Code Generation

Compilers course

Masters in Informatics and Computing Engineering (MIEIC), 3rd Year

U.PORTO

João M. P. Cardoso

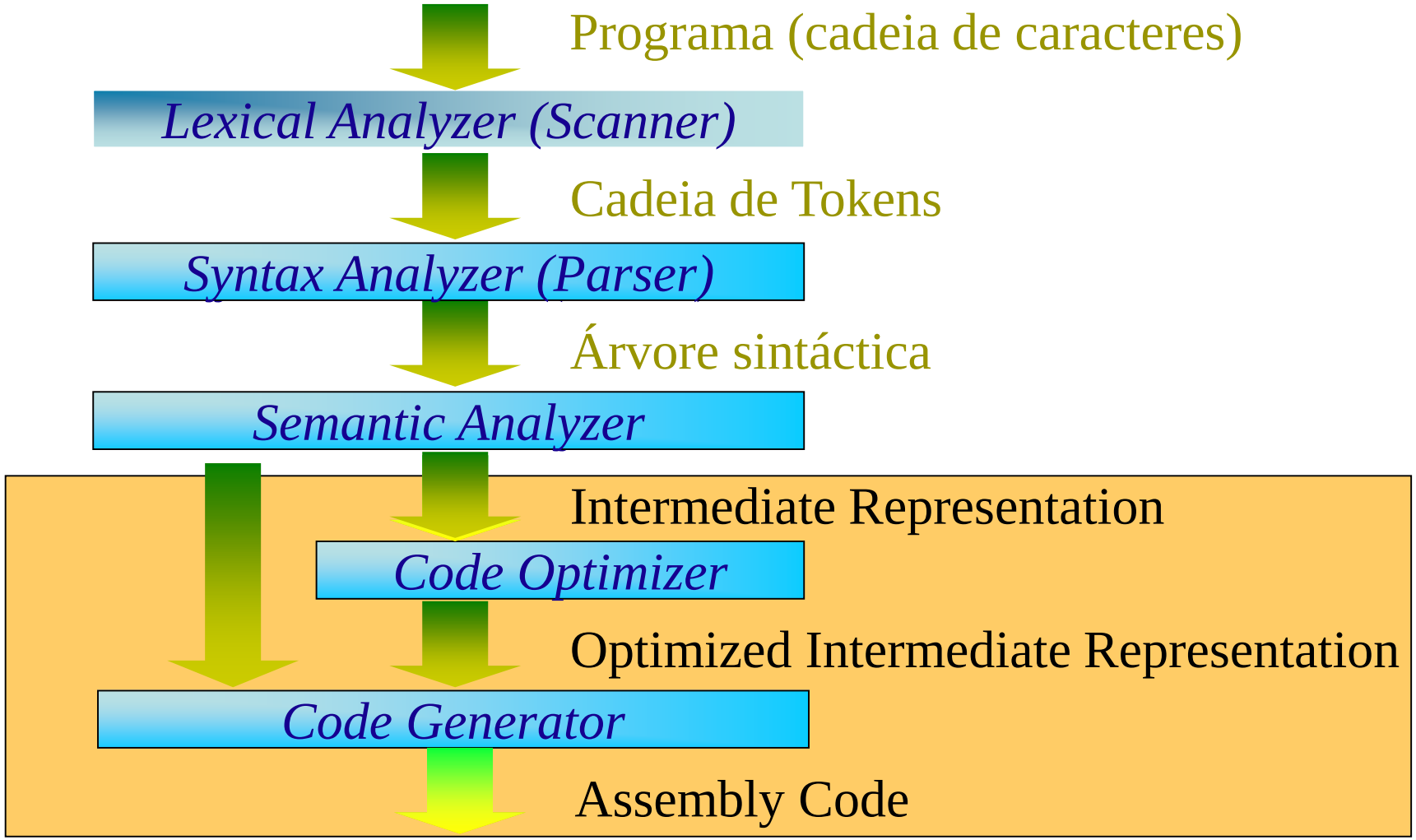
Universidade do Porto
FEUP Faculdade de Engenharia

Dep. de Engenharia Informática
Faculdade de Engenharia (FEUP), Universidade do Porto,
Porto, Portugal
Email:jmpc@acm.org

Problem

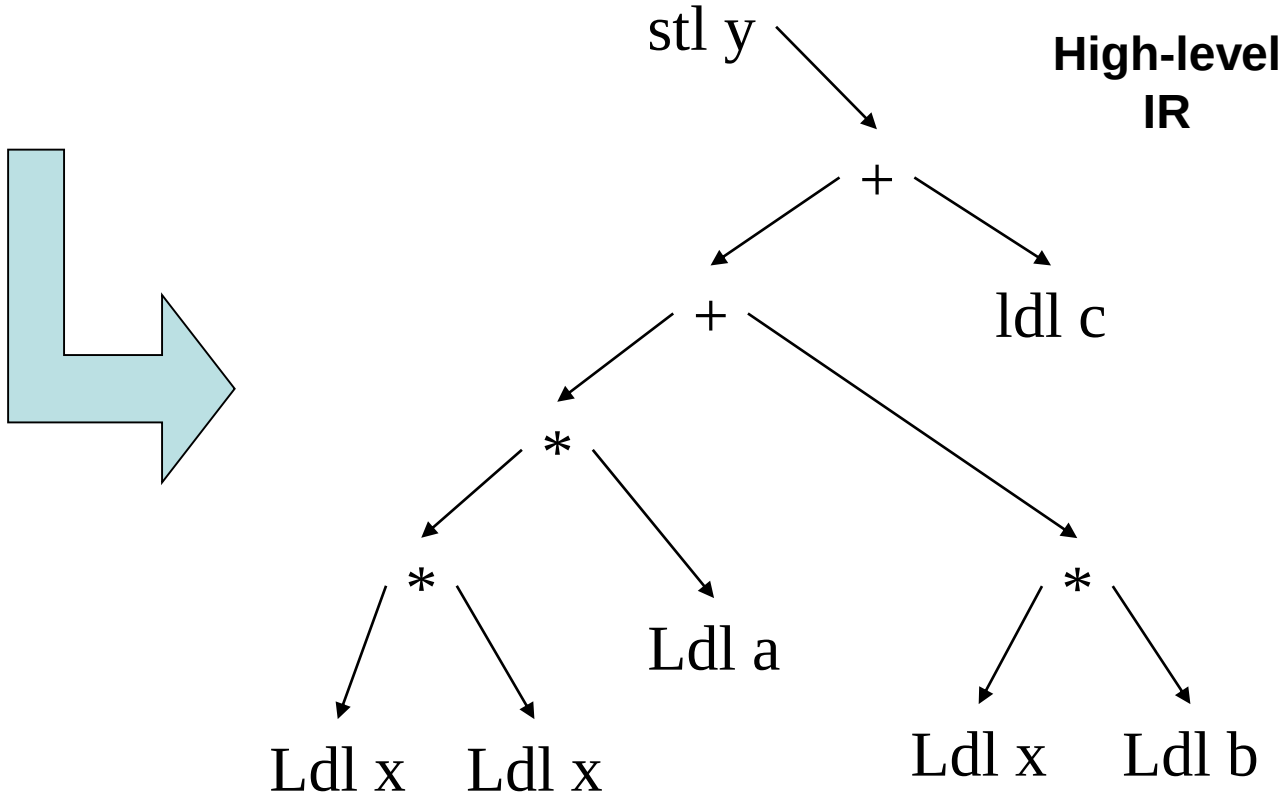
- How to generate assembly code given a low level intermediate representation?
- Not optimized:
  - Local variables and function parameters all assigned to distinct stack positions
- Optimized:
  - Sharing of relative stack positions by two or more local variables
  - Utilization of registers from the register file of the target microprocessor to accommodate local variables
  - ...

Final Code Generation



Final Code Generation

y=a*x*x+b*x+c;

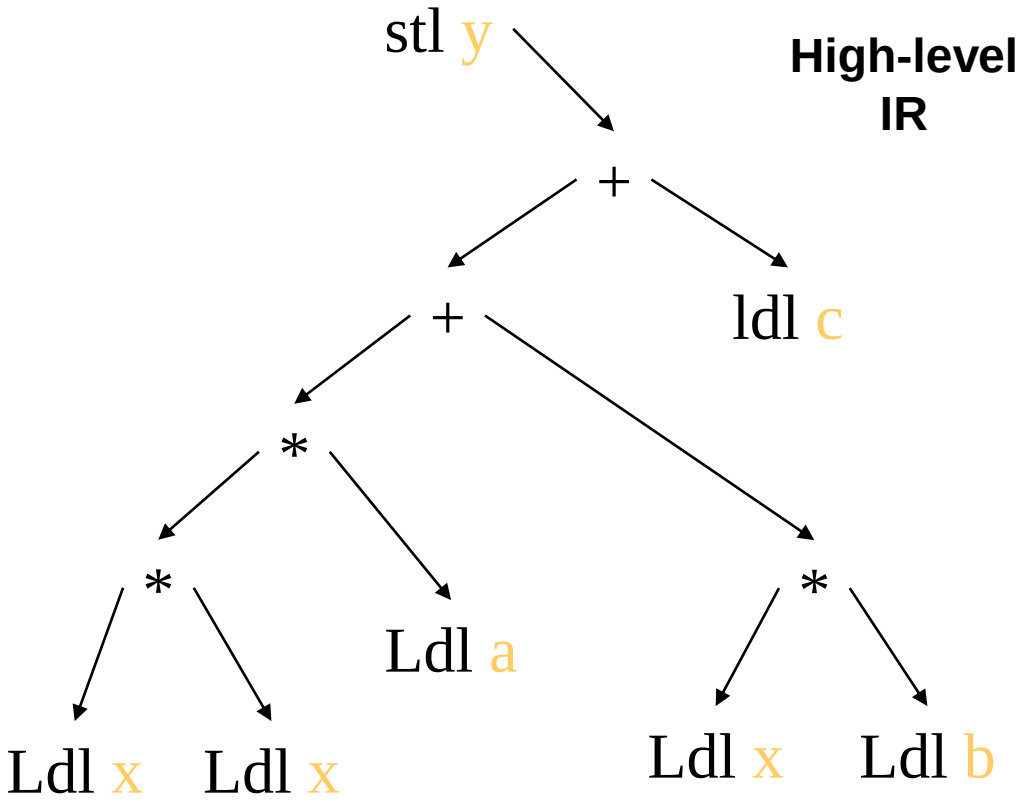


Final Code Generation

y=a*x*x+b*x+c;

Variables:

- a
- b
- x
- c
- y

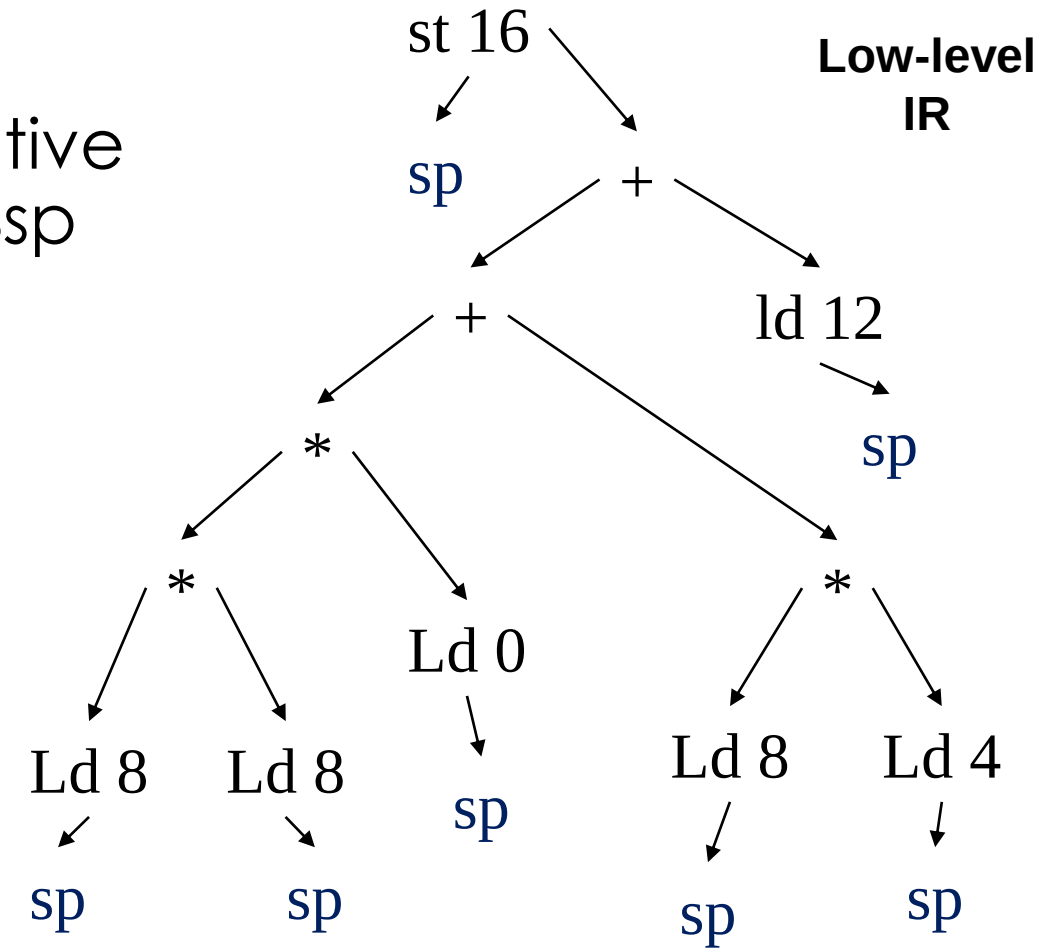


Final Code Generation

y=a*x*x+b*x+c;

