Compiler Construction: Practical Introduction

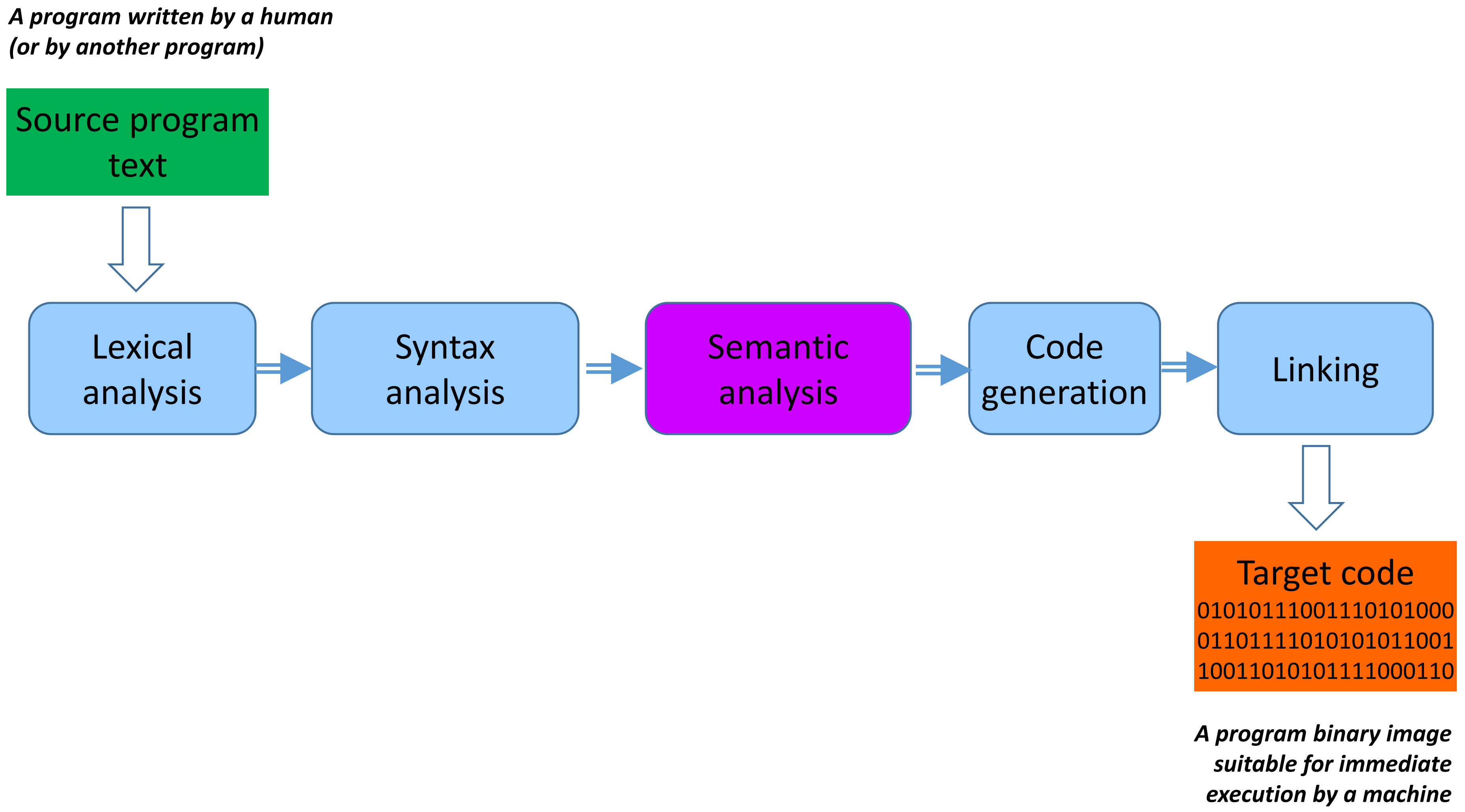
Samsung Compiler Bootcamp

Samsung Research Russia
Moscow 2019

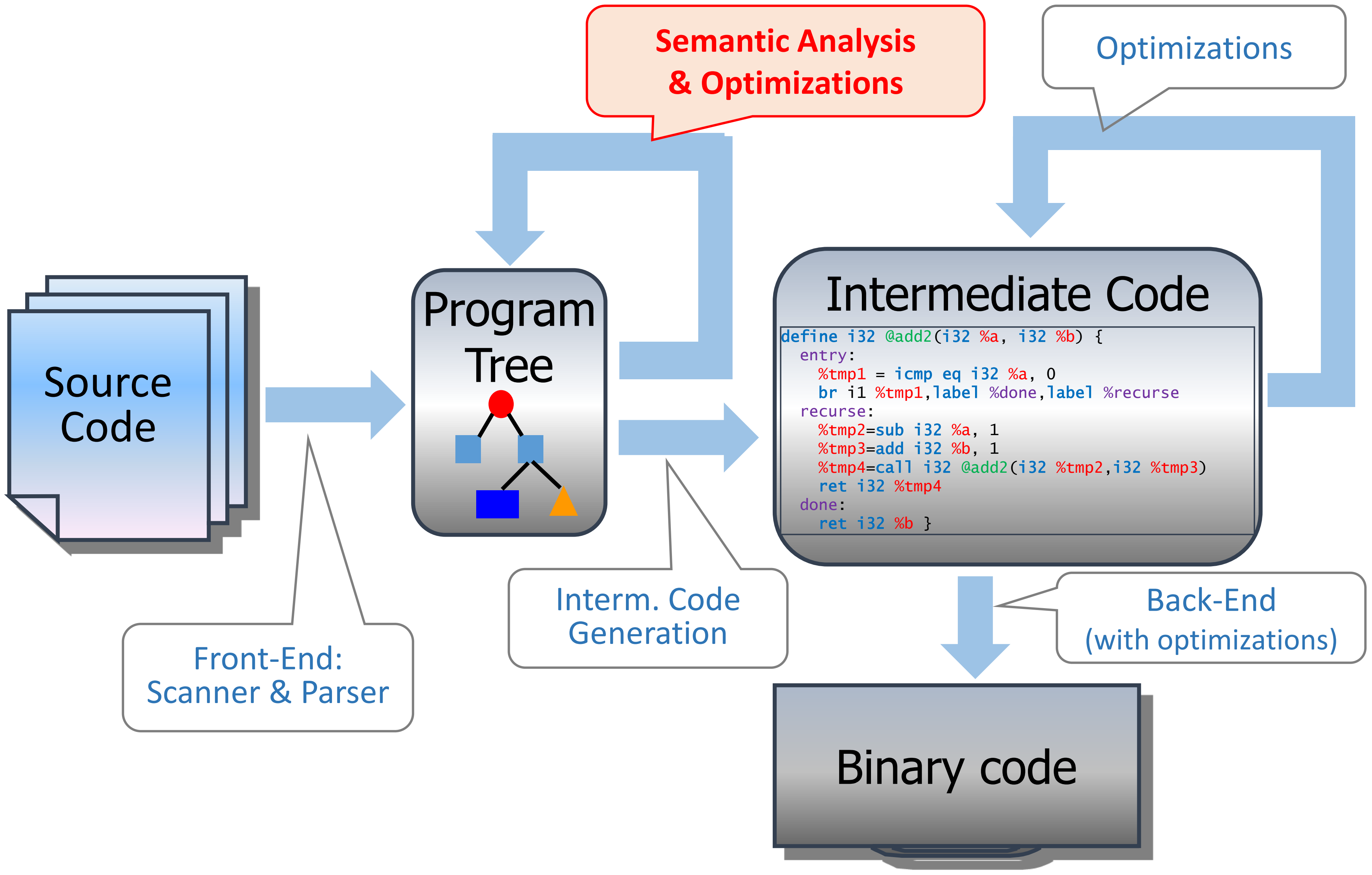
Lecture 5
Semantic Analysis
Optimization

- Why and for semantic analysis is?
- Examples: standard conversions, initialization semantics, user-defined conversions, calculating constant expressions
- Source code optimizations: the general idea
- Examples: eliminating repeated calculations, replacing slow instructions, excluding redundant calculations, constant propagation etc.

Compilation: An Ideal Picture



Where We Are Today



Semantic Analysis & Optimizations
Optimizations
Source Code
Program Tree
Intermediate Code
Front-End: Scanner & Parser
Interm. Code Generation
Back-End (with optimizations)
Binary code

```
define i32 @add2(i32 %a, i32 %b) {
  entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1,label %done,label %recurse
  recurse:
    %tmp2=sub i32 %a, 1
    %tmp3=add i32 %b, 1
    %tmp4=call i32 @add2(i32 %tmp2,i32 %tmp3)
    ret i32 %tmp4
  done:
    ret i32 %b }
```

What Is Semantic Analysis For?

```
void f(int p)
{
    int a, b;
    *a = 777;
    return xyz*a+b*f;
}
...
f();
int x = f(1,2);
```

Illegal operation (dereferencing) for the object of type **int**

Using uninitialized variables a and b

Undeclared variable xyz

Illegal operand types for operator *

Returning a value in **void** function

Illegal number of arguments in calls to function f

Illegal position for call to function f

Syntactically perfect program!..

What Is Semantic Analysis For?

Some remarks

1. Errors like "undeclared identifier» are typically detected on syntax analysis stage – while building symbol tables and/or program tree.
2. Errors like "uninitialized variable" usually are not detected by all compilers because it requires deeper control flow and data flow analysis.
3. Analysis of the code snippet ...xyz*a... typically results in a message like "illegal operand types for * operator". Formally that's true but in fact the reason is that xyz is not declared – this is an example of and ***induced error***.

