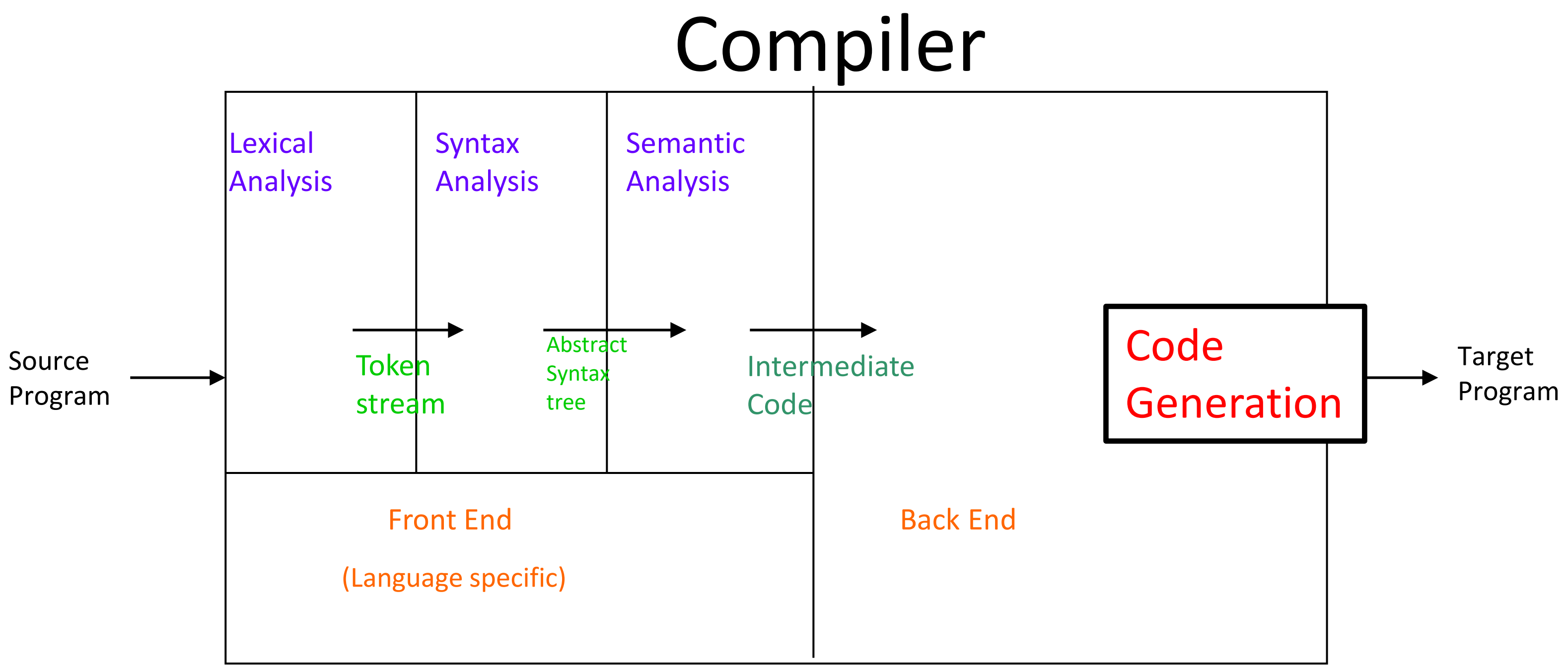
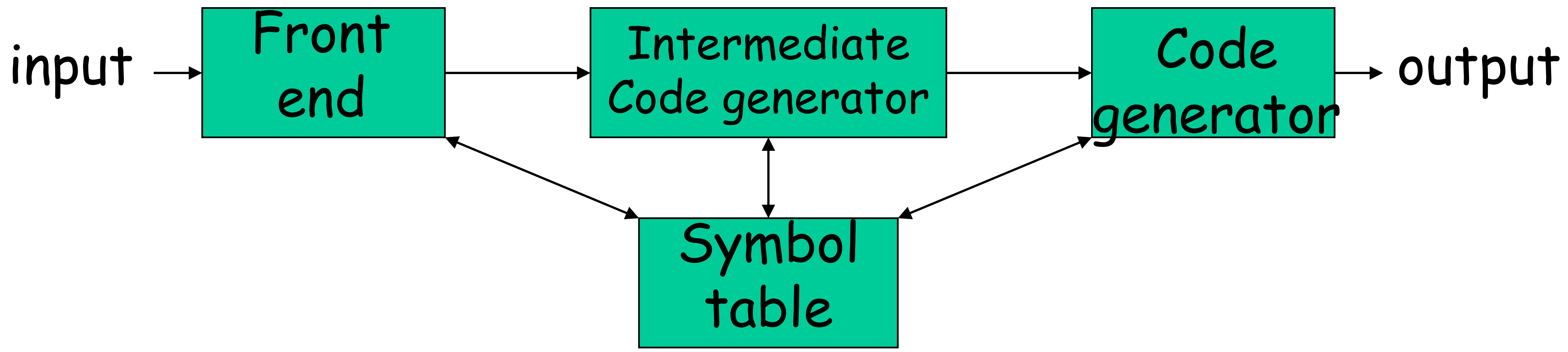
Principles of Compiler Design

Code Generation



Code generation and Instruction Selection



Requirements

- output code must be correct
- output code must be of high quality
- code generator should run efficiently

Design of code generator: Issues

- Input: Intermediate representation with symbol table
  - assume that input has been validated by the front end

- Target programs :
  - absolute machine language
    fast for small programs
  - relocatable machine code
    requires linker and loader
  - assembly code
    requires assembler, linker, and loader

More Issues…

- Instruction selection
  - Uniformity
  - Completeness
  - Instruction speed, power consumption
- Register allocation
  - Instructions with register operands are faster
  - store long life time and counters in registers
  - temporary locations
  - Even odd register pairs
- Evaluation order

Instruction Selection

- straight forward code if efficiency is not an issue

a=b+c	Mov b, R_0	
d=a+e	Add c, R_0	
	Mov R_0, a	
	Mov a, R_0	can be eliminated
	Add e, R_0	
	Mov R_0, d	
a=a+1	Mov a, R_0	Inc a
	Add #1, R_0	
	Mov R_0, a	

Example Target Machine

- Byte addressable with 4 bytes per word
- n registers R_0, R_1, ..., R_{n-l}
- Two address instructions of the form

  opcode source, destination
- Usual opcodes like move, add, sub etc.
- Addressing modes

MODE	FORM	ADDRESS
Absolute	M	M
register	R	R
index	c(R)	c+content(R)
indirect register	*R	content(R)
indirect index	*c(R)	content(c+content(R))
literal	#c	c

Flow Graph

- Graph representation of three address code
- Useful for understanding code generation (and for optimization)
- Nodes represent computation
- Edges represent flow of control

Basic blocks

- (maximum) sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end

Algorithm to identify basic blocks

- determine leader
  - first statement is a leader
  - any target of a goto statement is a leader
  - any statement that follows a goto statement is a leader
- for each leader its basic block consists of the leader and all statements up to next leader

Flow graphs

- add control flow information to basic blocks
- nodes are the basic blocks
- there is a directed edge from B_1 to B_2 if B_2 can follow B_1 in some execution sequence
  - there is a jump from the last statement of B_1 to the first statement of B_2
  - B_2 follows B_1 in natural order of execution
- initial node: block with first statement as leader

Next use information

- for register and temporary allocation
- remove variables from registers if not used
- statement **X = Y op Z**

  defines X and uses Y and Z
- scan each basic blocks backwards
- assume all temporaries are dead on exit and all user variables are live on exit

Computing next use information

Suppose we are scanning

i : X := Y op Z

in backward scan

1. attach to statement **i**, information in symbol table about **X, Y, Z**
2. set **X** to “not live” and “no next use” in symbol table
3. set **Y** and **Z** to be “live” and next use as **i** in symbol table

