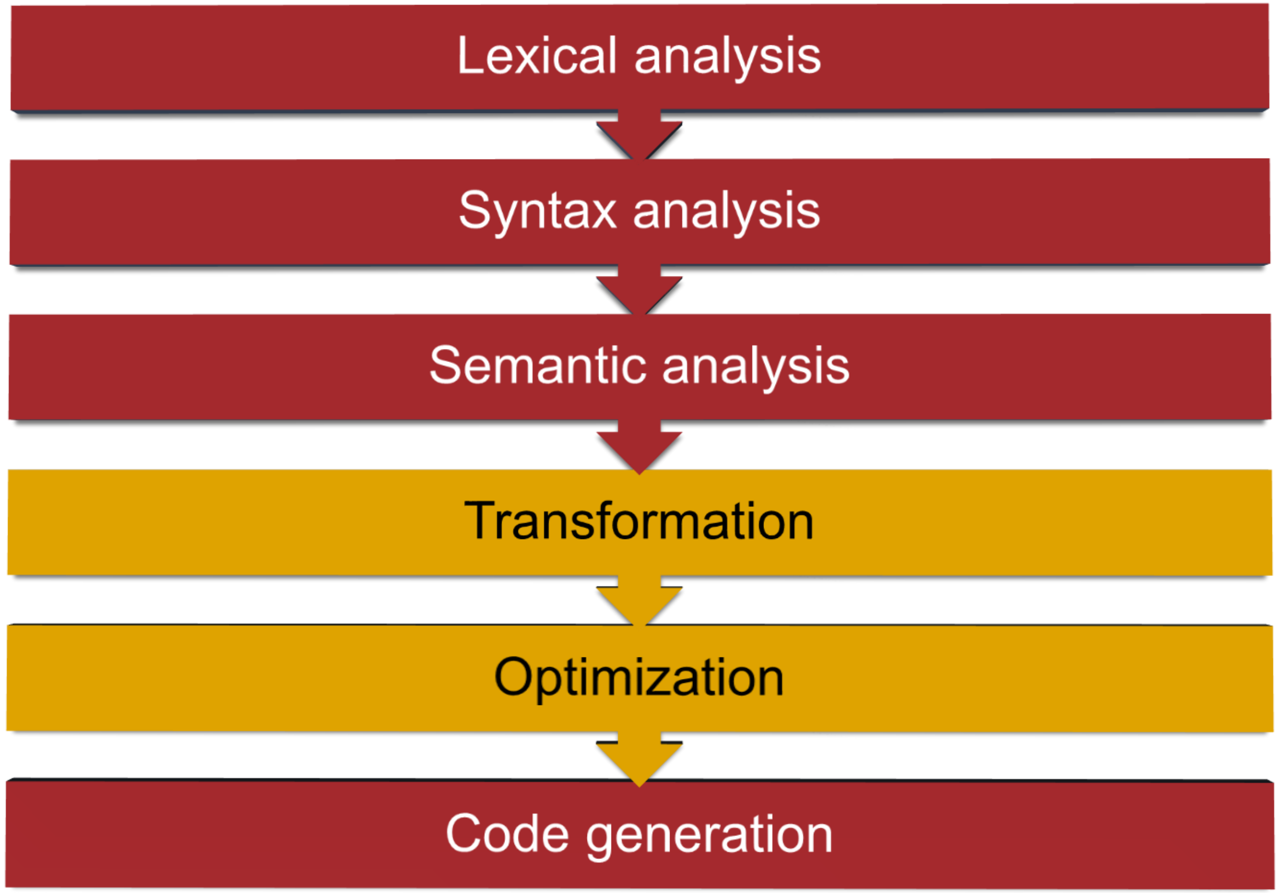
Model-based Software Development

Lecture 5

Transformation, optimization

Dr. Balázs Simon

Compilation phases



This lecture: Transformation, Optimization

- I. Transformation
- II. Type mapping
- III. Statement mapping: SSA
- IV. Optimization
- V. Optimization techniques

Transformation

- Goal: map syntax tree to intermediate representation (IR)
  - memory layout of types and data structures
  - statements to simple operations that can be mapped to machine code directly
  - no resource limit yet (e.g., no limit on the number of registers)
- Possible intermediate representations:
  - 1. No explicit IR: machine code directly from syntax tree
  - 2. Three-Address Code (TAC or 3AC): each operation has max. 3 operands (e.g., `t1 = t2 * t3`)
  - 3. Static Single-Assignment (SSA): like TAC, but all variables are immutable

