

# Compiler Construction: Practical Introduction

## Lecture 10 Compilation for Virtual Machines

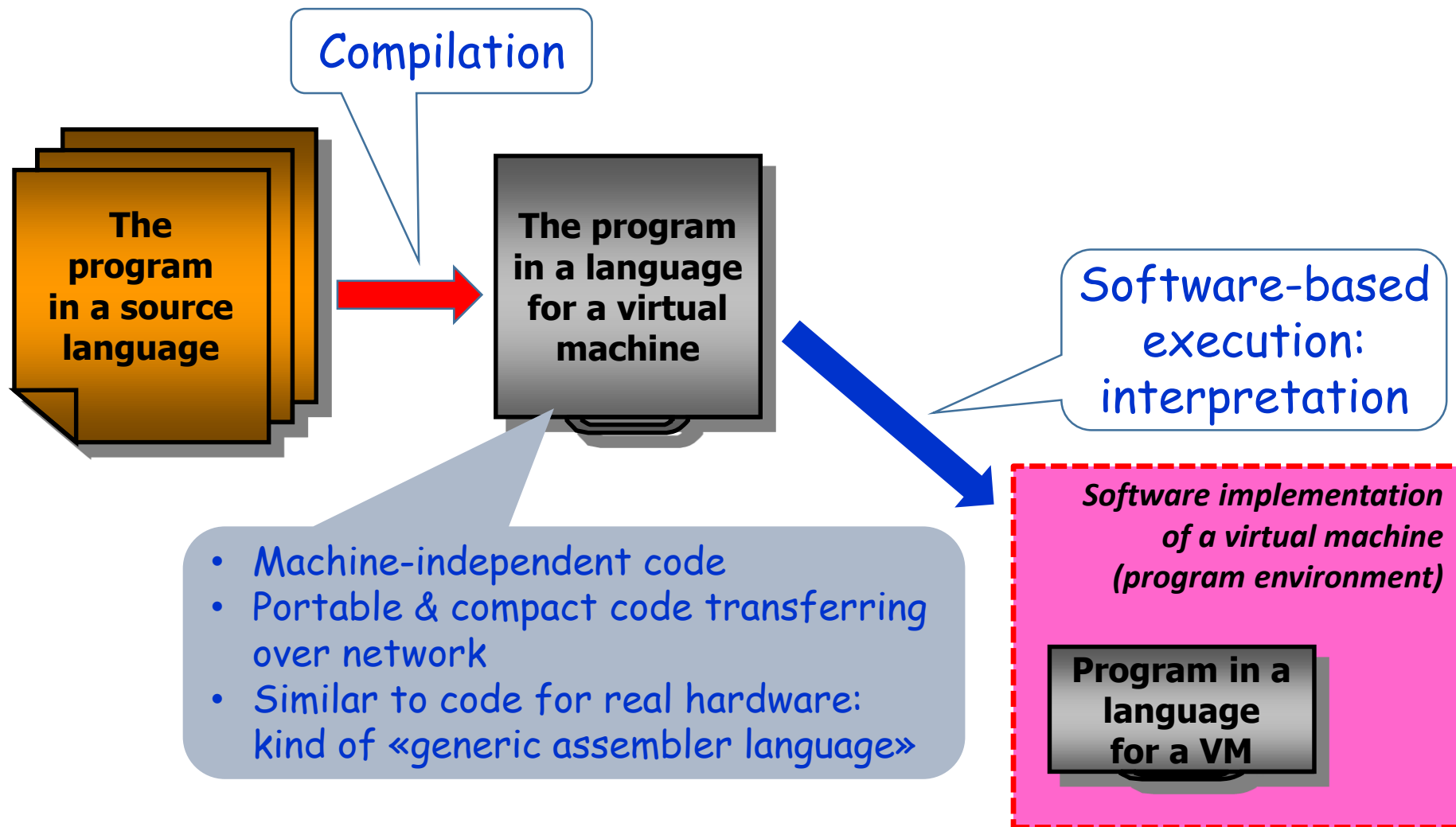
Eugene Zouev  
Spring Semester 2022  
Innopolis University

# Virtual Machine: The Idea

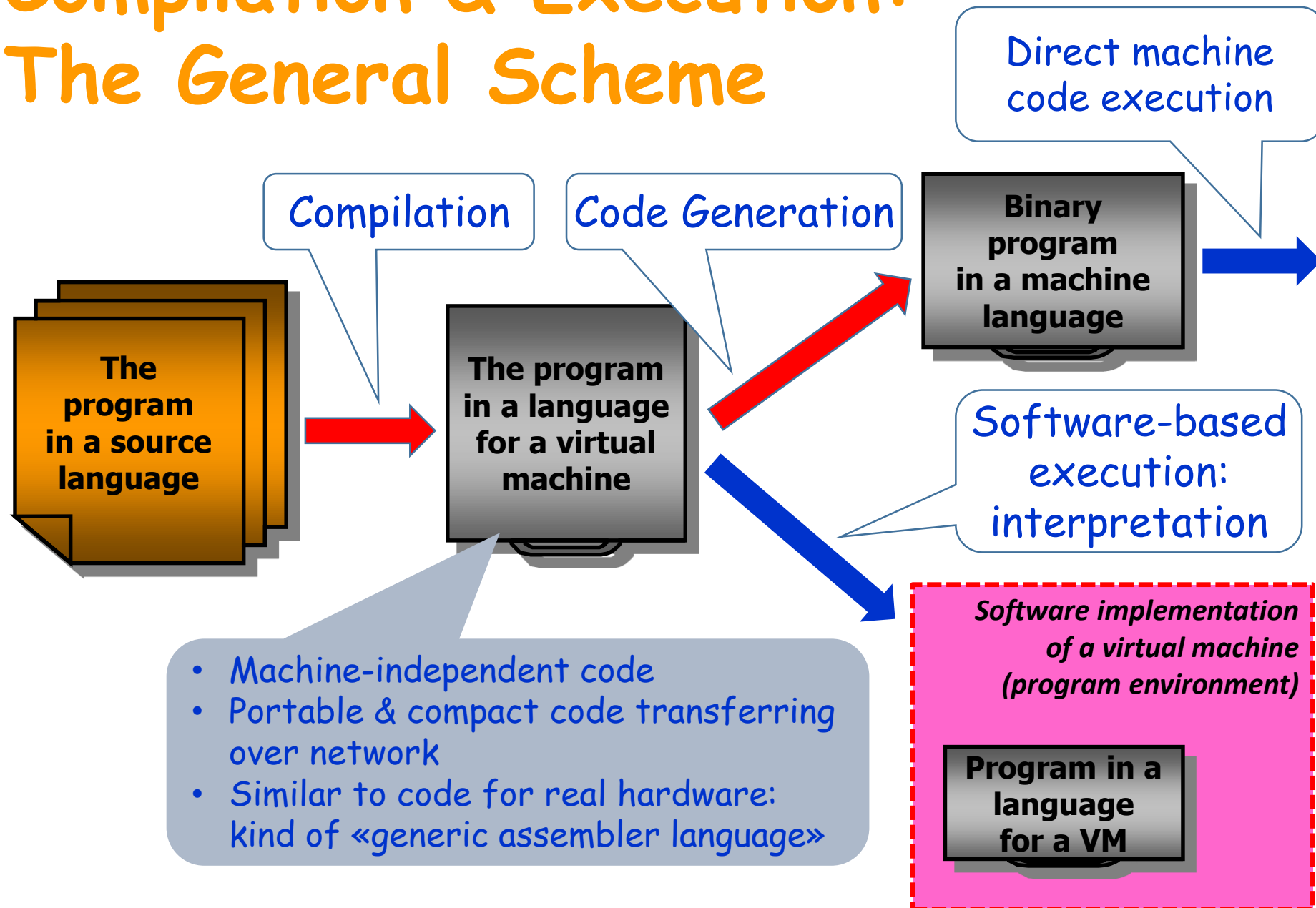
The source program gets compiled...

- Neither to an object code (or an executable program) for a particular hardware architecture;
- Nor to an intermediate representation carrying information about source code semantics -
  - But to a program for some hypothetical (abstract, virtual) computer with all architectural features of a real computer: a "CPU" with instruction set, with memory, registers etc.

# Compilation & Execution: The General Scheme



# Compilation & Execution: The General Scheme



# Virtual Machine: What's New?

What's the real difference between conventional *program intermediate representation* and virtual machine code??

# Virtual Machine: What's New?

What's the real difference between conventional *program intermediate representation* and virtual machine code??

- Virtual machine is designed not for adequate and complete **semantic representation** of the source program (as IR), but for portability and for program **execution**.
- Virtual machine architecture is made quite **similar to real hardware architecture**.

# Brief History

- Snobol-4: The language for symbolic manipulations: 1967 (!!!)

Snobol-4 programs translated into the code for SIL (System Implementation Language ) abstract machine

- N.Wirth's Pascal compiler: 1973 (!!)

Pascal source programs get compiled to code of an abstract Pascal machine: **P Code**.

The next generation was **M Code** for Modula-2 language and its compiler.

- Java Virtual Machine (JVM)

.NET Platform

- Python language

Has its own abstract machine

- LLVM infrastructure

# JVM & .NET: major features

*From the compiler writers' point of view ☺*

- Hardware independence
  - however, rather "close" to real machines
- **Stack-based execution model**
  - not only function calls, but expression calculations as well
- Rather high level of the instruction set
  - high-level function call mechanism; exception mechanism is supported
- Advanced code structure
  - constants, metadata (!), debug information
- Open format:
  - ISO standard for .NET,  
complete documentation for JVM



# JVM & .NET: Philosophy

*Java Slogan:*

Write once – run everywhere

# JVM & .NET: Philosophy

*Java Slogan:*

*...in Java*

*...on JVM*

Write once – run everywhere

# JVM & .NET: Philosophy

*Java Slogan:*

*...in Java*

*...on JVM*

Write once – run everywhere

*.NET Slogan:*

Write in any language –  
run under .NET

# JVM & .NET: Philosophy

*Java Slogan:*

*...in Java*

*...on JVM*

Write once – run everywhere

*.NET Slogan:*

*...on Windows(?)*

Write in any language –  
run under .NET

# JVM & .NET: Comparison (1)

- Official Java/JVM slogan:  
**Write once - run everywhere**  
(but only under JVM 😊)  
The single language and many hardware platforms
- (Unofficial) .NET slogan:  
**Write for .NET in any language - and get full interoperability** (but only for Windows 😊)  
Many languages - the single platform (Windows)

Now multiple of  
platforms actually!

# JVM & .NET: Comparison (2)

- **Implementation:**  
JVM: **many** implementations (Sun/Oracle was just the first) for several hardware architectures - *including non-stacked*.  
.NET: at least **four** implementations:  
the two of Microsoft («main version» и Rotor which is open source), **Mono** & Portable.NET.

# JVM & .NET: Comparison (2)

- **Implementation :**

JVM: **many** implementations (Sun/Oracle was just the first) for several hardware architectures - *including non-stacked*.