Variables: relative position to $sp

- a: 0
- b: 4
- x: 8
- c: 12
- y: 16



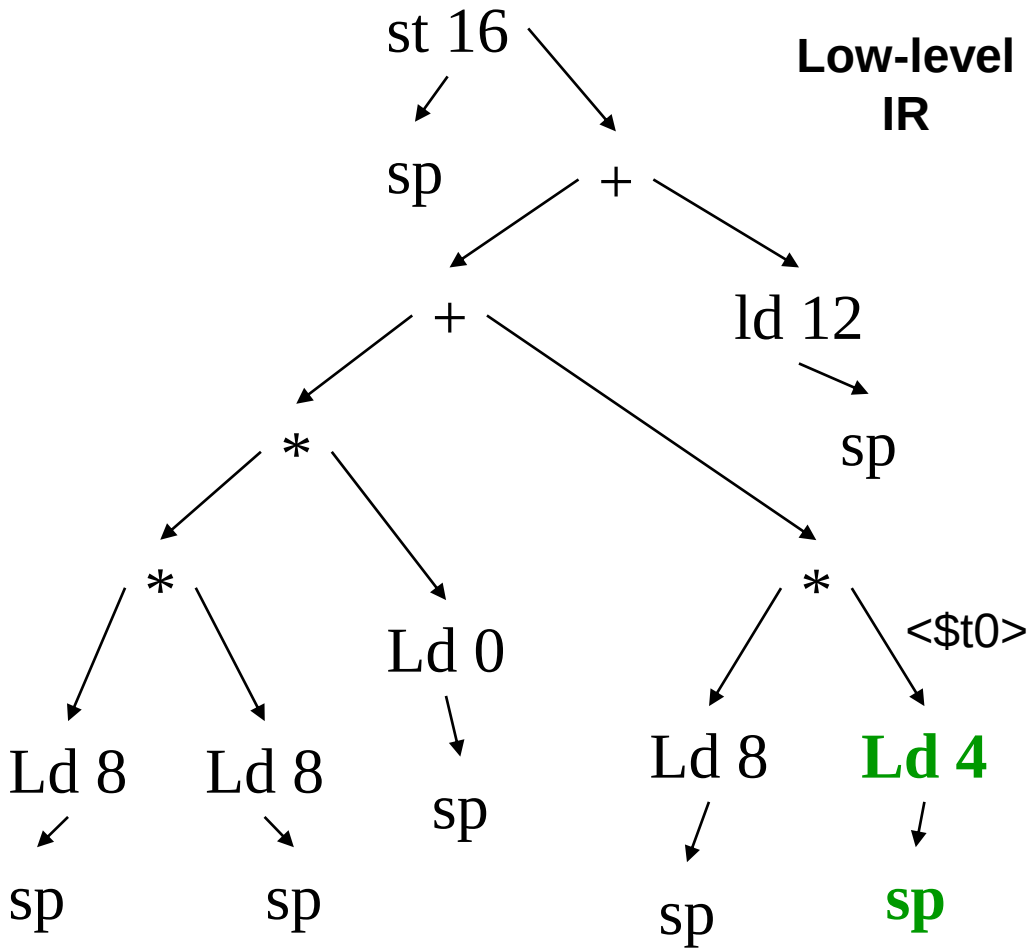
Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

Begin by leaves:

lw $t0, 4($sp)



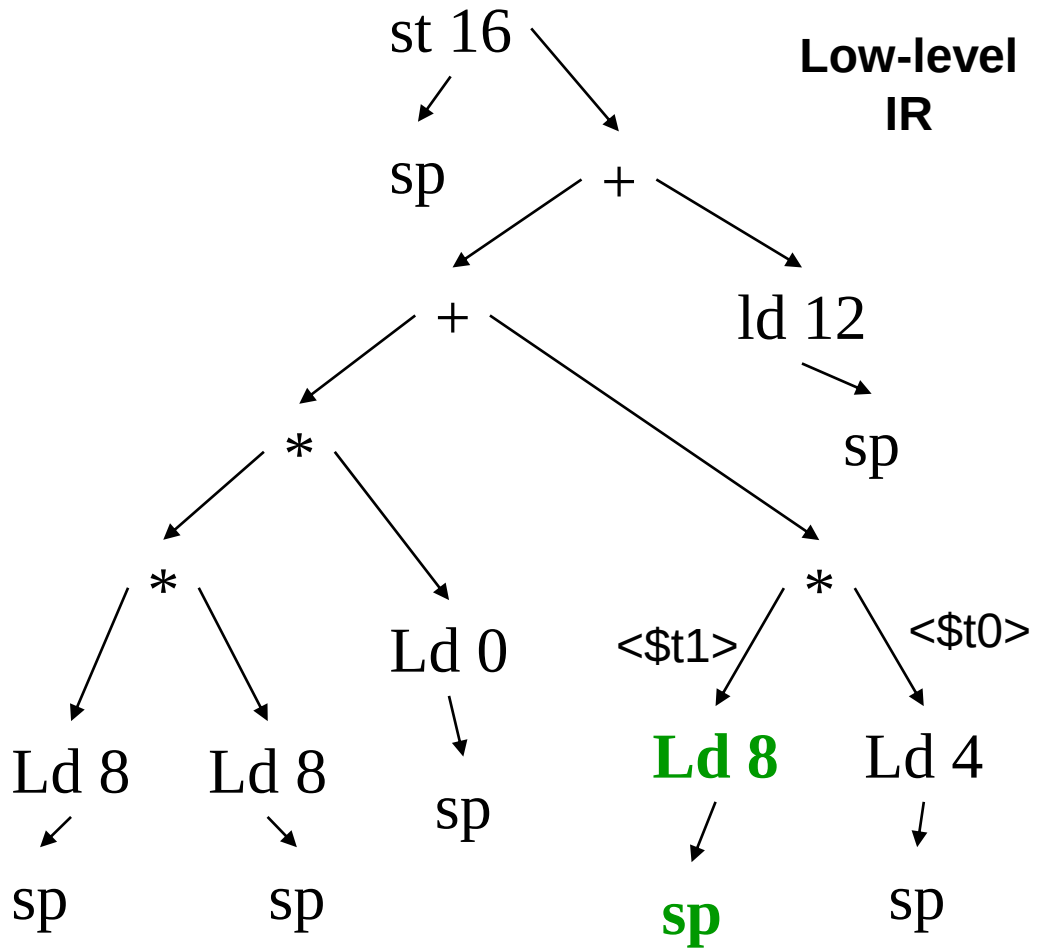
Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 4($sp)

lw $t1, 8($sp)

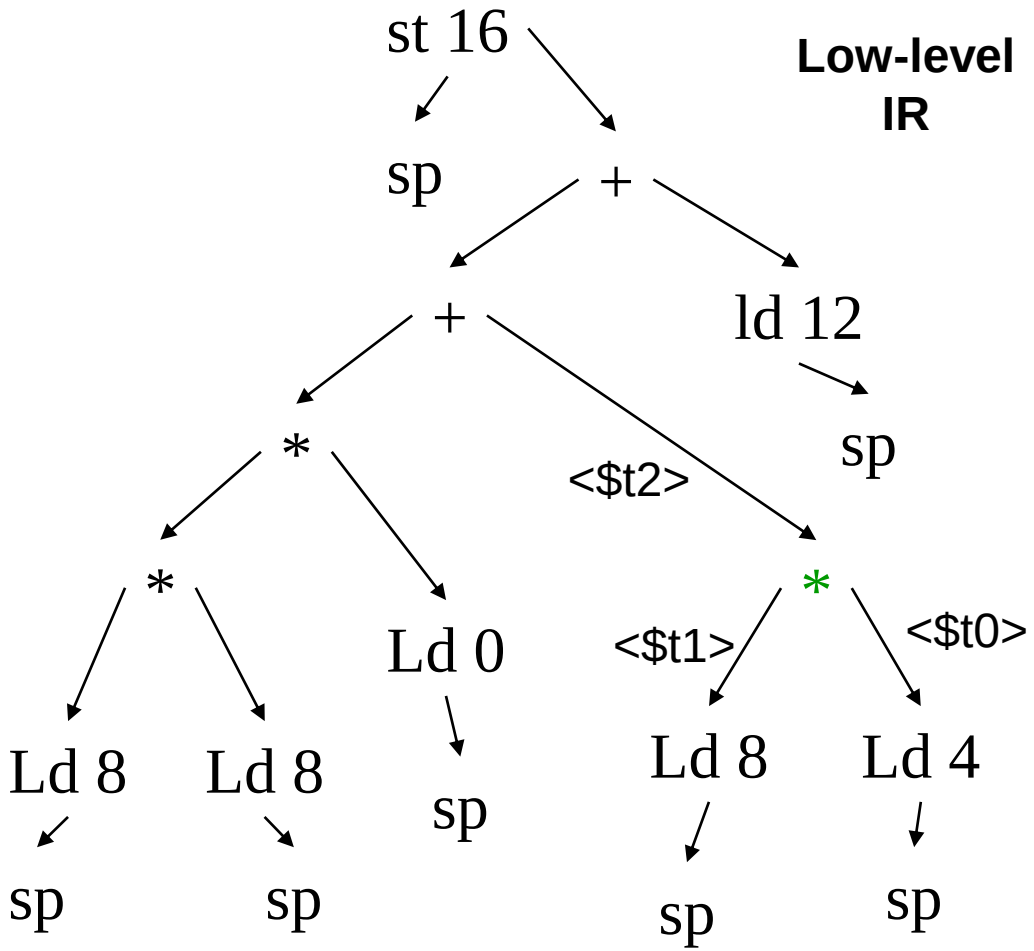


Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t2, $t1, $t0

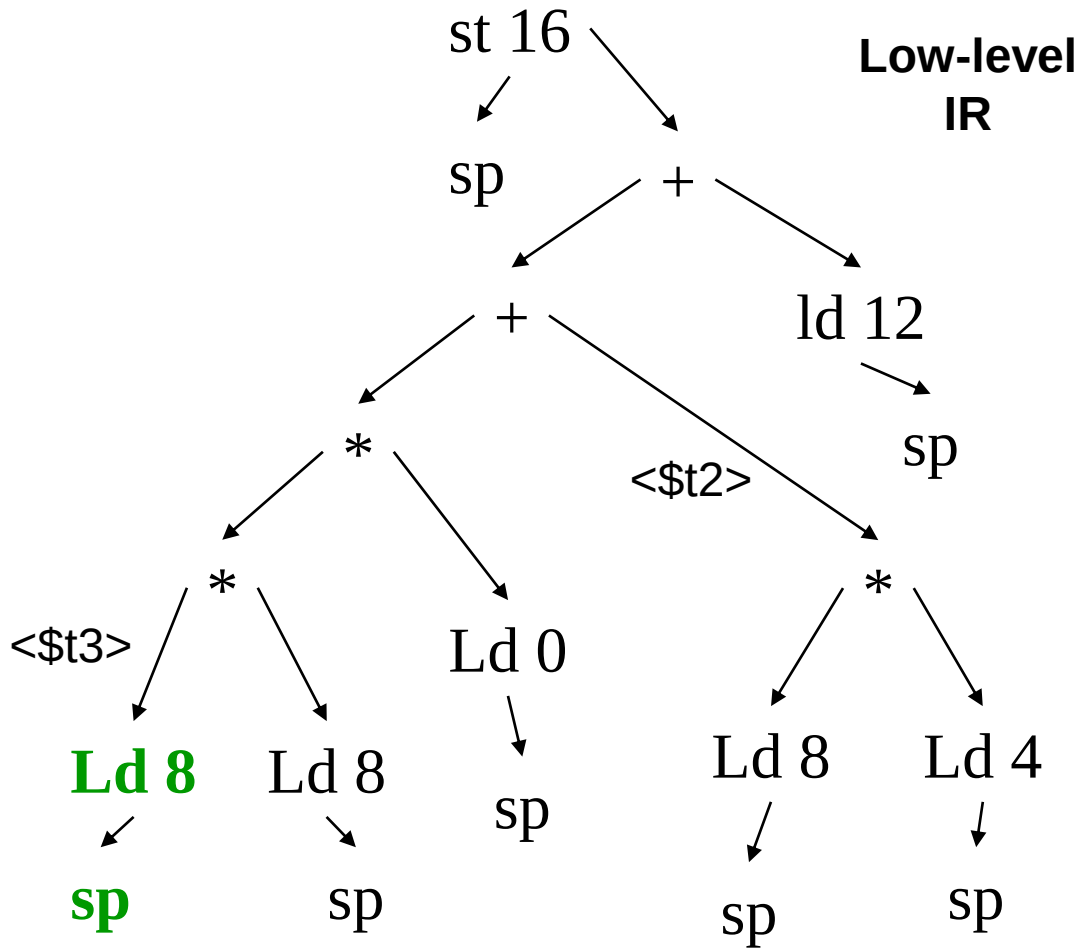


Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t2, $t1, $t0
lw $t3, 8($sp)

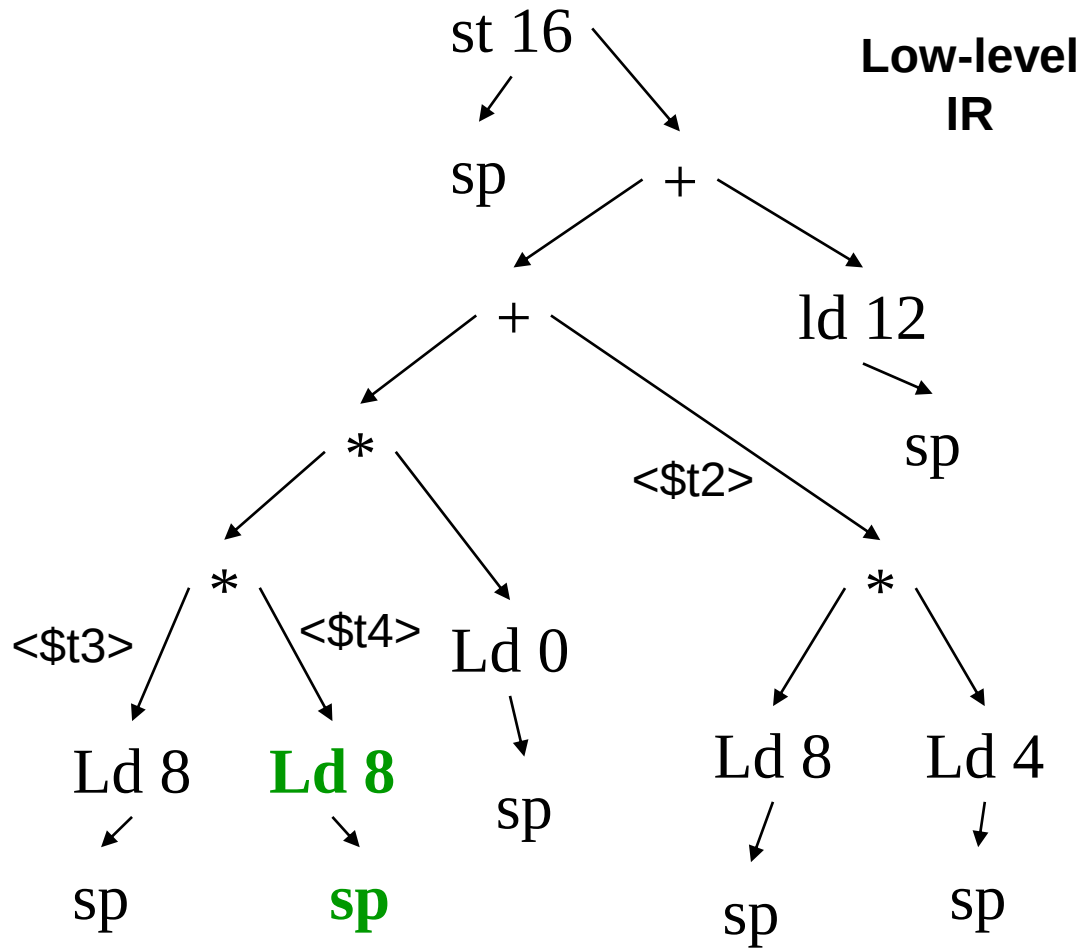


Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t2, $t1, $t0
lw $t3, 8($sp)
lw $t4, 8($sp)

