

# System Software Crash Course

Samsung Research Russia  
Moscow 2019

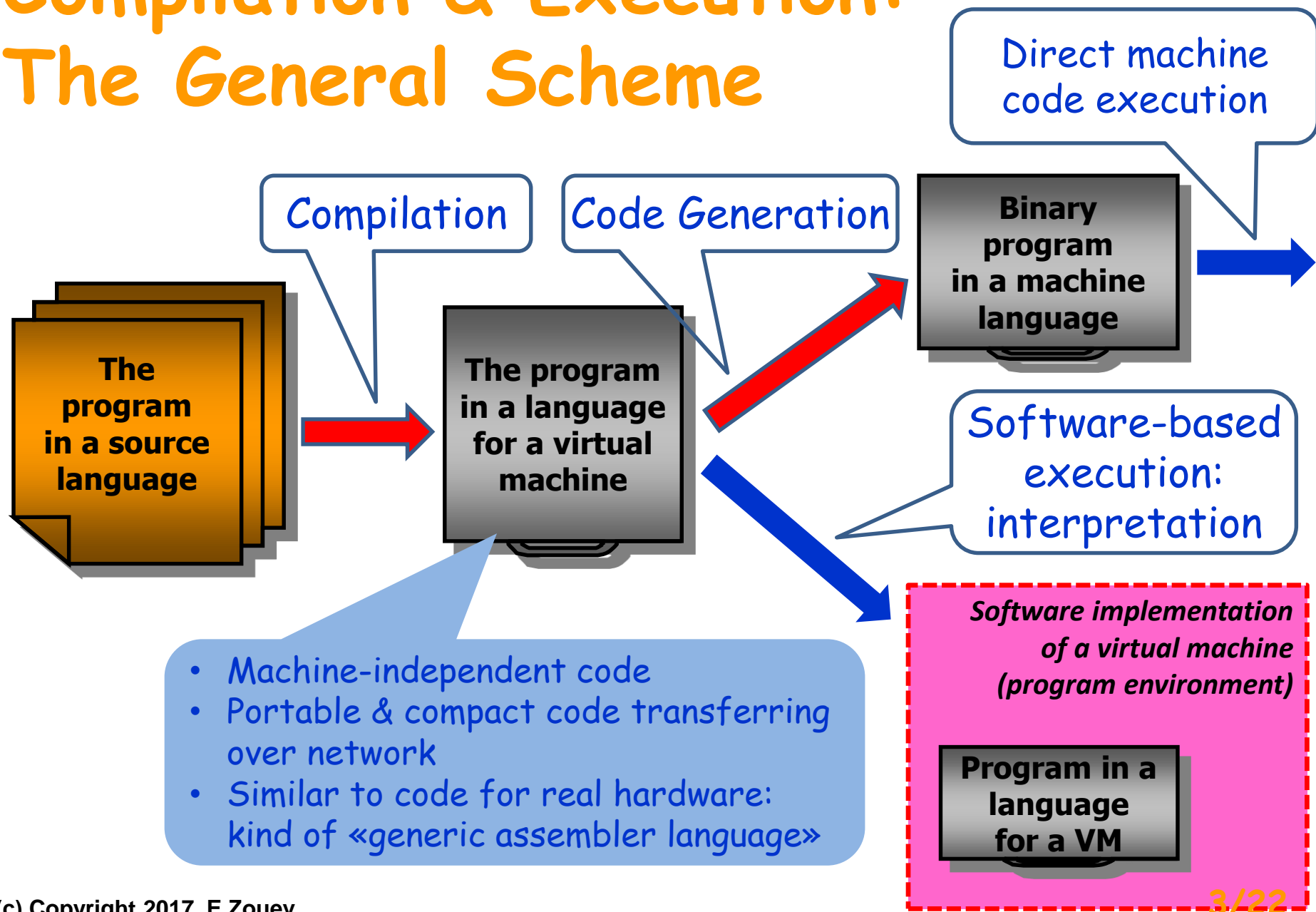
Block C Compiler Construction  
12. Virtual Machines  
Eugene Zouev

# 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



# 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 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:*

*...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)



# JVM & .NET: Comparison (2)

- **Implementation :**  
JVM: **many** implementations (Sun/Oracle was just the first) for several hardware architectures.  
.NET: at least **four** implementations:  
the two of Microsoft («main version» и Rotor which is open source), **Mono** & Portable.NET.
- **Source languages:**  
**Many** (other than Java) for JVM.  
**Many** (other than C#) for .NET.