.NET: at least **four** implementations:

the two of Microsoft («main version» и Rotor which is open source), **Mono** & Portable.NET.

- Java, Scala, Kotlin, Groovie, Clojure, ...
- JVM-based versions of Ruby, Python, Common Lisp, ...

- **Source languages:**

**Many** (other than Java) for JVM.

**Many** (other than C#) for .NET.

- C#, Spec#, Xen, Cw, Basic, F#, Nemerle, ...
- .NET-based versions: Python, Lisp, Ada, **ML**...

# JVM & .NET: Comparison (3)

## Standardization:

- Neither Java, nor Java Virtual Machine **are not yet standardized**.
- Not only C# language, but all .NET platform components (architecture, type system, instruction set, common language infrastructure etc.) - **are standardized** by both ECMA (European standard organization), and by ISO (International Standard Organization).



# MSIL Code Example

```
class Program
{
    int F(int a,int b)
    {
        int c = 7;
        int x = (a-b)*(a+c);
        return x;
    }
}
```

# MSIL Code Example

```
class Program
{
    int F(int a,int b)
    {
        int c = 7;
        int x = (a-b)*(a+c);
        return x;
    }
}
```

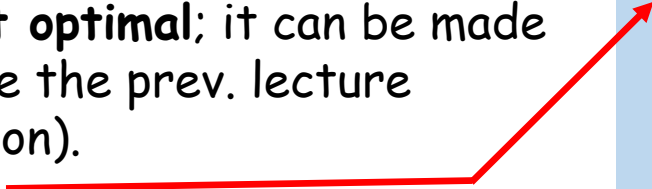
```
.method private hidebysig instance int32
    F(int32 a,
        int32 b) cil managed
{
    // Code size          17 (0x11)
    .maxstack 3
    .locals init ([0] int32 c,
                  [1] int32 x,
                  [2] int32 CS$1$0000)

    IL_0000:  nop
    IL_0001:  ldc.i4.7
    IL_0002:  stloc.0
    IL_0003:  ldarg.1
    IL_0004:  ldarg.2
    IL_0005:  sub
    IL_0006:  ldarg.1
    IL_0007:  ldloc.0
    IL_0008:  add
    IL_0009:  mul
    IL_000a:  stloc.1
    IL_000b:  ldloc.1
    IL_000c:  stloc.2
    IL_000d:  br.s      IL_000f
    IL_000f:  ldloc.2
    IL_0010:  ret
} // end of method Program::F
```

# MSIL Code Example

```
class Program
{
    int F(int a,int b)
    {
        int c = 7;
        int x = (a-b)*(a+c);
        return x;
    }
}
```

Even such a simple (actually trivial) code **is not optimal**; it can be made better. See the prev. lecture (optimization).



```
.method private hidebysig instance int32
    F(int32 a,
      int32 b) cil managed
{
    // Code size          17 (0x11)
    .maxstack 3
    .locals init ([0] int32 c,
                  [1] int32 x,
                  [2] int32 CS$1$0000)

    IL_0000:  nop
    IL_0001:  ldc.i4.7
    IL_0002:  stloc.0
    IL_0003:  ldarg.1
    IL_0004:  ldarg.2
    IL_0005:  sub
    IL_0006:  ldarg.1
    IL_0007:  ldloc.0
    IL_0008:  add
    IL_0009:  mul
    IL_000a:  stloc.1
    IL_000b:  ldloc.1
    IL_000c:  stloc.2
    IL_000d:  br.s      IL_000f
    IL_000f:  ldloc.2
    IL_0010:  ret
} // end of method Program::F
```

# Code Generation for VM (1)

**A + B**    *Infix notation*

( a - b ) \* ( a + c )

**+ A B**    *Polish (prefix) notation*

\* - a b + a c

**A B +**    *Polish reverse (postfix) notation*

ПОЛИЗ

a b - a c + \*

# Code Generation for VM (1)

**A + B**    *Infix notation*

( a - b ) \* ( a + c )

**+ A B**    *Polish (prefix) notation*

\* - a b + a c

**A B +**    *Polish reverse (postfix) notation*

ПОЛИЗ

a b - a c + \*

## Polish inverse notation characteristics:

- No parentheses.
- Operands go in the same order as in the source expression.
- Operators go immediately after operands they apply.

# Code Generation for VM (1)

$A + B$     *Infix notation*

$( a - b ) * ( a + c )$

$+ A B$     *Polish (prefix) notation*

$* - a b + a c$

$A B +$     *Polish reverse (postfix) notation*

ПОЛИЗ

$a b - a c + *$

Polish inverse notation characteristics:

- No parentheses.
- Operands go in the same order as in the source expression.
- Operators go immediately after operands they apply.
- **It reflects the order of operations on the stack!**

# Code Generation for VM (2)

a b - a c + \*

# Code Generation for VM (2)

a b - a c + \*

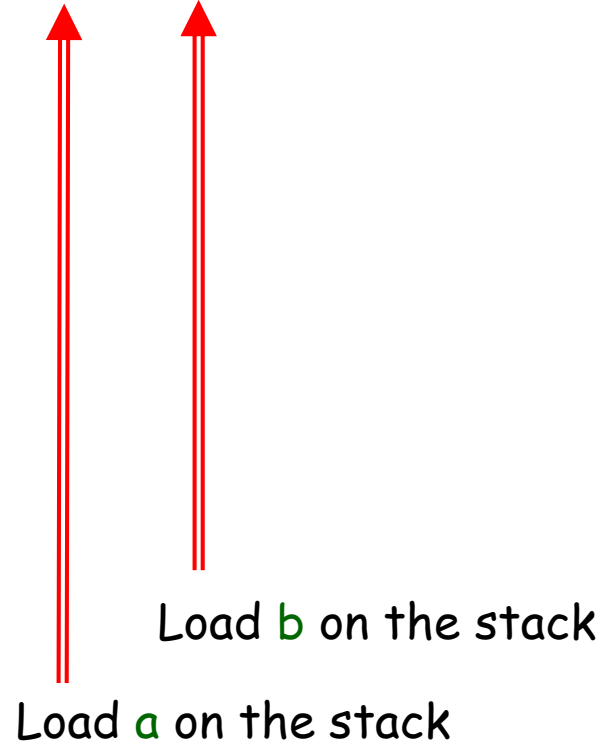


Load **a** on the stack



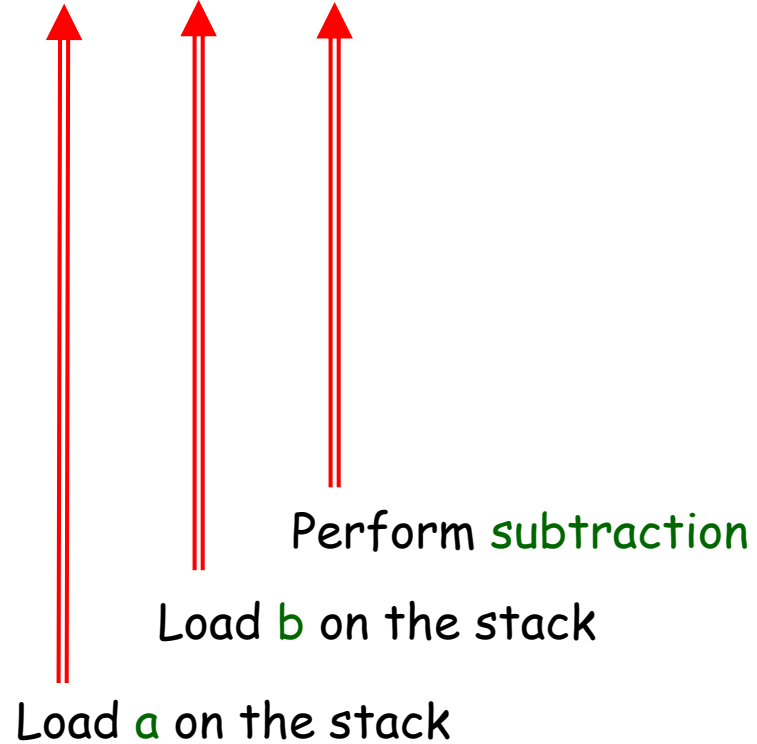
# Code Generation for VM (2)