Наведенные ошибки

Semantic Analysis

- Typically semantic analysis runs on the program tree built on previous compilation stages (while syntax analysis).
- Semantic analysis is typically implemented as a series of tree traverses with some actions related to the source language semantics.
- The more complex semantics is the more passes (traverses) are needed.
    - For relatively simple languages semantic analysis can be done **together** with syntax analysis while building the program tree.
    - Usually, the last tree walk implements target code generation – either an intermediate representation (like C--) or assembler code.
    - Often, before code generation, some **additional stages** after semantic analysis are necessary like building CFG & SSA representations…

Semantic Analysis

- One or several semantic actions are performed on each tree walk.
- What's the result of each tree walk?
  - Either a modified program tree with **the same node types**; perhaps complex nodes get replaced for simpler ones.

    Example is the C# compiler: after each tree walk the tree consists of the same node types.

  - Or a modified program tree **with different node types** that are more primitive but are "closer" to the target architecture.

    Example is the Scala compiler: node types representing source program constructs get replaced for more primitive nodes ("ICode"), and the JVM (or MSIL) code is generated form ICode finally.

Semantic Analysis

The result of each tree walk is typically twofold:

- The tree <u>changes its structure</u>: some nodes/subtrees are added or removed, some nodes/subtrees get replaced for other nodes/subtrees…
- Tree nodes <u>are annotated</u> (“decorated” ☺) <u>by attributes</u> reflecting various semantic features; the attributes are deduced during the analysis process.

=> The Abstract Syntax Tree (AST) is converted to the **Annotated** Syntax Tree (AAST).

(An alternative solution is **attribute grammars**.)