Example

1: $t_1 = a * a$

2: $t_2 = a * b$

3: $t_3 = 2 * t_2$

4: $t_4 = t_1 + t_3$

5: $t_5 = b * b$

6: $t_6 = t_4 + t_5$

7: $X = t_6$

Example

STATEMENT

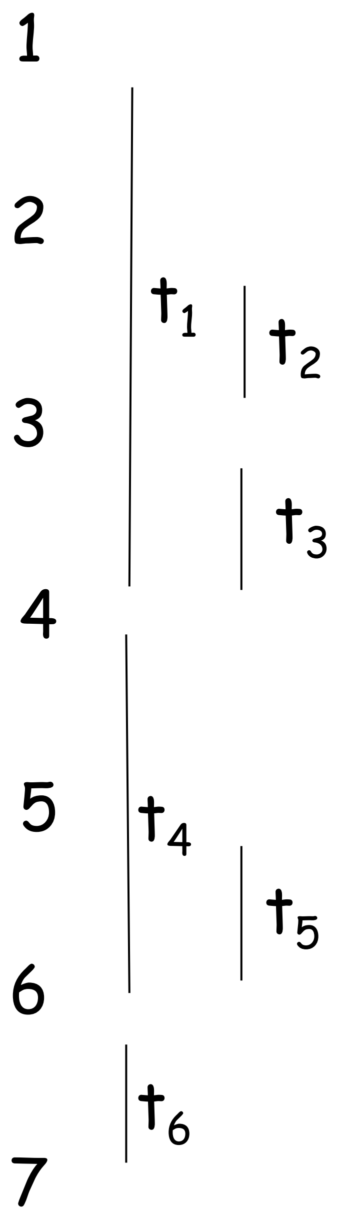
1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$

7: no temporary is live
6: t_6:use(7), t_4 t_5 not live
5: t_5:use(6)
4: t_4:use(6), t_1 t_3 not live
3: t_3:use(4), t_2 not live
2: t_2:use(3)
1: t_1:use(4)

Symbol Table

t_1	dead	Use in 4
t_2	dead	Use in 3
t_3	dead	Use in 4
t_4	dead	Use in 6
t_5	dead	Use in 6
t_6	dead	Use in 7

Example …



1: $t_1 = a * a$

2: $t_2 = a * b$

3: $t_2 = 2 * t_2$

4: $t_1 = t_1 + t_2$

5: $t_2 = b * b$

6: $t_1 = t_1 + t_2$

7: $X = t_1$

STATEMENT

1: $t_1 = a * a$

2: $t_2 = a * b$

3: $t_3 = 2 * t_2$

4: $t_4 = t_1 + t_3$

5: $t_5 = b * b$

6: $t_6 = t_4 + t_5$

7: $X = t_6$

Code Generator

- consider each statement
- remember if operands are in registers
- **Register descriptor**
  - Keep track of what is currently in each register.
  - Initially all the registers are empty
- **Address descriptor**
  - Keep track of location where current value of the name can be found at runtime
  - The location might be a register, stack, memory address or a set of those

Code Generation Algorithm

for each X = Y op Z do

- invoke a function getreg to determine location L where X must be stored. Usually L is a register.
- Consult address descriptor of Y to determine Y'. Prefer a register for Y'. If value of Y not already in L generate

```
Mov Y', L
```

Code Generation Algorithm

- Generate

  op Z', L

  Again prefer a register for Z. Update address descriptor of X to indicate X is in L.
- If L is a register, update its descriptor to indicate that it contains X and remove X from all other register descriptors.
- If current value of Y and/or Z have no next use and are dead on exit from block and are in registers, change register descriptor to indicate that they no longer contain Y and/or Z.

Function getreg

1. If Y is in register (that holds no other values) and Y is not live and has no next use after

   $X = Y \text{ op } Z$

   then return register of Y for L.
2. Failing (1) return an empty register
3. Failing (2) if X has a next use in the block or op requires register then get a register R, store its content into M (by Mov R, M) and use it.
4. else select memory location X as L

Example

Stmt	code	reg desc	addr desc
t_1=a-b	mov a,R_0		
	sub b,R_0	R_0 contains t_1	t_1 in R_0
t_2=a-c	mov a,R_1	R_0 contains t_1	t_1 in R_0
	sub c,R_1	R_1 contains t_2	t_2 in R_1
$t_3=t_1+t_2$	add R_1,R_0	R_0 contains t_3	t_3 in R_0
		R_1 contains t_2	t_2 in R_1
$d=t_3+t_2$	add R_1,R_0	R_0 contains d	d in R_0
	mov R_0,d		d in R_0 and memory

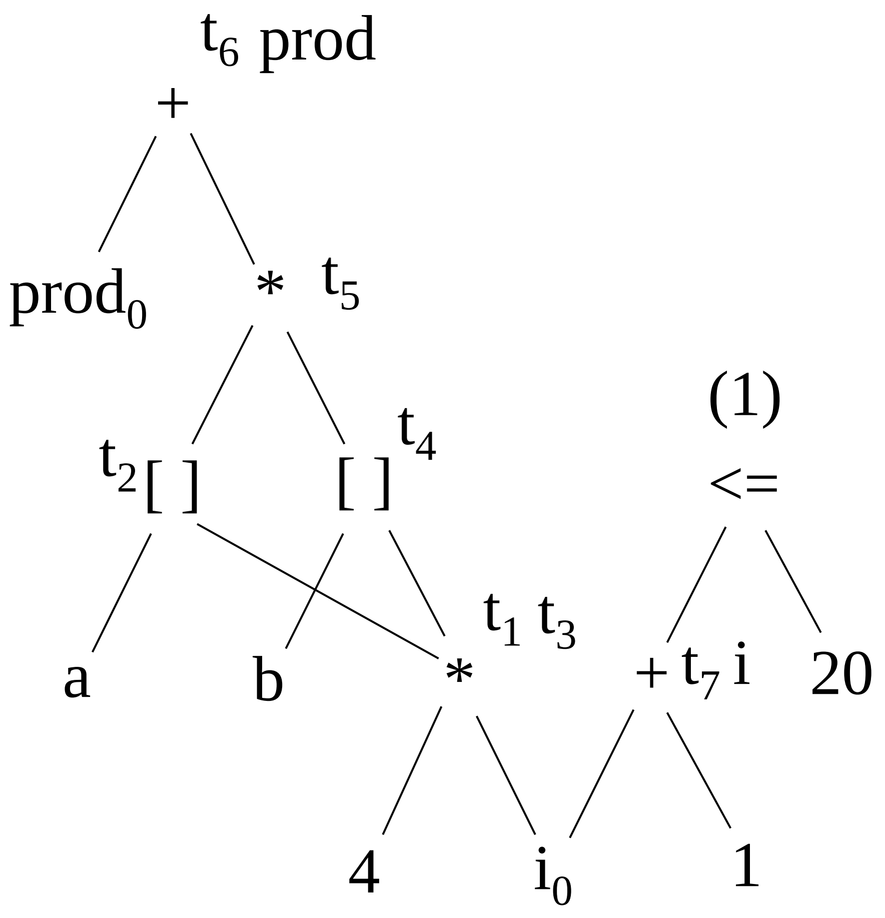
t_1=a-b
t_2=a-c
$t_3=t_1+t_2$
$d=t_3+t_2$

DAG representation of basic blocks

- useful data structures for implementing transformations on basic blocks
- gives a picture of how value computed by a statement is used in subsequent statements
- good way of determining common sub-expressions
- A dag for a basic block has following labels on the nodes
  - leaves are labeled by unique identifiers, either variable names or constants
  - interior nodes are labeled by an operator symbol
  - nodes are also optionally given a sequence of identifiers for labels