a b - a c + \*

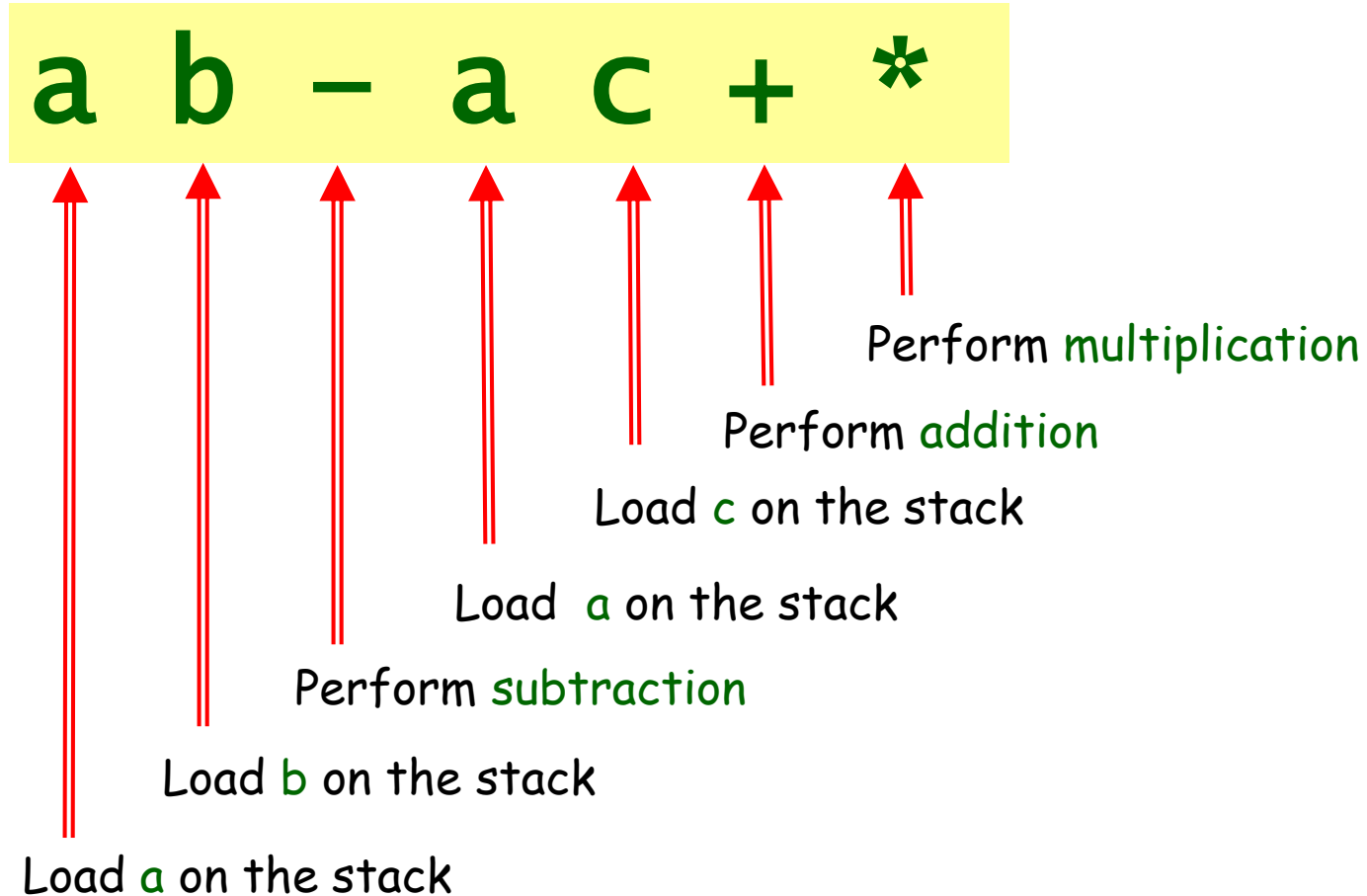


# Code Generation for VM (2)

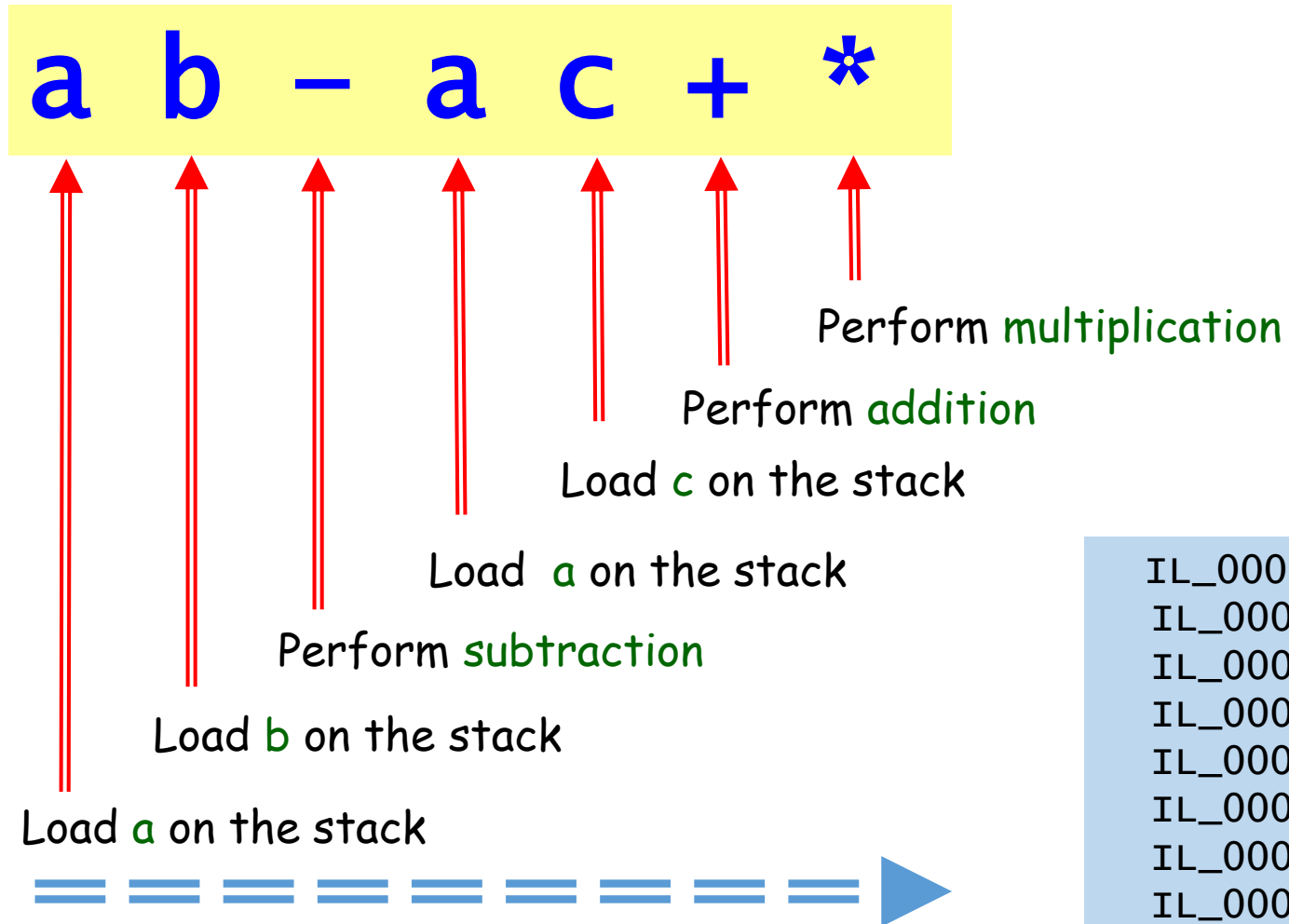
a b - a c + \*



# Code Generation for VM (2)



# Code Generation for VM (2)

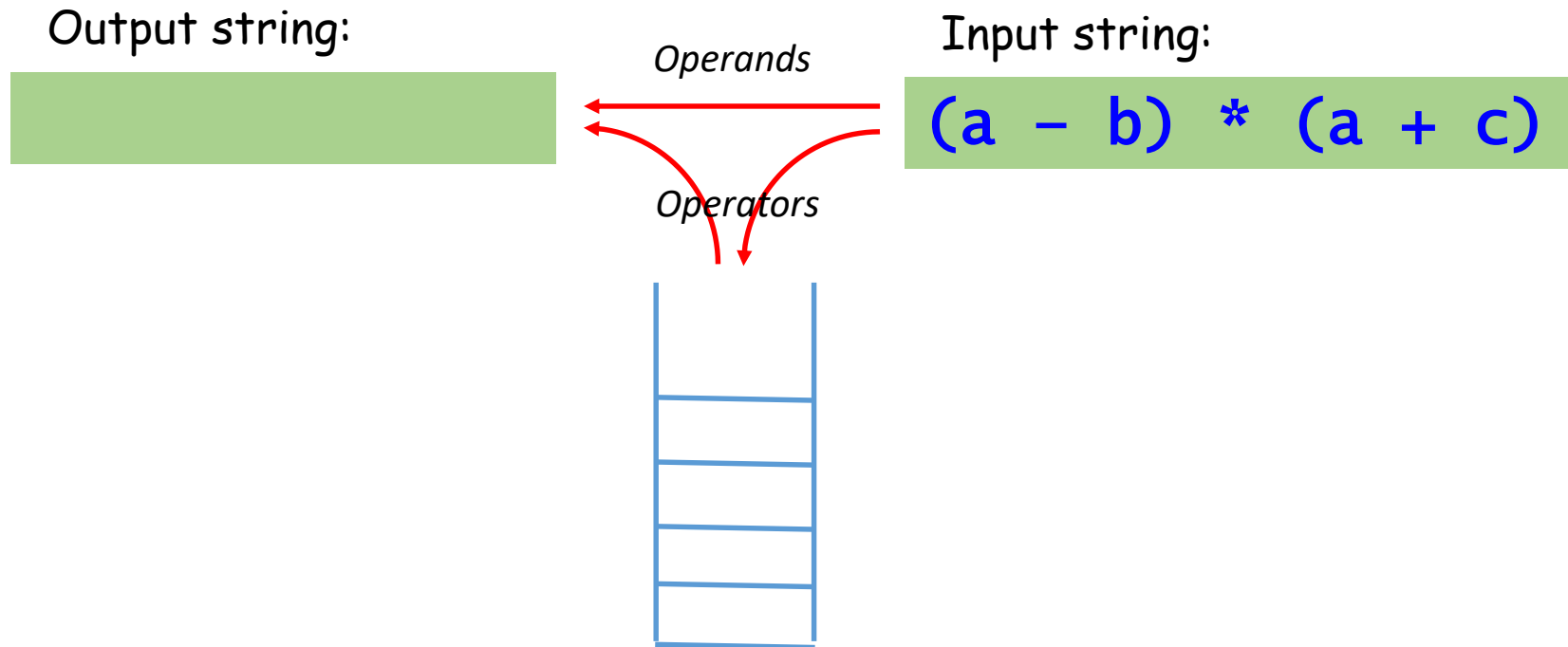


```
class Program
{
    int F(int a,int b)
    {
        int c = 7;
        int x = (a-b)*(a+c);
        return x;
    }
}
```

```
IL_0002: ...
IL_0003: ldarg.1
IL_0004: ldarg.2
IL_0005: sub
IL_0006: ldarg.1
IL_0007: ldloc.0
IL_0008: add
IL_0009: mul
IL_000a: ...
```

# Code Generation: E.Dijkstra algorithm (1)

Operator-precedence stack method  
("shunting-yard" algorithm)



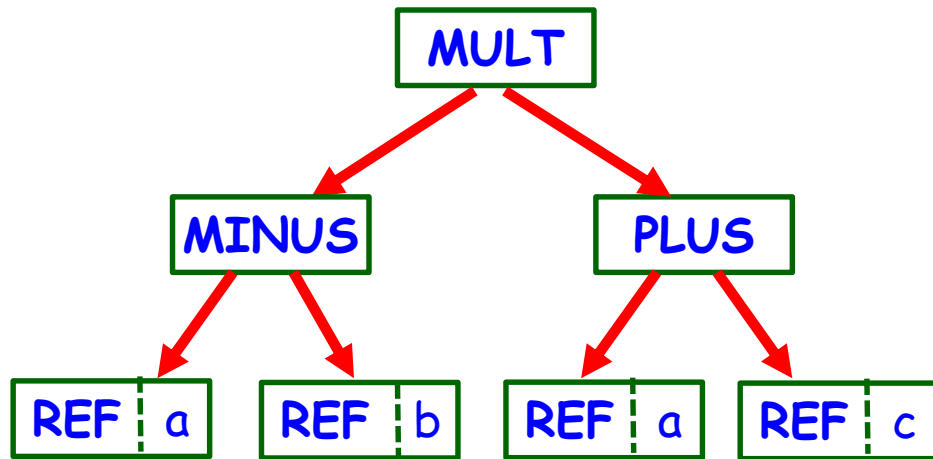
# Code Generation:

## E.Dijkstra algorithm (2)

1. Each operator is assigned a precedence number in accordance of conventional rules. The numbering starts from 2.
2. Opening parenthesis has the number of 0, closing parenthesis - 1.
3. The input expression is read from left to right. Operands go directly to the output string.
4. Opening parenthesis (with prec. number 0) is always put to the stack.
5. If the number of the current operator is bigger than the operator's number from the top, then the new operator is put to the stack.
6. If the number of the current operator is less or equal to the one from the top then all operators with greater or equal numbers popped from the stack to the output string.
7. If the current source element is closing parenthesis then all operators down to the opening parentheses are popped from the stack to the output string. Both parentheses are removed.
8. If there are no more symbols in the source string then the rest operators are popped from the stack to the output string.
9. **(The algorithm could be easily generalized for other operators: taking an array element, function call with parameter passing etc.).**