This lecture: Transformation, Optimization

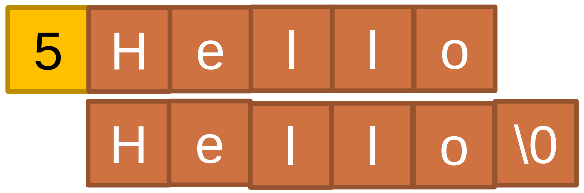
- I. Transformation
- II. Type mapping
- III. Statement mapping: SSA
- IV. Optimization
- V. Optimization techniques

Memory layout of simple types

- Primitive types:
  - pointer, bool, char, byte, short, int, long, float, double, …
  - must be aligned to addressable memory space, usually 1 byte is the minimal size
  - handle: bool values, char encoding, big/little endian, fixed/floating point, calling convention, …
- Enum/Flags:
  - mapped to integral types (byte, short, int, long, …)
  - size is dependant on the possible values, but in some languages it can be specified explicitly (e.g., C#)
- String type:
  - primitive or composite type, depending on the language
  - simple (e.g., Pascal, C#): length + characters
  - composite (e.g., C): character array ending with '\0'

"Hello"

"Hello"

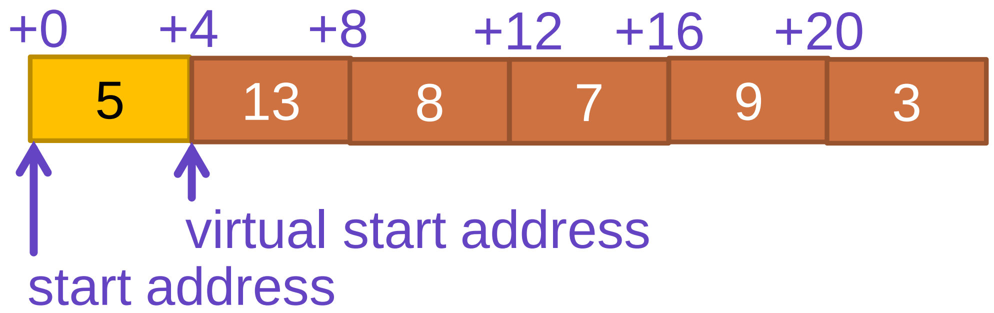


Memory layout of composite types: Array

(examples: for 32-bit architecture)

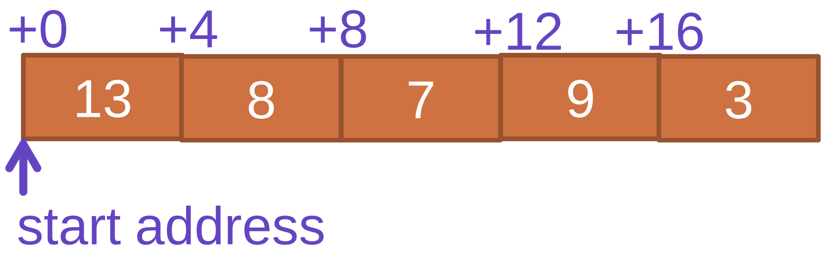
- Kinds:
  - static: fix size, reserved at compile time, on the stack (e.g., C)
  - dynamic: fix size, reserved at runtime, on the heap (e.g., C malloc, C#)
  - flexible: dynamic size at runtime (e.g., Python)
- Checked size (e.g., C#): length + items

```
int[] a = new int[] { 13, 8, 7, 9, 3 };
```



- Unchecked size (e.g., C): items

```
int a[] = { 13, 8, 7, 9, 3 };
```

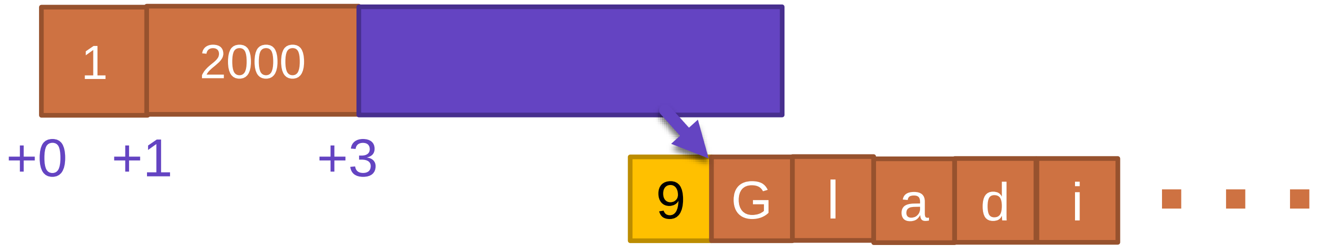


Memory layout of composite types: Struct (Union: overlapping fields)