DAG representation: example

1. t_1 := 4 * i
2. t_2 := a[t_1]
3. t_3 := 4 * i
4. t_4 := b[t_3]
5. t_5 := t_2 * t_4
6. t_6 := prod + t_5
7. prod := t_6
8. t_7 := i + 1
9. i := t_7
10. if i <= 20 goto (1)



Code Generation from DAG

S_1 = 4 * i
S_2 = addr(A)-4
S_3 = $S_2[S_1]$
S_4 = 4 * i
S_5 = addr(B)-4
S_6 = $S_5[S_4]$
S_7 = S_3 * S_6
S_8 = prod+S_7
prod = S_8
S_9 = I+1
I = S_9
If I <= 20 goto (1)

S_1 = 4 * i
S_2 = addr(A)-4
S_3 = $S_2[S_1]$

S_5 = addr(B)-4
S_6 = $S_5[S_4]$
S_7 = S_3 * S_6

prod = prod + S_7

I = I + 1
If I <= 20 goto (1)

Rearranging order of the code

- Consider following basic block

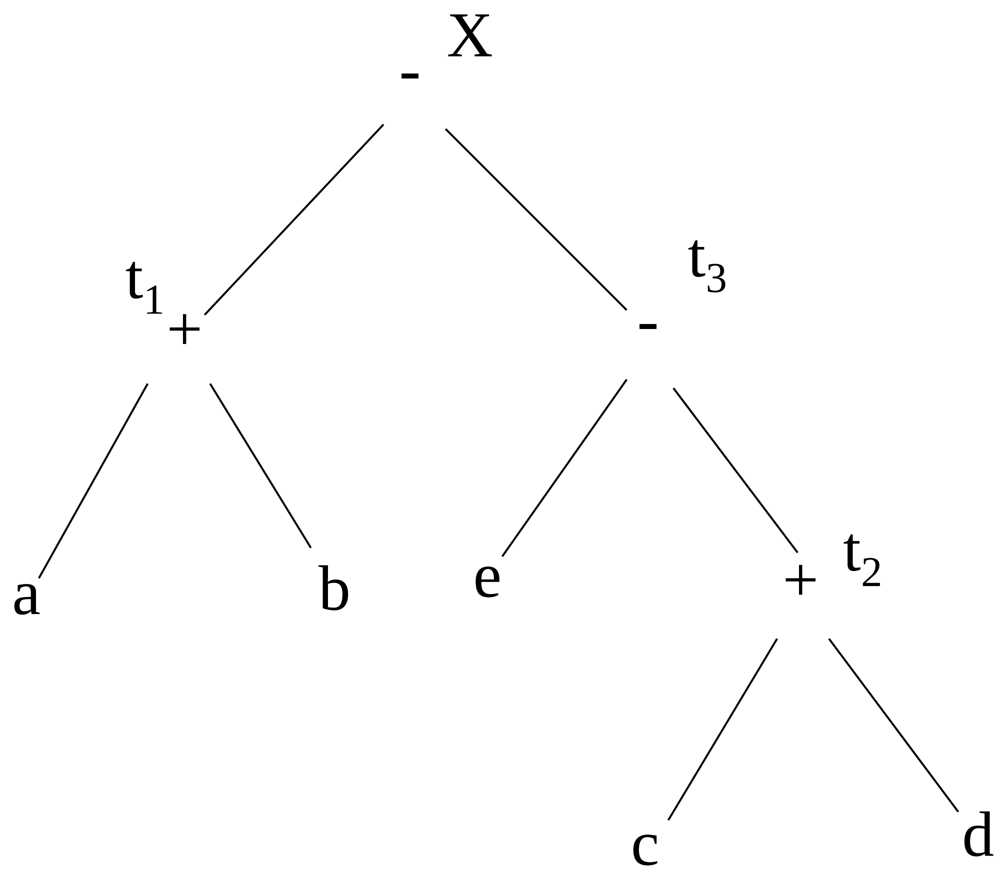
  $t_1 = a + b$

  $t_2 = c + d$

  $t_3 = e - t_2$

  $X = t_1 - t_3$

  and its DAG



Rearranging order …

Three adress code for the DAG (assuming only two registers are available)

MOV a, R_0
ADD b, R_0
MOV c, R_1
ADD d, R_1
MOV R_0, t_1
MOV e, R_0
SUB R_1, R_0
MOV t_1, R_1
SUB R_0, R_1
MOV R_1, X

Register spilling

Register reloading

Rearranging the code as

$t_2 = c + d$

$t_3 = e - t_2$

$t_1 = a + b$

$X = t_1 - t_3$

gives

MOV c, R_0
ADD d, R_0
MOV e, R_1
SUB R_0, R_1
MOV a, R_0
ADD b, R_0
SUB R_1, R_0
MOV R_1, X

Code Generator Generator

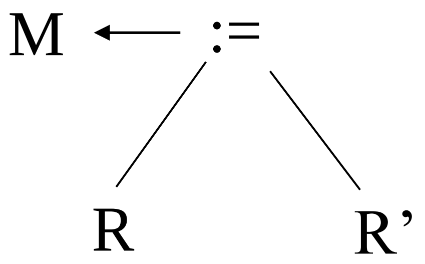
- Code generation by tree rewriting

- target code is generated during a process in which input tree is reduced to a single node

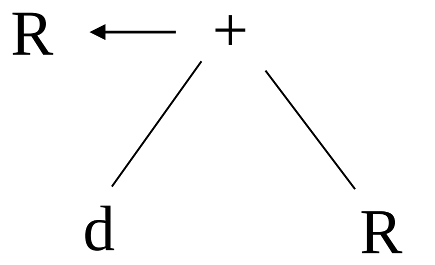
- each rewriting rule is of the form
  replacement → template { action}

  where
  - replacement is a single node
  - template is a tree
  - action is a code fragment

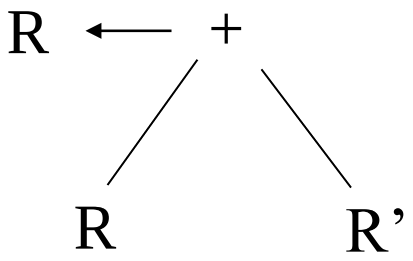
Instruction set for a hypothetical machine



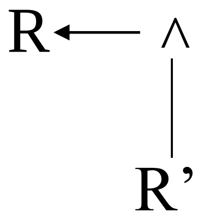
Move R’, R@



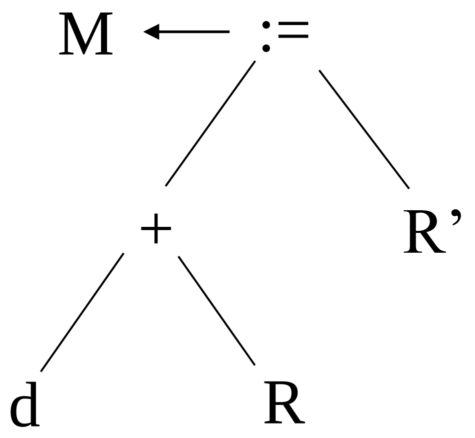
Add #d, R



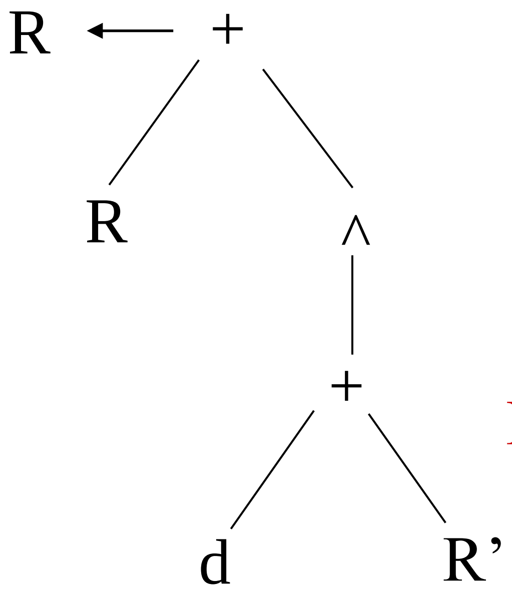
Add R’, R



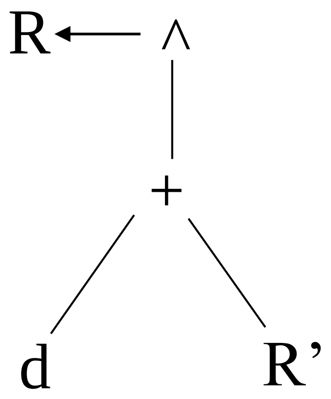
Move R’@, R



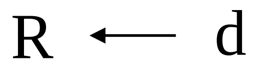
Move R’, R@(d)