Semantic Analysis: Actions

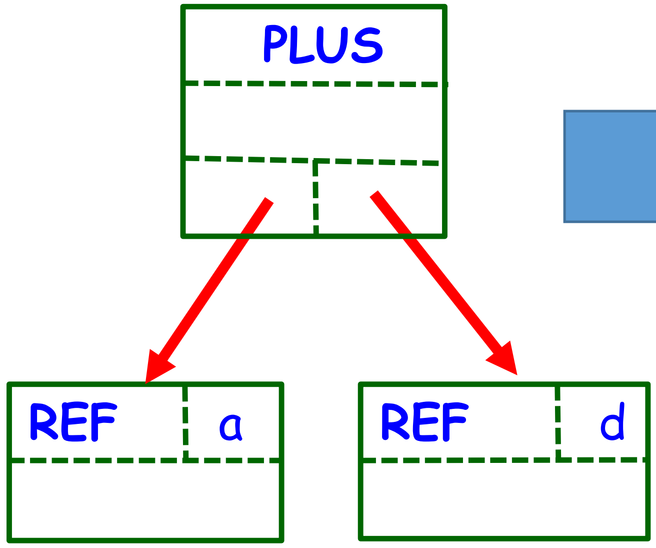
Four categories of actions while semantic analysis:

- Semantic checks
  - Operand types consistency in expressions
    Проверки корректности конструкций (деструктор)
- Semantic conversions
  - Replacing conversions for function calls
  - Replacing infix operators for operator function calls
  - Inserting necessary type conversions
  - Template instantiating
- Identification of hidden semantics
  - Implicit destructor calls
    Temporary objects
- Optimizations (!)

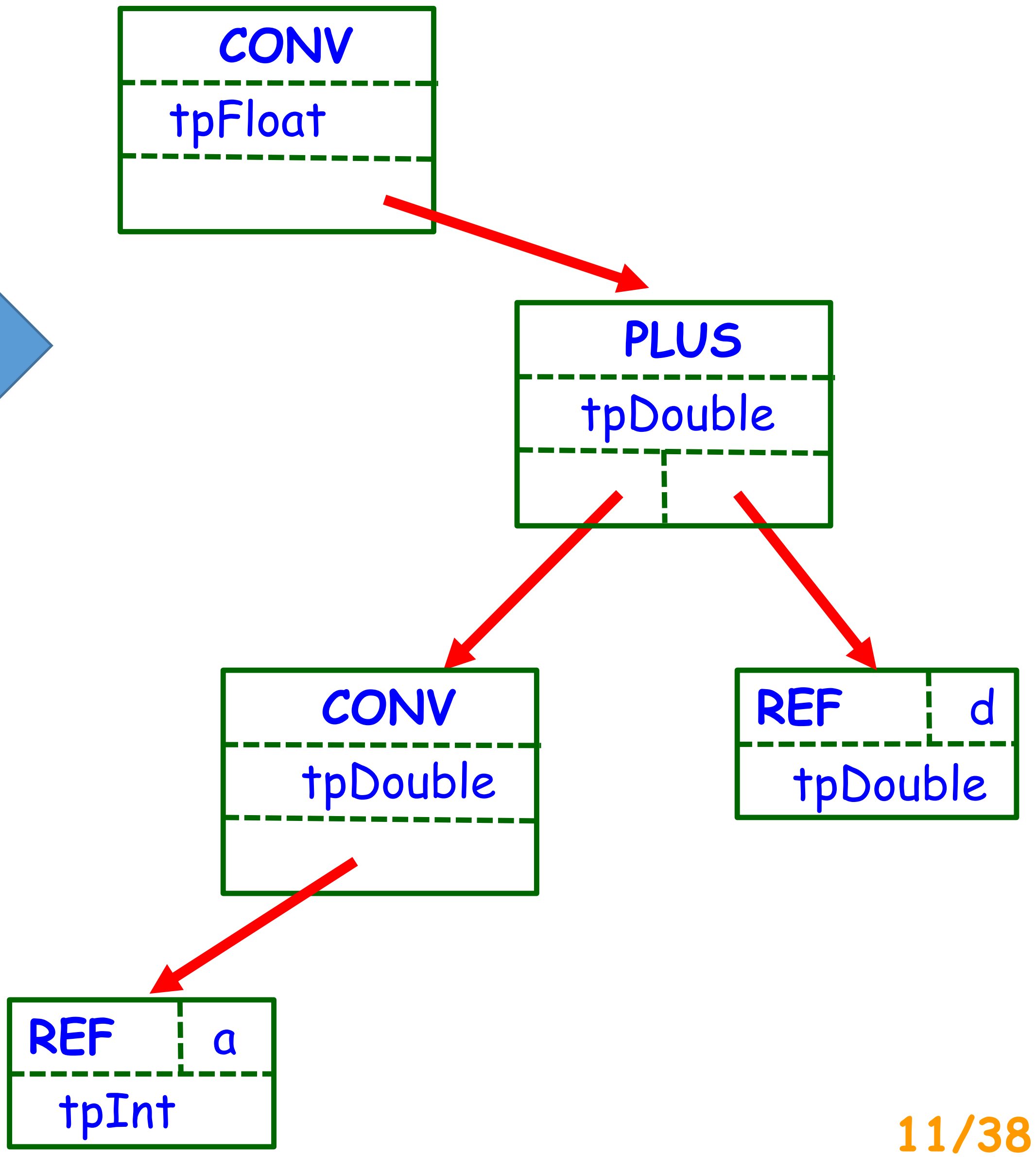
Semantic Analysis: Example 1

Standard conversions

```
int a = 3;
double d = 7.55;
float x = a + d;
```



```
...
float x = (float)((double)a + d);
```



Semantic Analysis: Example 2

Initialization semantics

```
class C { ... };
...
C c1;
C c2(1);
C c3(c2);
C c4 = 7;
C c5 = c1;
```

- Allocate memory for c1 object;
- Call **default constructor** of C for c1.