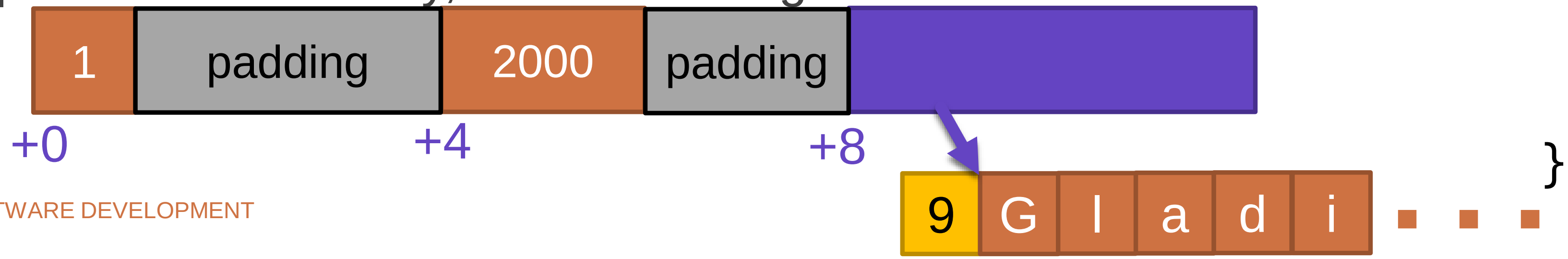
```
var m = new Movie() { Genre = Genre.Drama, Year = 2000, Title = "Gladiator" }
```

(examples: for 32-bit architecture)

- Packed:
  - size = sum of the sizes of the fields



- Aligned:
  - fields aligned to words
  - requires more memory, but addressing is more efficient




```
struct Movie
{
    Genre Genre;
    short Year;
    string Title;
}

enum Genre : byte
{
    Action = 0,
    Drama = 1,
    Romance = 2,
    Comedy = 3
}
```

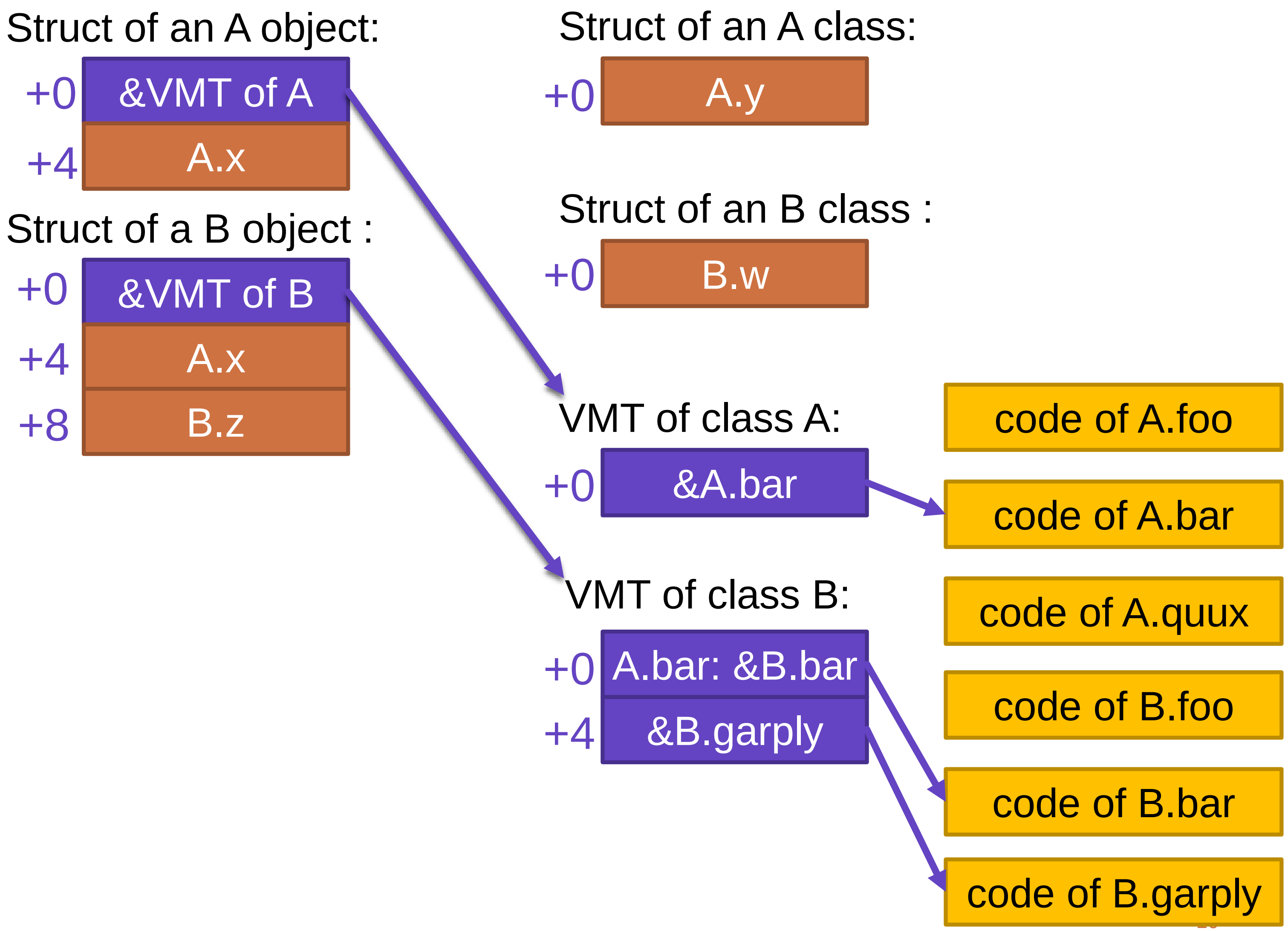
Memory layout of composite types: Class and object