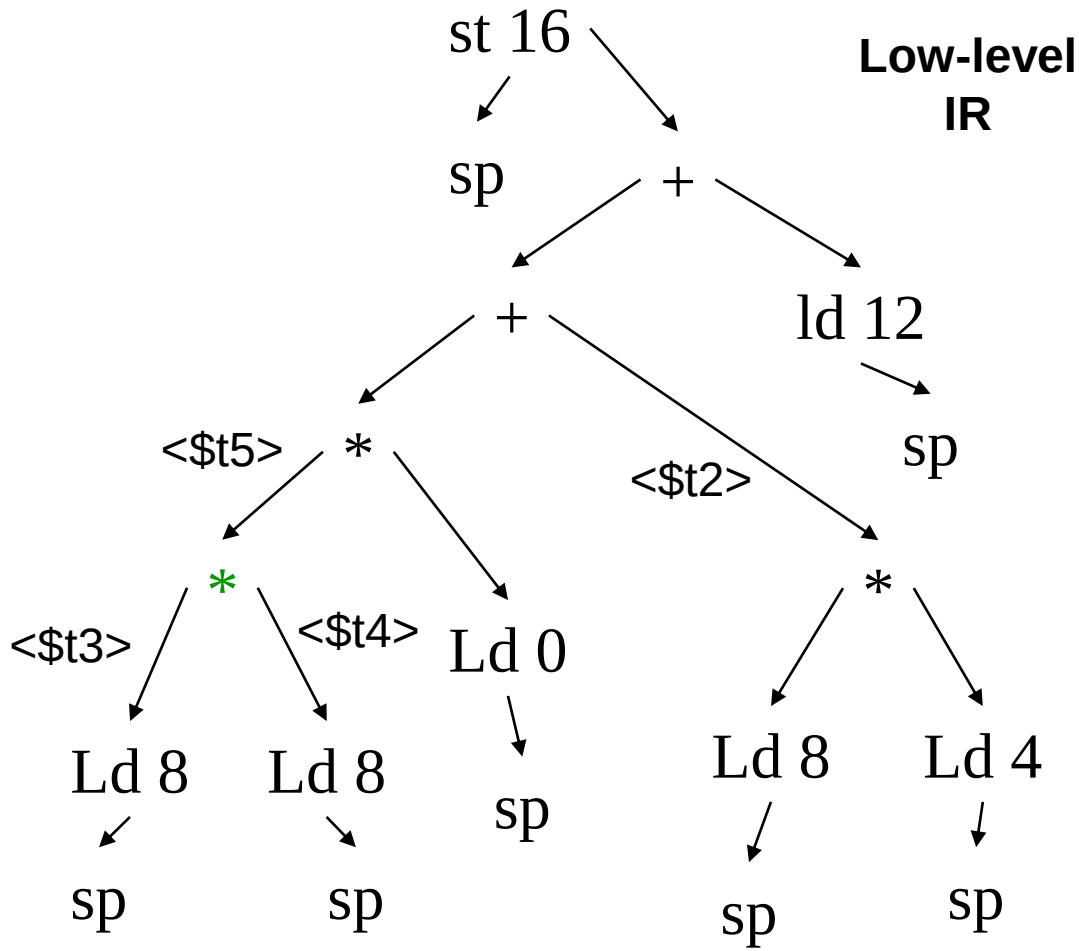


Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t2, $t1, $t0
lw $t3, 8($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4

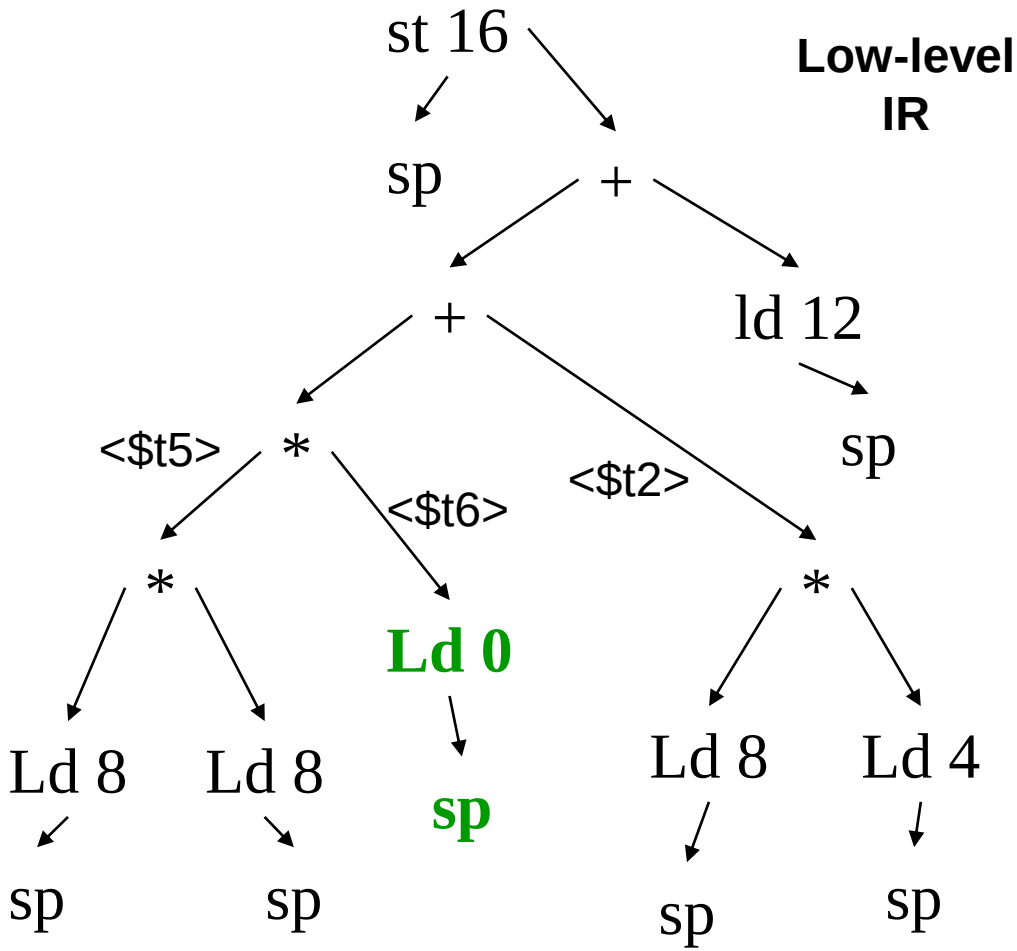


Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t2, $t1, $t0
lw $t3, 8($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 0($sp)

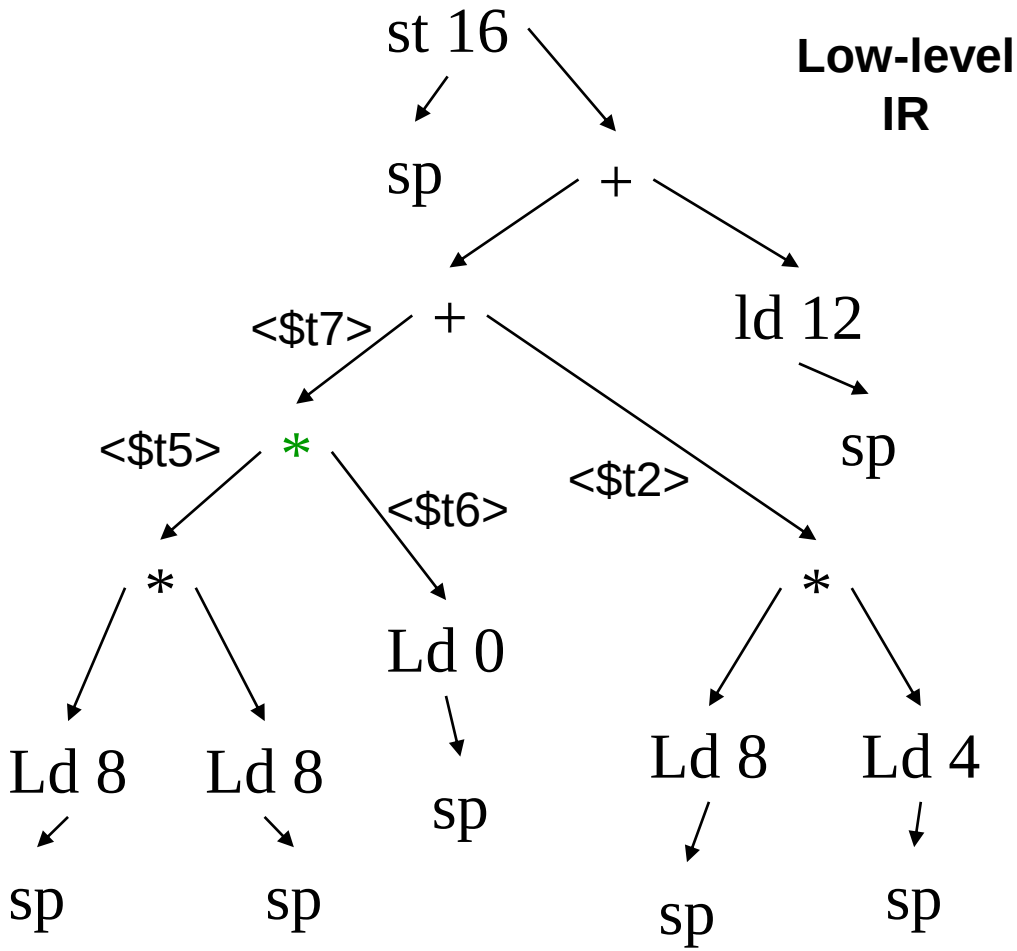


Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t2, $t1, $t0
lw $t3, 8($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 0($sp)
mult $t7, $t5, $t6

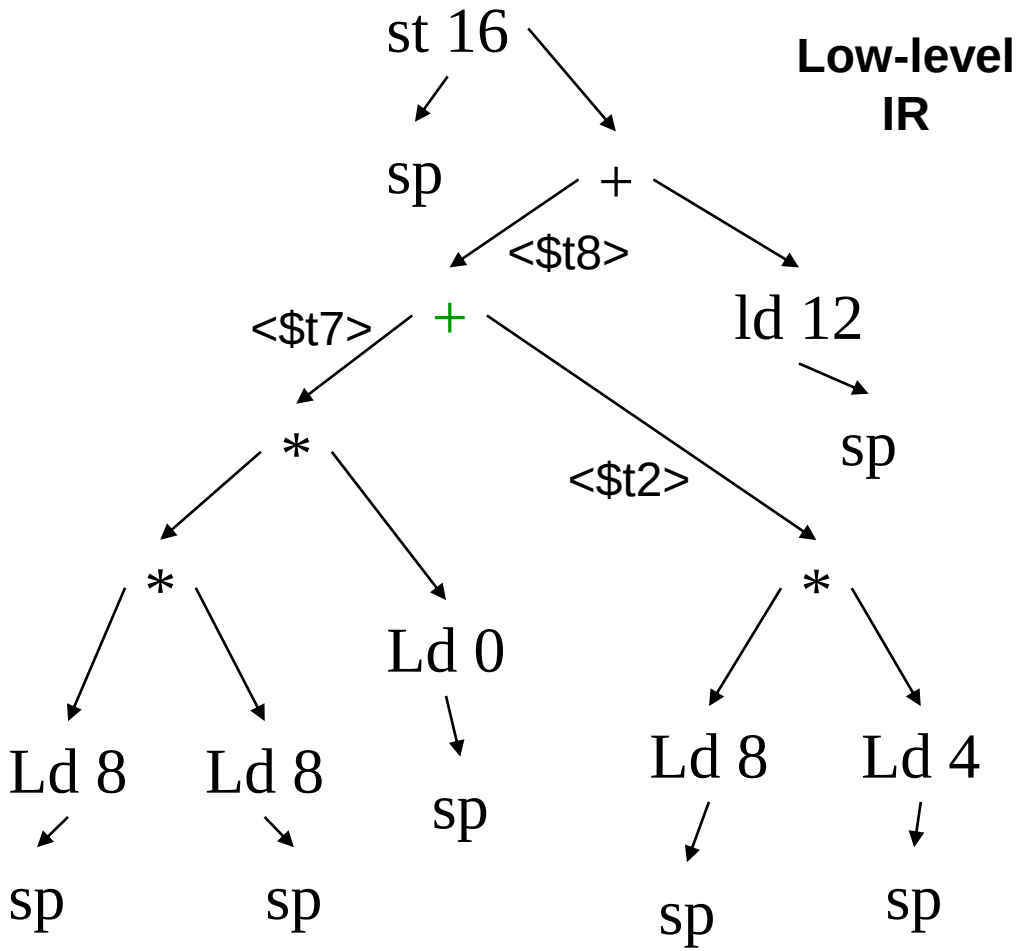


Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t2, $t1, $t0
lw $t3, 8($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 0($sp)
mult $t7, $t5, $t6
Add $t8, $t7, $t2