- Allocate memory for c2 object;
- Call **C(int)** constructor for c2.

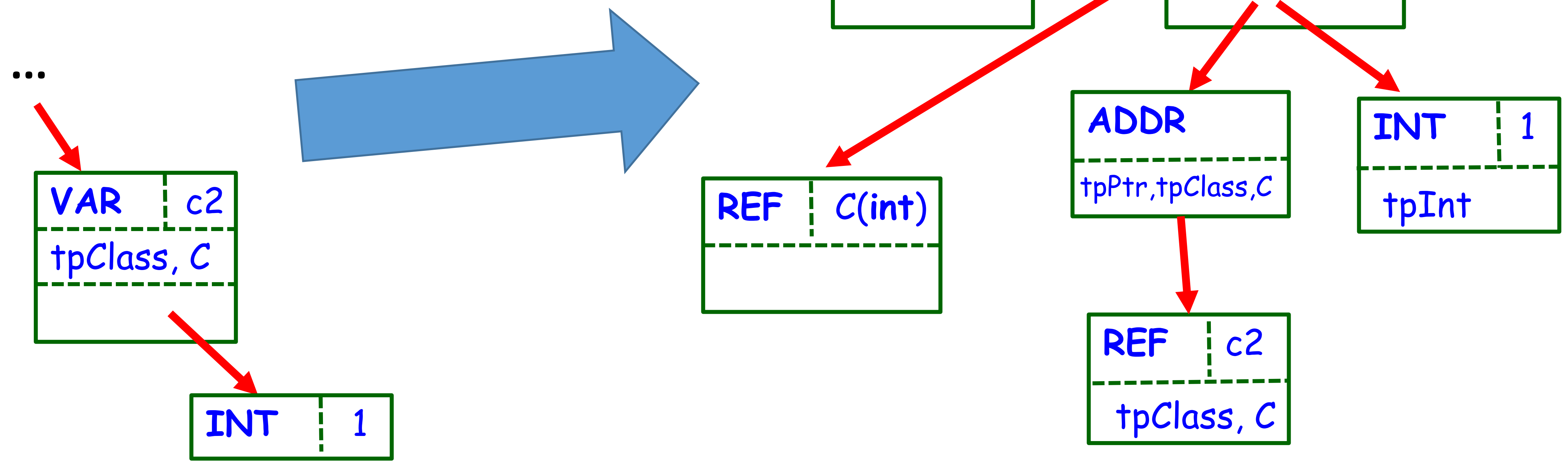
- Allocate memory for c3 object;
- Call **copy constructor** for c3.

- Allocate memory for c4 object;
- Create temporary object tmp;
- Call **C(int)** constructor for tmp;
- Call **copy constructor** for c4.

Semantic Analysis: Example 2

Initialization semantics

```
class C { … };
…
C c2(1);
```



Semantic Analysis: Example 3

User-defined conversions

```
class C {
private:
    bool m;
public:
    operator bool() { return m; }
};
...
C c;
...
if ( c ) ...
```

$$\texttt{if (c) ...} \leftarrow \boxed{\texttt{if ((bool)c) ...}} \leftarrow \boxed{\texttt{if (c.operator bool()) ...}}$$

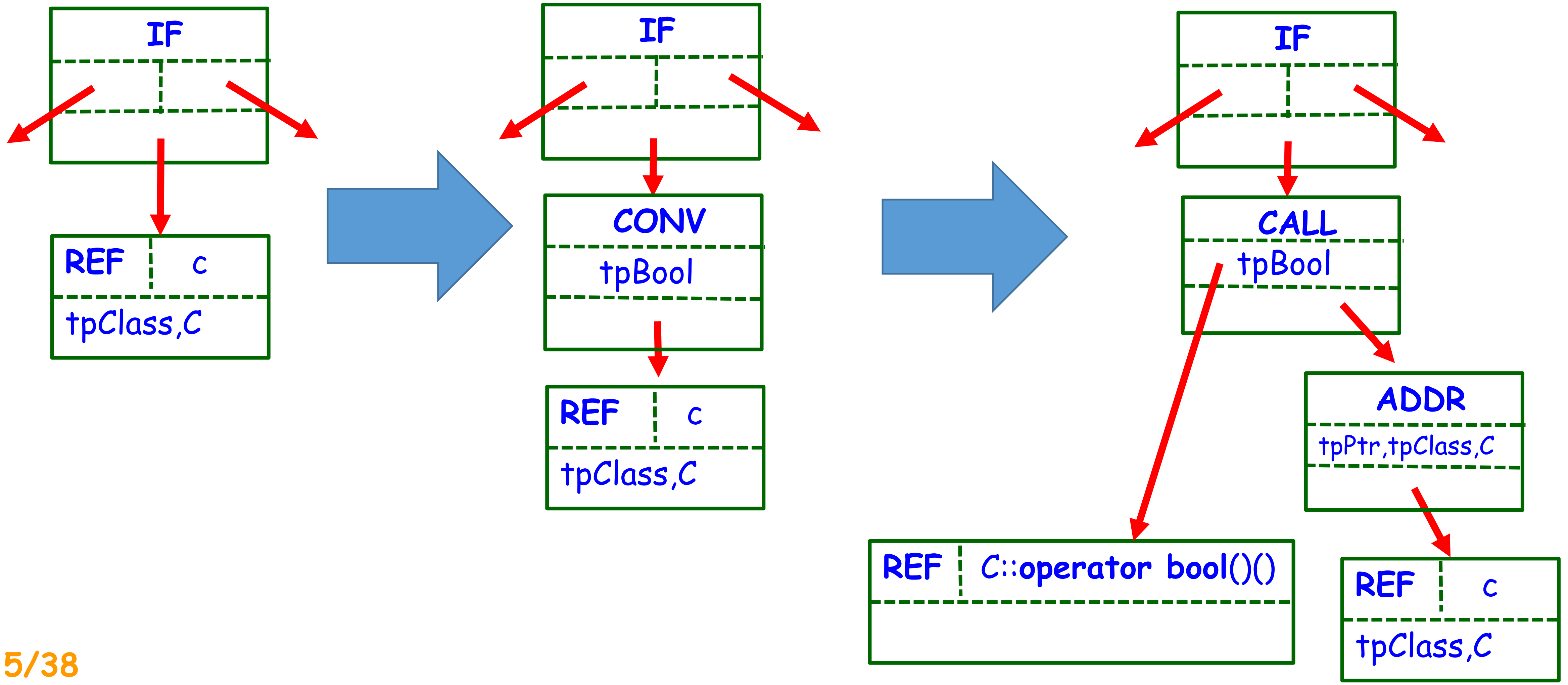
Semantic Analysis: Example 3

```
if ( c ) …
```

```
if ( (bool)c ) …
```

```
if ( c.operator bool() ) …
```