Add R’@(d), R



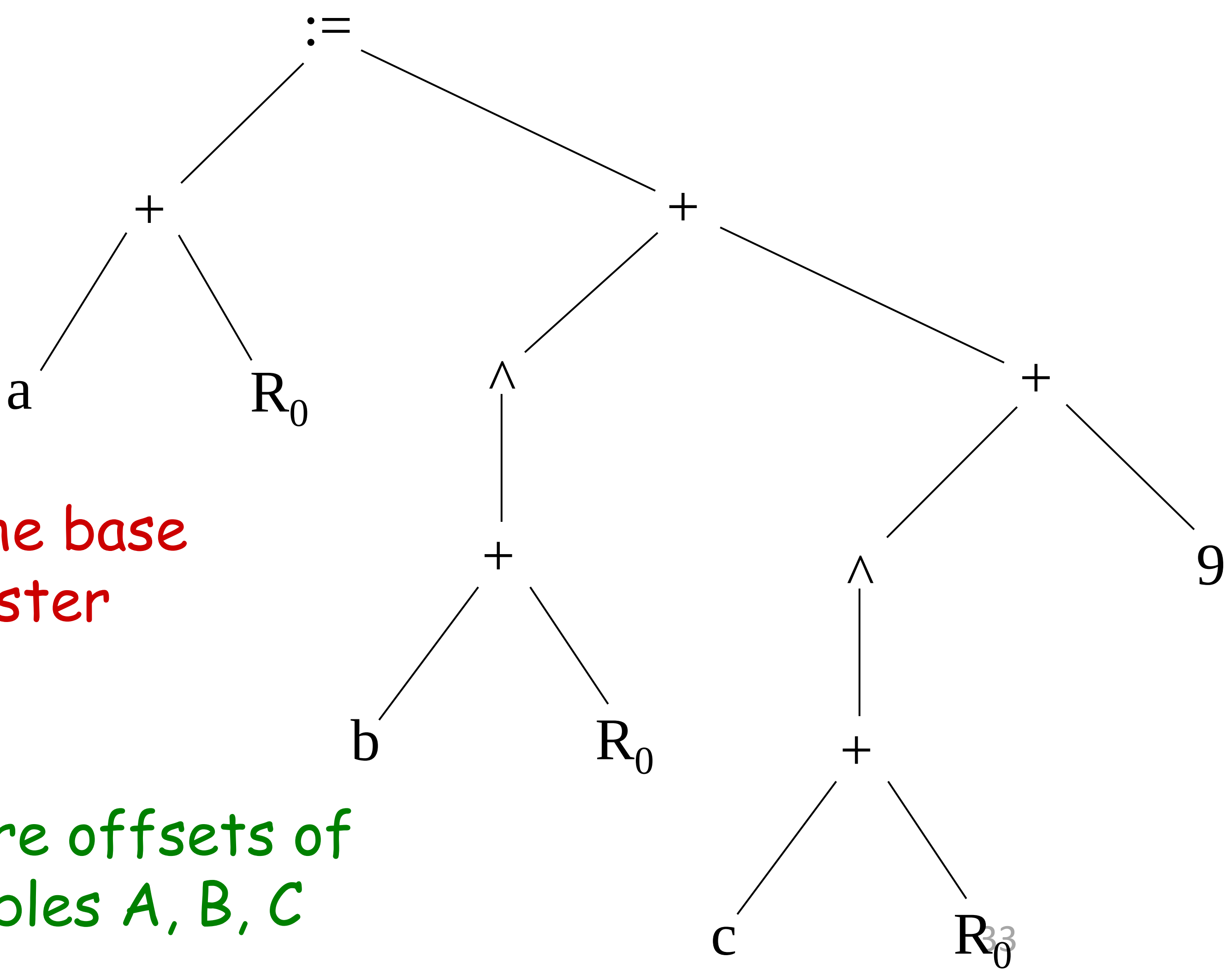
Move R’@(d), R



Move #d, R

Example

IR tree for A:=B+C+9



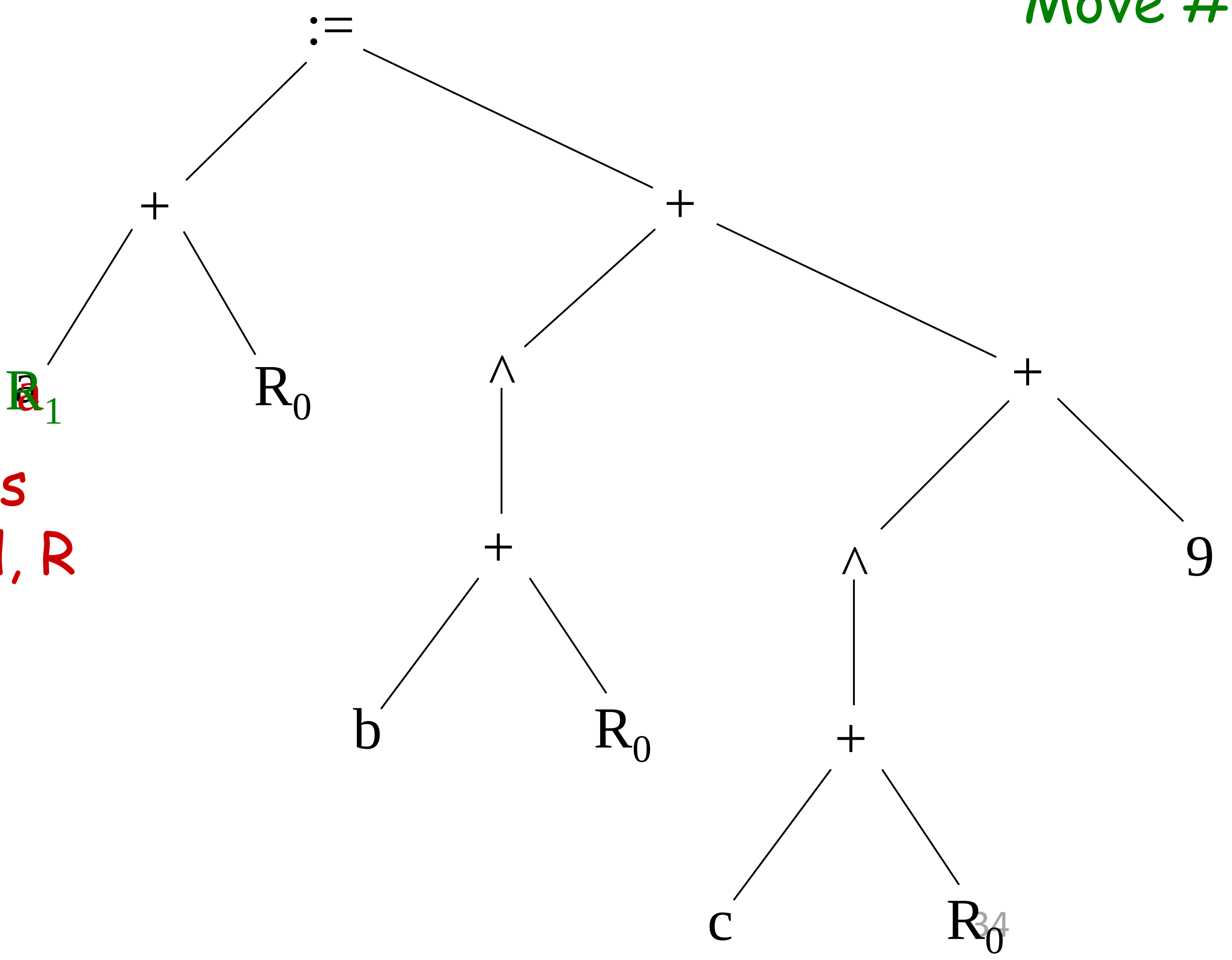
R_0 is the base register

a, b, c are offsets of Variables A, B, C

Example ...

