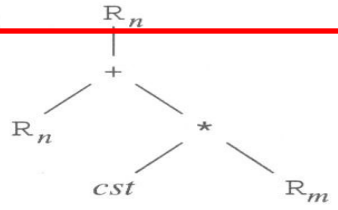
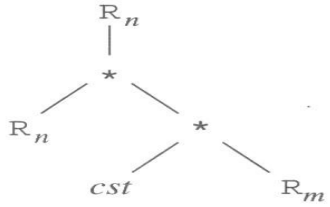
Load_Byte (b+rd) [Rc], 4, Rt

Load_Addr 9[Rt], 2, Ra

Bottom-up rewriting system (BURS) code generation

- The figure next shows a small set of instructions, which is representative of modern machines, large enough to show the principles involved and small enough to make explanation.

- For each instruction we show:
  1) an identifying number
  2) the AST it represents,
  3) the instruction name,
  4) its semantics, and
  5) its cost of execution.

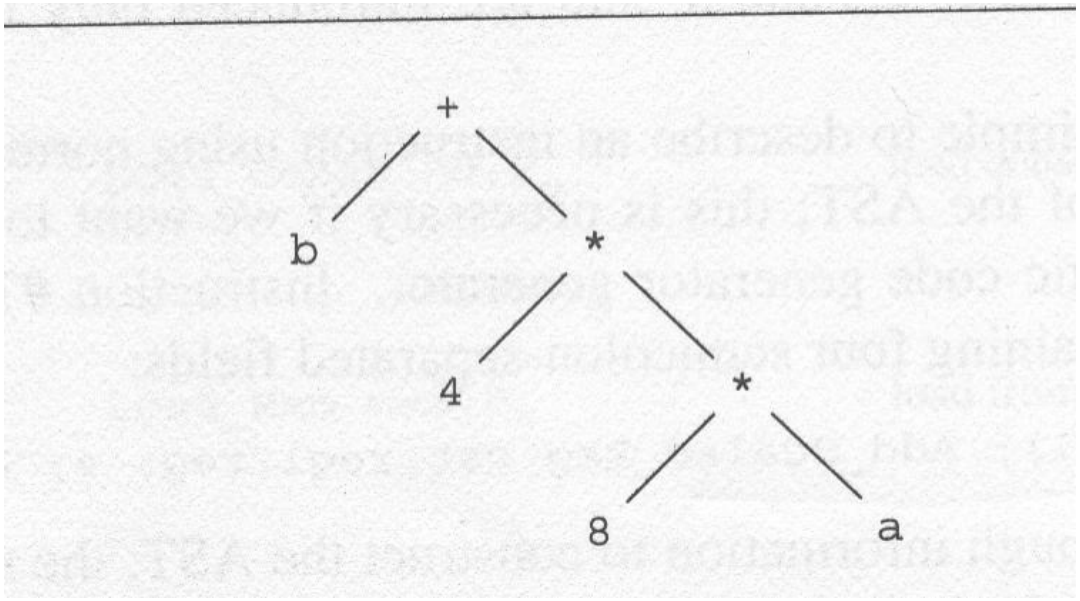
#1		`Load_Const` *cst*, R_n	load constant	cost = 1
#2		`Load_Mem` *mem*, R_n	load from memory	cost = 3
#3		`Add_Mem` *mem*, R_n	add from memory	cost = 3
#4		`Add_Reg` R_1, R_n	add registers	cost = 1
#5		`Mult_Mem` *mem*, R_n	multiply from memory	cost = 6
#6		`Mult_Reg` R_m, R_n	multiply registers	cost = 4
#7		`Add_Scaled_Reg` *cst*, R_m, R_n	add scaled register	cost = 4
#8		`Mult_Scaled_Reg` *cst*, R_m, R_n	multiply scaled register	cost = 5

NOTES

- It is common to provide additional instruction sets for similar purposes.
- Since we have alternatives to choose from, it creates a problem of optimization.