# 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).

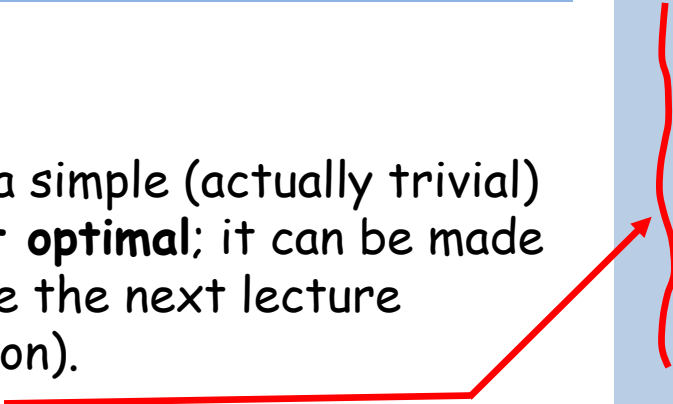
# MSIL Code Example

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

```
.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
```

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



# 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*

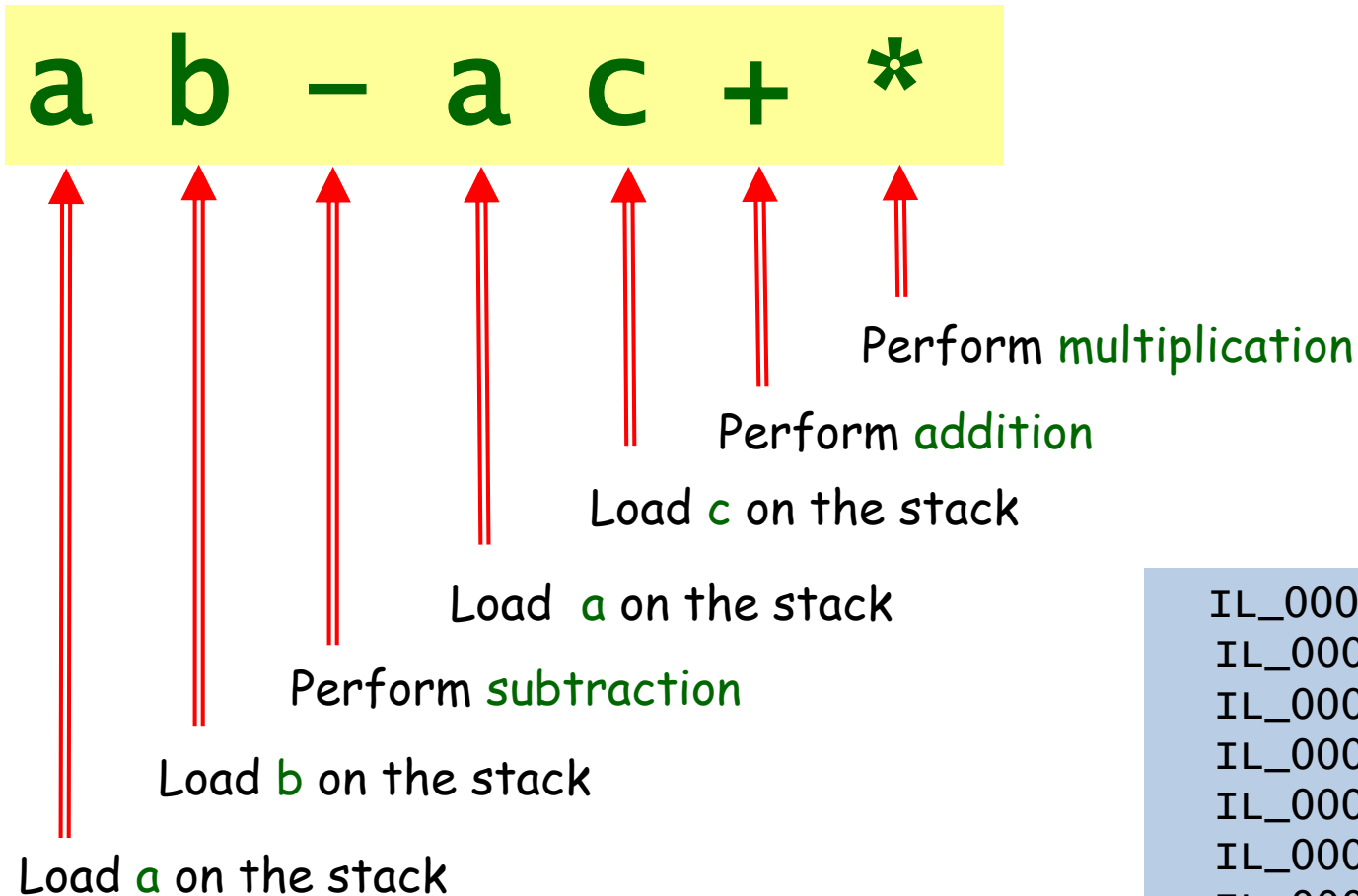
ПОЛИЗ

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!**

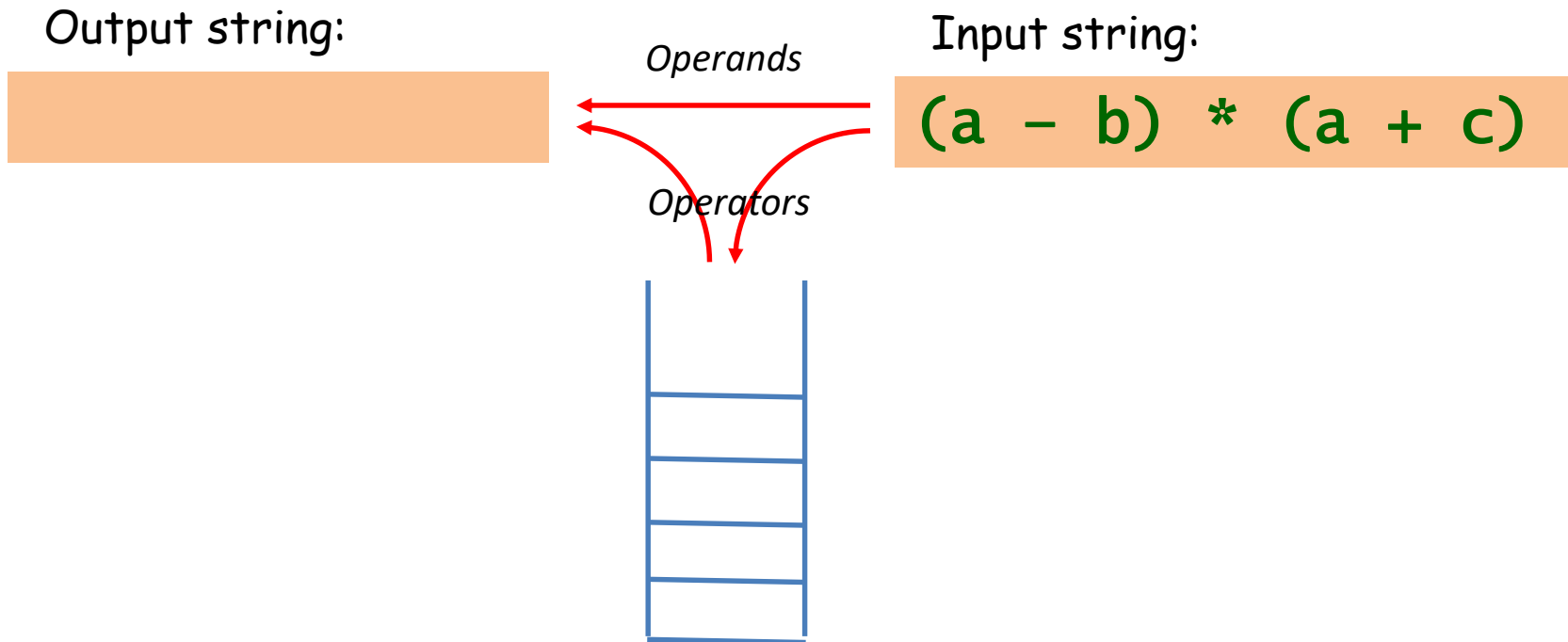
# Code Generation for VM (2)



```
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)



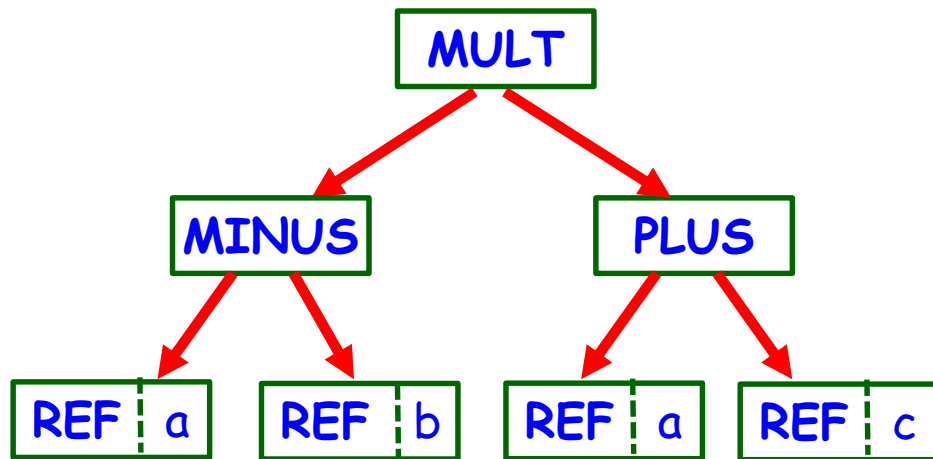