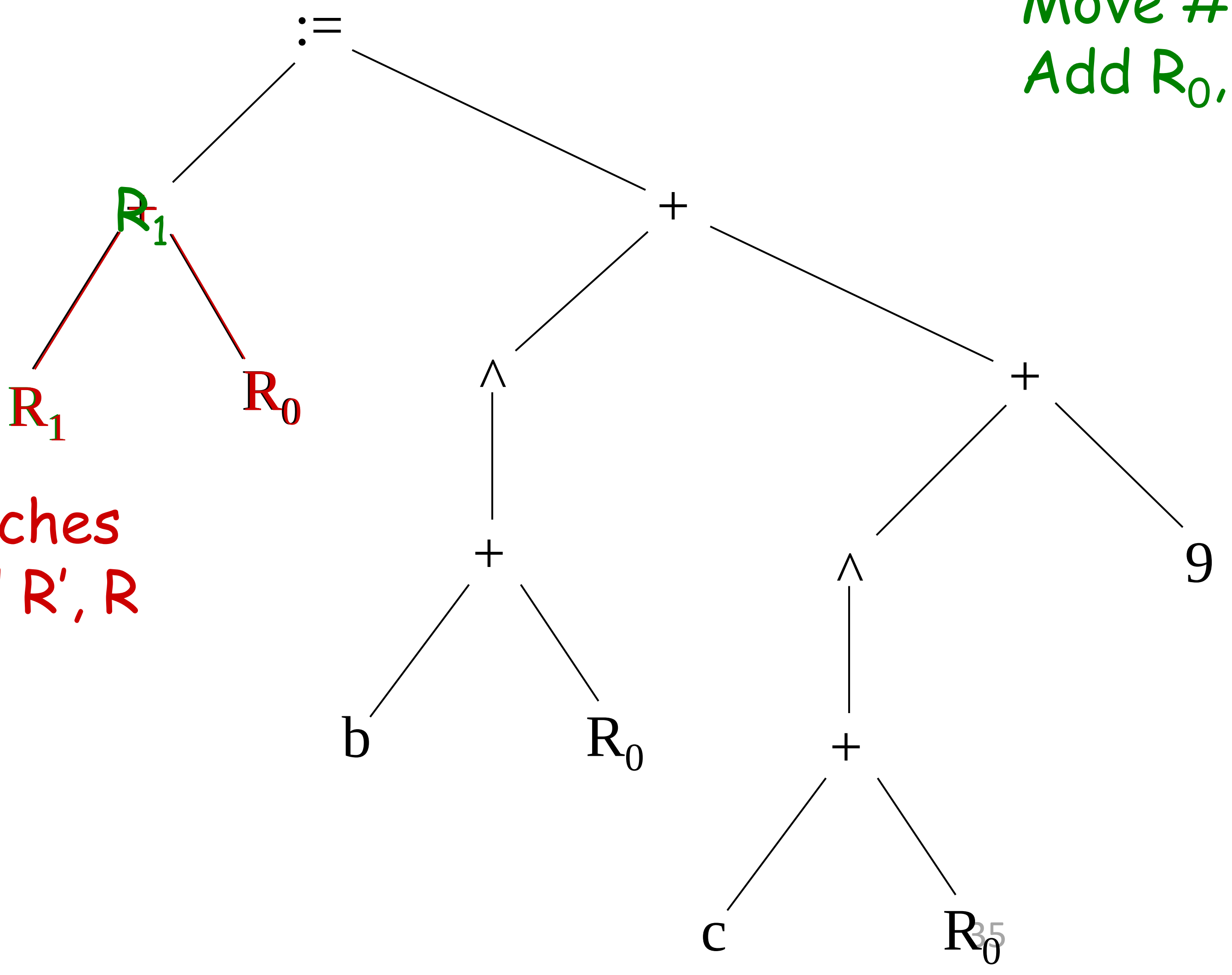
Generate

Move #a, R_1



matches
move #d, R

Example …



matches
Add R', R

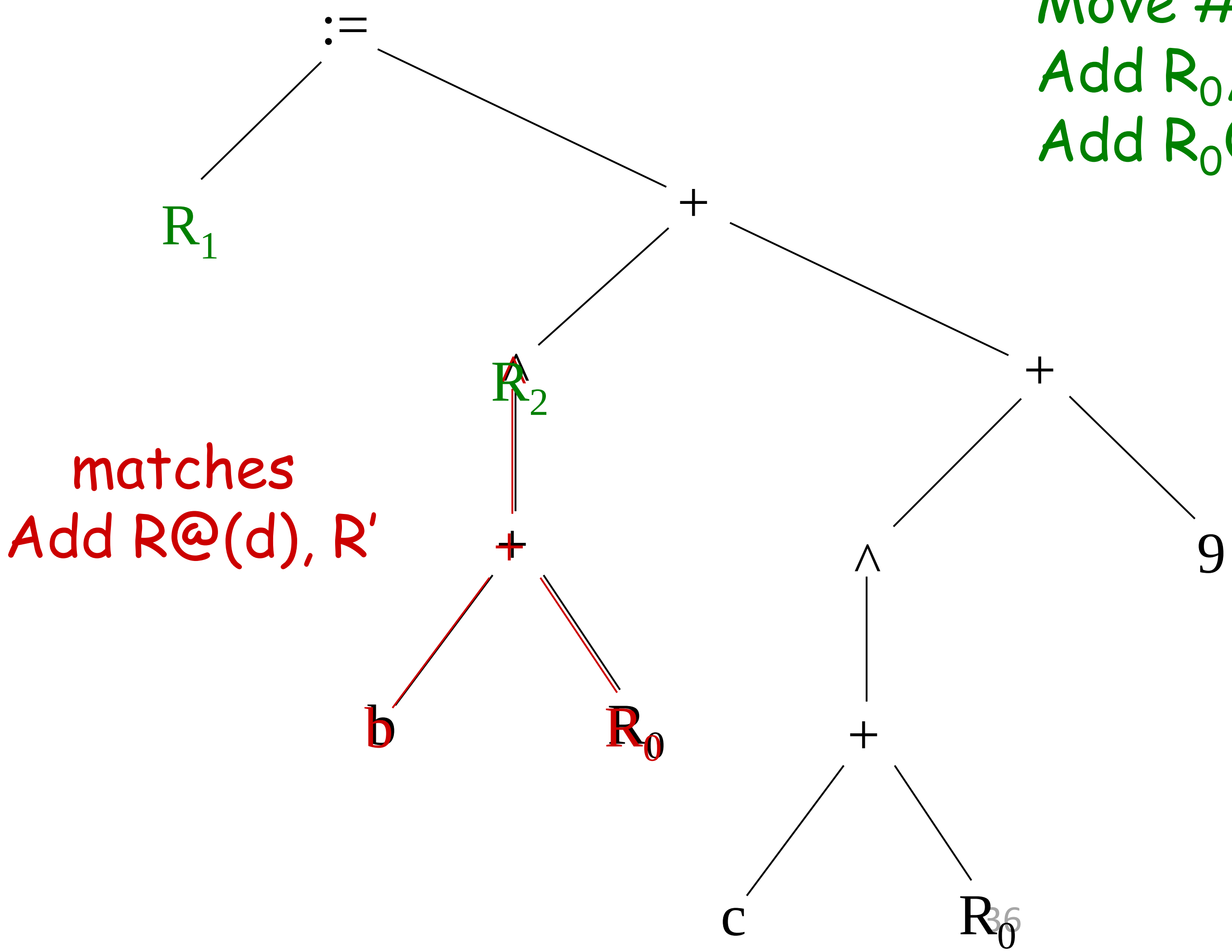