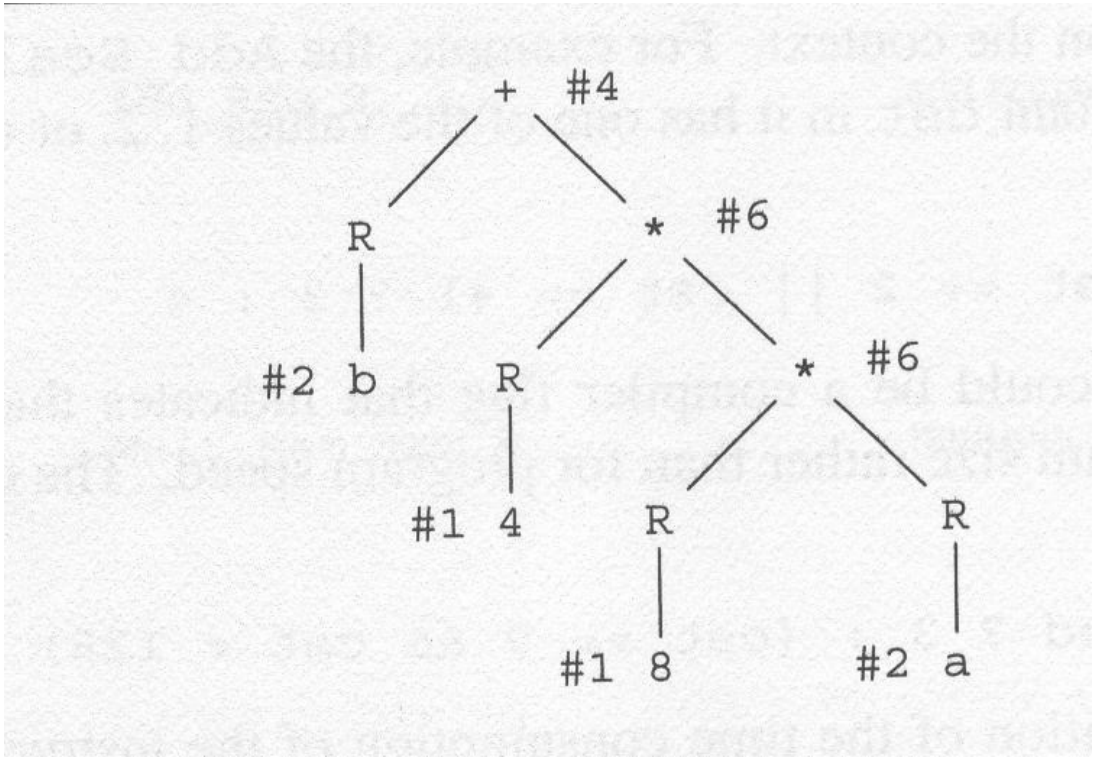
Bottom-up rewriting system (BURS) code generation (Cont.)

- The AST for which we are going to generate code is given below.



Bottom-up rewriting system (BURS) code generation (Cont.)

- Naïve bottom-up rewrite of the AST:



Bottom-up rewriting system (BURS) code generation (Cont.)

- Code generated from the naïve rewrite:

```
Load_Const   8,R1     ; 1 unit
Load_Mem     a,R2     ; 3 units
Mult_Reg     R2,R1    ; 4 units
Load_Const   4,R2     ; 1 unit
Mult_Reg     R1,R2    ; 4 units
Load_Mem     b,R1     ; 3 units
Add_Reg      R2,R1    ; 1 unit
             Total    = 17 units
```

Bottom-up rewriting system (BURS) code generation (Cont.)

- The figure below illustrates another rewrite possibility.