- Object: instance fields of the class stored as a struct
  - 0^{th} field: pointer to the type descriptor (class)
  - the struct also contains the fields of the base classes
- Class: static fields + pointers to static and instance methods
  - static fields: stored as a struct
  - instance methods: implicit 0^{th} parameter for the object (this)
  - statikus methods: no extra parameter for the object
  - polymorphism: virtual method table (VMT)
- Constructor:
  - reserve object struct on the heap + initialization

Memory layout of composite types: Class and object

(examples: for 32-bit architecture)

```
class A
{
  int x;
  static int y;
  void foo() { ... }
  virtual void bar() { ... }
  static void quux() { ... }
}

class B : A
{
  int z;
  static int w;
  new void foo() { ... }
  override void bar() { ... }
  virtual void garply() { ... }
}
```



This lecture: Transformation, Optimization

- I. Transformation
- II. Type mapping
- **III. Statement mapping: SSA**
- IV. Optimization
- V. Optimization techniques

Static Single Assignment (SSA)

- Each variable gets its value exactly once
  - at the definition, before it is used
- Notation: original variable name numbered (versioned)
- Each operation has max. 3 operands (e.g., `t1: t2 * t3`)
- Example:

Program code:

```
x = 3;
y = 4;
z = 5;
x = x+y;
z = (x+y)*z;
```

→

SSA code:

```
x1: 3
y1: 4
z1: 5
x2: x1+y1
t1: x1+y1
z2: t1*z1