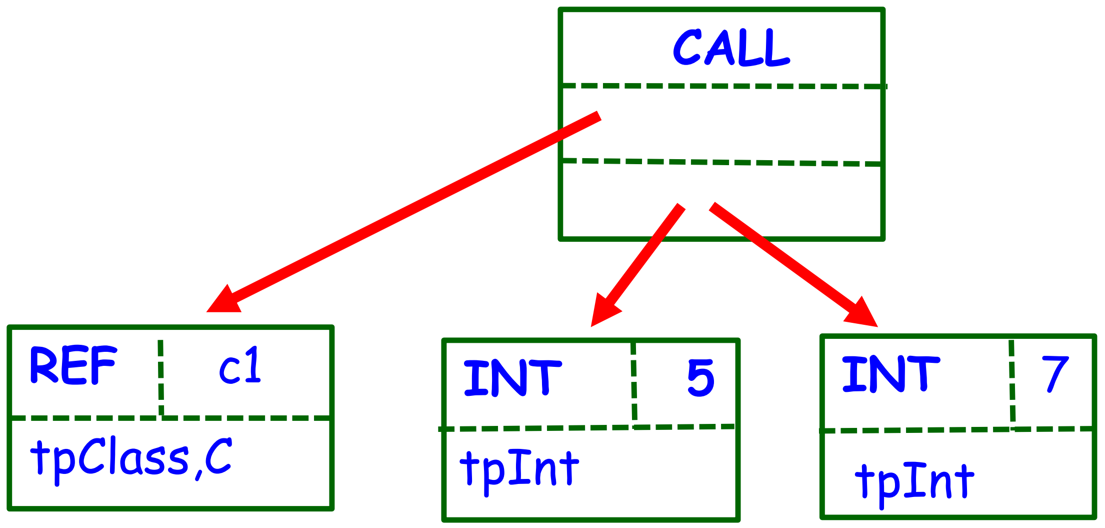
```
if ( C::operator bool(&c) ) …
```



Semantic Analysis: Example 4

Functional objects ("functors")

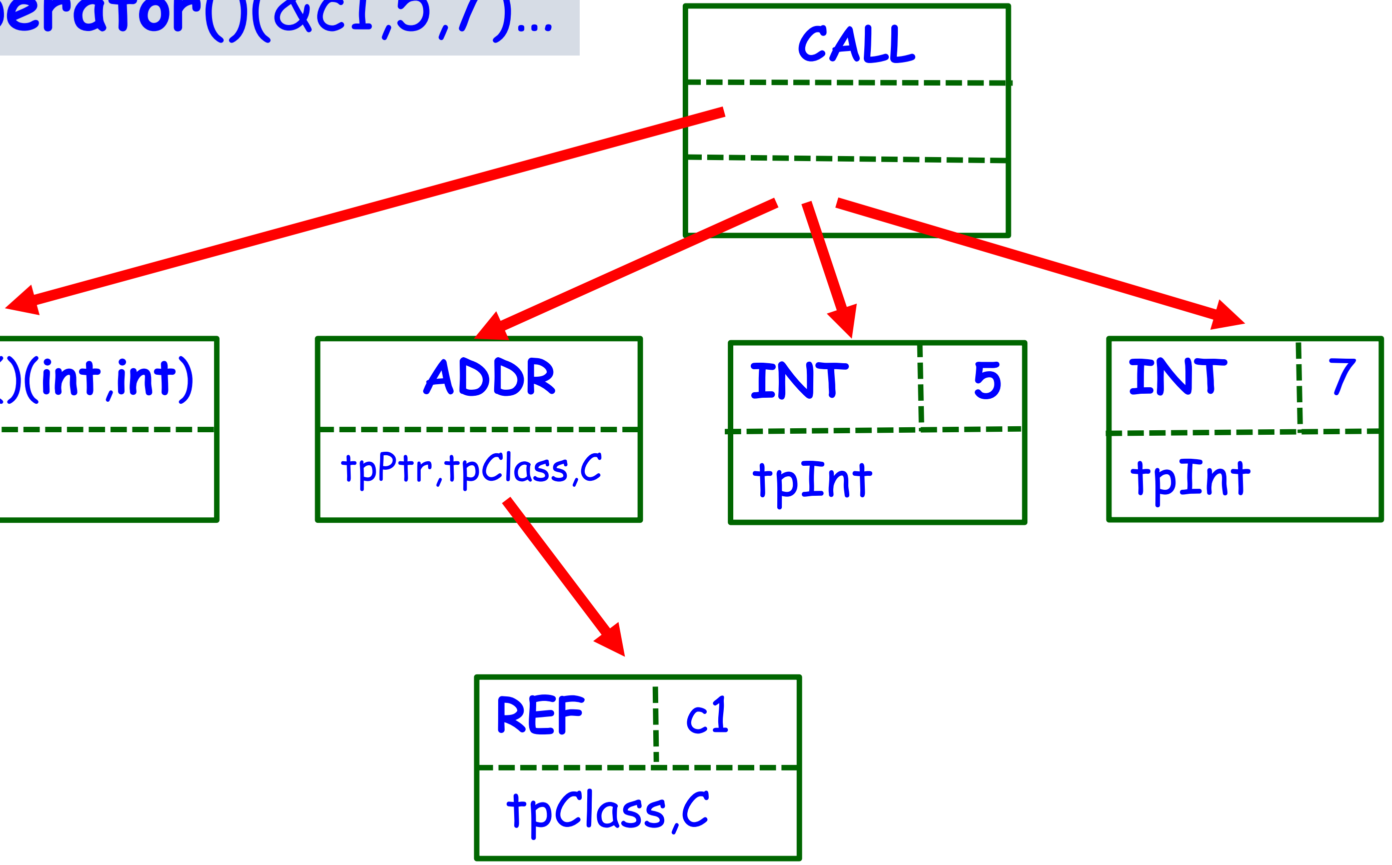
```
class C {
public:
    int operator()(int a,int b)
    { return a+b; }
};
...
C c1;
...
int res = c1(5,7);
```