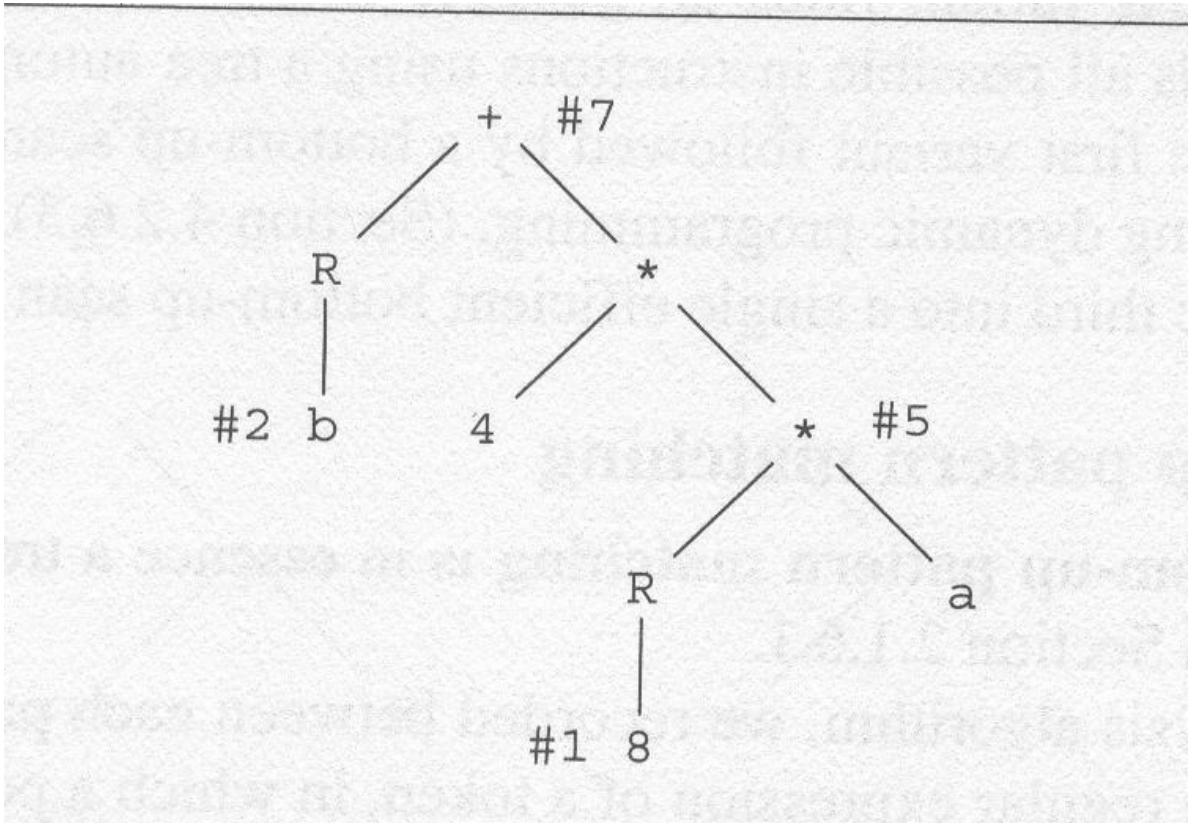
- This one applies a **top-down largest-fit algorithm**: starting from the top, **the largest instruction that would fit the operators in the tree was chosen**, and the operands were made to conform to that instruction.

  This rewrite is better than the naïve one:

  It uses 4 instructions, and its cost is 14 units.

Bottom-up rewriting system (BURS) code generation (Cont.)

Top-down largest-fit rewrite (#7 chosen first)



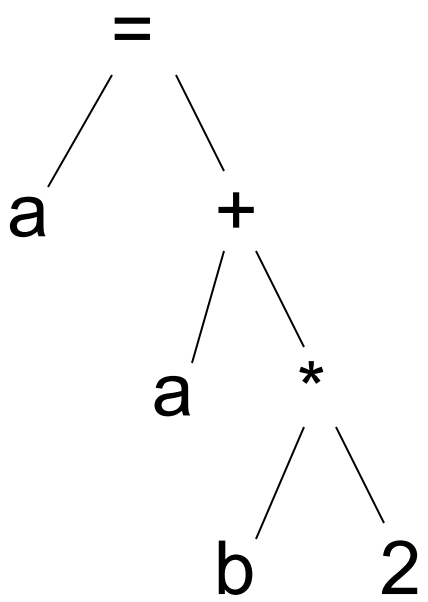
Bottom-up rewriting system (BURS) code generation (Cont.)