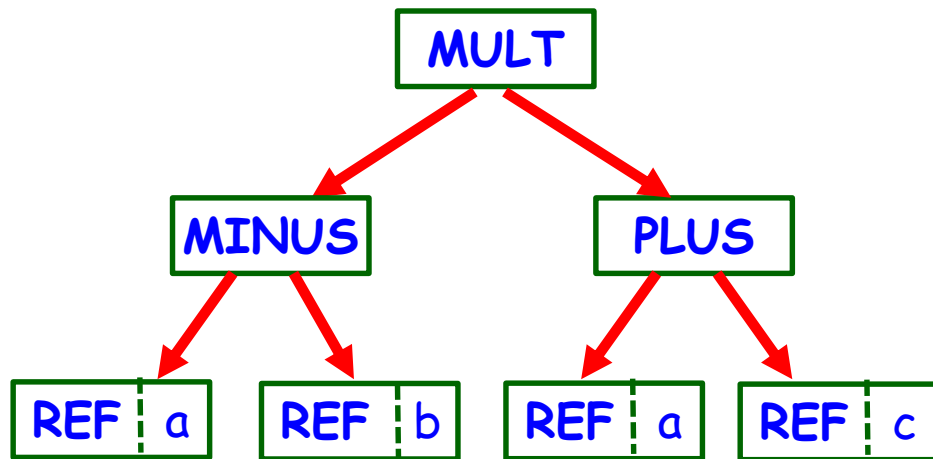
# VM Code Generation on the Program Tree

Tree traversing in order  
"bottom-up from left to right"



# VM Code Generation on the Program Tree

Tree traversing in order  
"bottom-up from left to right"



## Process the operator node:

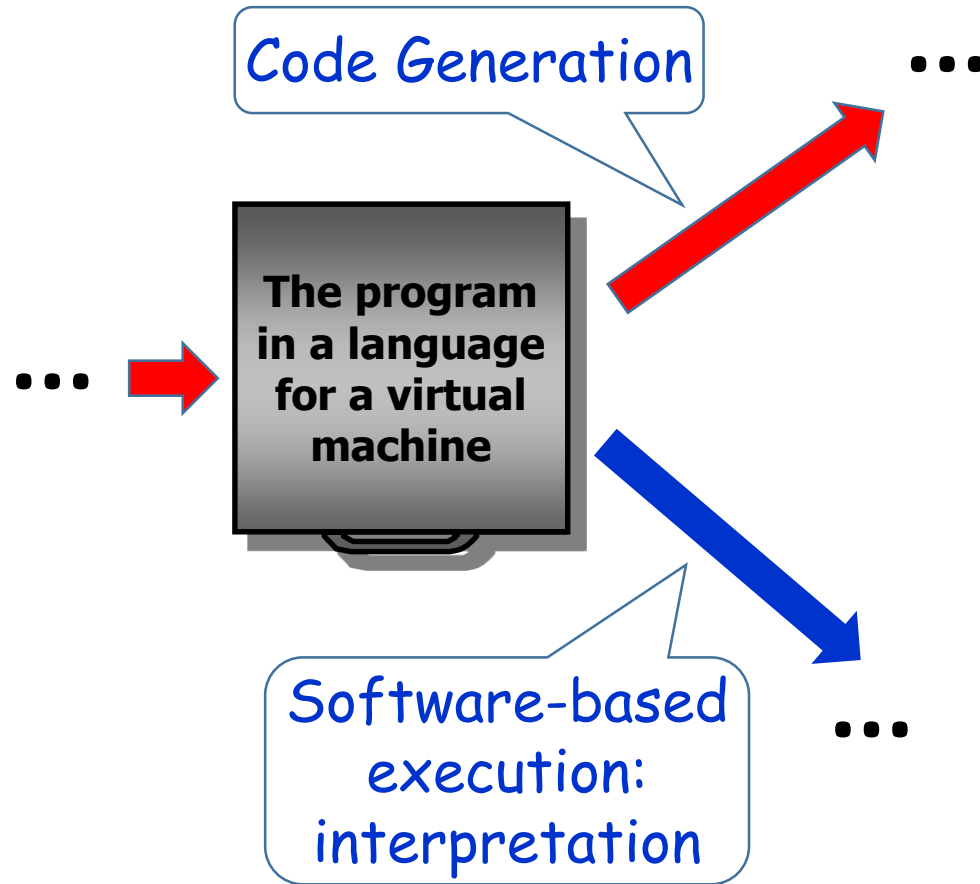
- Process the left subtree.
- Process the right subtree.
- Generate instruction code performing the operator from the root node.

## Process the terminal node:

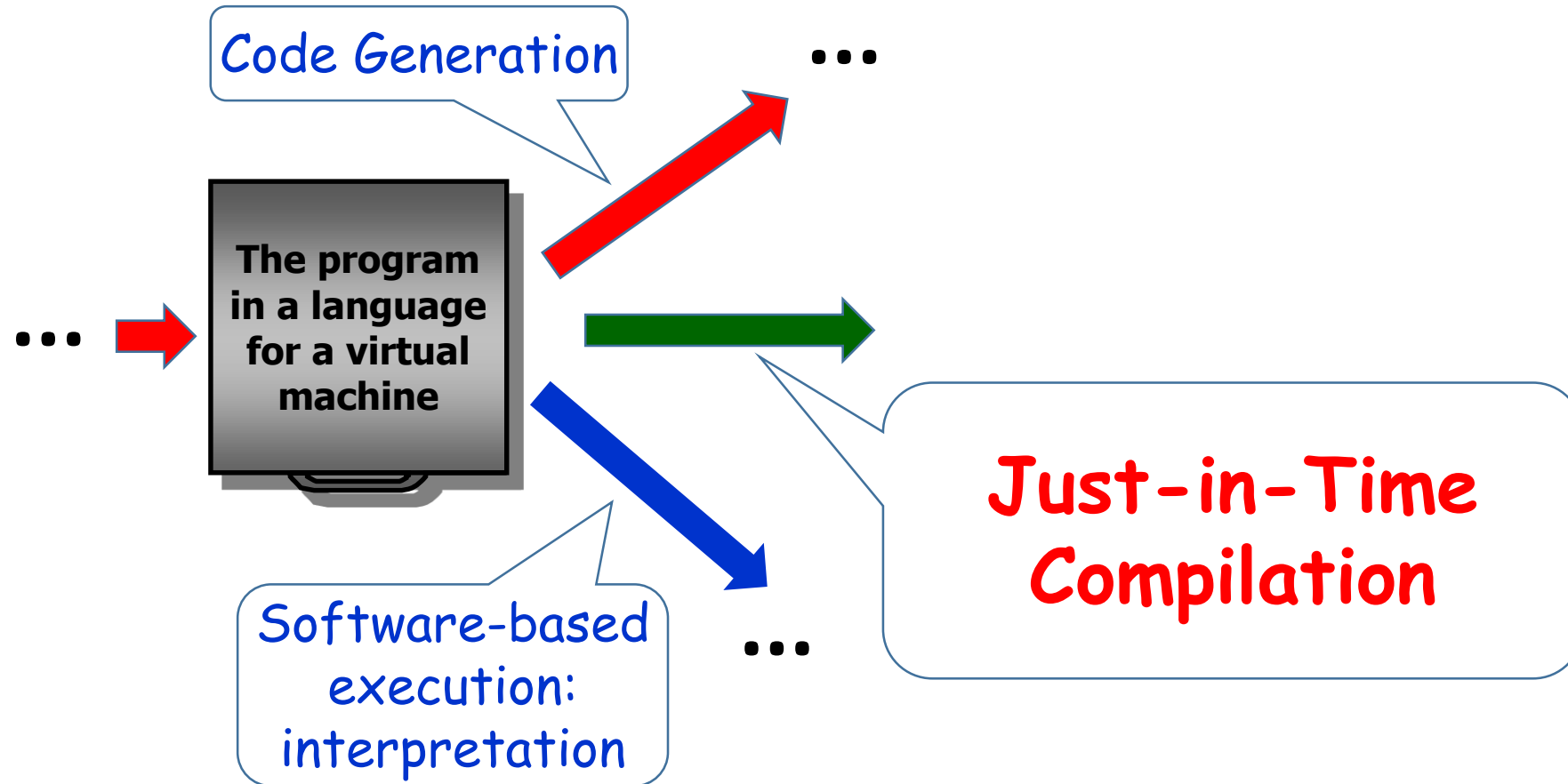
- Generate instruction code for loading operand to the stack.