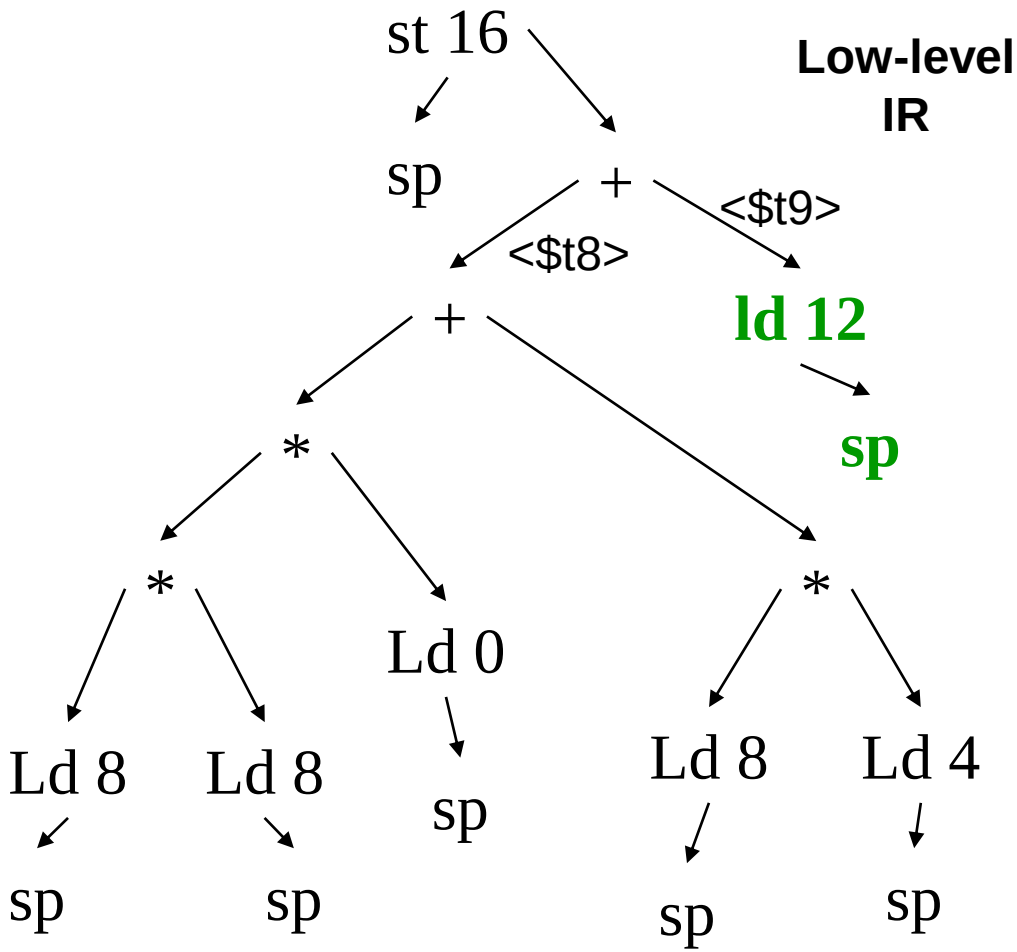
Low-level IR

Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

```
lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t2, $t1, $t0
lw $t3, 8($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 0($sp)
mult $t7, $t5, $t6
Add $t8, $t7, $t2
lw $t9, 12($sp)
```

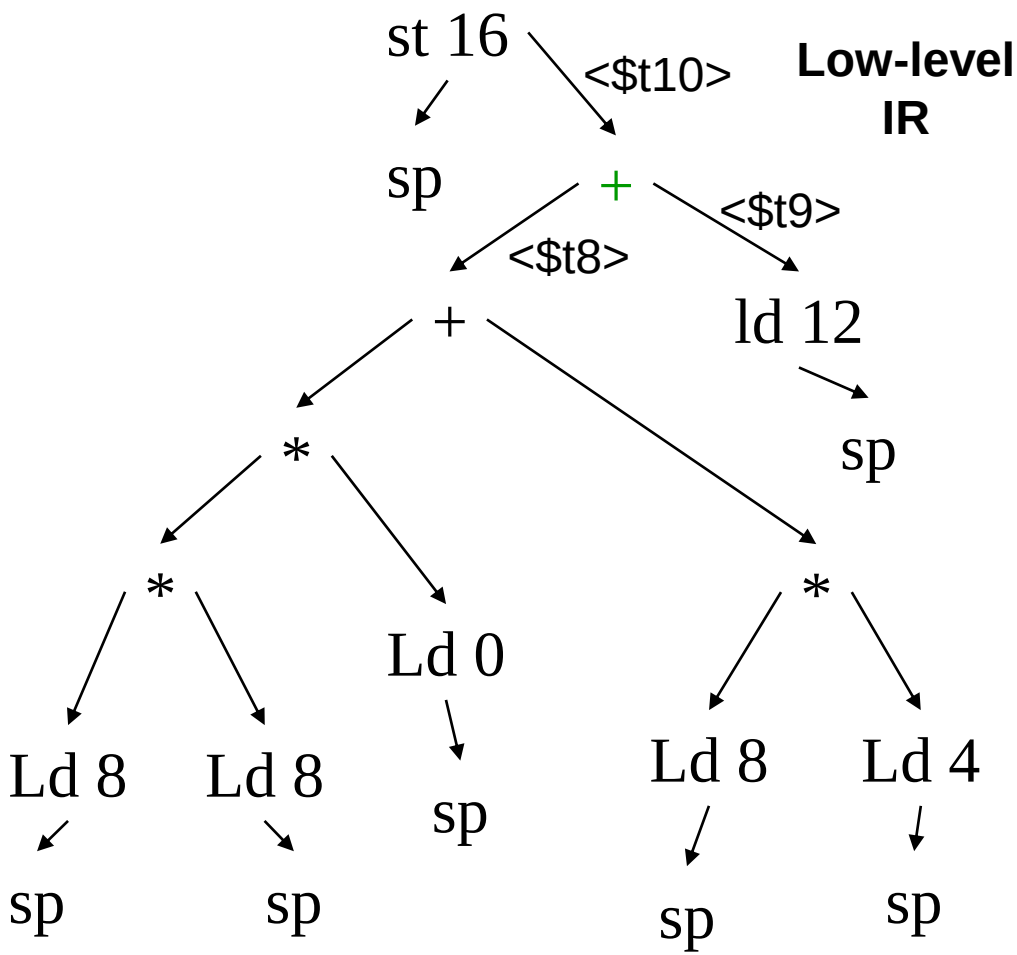


Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

```
lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t2, $t1, $t0
lw $t3, 8($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 0($sp)
mult $t7, $t5, $t6
Add $t8, $t7, $t2
lw $t9, 12($sp)
Add $t10, $t8, $t9
```

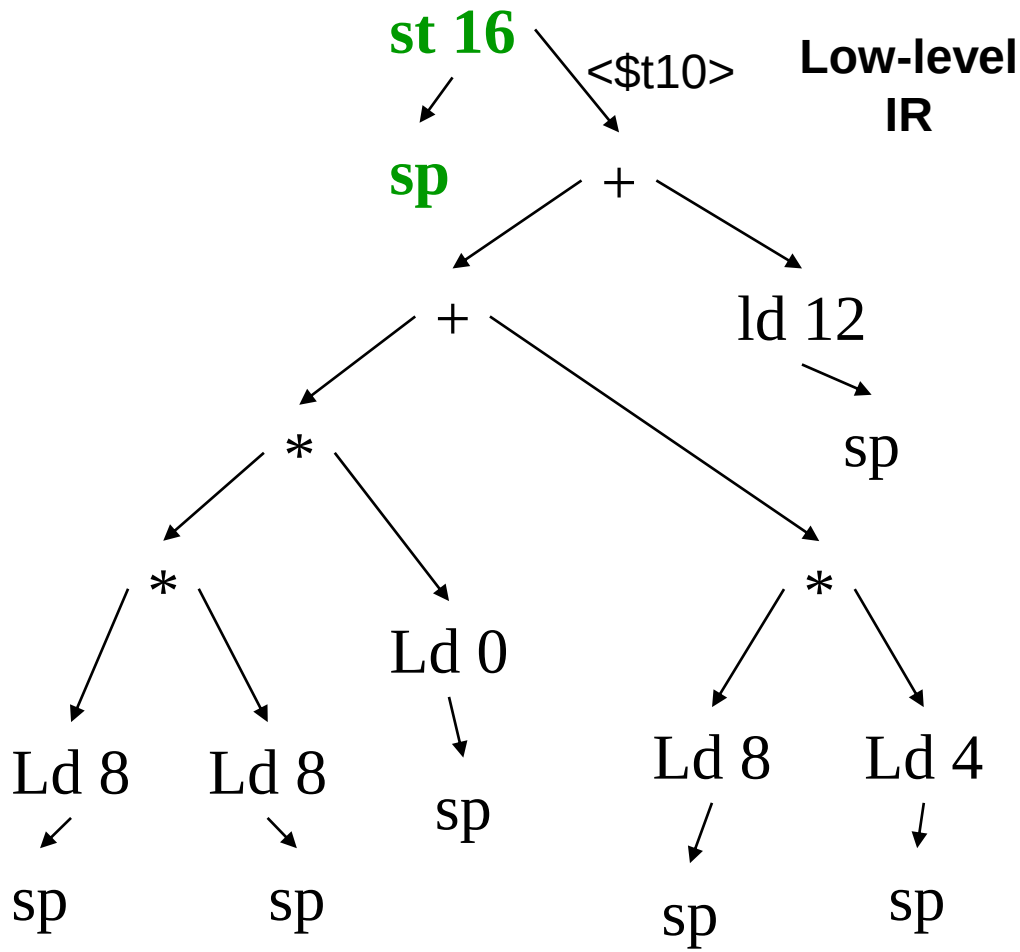


Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

```
lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t2, $t1, $t0
lw $t3, 8($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 0($sp)
mult $t7, $t5, $t6
Add $t8, $t7, $t2
lw $t9, 12($sp)
Add $t10, $t8, $t9
Sw  $t10, 16($sp)
```



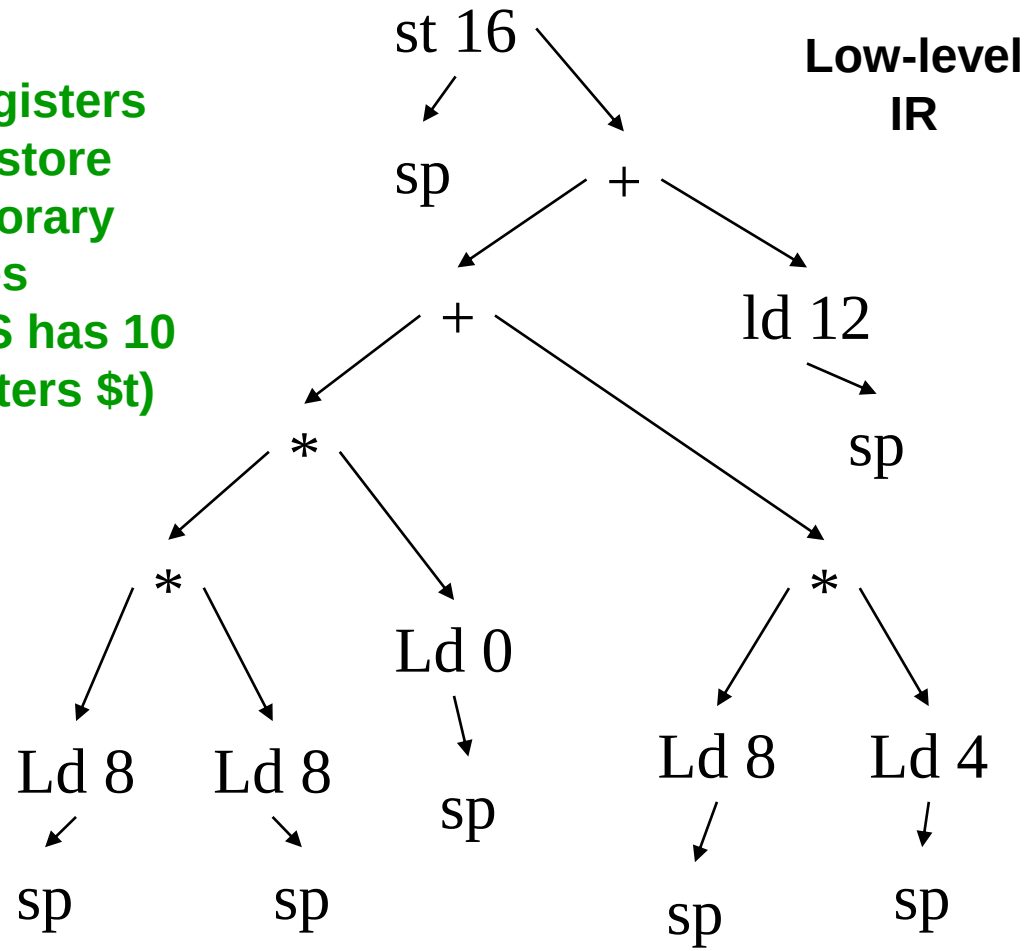
Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

```
lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t2, $t1, $t0
lw $t3, 8($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 0($sp)
mult $t7, $t5, $t6
add $t8, $t7, $t2
lw $t9, 12($sp)
add $t10, $t8, $t9
sw  $t10, 16($sp)
```

11 registers $t to store temporary values (MIPS has 10 registers $t)