Generate

Move #a, R_1
Add R_0, R_1

Example …

Generate

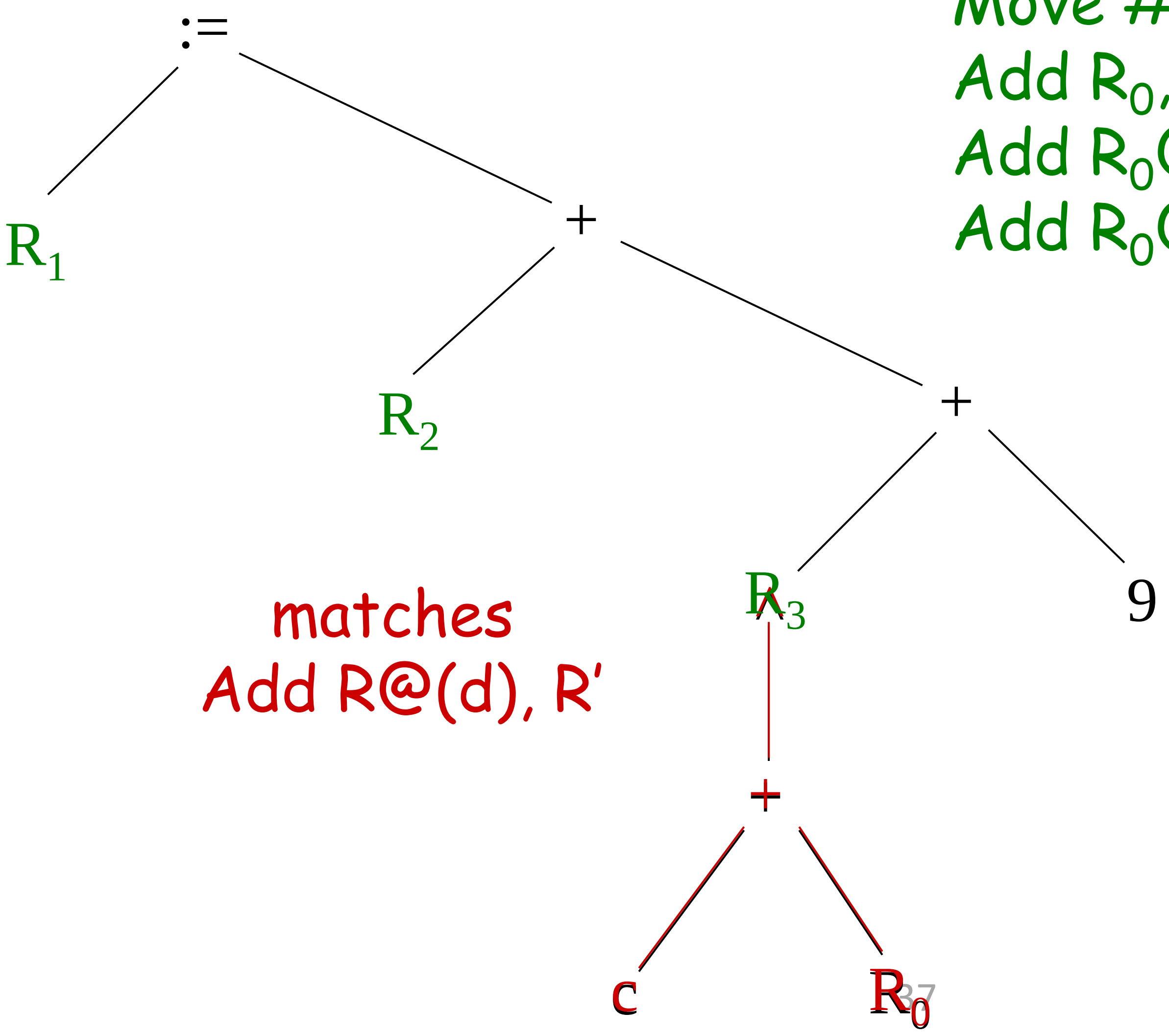
Move #a, R_1
Add R_0, R_1
Add R_0@(b), R_2