# Compilation & Execution: Addition to the Common Scheme - JIT

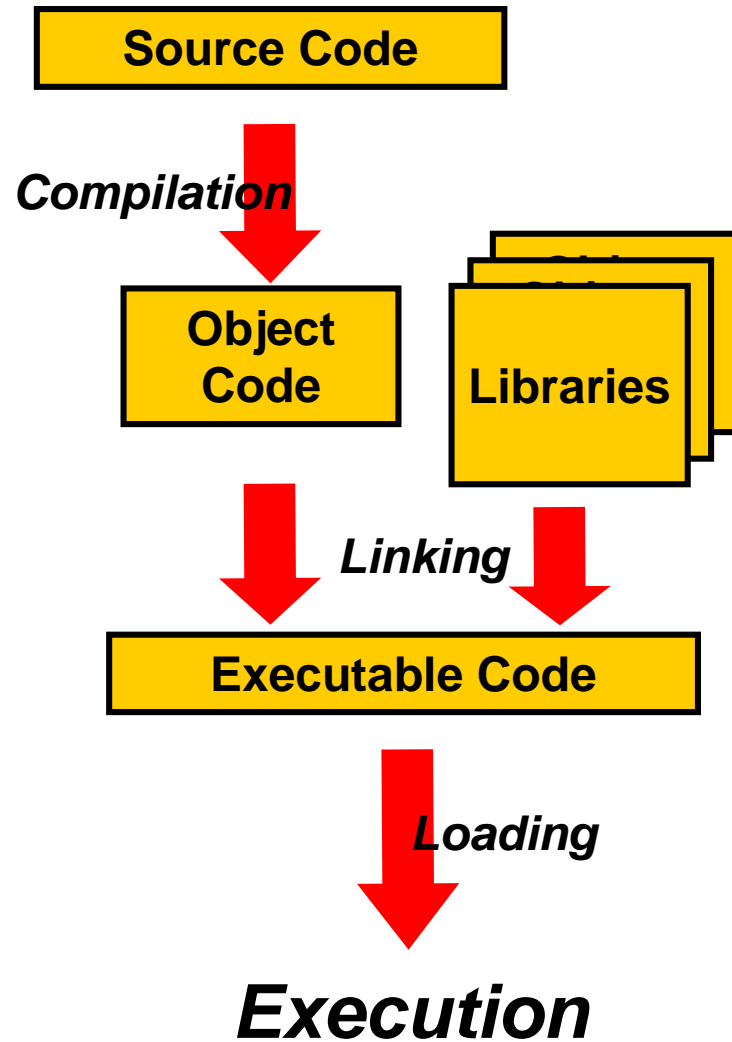


# Compilation & Execution: Addition to the Common Scheme - JIT

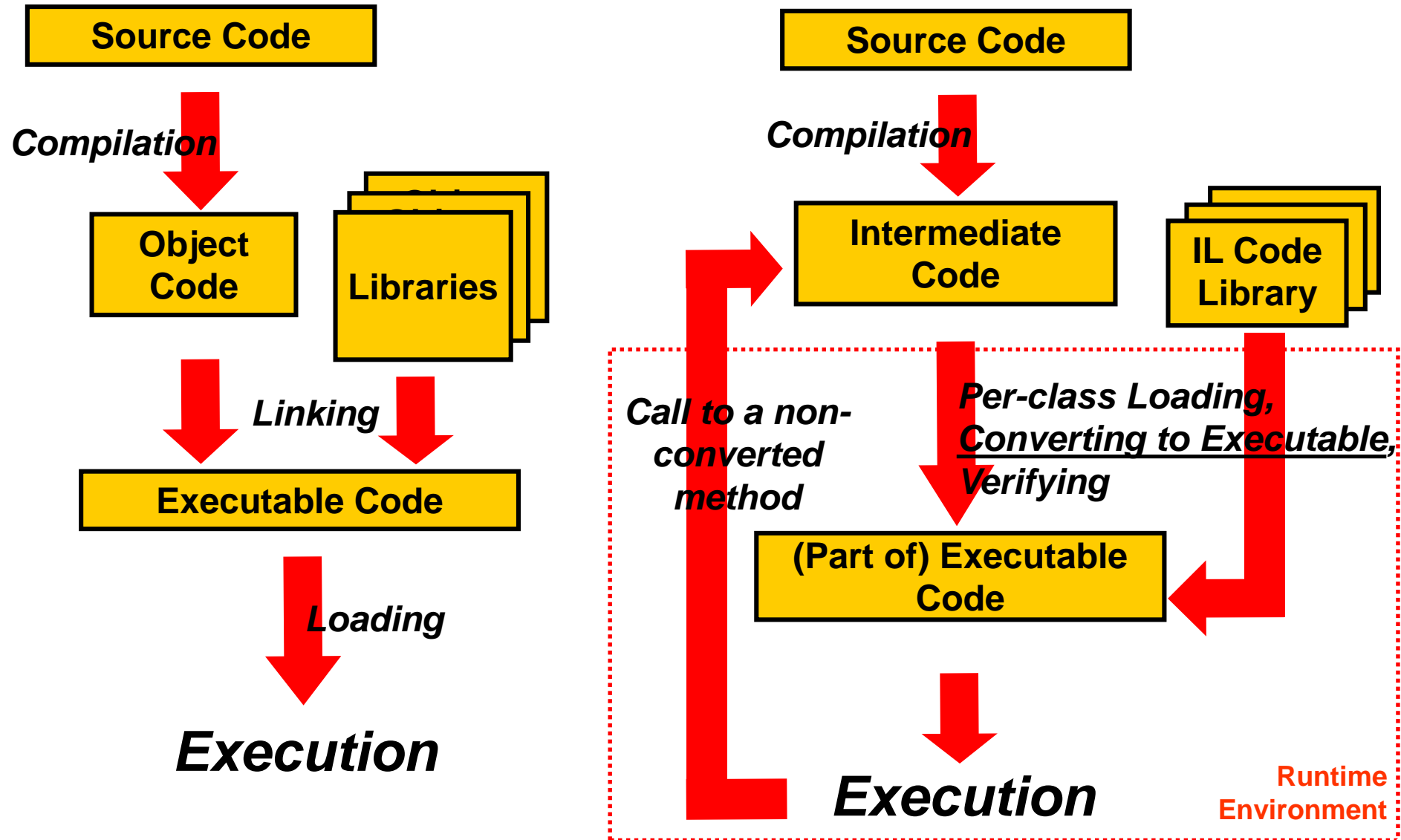


# Introduction to CLR: Execution Scheme

---



# Introduction to CLR: Execution Scheme



# Low Level Virtual Machine: LLVM

- Non-stacked abstract machine: the stack is used only for function calls but not for calculating expression.
- Register-based machine: execution model is based on operations on registers.
- The main operation is assignment: the SSA (Single Static Assignment) principle is deep inside.

There are some other important features; now we are interested only LLVM aspects related to compilation...

# LLVM Code Example

```
class Program
{
    int F(int a,int b)
    {
        int c = 7;
        int x = (a-b)*(a+c);
        return x;
    }
}
```

```
...
i32 @_ZlFi(i32 %a, i32 %b) #0
{
entry:
    %a.addr = alloca i32,      align 4
    %b.addr = alloca i32,      align 4
    %c       = alloca i32,      align 4
    %x       = alloca i32,      align 4
    store i32 %a, i32* %a.addr, align 4
    store i32 %b, i32* %b.addr, align 4
    store i32 7, i32* %c,       align 4
    %0 = load i32* %a.addr,      align 4
    %1 = load i32* %b.addr,      align 4
    %sub = sub nsw i32 %0, %1
    %2 = load i32* %a.addr,      align 4
    %3 = load i32* %c,           align 4
    %add = add nsw i32 %2, %3
    %mul = mul nsw i32 %sub, %add
    store i32 %mul, i32* %x,      align 4
    %4 = load i32* %x,           align 4
    ret i32 %4
}
...
```

# LLVM Code: Some Features

- Stack mechanism is used for function calls only.
  - Execution model is based on operations on registers.
- The result of (almost) each operator is assigned to some variable (register).
  - Each assignment works with a **new** variable..
  - (Almost) each instruction is a **triplet**: two operands and the result.

```
...  
i32 @_z1Fi(i32 %a, i32 %b) #0  
{  
  entry:  
    %a.addr = alloca i32,      align 4  
    %b.addr = alloca i32,      align 4  
    %c       = alloca i32,      align 4  
    %x       = alloca i32,      align 4  
    store i32 %a, i32* %a.addr, align 4  
    store i32 %b, i32* %b.addr, align 4  
    store i32 7, i32* %c,       align 4  
    %0 = load i32* %a.addr,      align 4  
    %1 = load i32* %b.addr,      align 4  
    %sub = sub nsw i32 %0, %1  
    %2 = load i32* %a.addr,      align 4  
    %3 = load i32* %c,          align 4  
    %add = add nsw i32 %2, %3  
    %mul = mul nsw i32 %sub, %add  
    store i32 %mul, i32* %x,      align 4  
    %4 = load i32* %x,          align 4  
    ret i32 %4  
}  
...
```

# LLVM versus .NET/JVM

- Stack-based architecture is the fundamental basis for the most of widely used programming languages.
- It's much simpler to generate code for a stack-based machine (either for a real or for a virtual machine).
- Stack mechanism is supported by many hardware architectures.



# LLVM versus .NET/JVM

- Stack-based architecture is the fundamental basis for the most of widely used programming languages.
- It's much simpler to generate code for a stack-based machine (either for a real or for a virtual machine).
- Stack mechanism is supported by many hardware architectures.

----- However... -----

- Stack-based execution model is **less efficient** than the register-based model.
- Stack-based model is not so suitable for **optimizations**; usually it's harder to implement opts than for register model.
- SSA approach supported by LLVM is specially oriented for implementing deep optimizations.

# Code Generation & Interpretation: Some Hints & Directions

- Mappings (projections):  
an approach for developing code generator
- Tree calculation:  
an approach for developing an interpreter

# Language Mappings

The idea is quite straightforward:

- Create a list of basic language constructs like expressions, statements etc...
- ...And write the corresponding machine code that should be generated by the code generator for the construct.

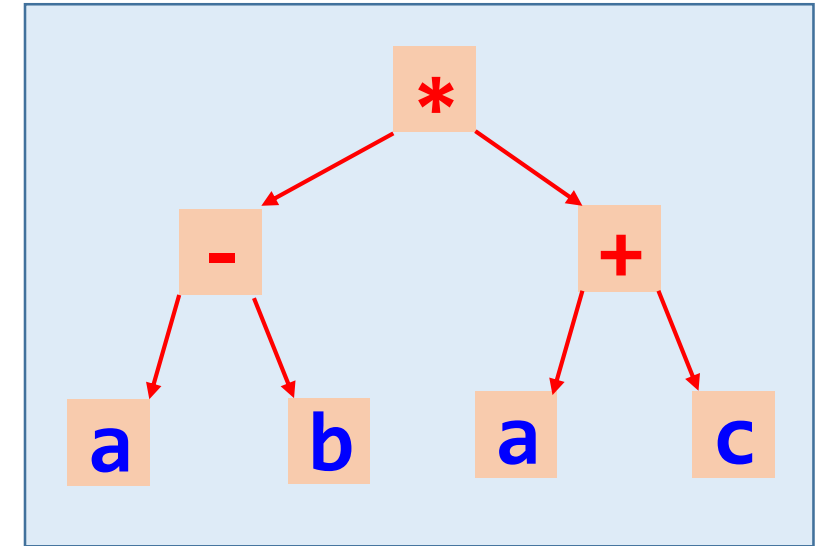

The rest are examples (for stack machines)

# Mapping Example: Expressions

```
class Program
{
  int F(int a, int b)
  {
    int c = 7;
    int x = (a-b)*(a+c);
    return x;
  }
}
```

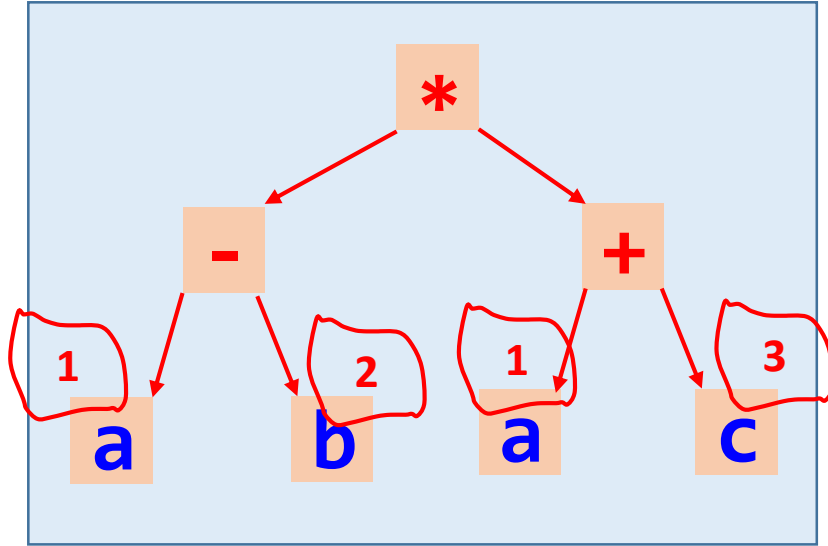
Red annotations in the code: 1 around 'a', 2 around 'b', 3 around 'c', and 4 around the entire expression '(a-b)\*(a+c)'.