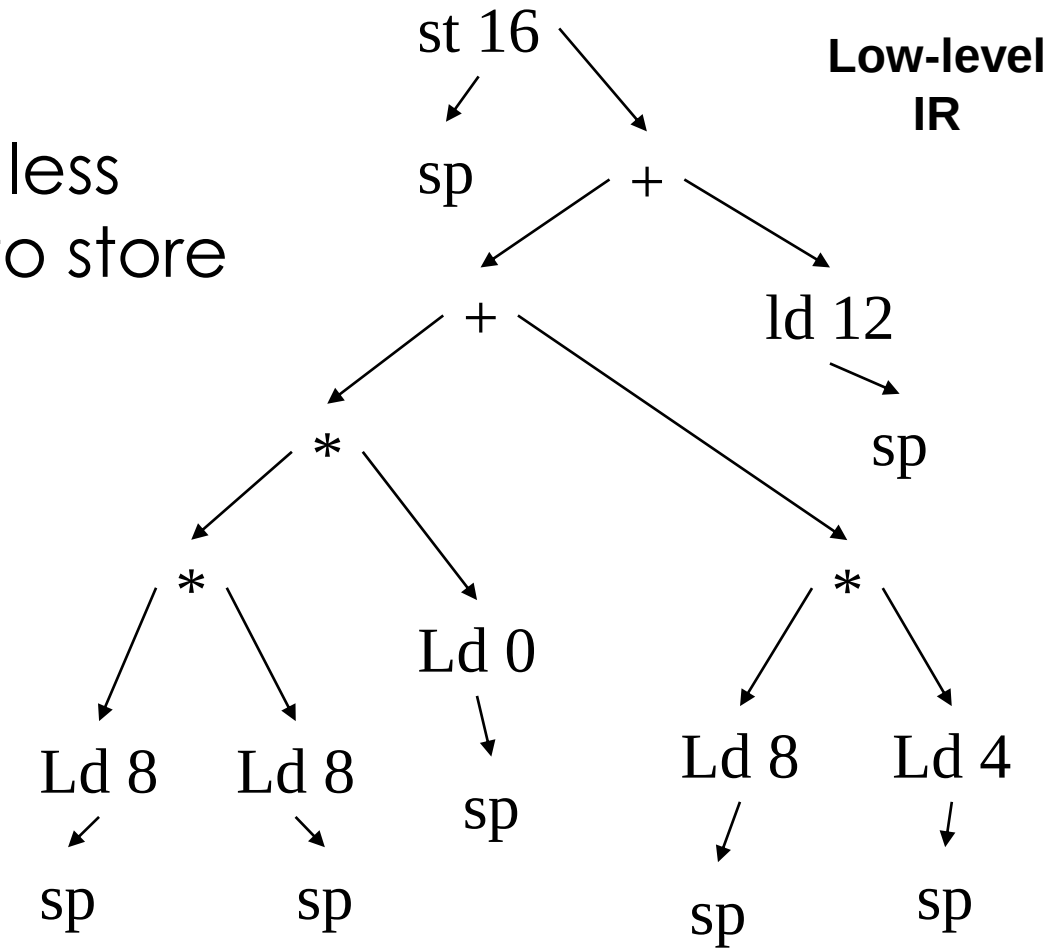
Low-level IR

Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

Solution using less registers $t to store temporary values?



Low-level IR

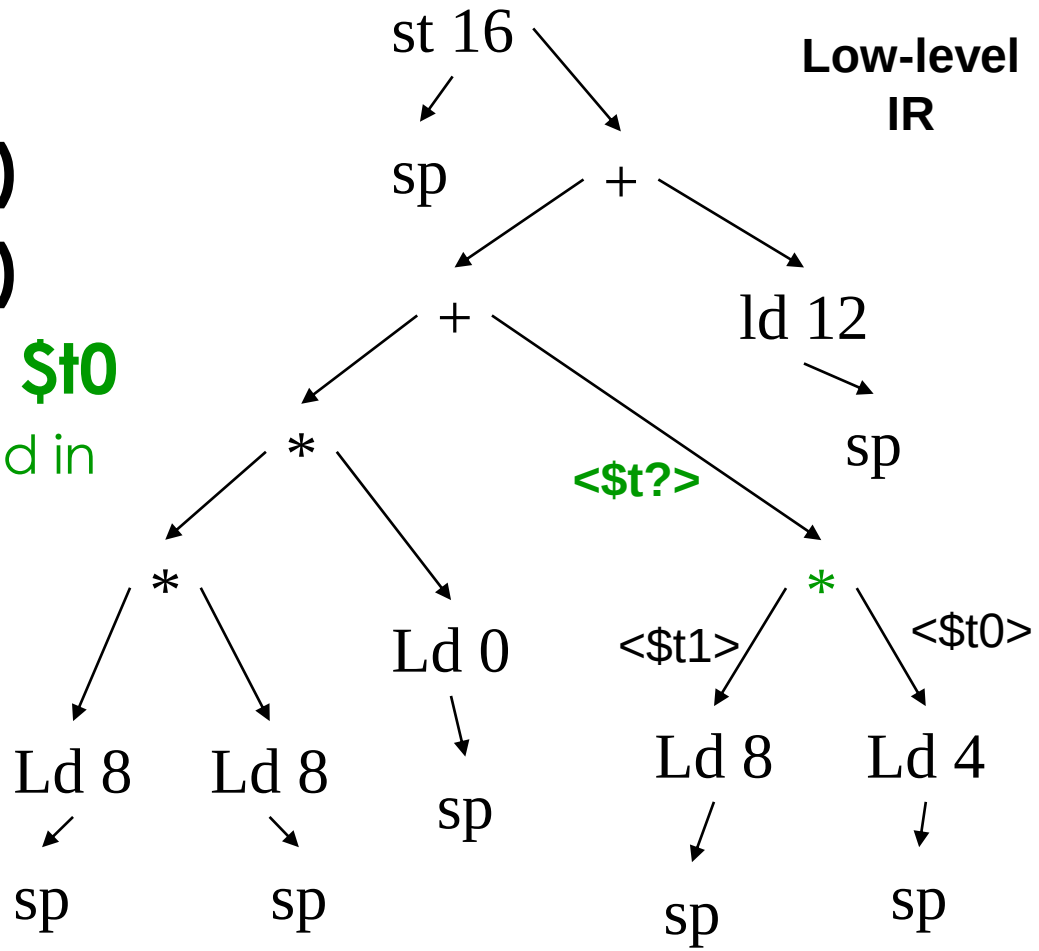
Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

- y=a*x*x+b*x+c;

- **lw $t0, 4($sp)**
- **lw $t1, 8($sp)**
- **mult $t?, $t1, $t0**

Result can be stored in
$t1 or in $t0

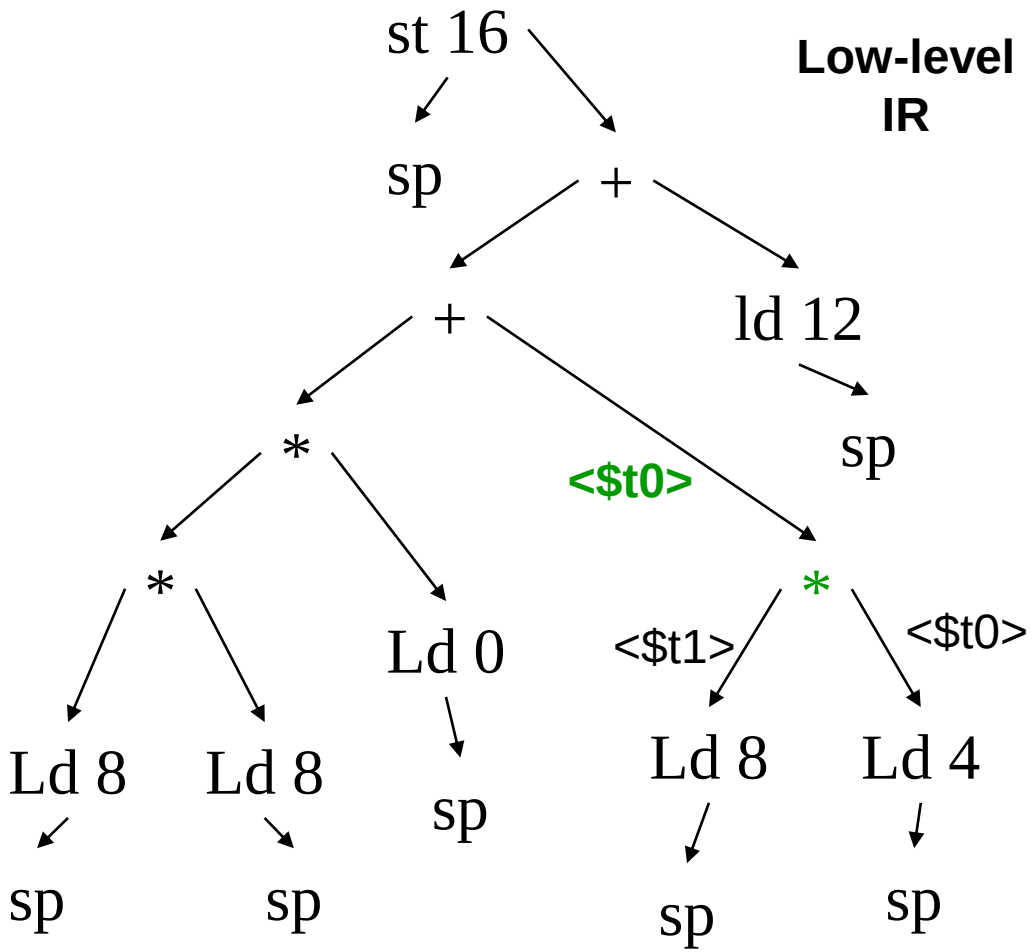


Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

- y=a*x*x+b*x+c;

- **lw $t0, 4($sp)**
- **lw $t1, 8($sp)**
- **mult $t0, $t1, $t0**
- **…**



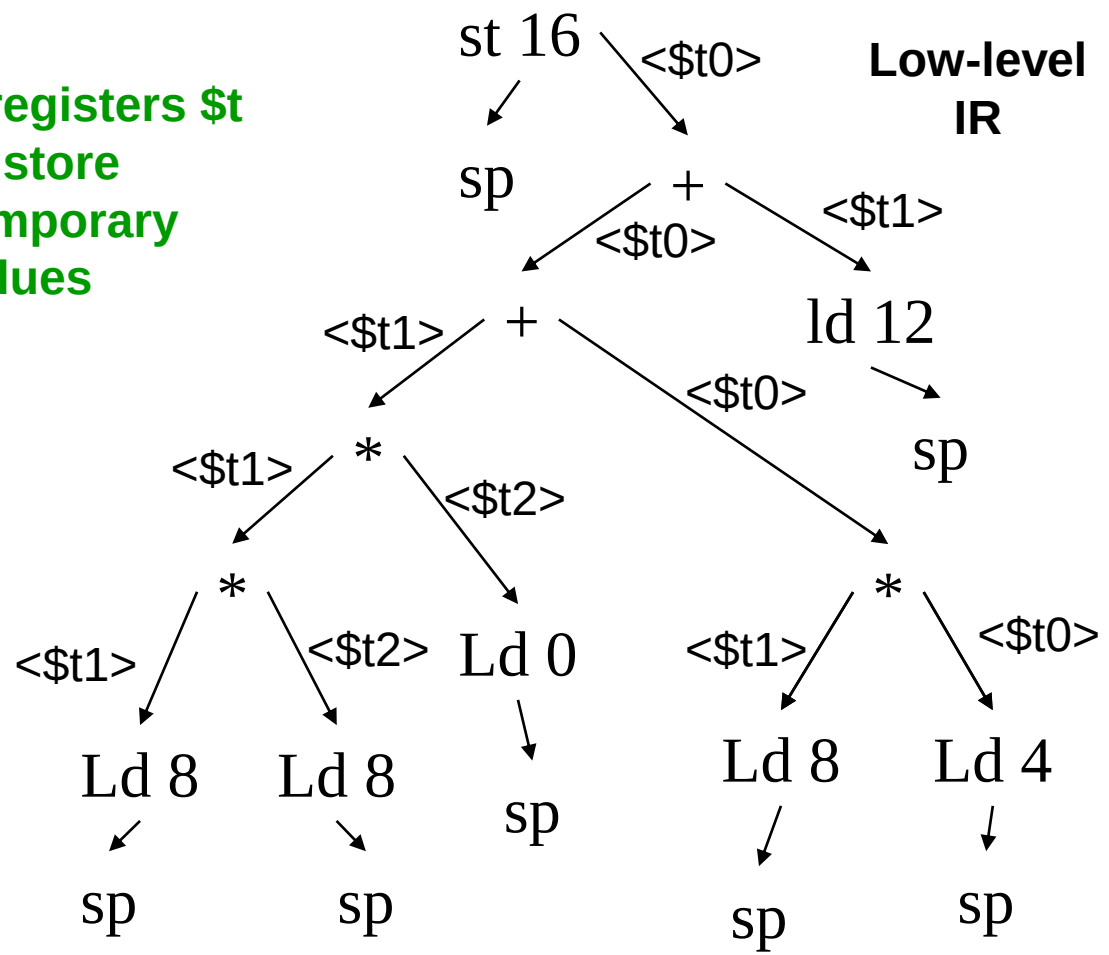
Final Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

- y=a*x*x+b*x+c;

```
lw $t0, 4($sp)
lw $t1, 8($sp)
mult $t0, $t1, $t0
lw $t1, 8($sp)
lw $t2, 8($sp)
mult $t1, $t1, $t2
lw $t2, 0($sp)
mult $t1, $t1, $t2
add $t0, $t1, $t0
lw $t1, 12($sp)
add $t0, $t0, $t1
sw  $t0, 16($sp)
```

3 registers $t to store temporary values



Code Generation Sequence

- Tree-based IR
  - Sethi-Ullman Algorithm (see Dragon, Tiger, Books)
- DAG-based IR
- See slides from José Nelson Amaral:
  - http://www.cs.ualberta.ca/~amaral/courses/680/webslides/T6-CodeGeneration/index.htm

Not Optimized

- Stack accesses require more clock cycles than accesses to internal microprocessor registers
- Utilization of the stack to all the local variables requires more instructions

Code Generation

- Use of *templates* to generate assembly code for:
  - If-then and if-then-else constructs
  - Loops

Code Generation

- Templates for If-then and if-then-else constructs

if (test)
true_body
else
false_body

```
        <do the test>
        boper …, lab_true
        <false_body>
        jump  lab_end
lab_true:
        <true_body>
lab_end:
```

Code Generation

- Templates for loops

for(stmt1;test;stmt2)
body

```
            <stmt1>
lab_init:   <test>
            boper …, lab_true
            jump lab_end
lab_true:
            <body>
            <stmt2>
            jump lab_init
lab_end:
```