Example …

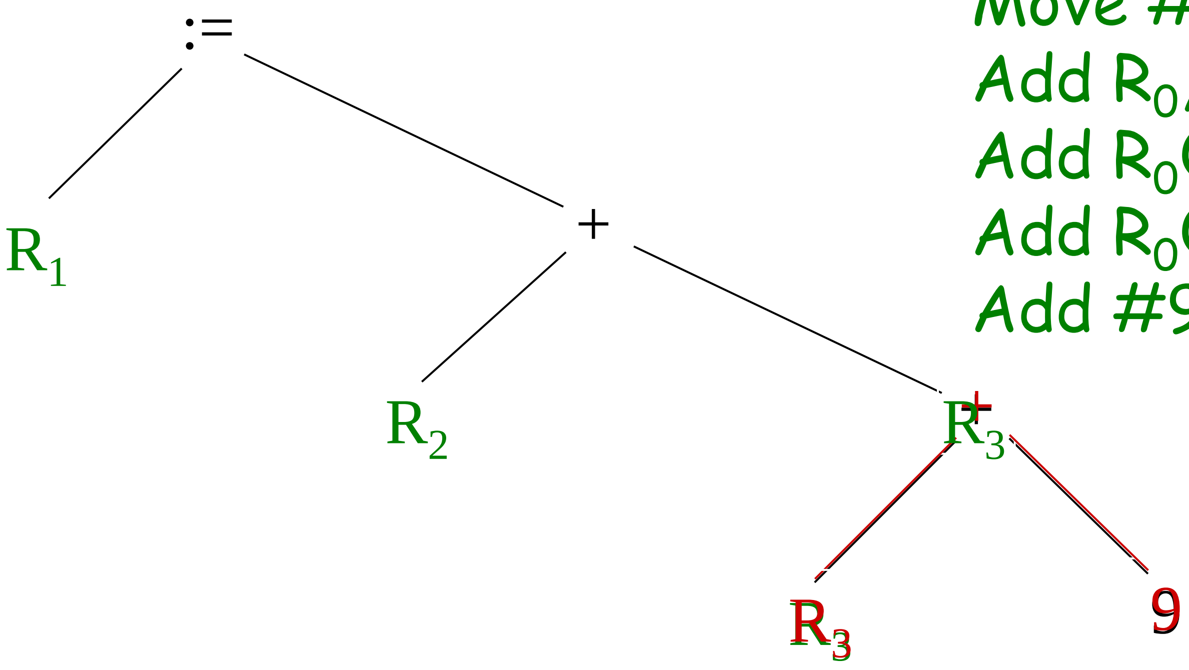
Generate

Move #a, R_1
Add R_0, R_1
Add R_0@(b), R_2
Add R_0@(c), R_3



matches
Add R@(d), R′

Example …



matches

Add #d, R

Generate

Move #a, R_1

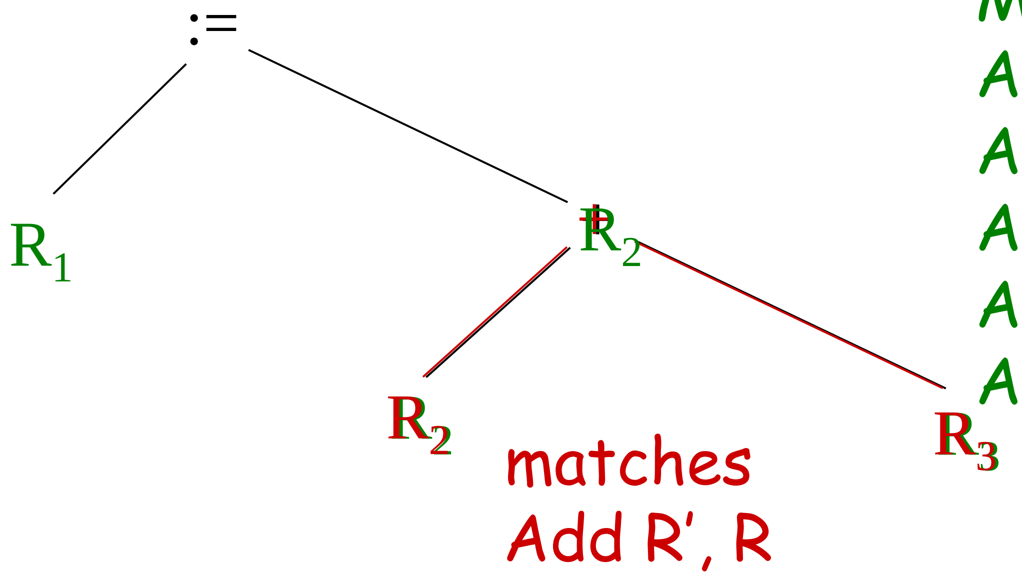
Add R_0, R_1

Add R_0@(b), R_2

Add R_0@(c), R_3

Add #9, R_3

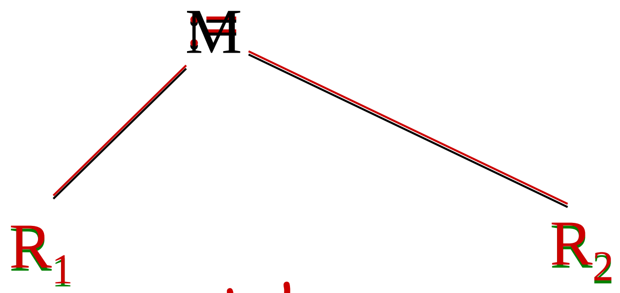
Example …



Generate

Move #a, R_1
Add R_0, R_1
Add R_0@(b), R_2
Add R_0@(c), R_3
Add #9, R_3
Add R_3, R_2

Example …



matches
Add R', R@

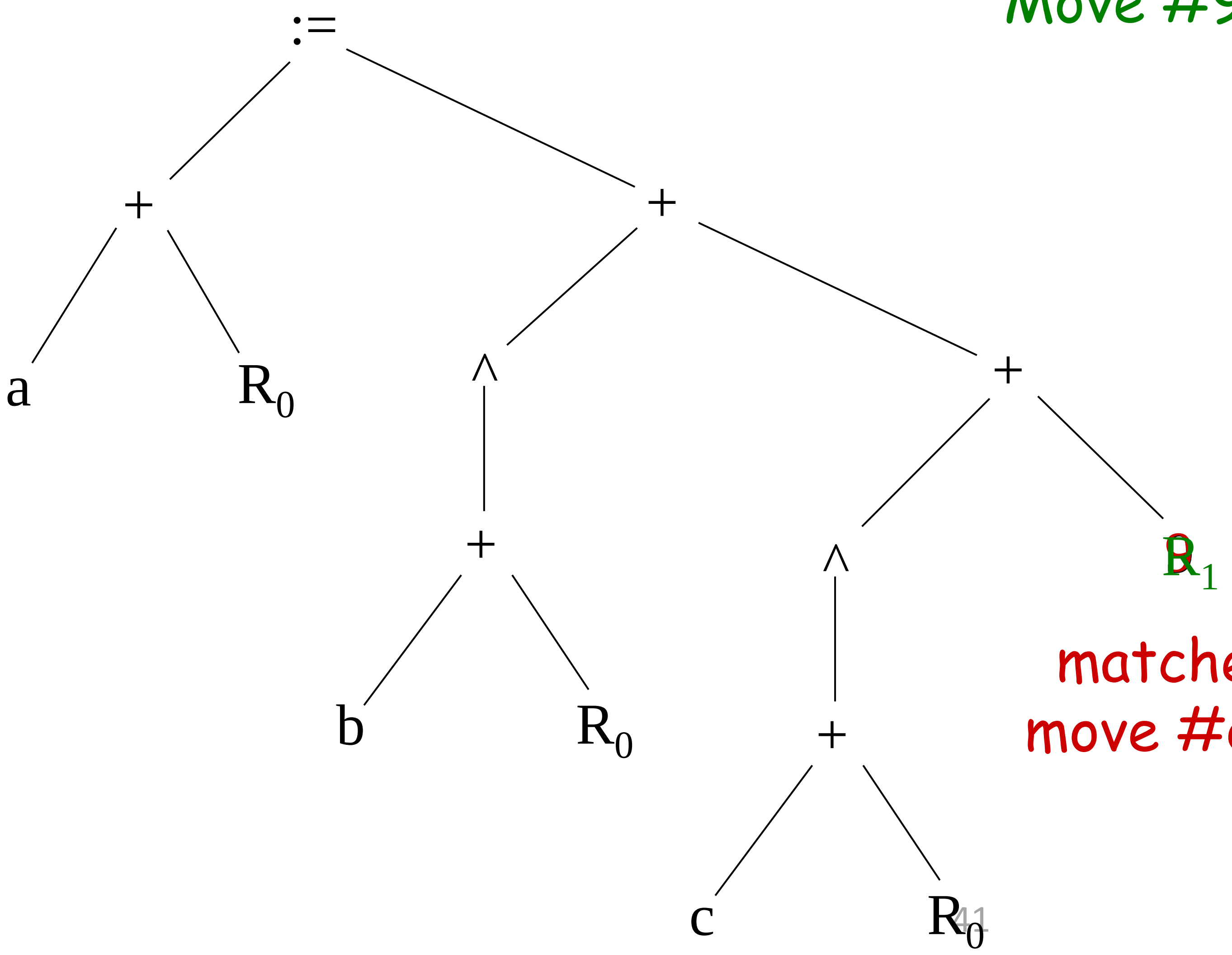
Generate

Move #a, R_1
Add R_0, R_1
Add R_0@(b), R_2
Add R_0@(c), R_3
Add #9, R_3
Add R_3, R_2
Move R_2, R_1@