- Code generated from the top-down largest-fit rewrite:

```
Load_Const        8,R1        ; 1 unit
Mult_Mem          a,R1        ; 6 units
Load_Mem          b,R2        ; 3 units
Add_Scaled_Reg    4,R1,R2     ; 4 units
                  Total       = 14 units
```

Register machine code vs. weighted register allocation

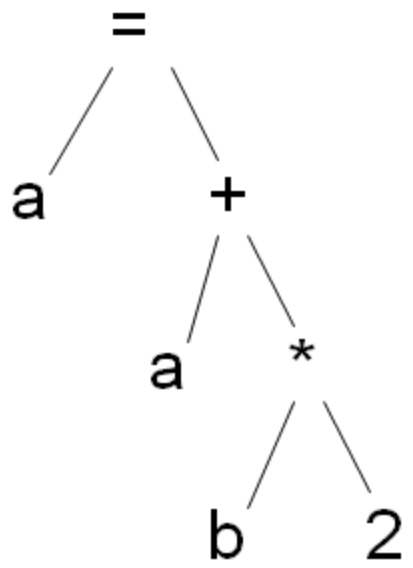
- The assignment: a = a + b * 2
- The AST:



Register machine code vs. weighted register allocation (Cont.)

- Register machine code:

```
Load_mem a, R1
Load_mem a, R2
Load_mem b, R3
Load_Const 2, R4
Multi_Reg   R4, R3
Add_Reg     R3, R2
Store_Reg   R2, a
```

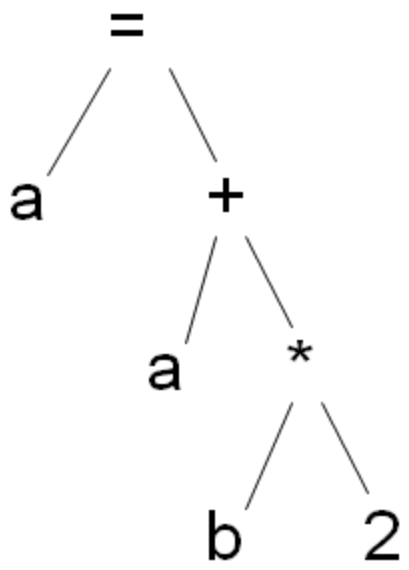


Use 4 registers.

Register machine code vs. weighted register allocation (Cont.)

- Weighted register allocation code:

```
Load_mem b, R1
Load_Const 2, R2
Multi_Reg   R2, R1
Load_mem a, R2
Add_Reg R2, R1
Store_Reg R1, a
```



Use 2 registers.