Code Generation

- Templates for loops

for(stmt1;test;stmt2)
body

```
                <stmt1>
lab_init:       <test>
                bnoper …, lab_end
                <body>
                <stmt2>
                jump lab_init
lab_end:
```

Code Generation

- Templates for loops

for(stmt1;test;stmt2)
body

```
            <stmt1>
            jump lab_init
lab_true:   <body>
            <stmt2>
lab_init:   <test>
            boper …, lab_true
lab_end:
```

Code Generation

- Sequence of instructions
  - It has impact on registers needed, stack depth, etc.
  - Dealing with pipelining needs instruction scheduling

Hints for Code Generator

- Go down slowly and step-by-step in the abstraction level: use the number of stages you need:
  - Even if each stage does only one thing!
  - Easier to debug, easier to handle the problems
- Mantain the abstraction level consistent
  - IR must mantain the semantic correct everytime!
  - One may need to do optimizations between stages
- Use *assertions*
  - An *assertion* to verify some condition that should apply
  - They help to find bugs

Hints for Code Generator

- Start doing simple code generation, even if naïf
  - Ok to generate: 0 + 1*x + 0*y
- The *runtime* library is our friend!
  - Do not try to generate assembly code when there exist library routines with the same functionality
  - Example: malloc

Hints for Code Generator

- Remember that the optimizations come later
  - It is the role of the optimizer to perform the optimizations
  - Think that the optimizer needs to restructure the code according to the portfolio of optimizations it integrates
  - Examples: register allocation, algebraic simplifications, constant propagations
- Use a good test infrastructure
  - Regressive Test
    - If a program originates a bug then add it to the test suite
  - Use makefiles
    - to execute automatically the compiler under development over the test suite and to verify if all of the examples in the test suite pass in the tests (it can imply the use of a simulator of the target architecture)
  - Use the best software engineering practices

Code Generation Example

TARGETING THE JVM

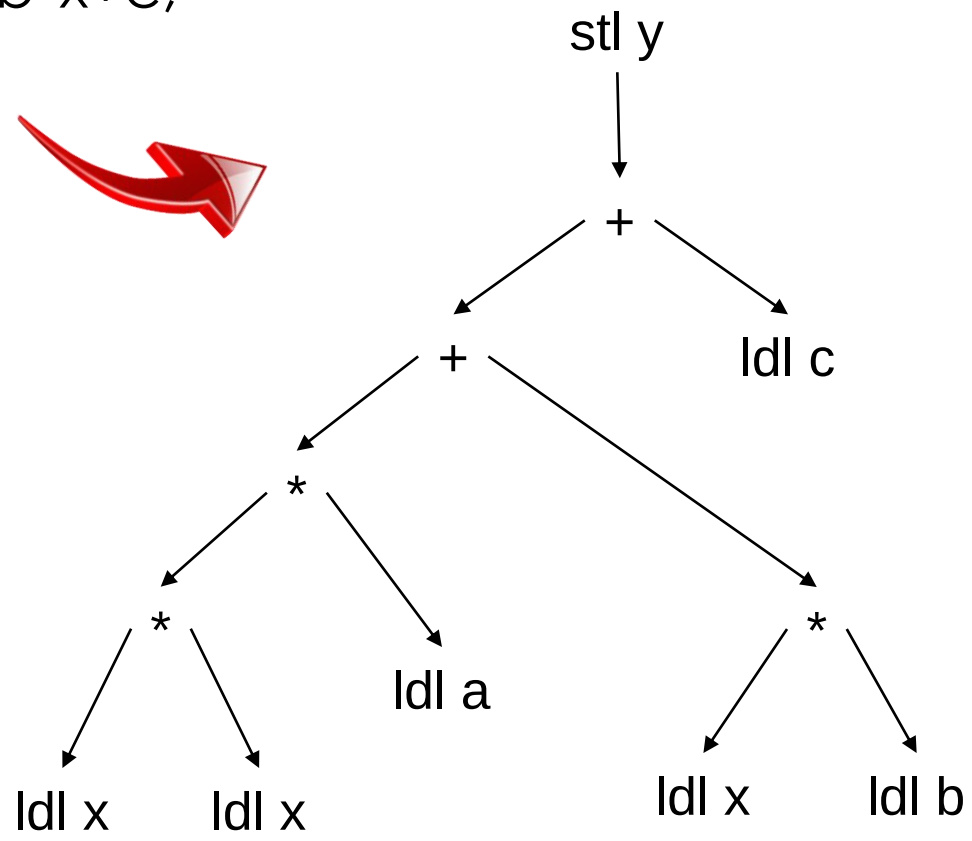
Targeting the JVM

- Instruction Selection does not require complex algorithms
- JVM instructions have very few tree patterns with overlapping
- We can use a Naive/Canonical Generation prioritizing the use of JVM instructions that cover the largest number of tree nodes
  - Example, considering that variable "i" is assigned to JVM local variable "3"

i++;	iload 3 iconst 1 iadd istore 3	iinc 3, 1

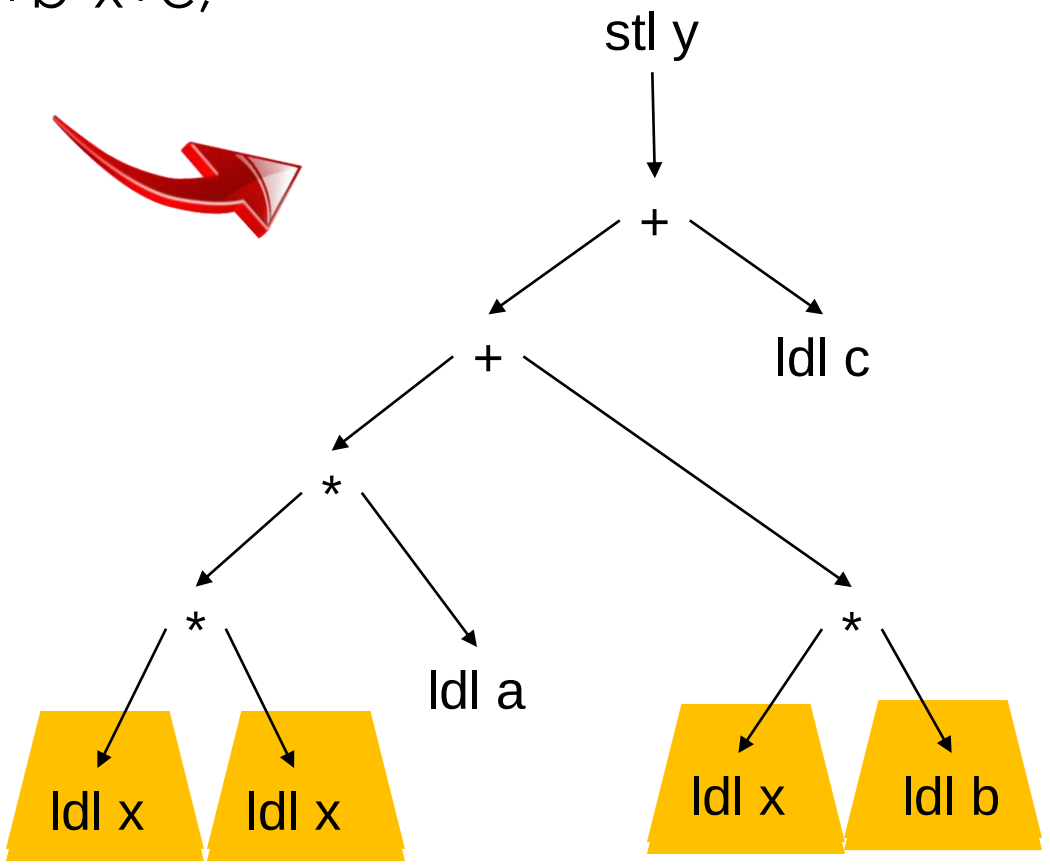
Targeting the JVM

- JVM
  - y=a*x*x+b*x+c;



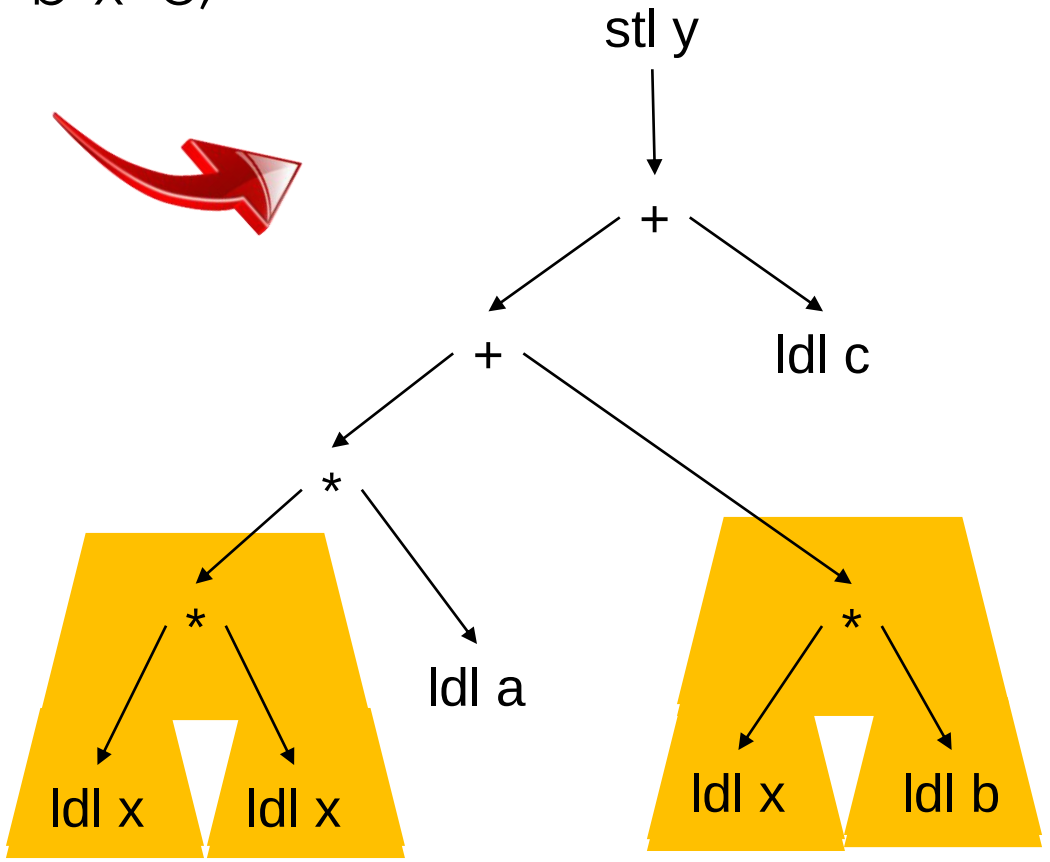
Targeting the JVM

- JVM
  - y=a*x*x+b*x+c;



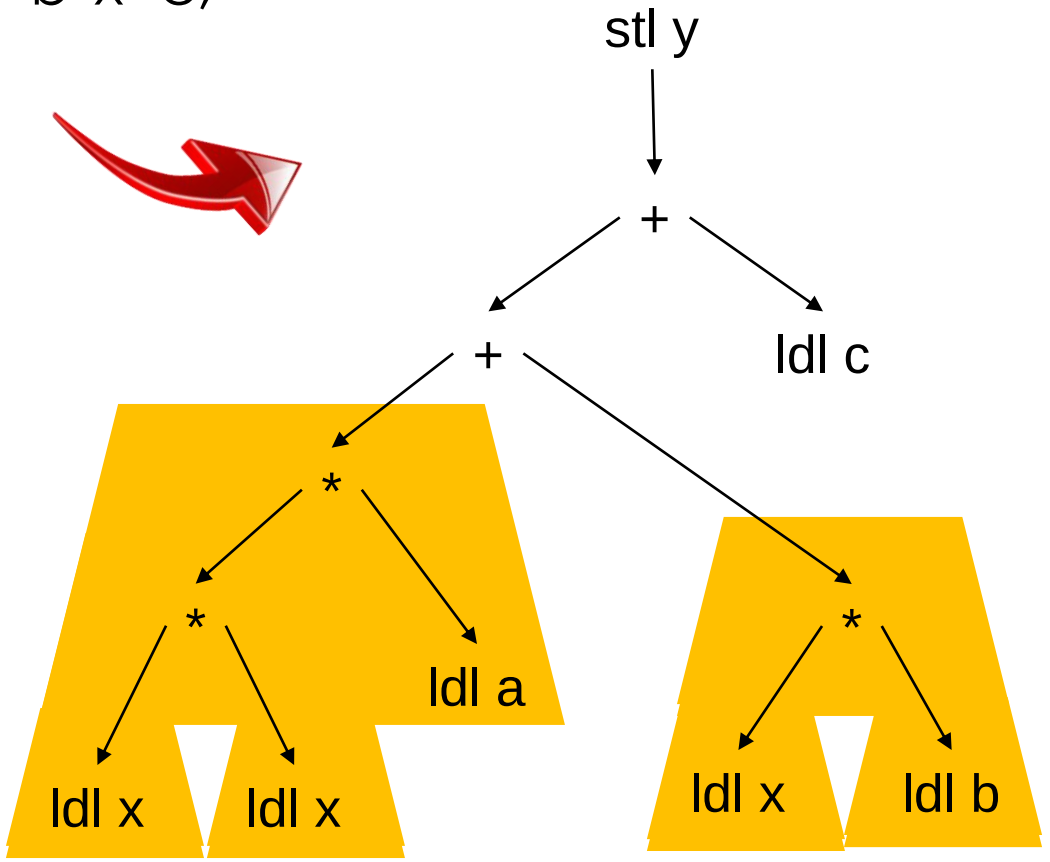
Targeting the JVM

- JVM
  - y=a*x*x+b*x+c;