Syntax &  
semantic  
analyses

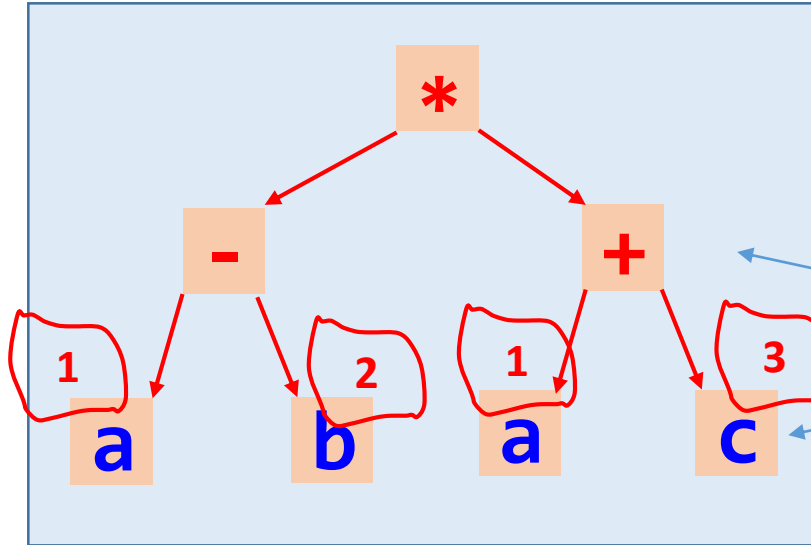


Code generation phase is implemented on the program tree - after it was built and semantically verified

# Mapping Example: Expressions



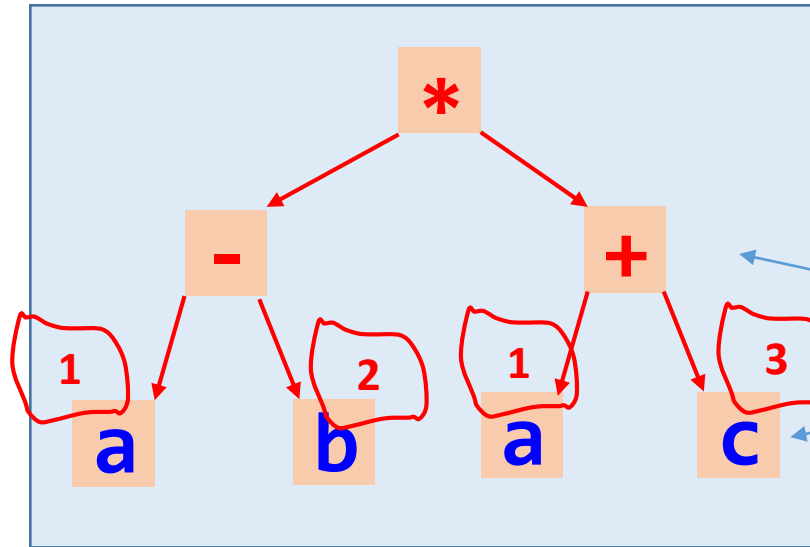
# Mapping Example: Expressions



Notice that in expression trees:

- Non-terminal nodes are always **operators**
- Terminal nodes are always **variables**

# Mapping Example: Expressions



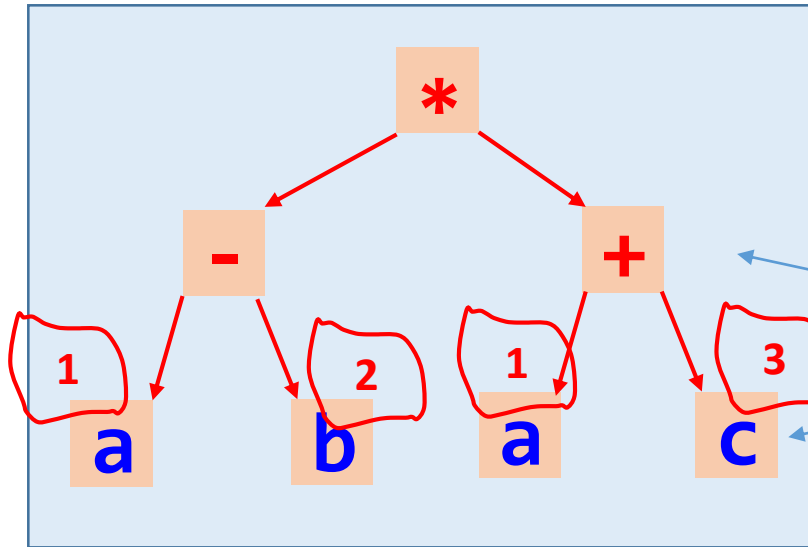
Notice that in expression trees:

- Non-terminal nodes are always **operators**
- Terminal nodes are always **variables**

The general mapping algorithm of the code generation for expressions for a stack based machine is as follows:

- Visit the left subtree of the tree.
- Visit the right subtree of the tree.
- Generate (issue, emit) an instruction for the operator from the root tree

# Mapping Example: Expressions



Notice that in expression trees:

- Non-terminal nodes are always **operators**
- Terminal nodes are always **variables**

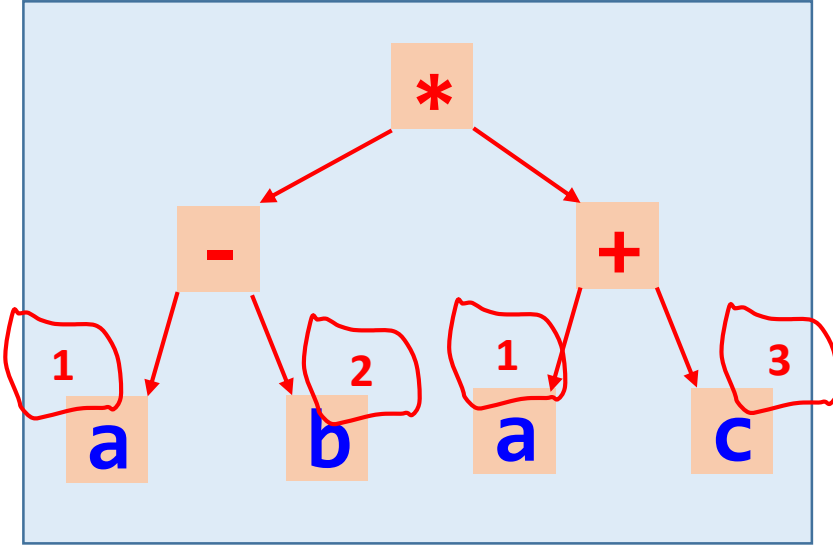
The general mapping algorithm of the code generation for expressions for a stack based machine is as follows:

- Visit the left subtree of the tree.
  - Visit the right subtree of the tree.
  - Generate (issue, emit) an instruction for the operator from the root tree
- Each terminal node (i.e., for each occurrence of a variable) gets **mapped** to an instruction "load a variable to the top of the execution stack".
  - Each non-terminal node (i.e., each occurrence of an operator) gets **mapped** to an instruction "perform the operator on two topmost values, remove them from the stack and push the result on top of the stack".



# Mapping Example: Expressions

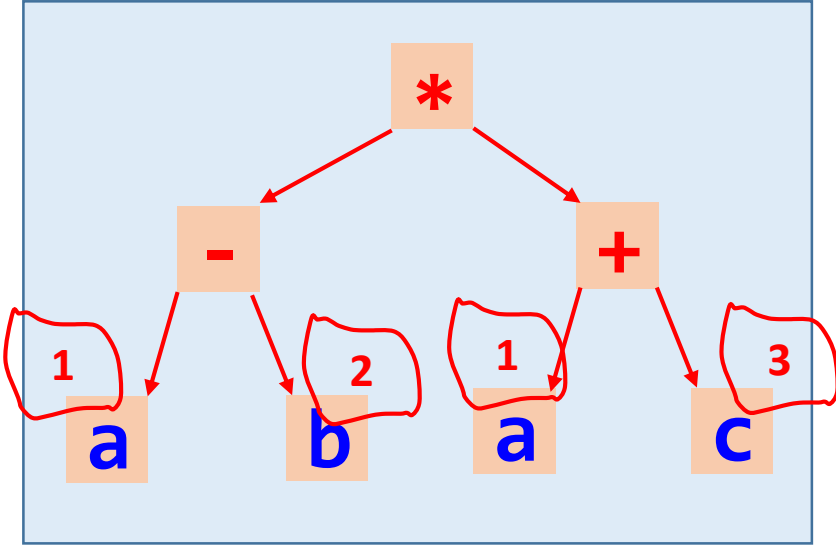
- Visit the left subtree of the  $*$  tree.



- Visit the right subtree of the  $*$  tree.

- Generate code for the  $*$  operator: `MULT`

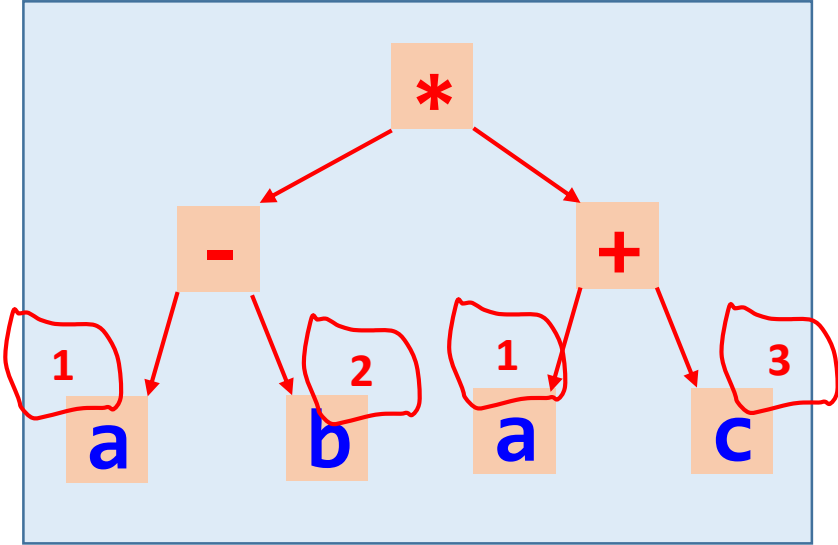
# Mapping Example: Expressions



- Visit the left subtree of the  $*$  tree.
  - Visit the left subtree of the  $-$  tree
    - Visit terminal node  $a$   
Generate code `LOAD_ARG1`
  - Visit the right subtree of the  $-$  tree
    - Visit terminal node  $b$   
Generate code `LOAD_ARG2`
  - Generate code for the  $-$  operator: `SUB`
- Visit the right subtree of the  $*$  tree.