Semantic Analysis: Example 4

Functional objects ("functors")

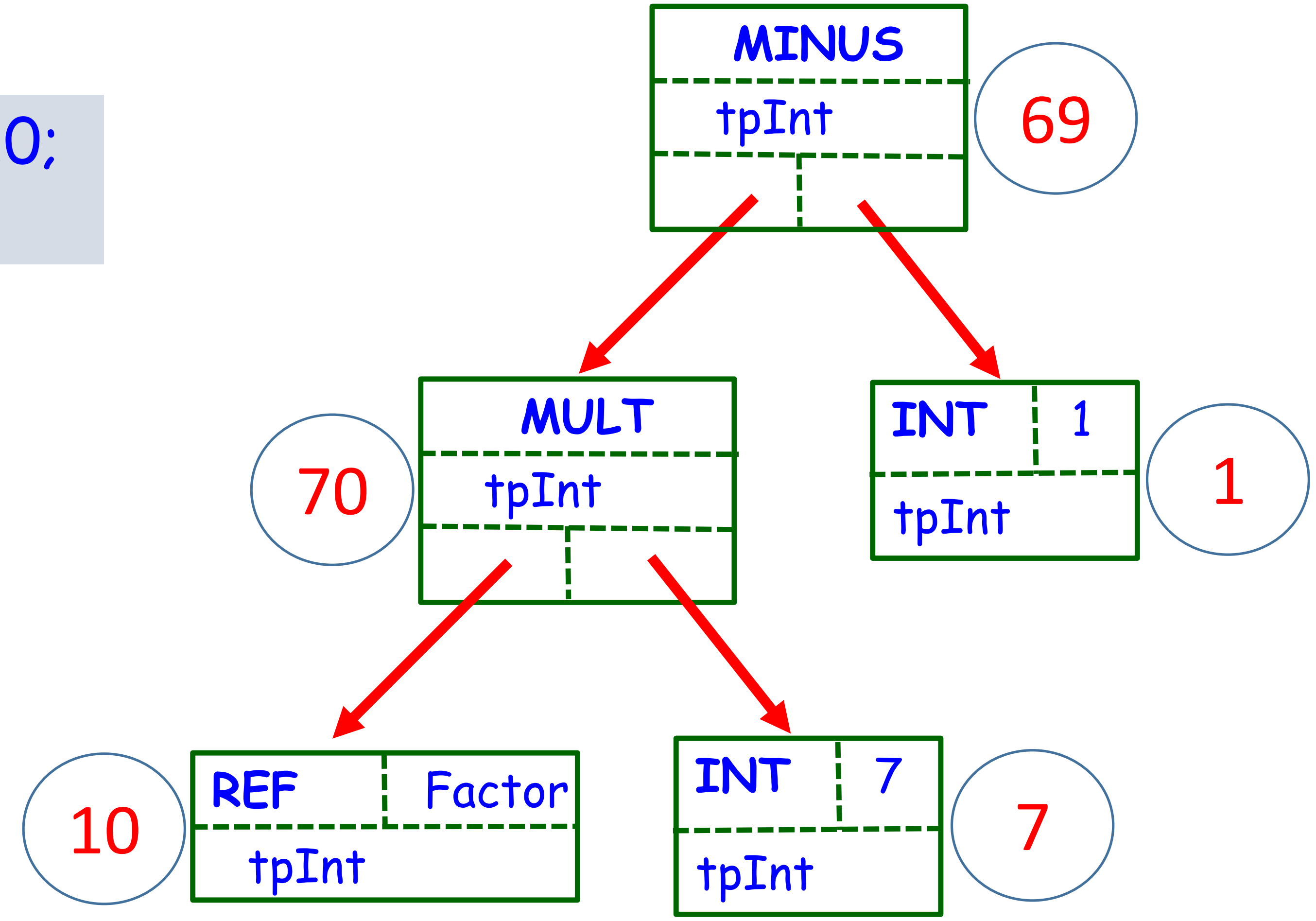
...c1(5,7)... → ...C::**operator**()(&c1,5,7)...