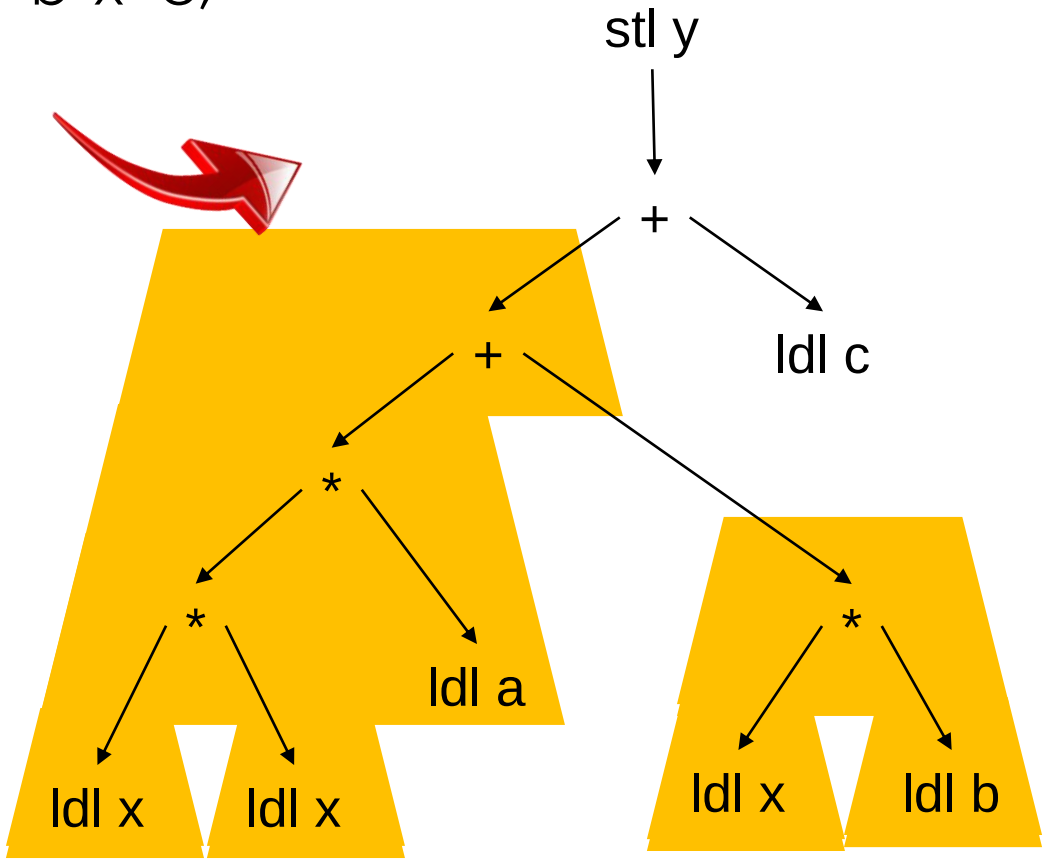
Targeting the JVM

- JVM
  - y=a*x*x+b*x+c;



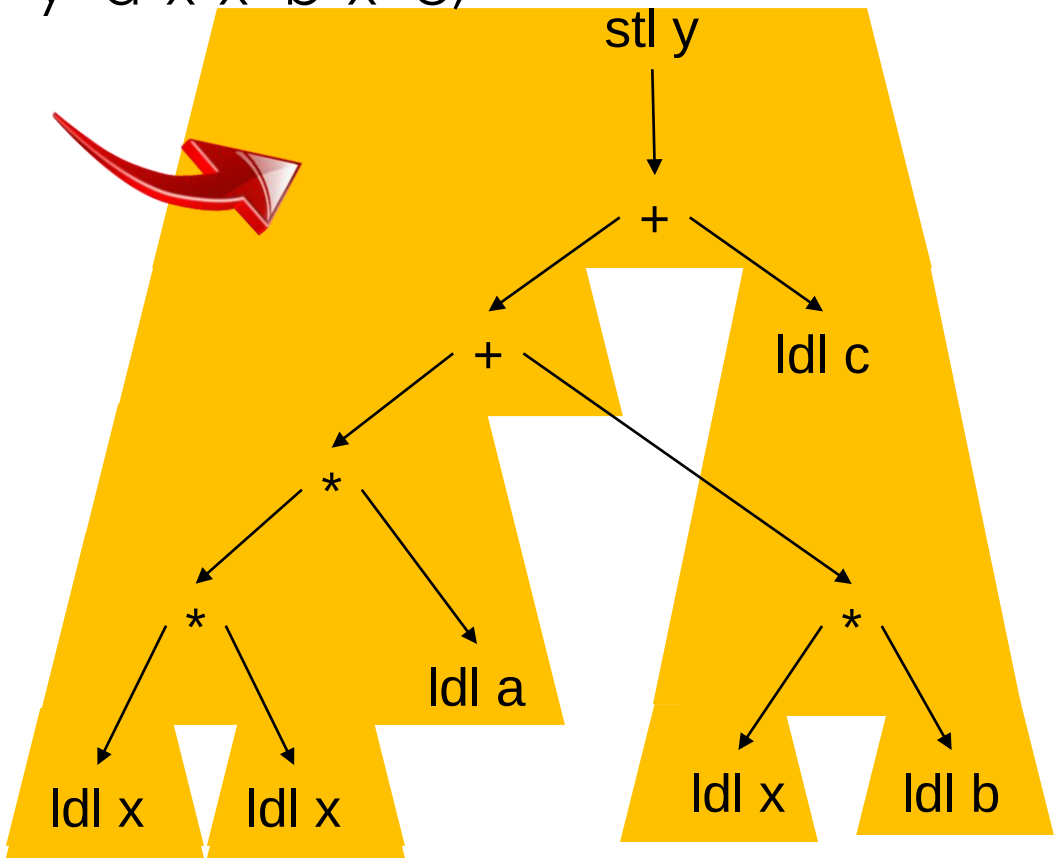
Targeting the JVM

- JVM
  - y=a*x*x+b*x+c;



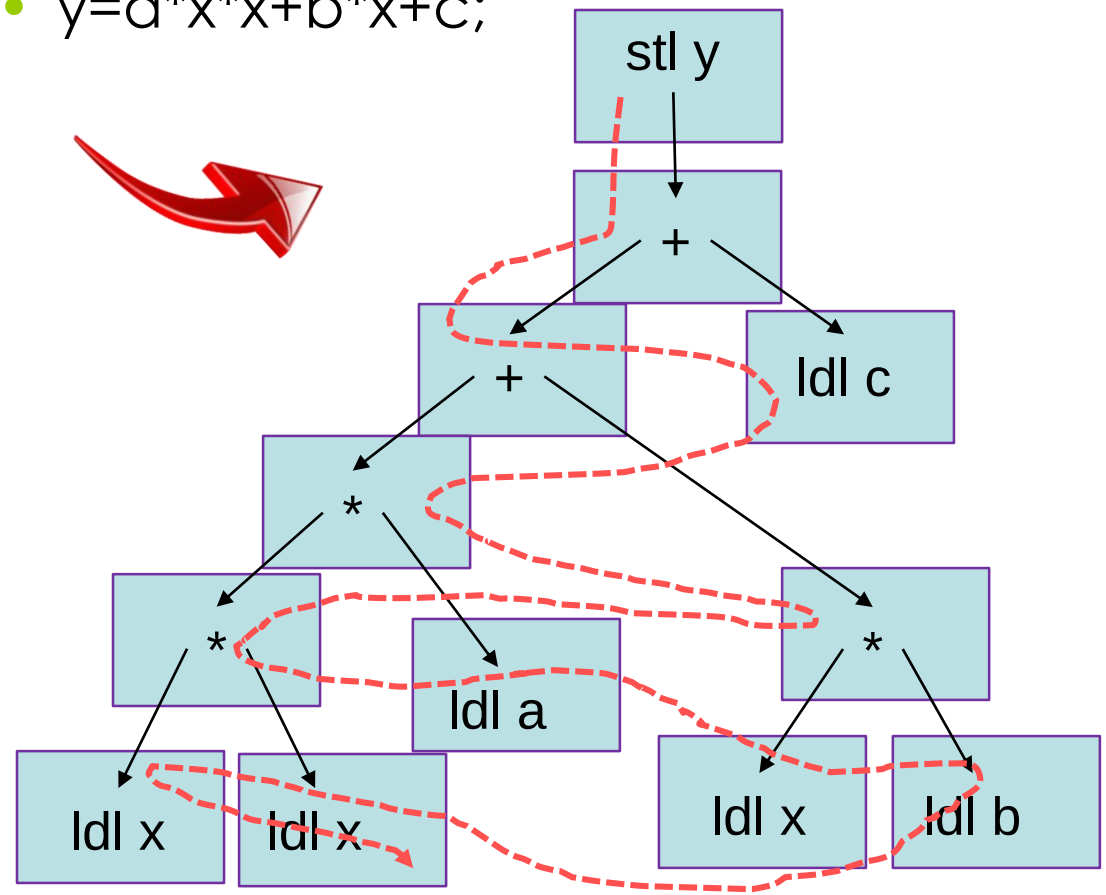
Targeting the JVM

- JVM
  - y=a*x*x+b*x+c;



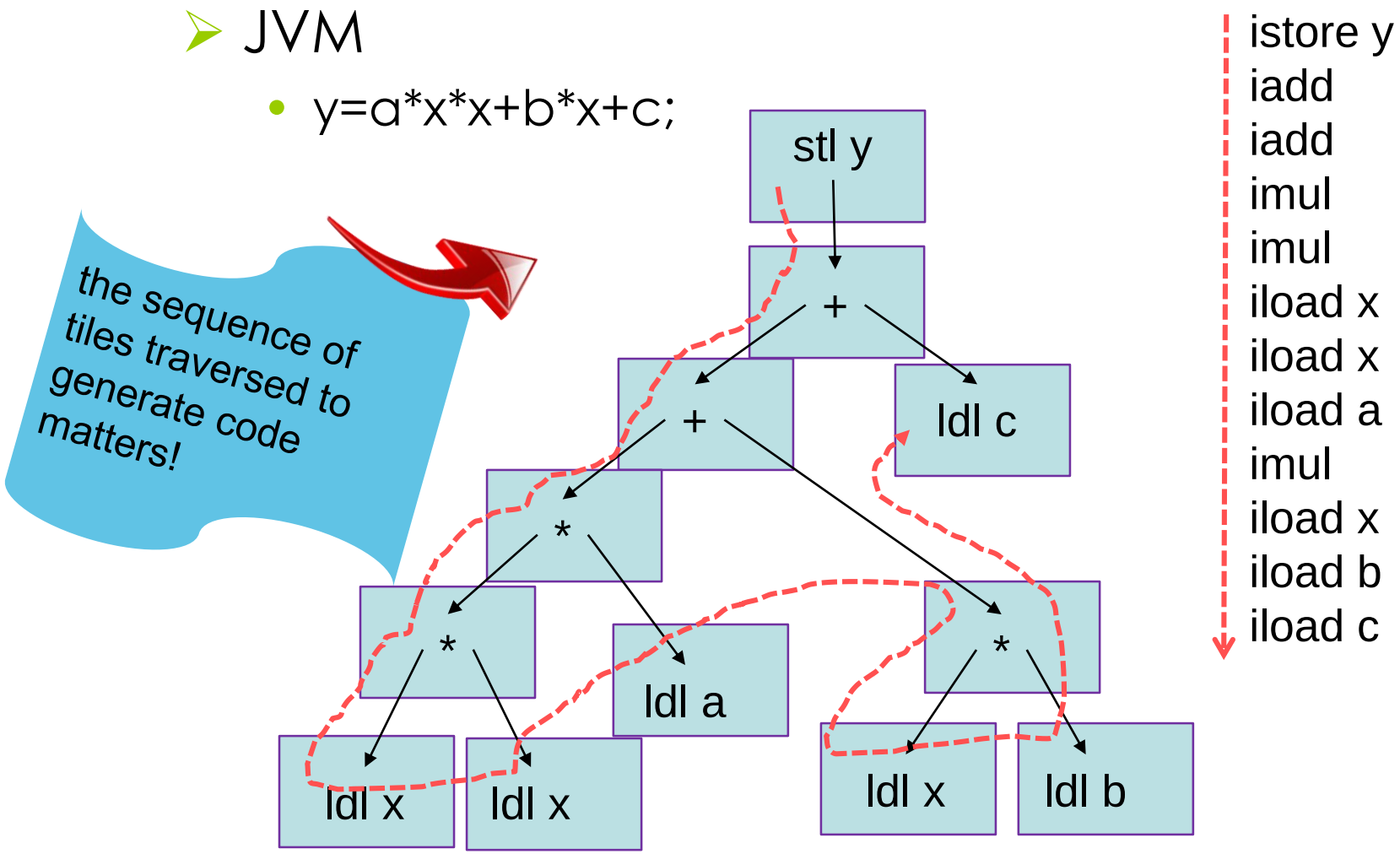
Targeting the JVM

- JVM
  - y=a*x*x+b*x+c;