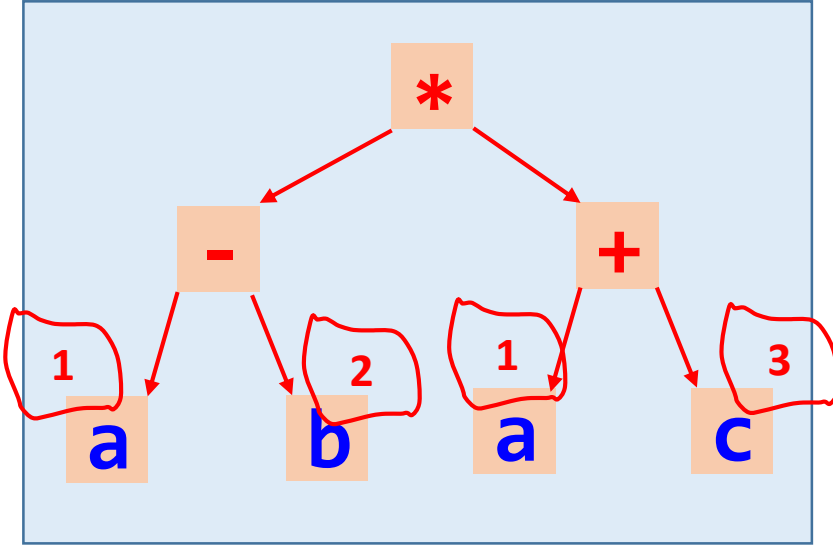
- Generate code for the  $*$  operator: `MULT`

# Mapping Example: Expressions



- Visit the left subtree of the \* tree.
  - Visit the left subtree of the - tree
    - Visit terminal mode a  
Generate code `LOAD_ARG1`
  - Visit the right subtree of the - tree
    - Visit terminal mode b  
Generate code `LOAD_ARG2`
  - Generate code for the - operator: `SUB`
- Visit the right subtree of the \* tree.
  - Visit the left subtree of the + tree
    - Visit terminal mode a  
Generate code `LOAD_ARG1`
  - Visit the right subtree of the + tree
    - Visit terminal mode c  
Generate code `LOAD_LOC1`
  - Generate code for the + operator: `ADD`
- Generate code for the \* operator: `MULT`

# Mapping Example: Expressions



Resulting code

```
LOAD_ARG1
LOAD_ARG2
SUB
LOAD_ARG1
LOAD_LOC1
ADD
MULT
```

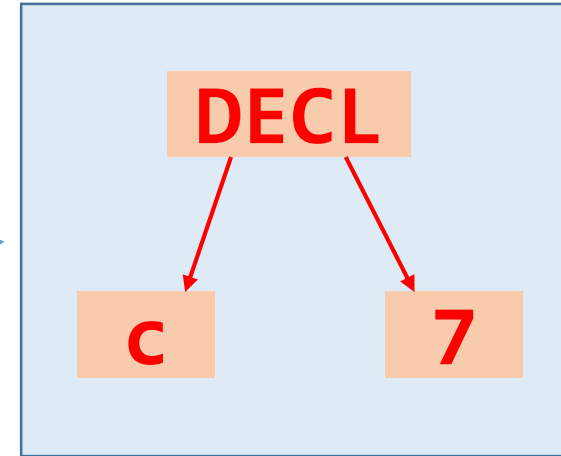

This is **pseudocode**:  
.NET, JVM, Python  
use very similar  
instructions

- Visit the left subtree of the \* tree.
  - Visit the left subtree of the - tree
    - Visit terminal mode a  
Generate code `LOAD_ARG1`
  - Visit the right subtree of the - tree
    - Visit terminal mode b  
Generate code `LOAD_ARG2`
  - Generate code for the - operator: `SUB`
- Visit the right subtree of the \* tree.
  - Visit the left subtree of the + tree
    - Visit terminal mode a  
Generate code `LOAD_ARG1`
  - Visit the right subtree of the + tree
    - Visit terminal mode c  
Generate code `LOAD_LOC1`
  - Generate code for the + operator: `ADD`
- Generate code for the \* operator: `MULT`

# Mapping Example: Declarations

```
class Program
{
  int F(int a,int b)
  {
    int c = 7;
    int x = (a-b)*(a+c);
    return x;
  }
}
```

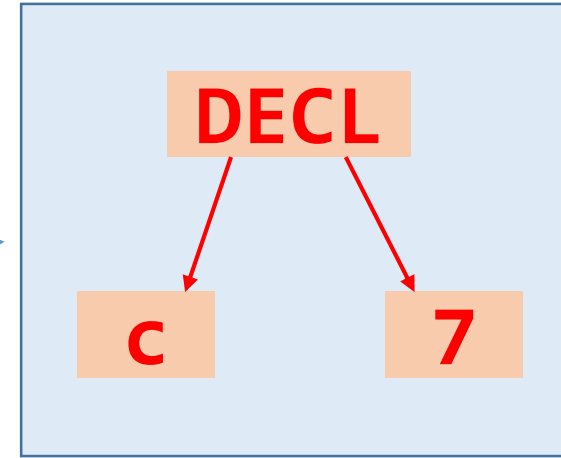
Syntax &  
semantic  
analyses



# Mapping Example: Declarations

```
class Program
{
  int F(int a,int b)
  {
    int c = 7;
    int x = (a-b)*(a+c);
    return x;
  }
}
```

Syntax &  
semantic  
analyses



Mapping

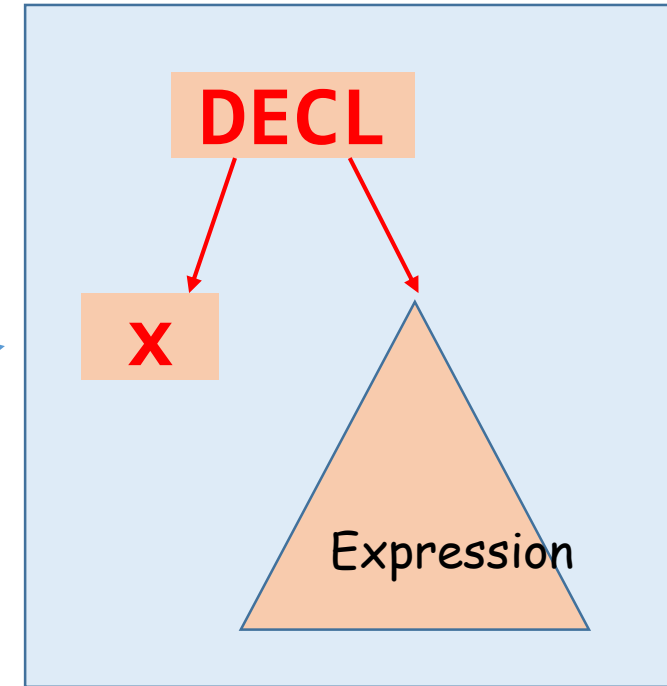
Resulting code

```
LOAD_ADDR3
LOAD_CONST 7
STORE
```

# Mapping Example: Declarations

```
class Program
{
  int F(int a,int b)
  {
    int c = 7;
    int x = (a-b)*(a+c);
    return x;
  }
}
```

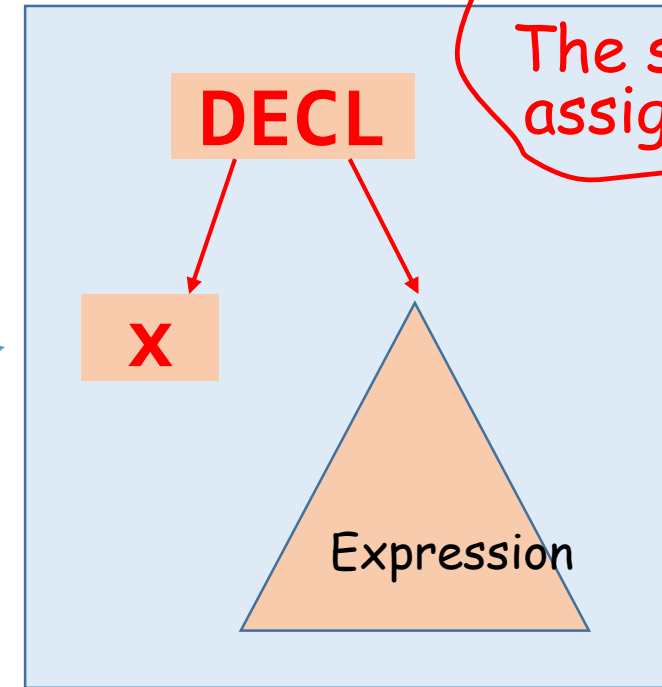
Syntax &  
semantic  
analyses



# Mapping Example: Declarations

```
class Program
{
  int F(int a,int b)
  {
    int c = 7;
    int x = (a-b)*(a+c);
    return x;
  }
}
```