Example

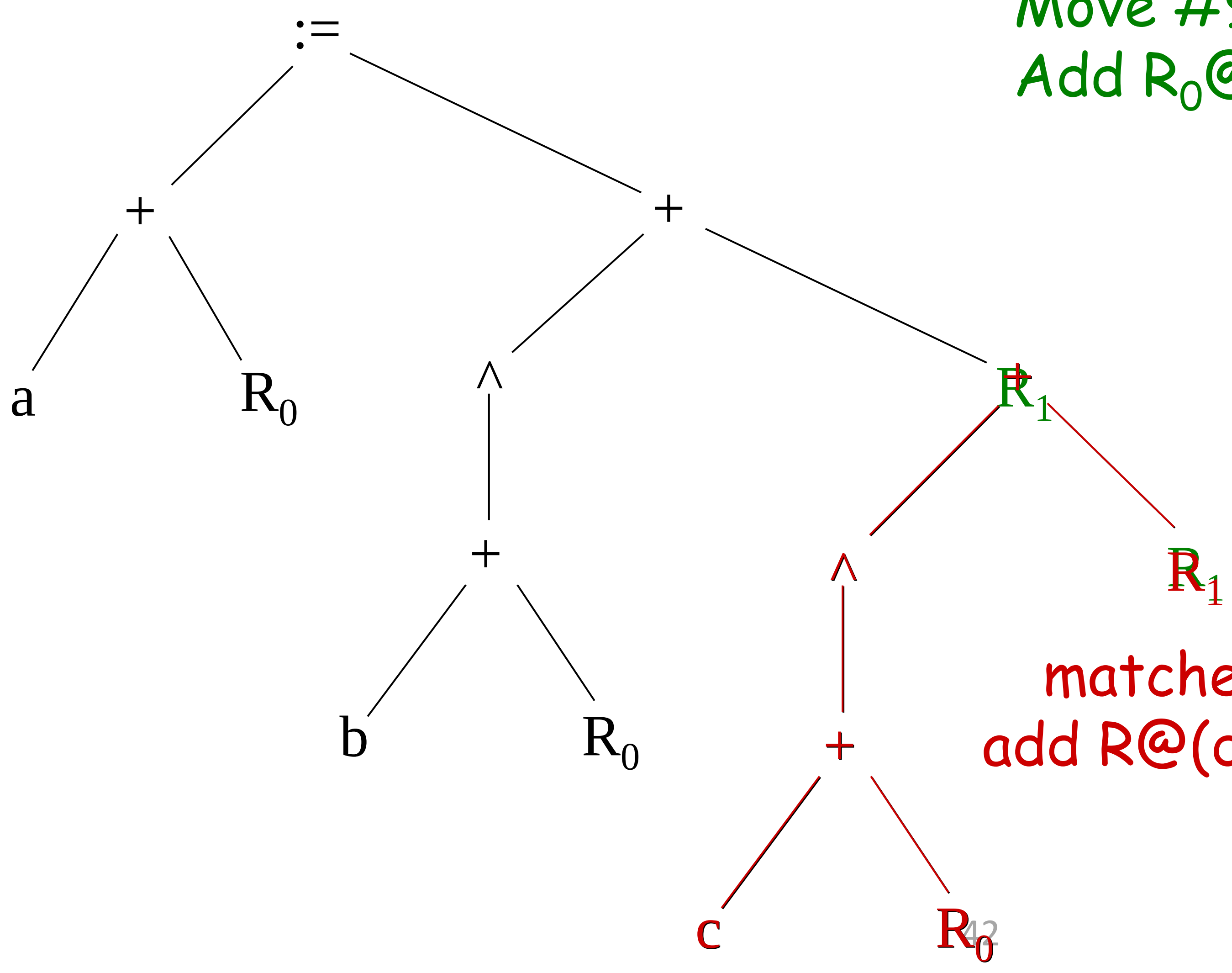
Generate

Move #9, R_1



matches
move #d, R

Example …



Generate

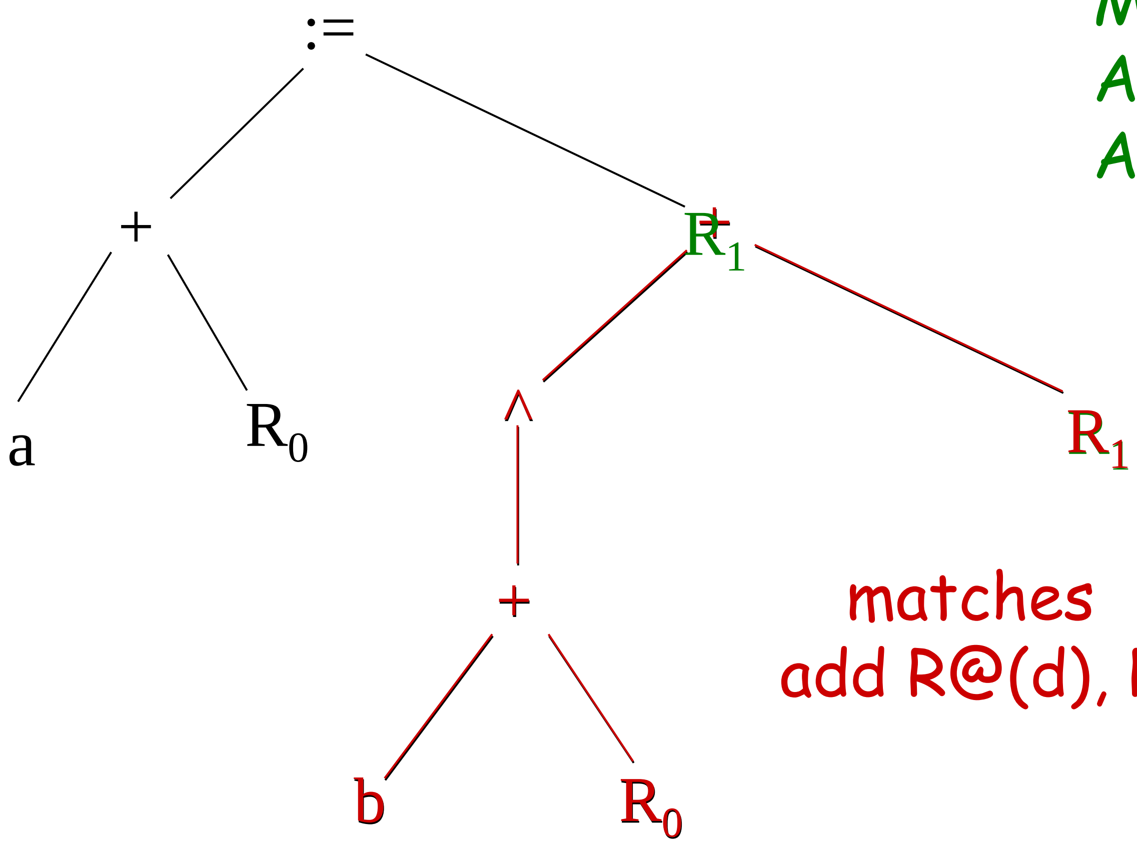
Move #9, R_1
Add R_0@(c), R_1

matches
add R@(d), R'

Example …

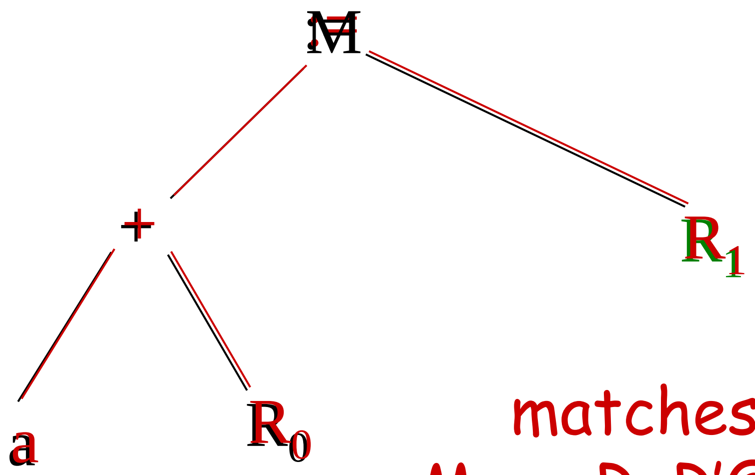
Generate

Move #9, R_1
Add R_0@(c), R_1
Add R_0@(b), R_1



matches
add R@(d), R′

Example …



matches

Move R, R'@(d)

Generate

Move #9, R_1
Add R_0@(c), R_1
Add R_0@(b), R_1
Move R_1, R_0@(a)