# Code Generation:

## 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"



## Process the operator node:

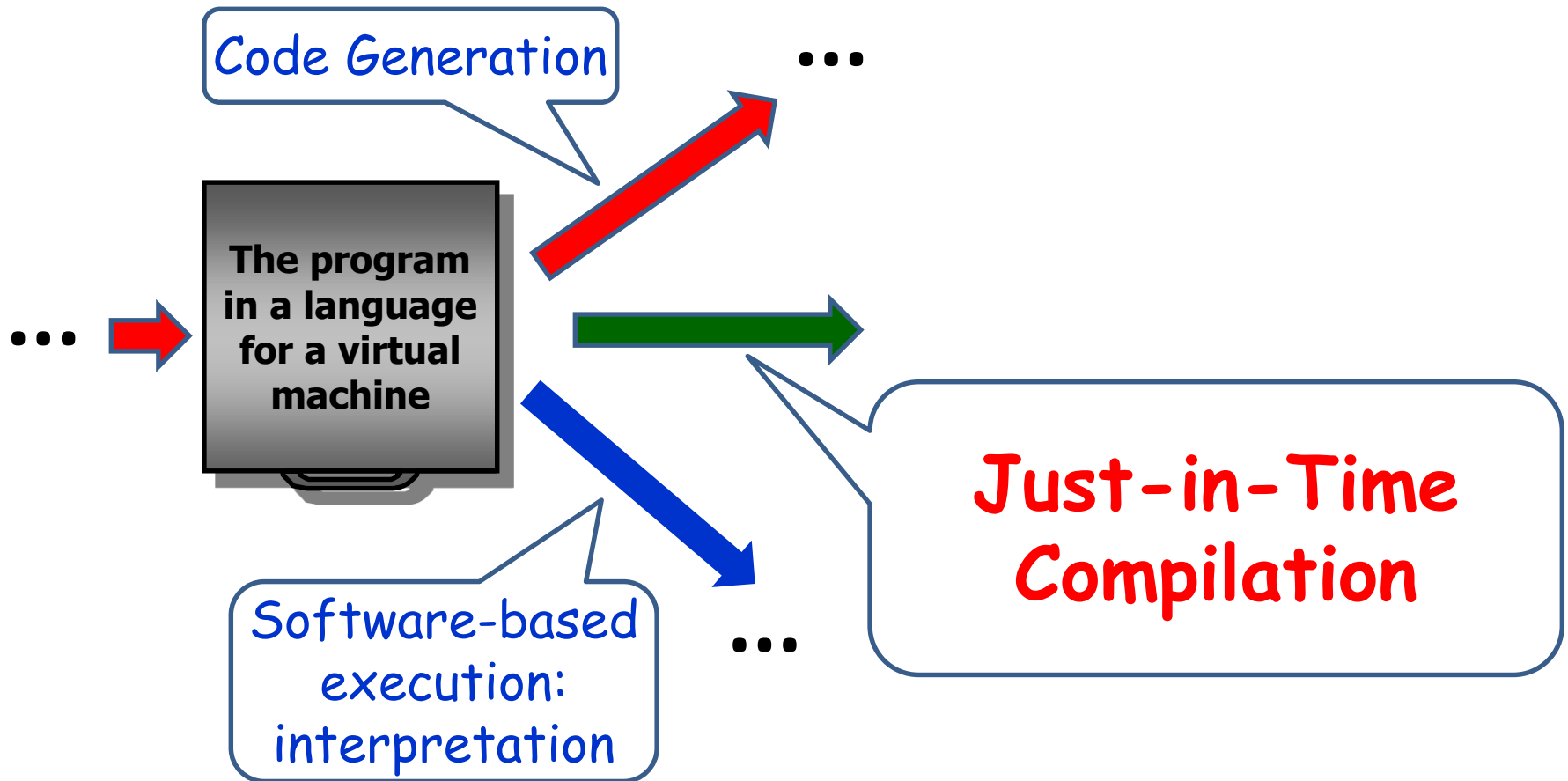
- Process the left subtree.
- Process the right subtree.
- Generate instruction code performing the operator from the root node.

## Process the terminal node:

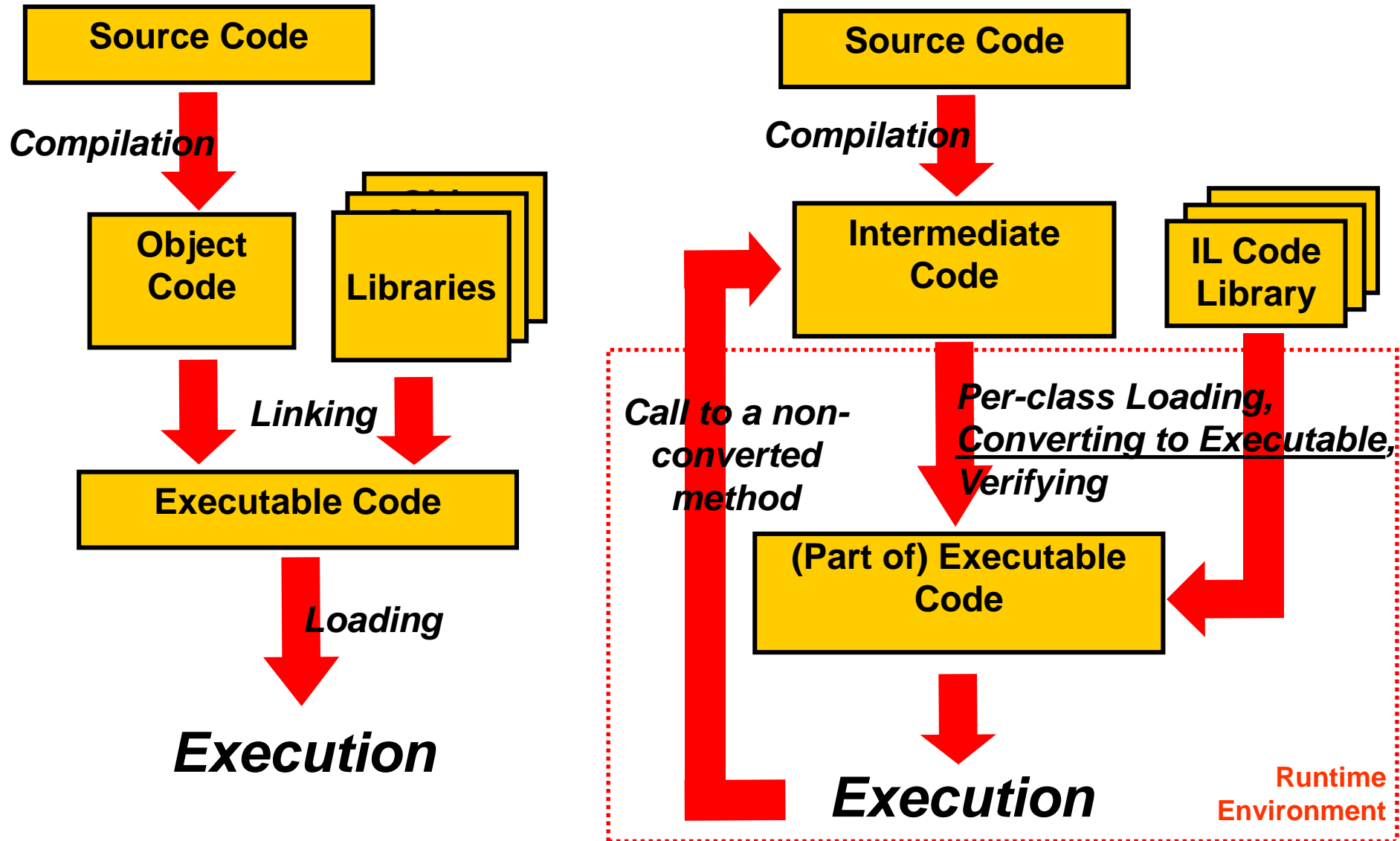
- Generate instruction code for loading operand to the stack.



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



# 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 @_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 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.

----- 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.