Syntax &  
semantic  
analyses



Mapping

Resulting code

LOAD\_ADDR4

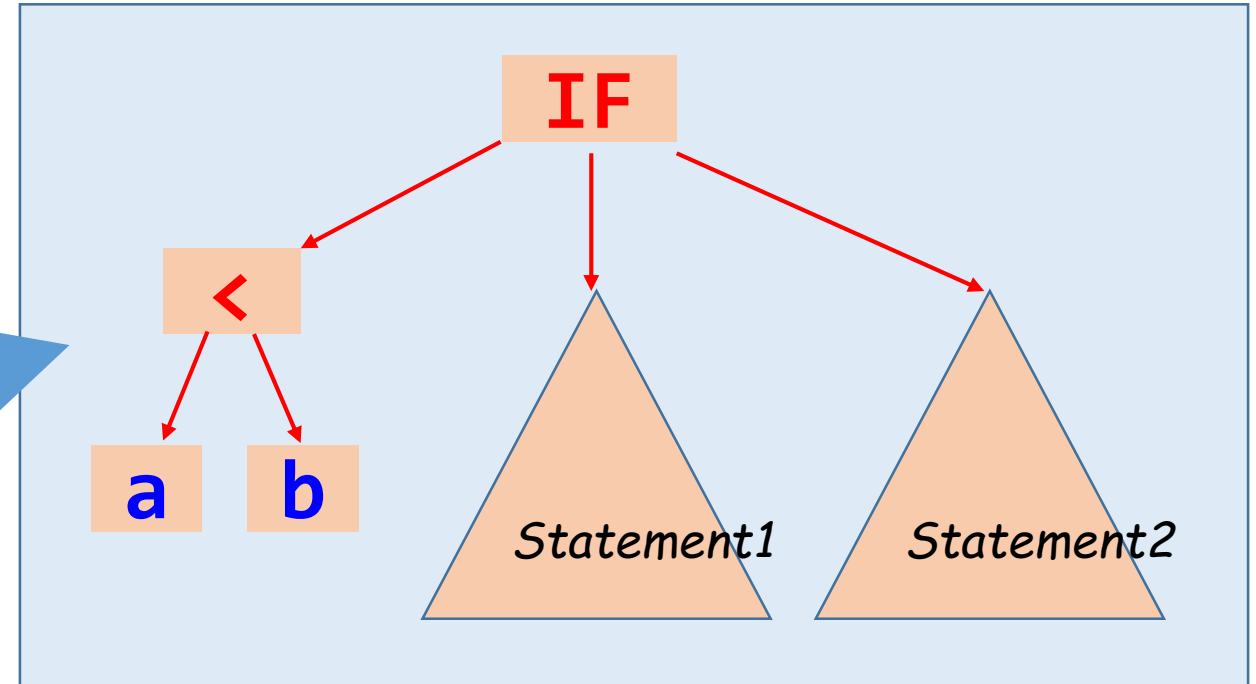
Code for Expression

STORE

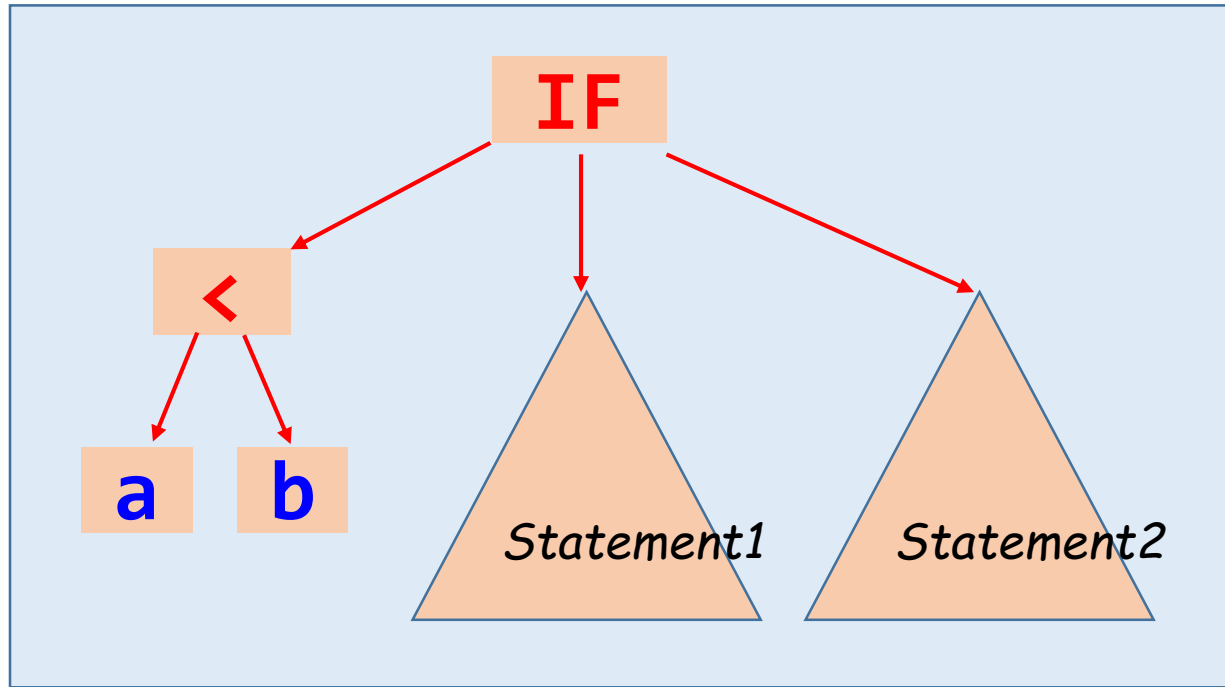


# Mapping Example: IF Statements

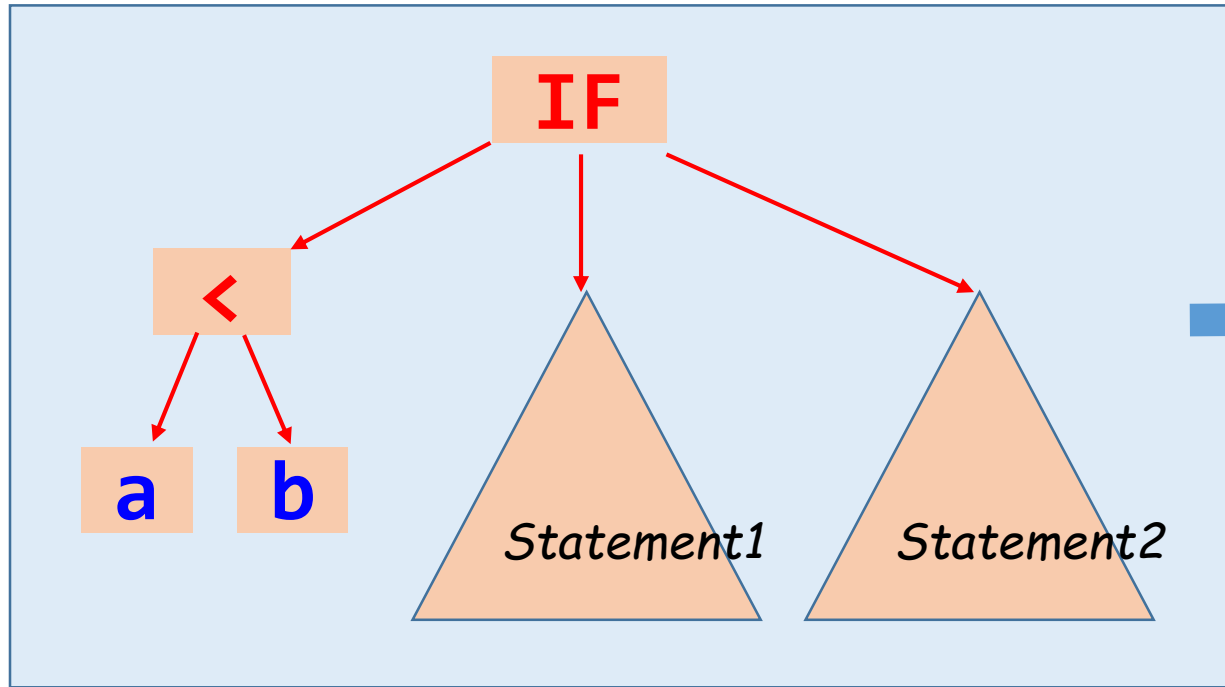
```
class Program
{
    void F(int a,int b)
    {
        if (a > b )
            Statement1
        else
            Statement2
    }
}
```



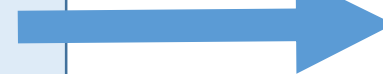
# Mapping Example: IF Statements



# Mapping Example: IF Statements



Mapping



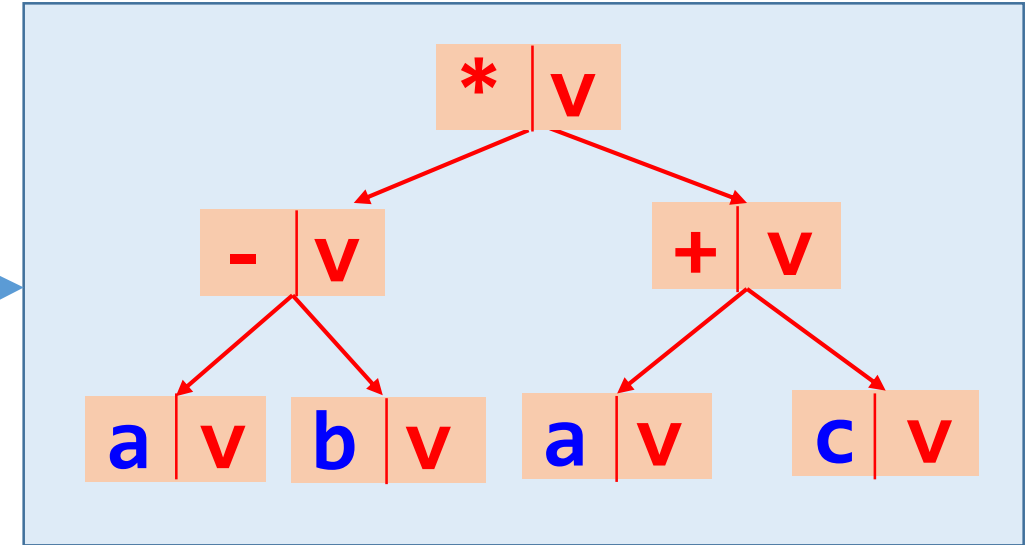
Resulting code

A1	LOAD_ARG1
A2	LOAD_ARG2
A3	COMP_LESS
A4	JUMP_IF_FALSE AN
...	
	<div>Code for Statement1</div>
	JUMP AM
AN	
...	
	<div>Code for Statement2</div>
AM	...

# How to Calculate an Expression

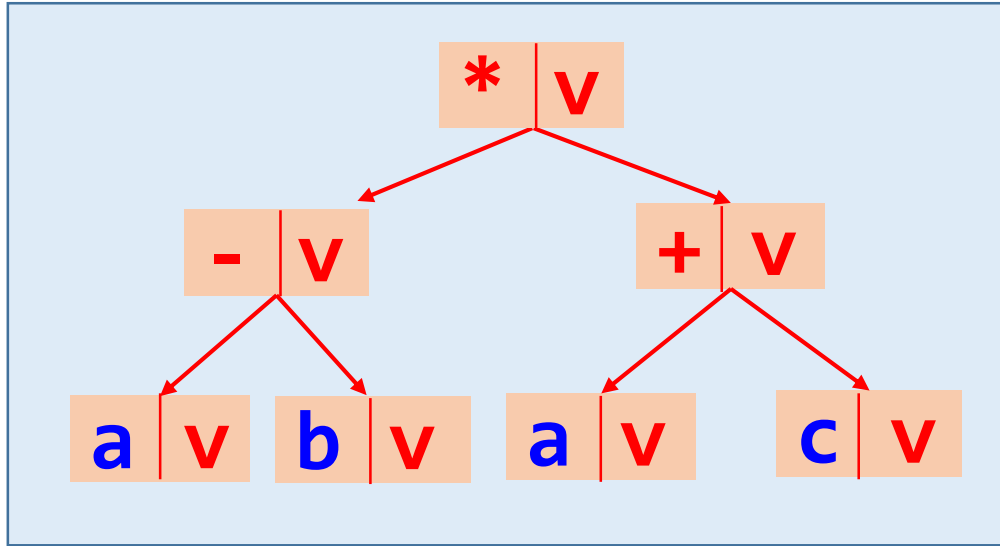
```
class Program
{
  int F(int a,int b)
  {
    int c = 7;
    int x = (a-b)*(a+c);
    return x;
  }
}
```

Syntax &  
semantic  
analyses



Interpretation can be performed  
right on the program tree - using  
the same traversing algorithm

# How to Calculate an Expression



- Visit the left subtree of the  $*$  tree.
  - Visit the left subtree of the  $-$  tree
    - Visit terminal mode  $a$   
 $a.v = \text{current-value-of-}a$
  - Visit the right subtree of the  $-$  tree
    - Visit terminal mode  $b$   
 $b.v = \text{current-value-of-}b$
  - $-.v = a.v * b.v$
- Visit the right subtree of the  $*$  tree.
  - Visit the left subtree of the  $+$  tree
    - Visit terminal mode  $a$   
 $a.v = \text{current-value-of-}a$
  - Visit the right subtree of the  $+$  tree
    - Visit terminal mode  $c$   
 $c.v = \text{current-value-of-}c$
  - $+.v = a.v + c.v$
- $*.v = -.v * +.v$