Homework

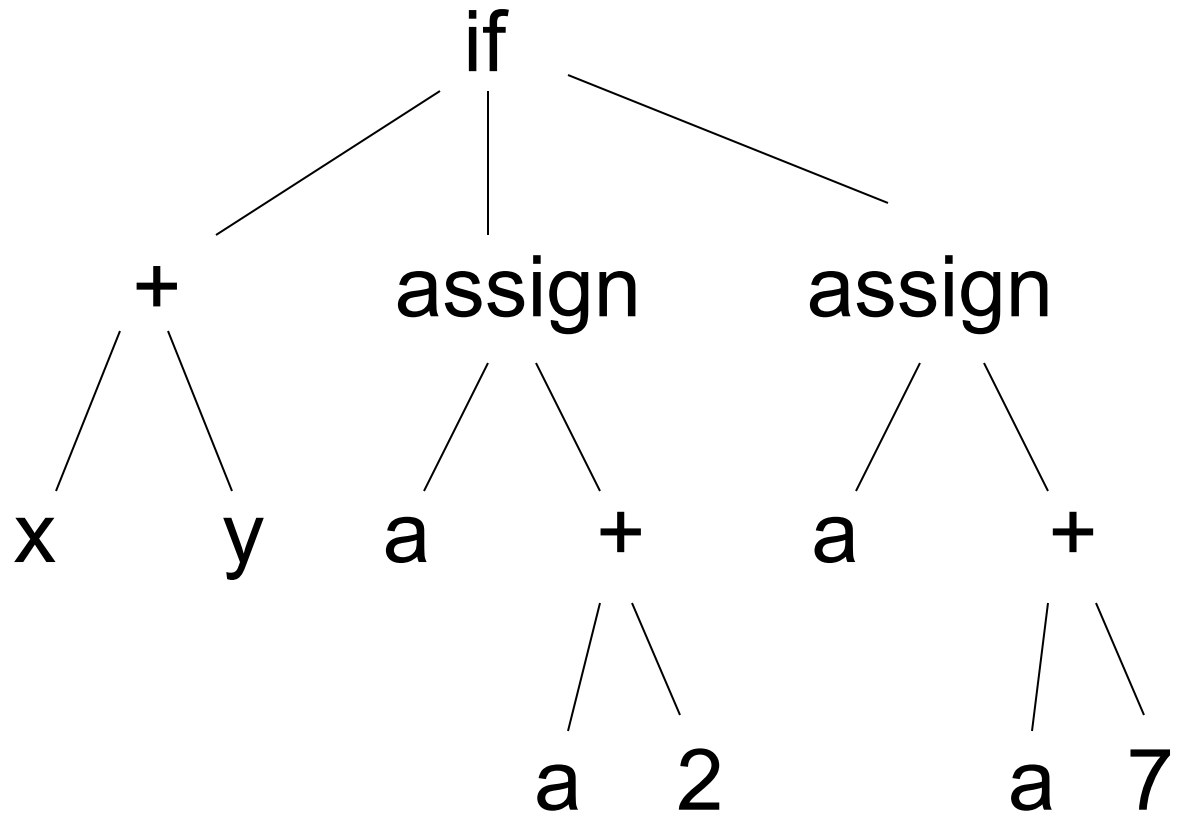
1. Given the grammar:

Start → Stmt $
Stmt → id assign E
| if lparen E rparen Stmt else Stmt fi
| begin Stmts end
Stmts → Stmts semi Stmt
| Stmt
E → E plus T
| T
T → id
| num

Show the AST for expression:

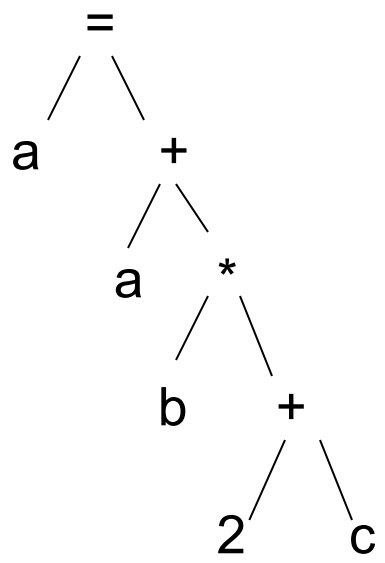
if (x + y) a = a + 2 else a = a + 7 fi

Homework 1 Solution



Homework

2. For the assignment: a = a + b * (2 + c)
And its AST:



a). generate target code for register machine
b). generate target code using weighted register allocation

Homework Solution

a). Generate target code using register machine

```
Load_mem a, R1
Load_mem a, R2
Load_mem b, R3
Load_Const 2, R4
Load_mem c, R5
Add_Reg R5, R4
Multi_Reg   R4, R3
Add_Reg    R3, R2
Store_Reg  R2, a
```

Homework Solution

b). Generate target code using weighted register allocation

```
Load_Const 2, R1
Load_mem c, R2
Add_Reg R2, R1
Load_mem b, R2
Multi_Reg   R2, R1
Load_mem a, R2
Add_Reg R2, R1
Store_Reg R1, a
```