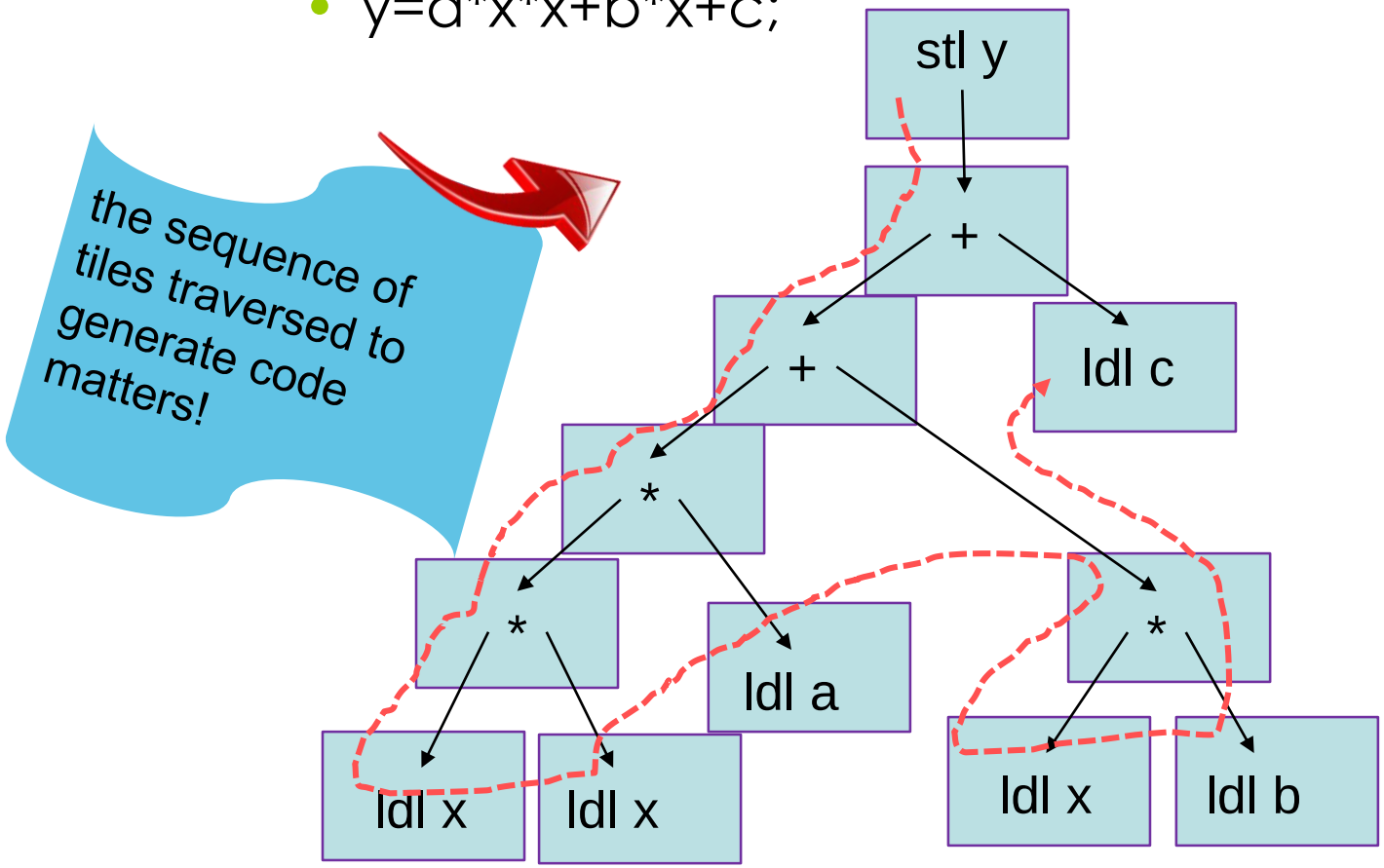
Targeting the JVM

- JVM
  - y=a*x*x+b*x+c;



Targeting the JVM

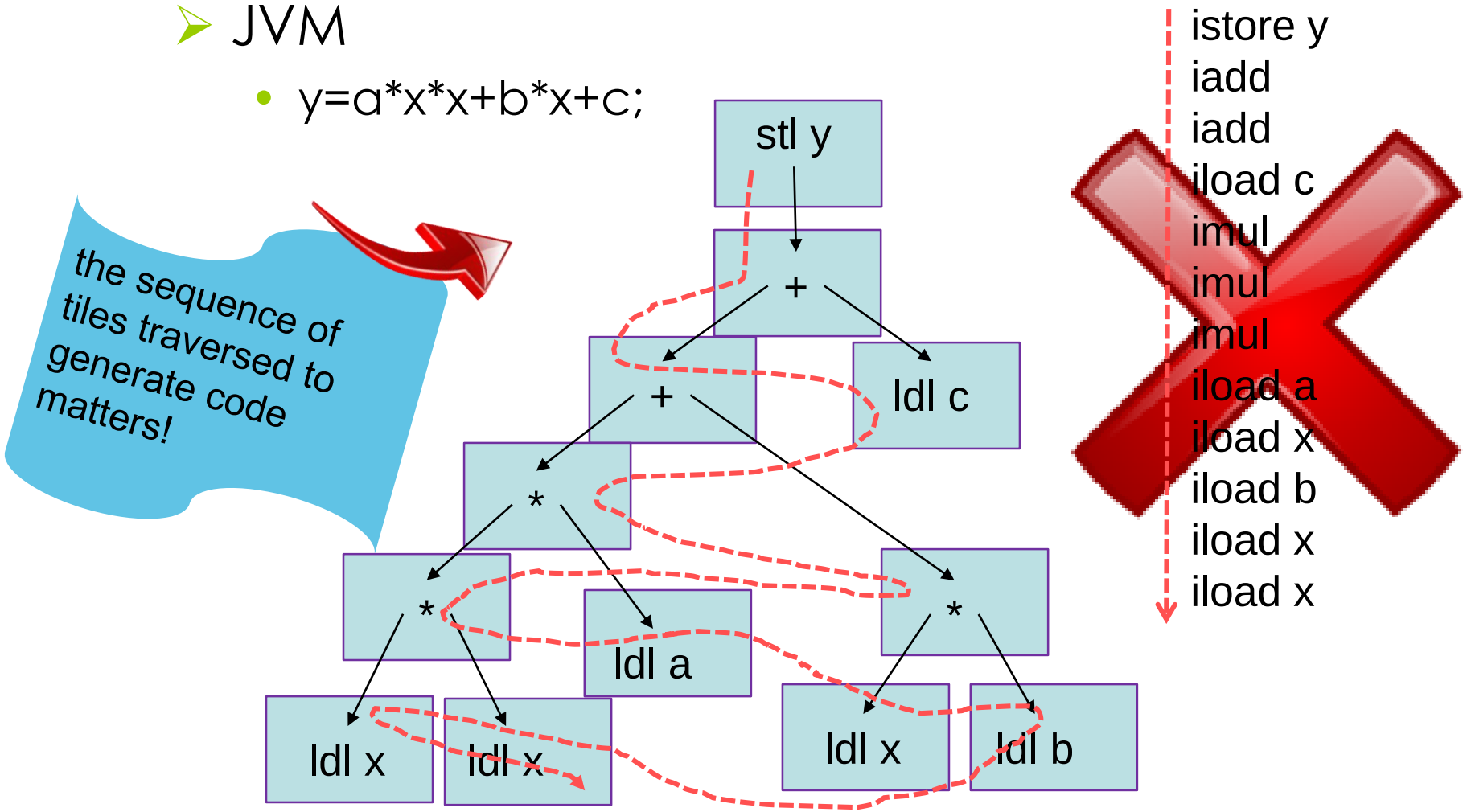
- JVM
  - y=a*x*x+b*x+c;




```
iload c
iload b
iload x
imul
iload a
iload x
iload x
imul
imul
iadd
iadd
istore y
```

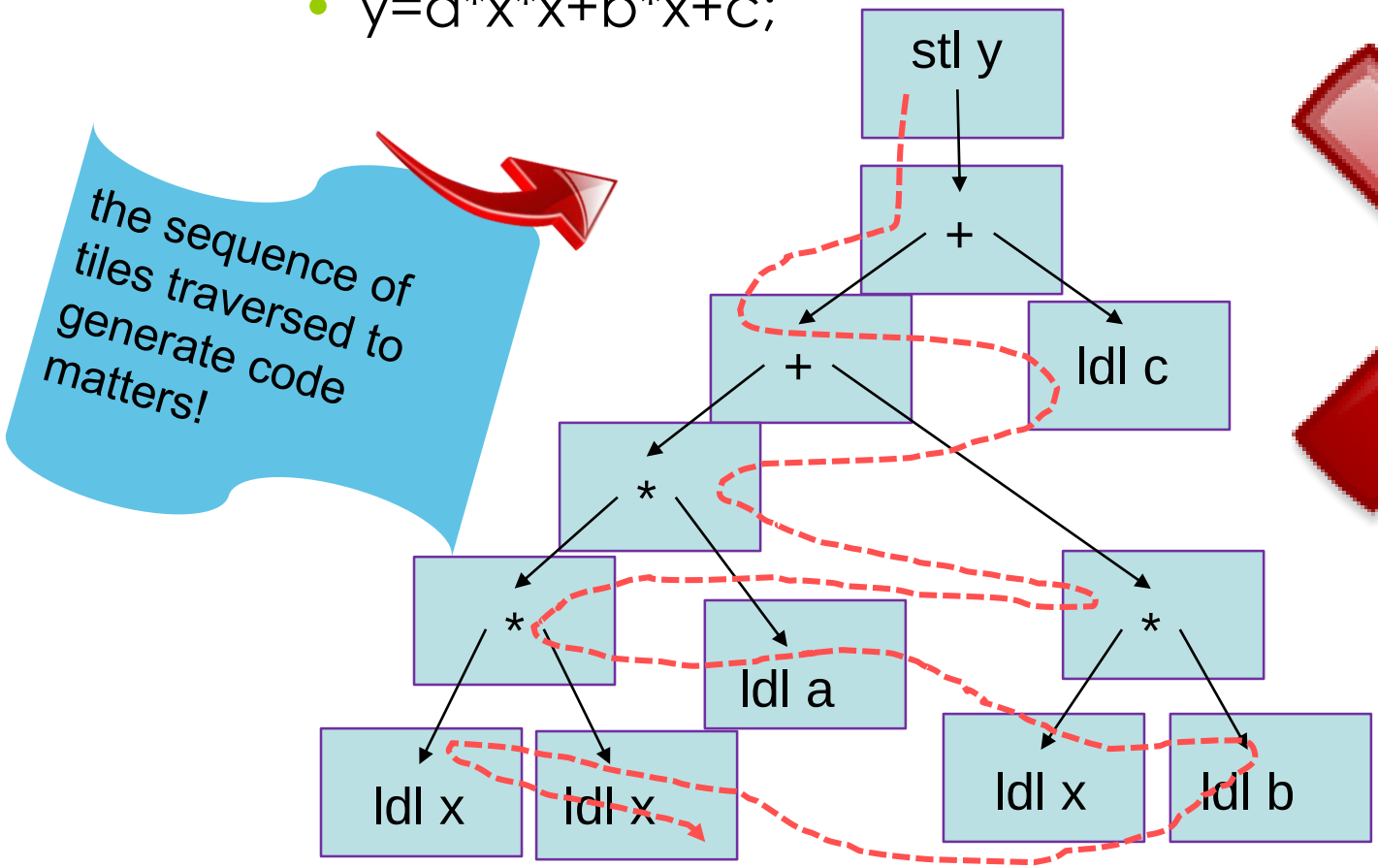
Targeting the JVM

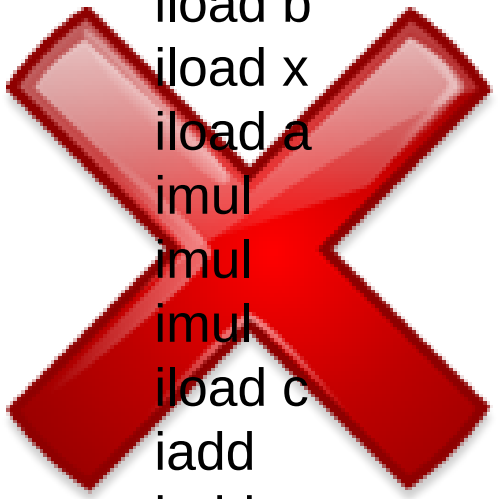
- JVM
  - y=a*x*x+b*x+c;



Targeting the JVM

- JVM
  - y=a*x*x+b*x+c;

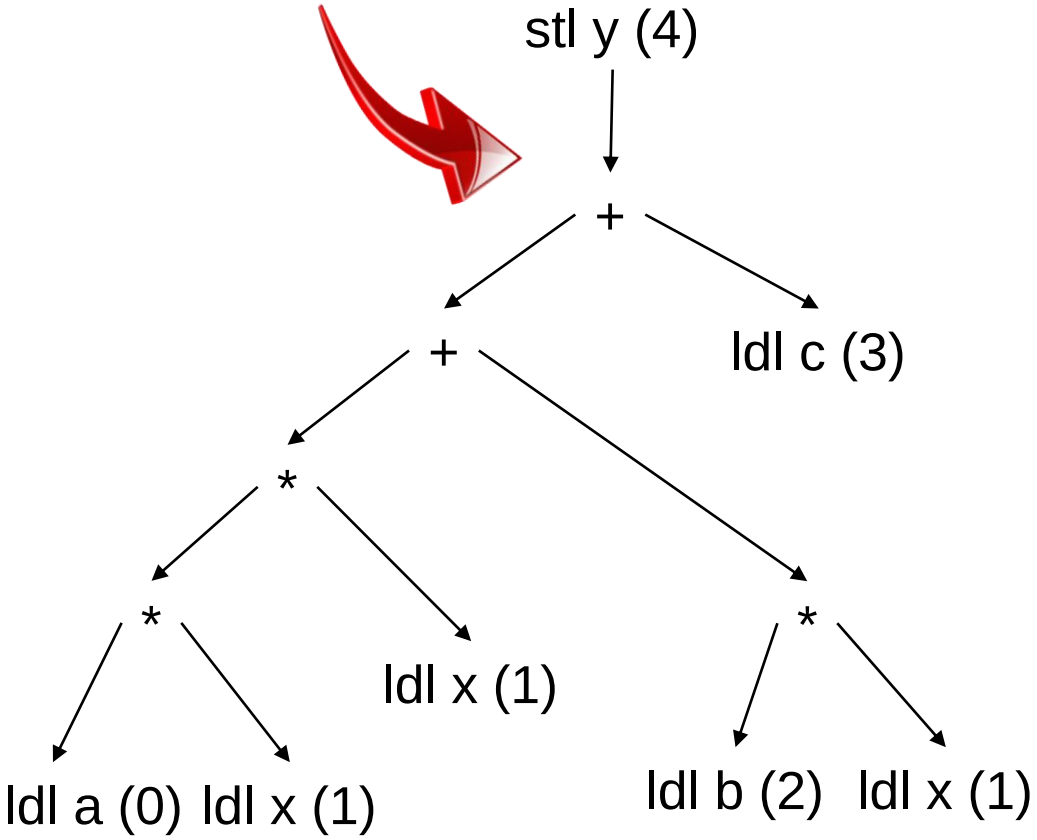




```
iload x
iload x
iload b
iload x
iload a
imul
imul
imul
iload c
iadd
iadd
istore y
```

Targeting the JVM

- JVM
  - y=a*x*x+b*x+c;



Javac generates (stack depth=3):

```
iload_0
iload_1
imul
iload_1
imul
iload_2
iload_1
imul
iadd
iload_3
iadd
istore  4
```

Targeting the JVM

- Uses an operand stack and a table of local variables
- Content on top positions of operand stack must be according to the needed operands foreach operation
  - E.g., iadd requires the top two operands on the stack