Semantic Analysis: Example 5

Calculating constant expressions

```
const int Factor = 10;
int A[Factor*7-1];
```



Program Optimization

Some general points

- Optimization can be performed on each stage of the program lifecycle: not only while compilation but while design, development and maintenance.
- Do we **really need** optimization?
  The best way to optimize a program – is **to design it correctly** (then perhaps we do not need to optimize it ☺)
- "Optimization-in-the-small" vs "optimization-in-the-large"

Program Optimization

- Finding places in programs which could be optimized (by some criteria) is very much empirical job; in the best case, there is just a set of techniques taken from experience.
- At the same time, there is a number of formal and/or constructive approaches for some kind of optimizations.

Today, we will be discussing what to optimize, but not how to do this…

Program Optimization

- While source code processing (lexical & syntax analysis)

  Big spectrum of optimization techniques.
- While semantic analysis (AST processing).

  Sequential AST traversing.
  Optimizations depend on the language semantics heavily.
- While target code generation (machine-dependent optimizations)

  Depend on the target architecture & on the instruction set.
- While linking: ***global*** code optimizations.

  Example: – C++ code bloat removing.

Common Subexpression Elimination(1)

```
long a = x*(1-sin(y));
long b = x + y/z;
long c = y/z + 1 - sin(y);
```

```
long tmp1 = 1-sin(y);
long tmp2 = z/y;

long a = x*tmp1;
long b = x + tmp2;
long c = tmp2 + tmp1;
```

The place:

While AST analysis.

Limitations:

1. Factorized functions cannot issue side effects.
2. Operands of factorized expressions cannot modify their values.

Common Subexpression Elimination (2)

```
static int x = 0;
long F(long y)
{
    x++;
    return <expression>;
}
```

Side effect

```
long a = x*(1-F(y));
long b = x + y/z;
...
z = <expression>;
...
long c = z/y + 1 - F(y);
```

Modifying value

Common Subexpression Elimination (3)

An expression may look different but still calculate the same value as some other expression => it can also get optimized.

```
long a = b*c - d;

long e = b;

b = b + 1 - b*c;

long f = b*c + c*e;
```

b*c: the second calculation of the same value.

The second calculation of b*c results in a different value.

The value of c*e is the same as b*c.

Operation Strength Reduction (1)

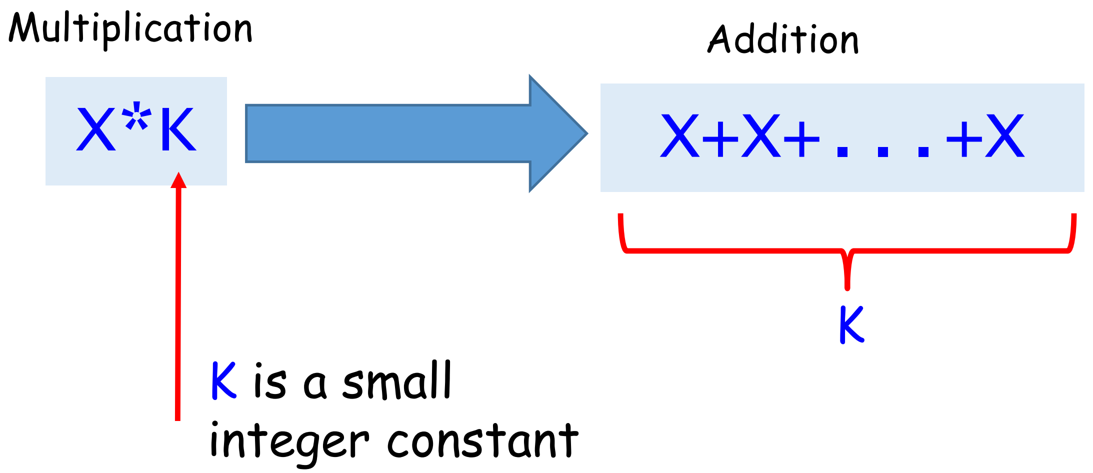
Actions: comparative performance

Multiplication/division on a power of two
Addition/subtraction
Multiplication
Division
Calculation of an integer power
Calculation of an arbitrary power

$\Rightarrow$ Replacing slower operations for faster ones (where possible)

For some target architectures it's **mandatory**: e.g., some RISCs just do not support multiplication!

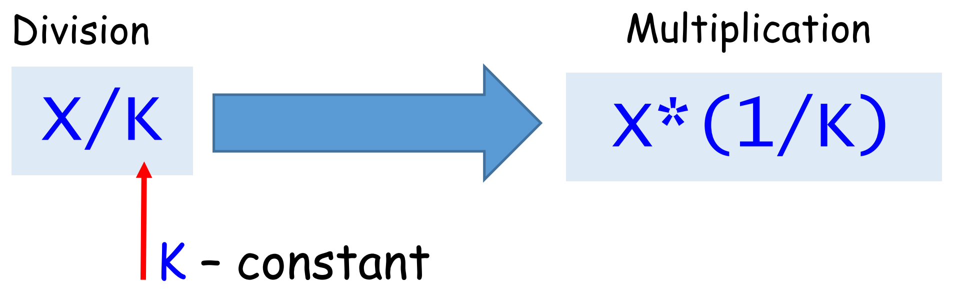
Operation Strength Reduction (2)



In general case it's impossible...

- At least one operand must be an integer constant.
- The constant should be relatively small; otherwise rounding errors will accumulate.

Operation Strength Reduction (3)



```
double x = c/b;
double y = (e+f)/b + d;
double z = b;
b = b+1;
...
z = sin(x)/z + e/b;
```

```
double tmp = (double)1/b;
double x = c*tmp;
double y = (e+f)*tmp + d;
double z = b;
b = b+1;
...
z = sin(x)*tmp + e/b;
```

Dead Code Elimination

```
double a;
...
a = (x+y)*sin(z);
...
a = x/y;
```

```
double a;
...
a = x/y;
```

If the value of a does not change between two assignments, then the first assignment can be removed.

Limitation: the action being removed cannot make side effects.

Constant propagation

If the value of a variable is known
then the variable reference can be replaced
for the value itself.

```
long a = 2;
long b = 3;
...
long c = a*b;
...
long t = (b+c)*a+x;
```

```
long a = 2;
long b = 3;
...
long c = 6; // a*b
...
long t = 18 + x; // (b+c)*a
```

Conditional Constant propagation

If the value in a loop condition is known in advance then the loop could be simplified.

```
while ( <Expression> )
{
    <Statements>
}
```

```
Loop:
  if ( !<Expression> ) goto Exit;
  <Statements>
  goto Loop;
Exit:
  ;
```

```
while ( 1 )
{
    <Statements>
}
```

```
Loop:
  ~~if ( !<Expression> ) goto Exit;~~
  <Statements>
  goto Loop;
~~Exit:~~
  ~~;~~
```

Type conversion optimizations

Type conversion is a potentially costly operation; therefore it's a good candidate for optimizations.

```
double a, b;
long i, j;
...
double c = a + i + b - j;
```

```
... a + (double)i + b - (double)j ...
```

```
double a, b;
long i, j;
...
double c = (a+b) + (i-j);
```

```
... a+b + (double)(i-j) ...
```

Code Hoisting

Access to array elements and function calls are also good candidates for optimizations...

```
for i:integer range 1..100 loop
    x(i) = y(i)+1/y(i);
    z(i) = y(i)**2;
end loop;
```

Address of `y(i)` gets calculated **300 times**

```
for i:integer range 1..100 loop
    declare
        tmp : real := y(i);
    begin
        x(i) = tmp+1/tmp;
        z(i) = tmp**2;
    end;
end loop;
```

Address of `y(i)` gets calculated **100 times**

Loop Fusion (1)

Loops are main consumers of CPU time!

```
for i:integer range 1..100 loop
    x(i) = 0;
end loop;
```

```
for i:integer range 1..100 loop
    z(i) = y(i)**2;
end loop;
```

```
for i:integer range 1..100 loop
    x(i) = 0;
    z(i) = y(i)**2;
end loop;
```

Costs for loop organization are reduced

Loop Fusion (2)

(More general case)

```
for i:integer range 1..100 loop
    x(i) = 0;
end loop;
```

```
for i:integer range 1..200 loop
    z(i) = y(i)**2;
end loop;
```

Overall amount of iterations: **300**

```
for i:integer range 1..100 loop
    x(i) = 0;
    z(i) = y(i)**2;
end loop;
```

```
for i:integer range 101..200 loop
    z(i) = y(i)**2;
end loop;
```

Overall amount of iterations: **200**

Loop Unrolling (1)

```
for (int i=0; i<100; i++)
{
    x[i] = y[i]*z[i];
}
```

Loop step = **1**
Overall amount
of iterations: **100**

```
for (int i=0; i<100; i+=2)
{
    x[i] = y[i]*z[i];
    x[i+1] = y[i+1]*z[i+1];
}
```

Loop step = **2**
Overall amount
of iterations: **50**

Loop Unrolling (2)

```
-- Skip past blanks, loop is opened up for speed
while Source (Scan_Ptr) = ' ' loop
    if Source (Scan_Ptr + 1) /= ' ' then
        Scan_Ptr := Scan_Ptr + 1; exit;
    end if;
    if Source (Scan_Ptr + 2) /= ' ' then
        Scan_Ptr := Scan_Ptr + 2; exit;
    end if;
    if Source (Scan_Ptr + 3) /= ' ' then
        Scan_Ptr := Scan_Ptr + 3;  exit;
    end if;
    if Source (Scan_Ptr + 4) /= ' ' then
        Scan_Ptr := Scan_Ptr + 4; exit;
    end if;
    if Source (Scan_Ptr + 5) /= ' ' then
        Scan_Ptr := Scan_Ptr + 5; exit;
    end if;
    if Source (Scan_Ptr + 6) /= ' ' then
        Scan_Ptr := Scan_Ptr + 6; exit;
    end if;
    if Source (Scan_Ptr + 7) /= ' ' then
        Scan_Ptr := Scan_Ptr + 7; exit;
    end if;
    Scan_Ptr := Scan_Ptr + 8;
end loop;
```

A real example:
The scanner of the Ada GNAT compiler

Tail Recursion Elimination (1)

```
void f(int x)
{
    if ( x == 0 ) return;
    ...Some actions...
    f(x-1);
}
```

```
void f(int x)
{
    if ( x == 0 ) return;
    ... Some actions...
    if ( Some_condition )
        f(x-1);
    else
        f(x-2);
}
```

The idea:
If the recursive call is the very last operation in the function body then it can be replaced for the direct jump to the beginning of the body – perhaps with argument (re)initialization.

See more details in:
http://en.wikipedia.org/wiki/Tail_call

Tail Recursion Elimination (2)

```
long factorial(long n)
{
   if (n == 0) return 1;
   else return n*factorial(n-1);
}
```

This is **not** tail recursion.
Why?

```
int fac_times(int n, int acc)
{
    if (n == 0) return acc;
    else return fac_times(n-1,acc*n);
}

int factorial(int n)
{
    return fac_times(n,1);
}
```

Equivalent program **with** tail recursion.