```

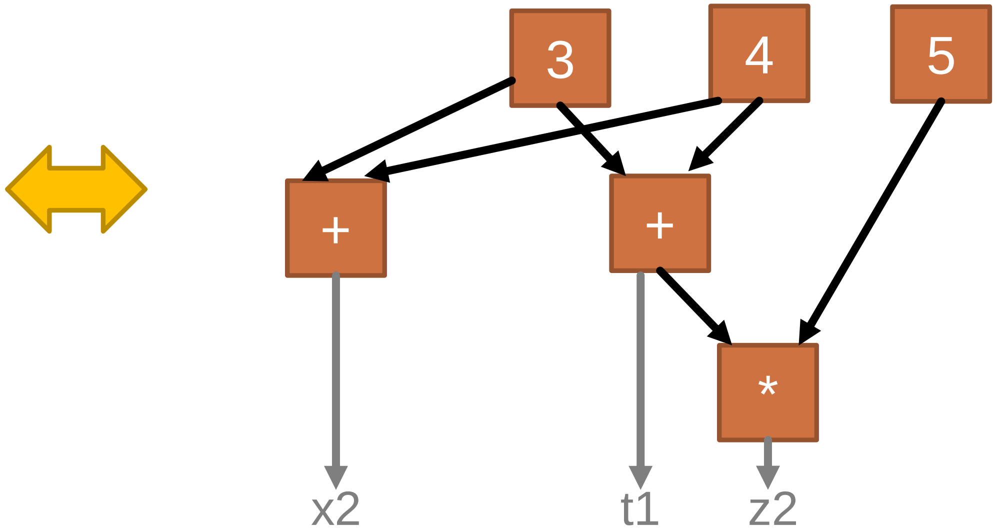
SSA graph

- Visual representation of the SSA code as a dataflow-graph (DFG)
  - Nodes: constants or operators
  - Directed edges: definition-use connections (reversed edges: data dependencies)
- Some optimizations are more intuitive in graph form

SSA code:

```
x1: 3
y1: 4
z1: 5
x2: x1+y1
t1: x1+y1
z2: t1*z1
```

Graph:



Φ-function

Question: what happens when multiple control flows meet?

```
if (x < y) z = x;
else z = y;
w = z;
```

⇨

```
if (x1 < y1) z1 = x1;
else z2 = y1;
w1 = ???
```

A special notation is needed: Φ-function

Value: the parameter corresponding to the actual control branch.

```
if (x < y) z = x;
else z = y;
w = z;
```

⇨

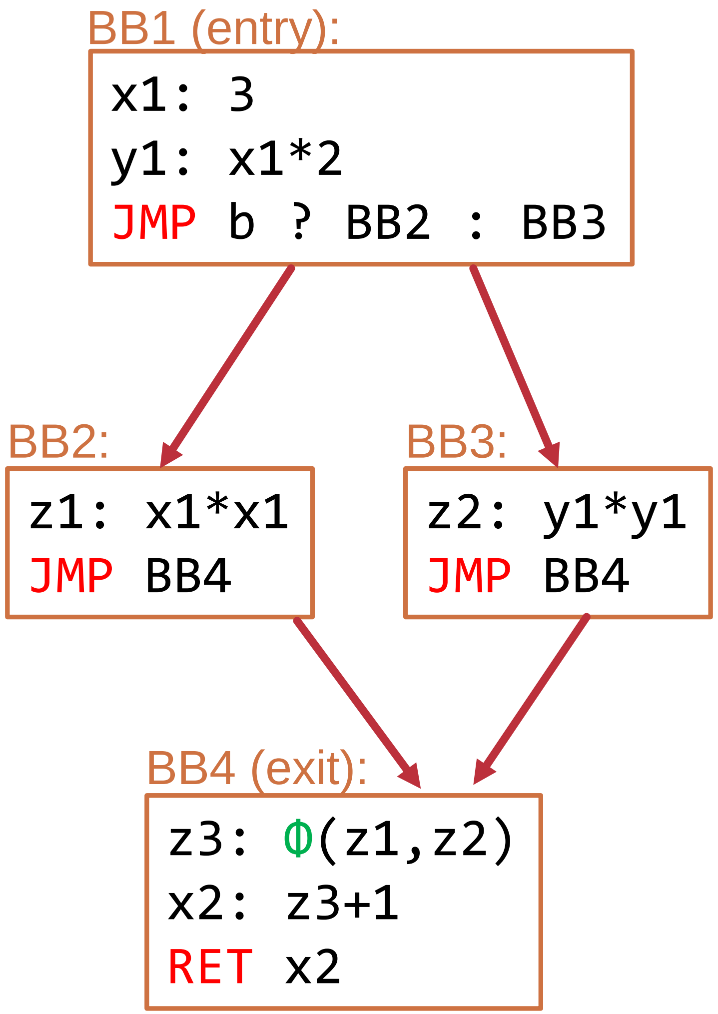
```
if (x1 < y1) z1 = x1;
else z2 = y1;
z3 = Φ(z1,z2);
w1 = z3;
```

Mapping of the program code

- Whole program: split into functions
- Function: control-flow graph (CFG)
  - node: basic block
  - edge: jump from one block to another
  - two special blocks: entry block and exit block
- Basic block:
  - statement series of maximum length (SSA statements)
  - atomic: if one statement is executed in a block, then all other statements within the block are executed, too
  - starts with a label, does not contain any other labels
  - ends with a jump to a label or with a conditional jump to a label, contains no other jumps (calling another function is not a jump)

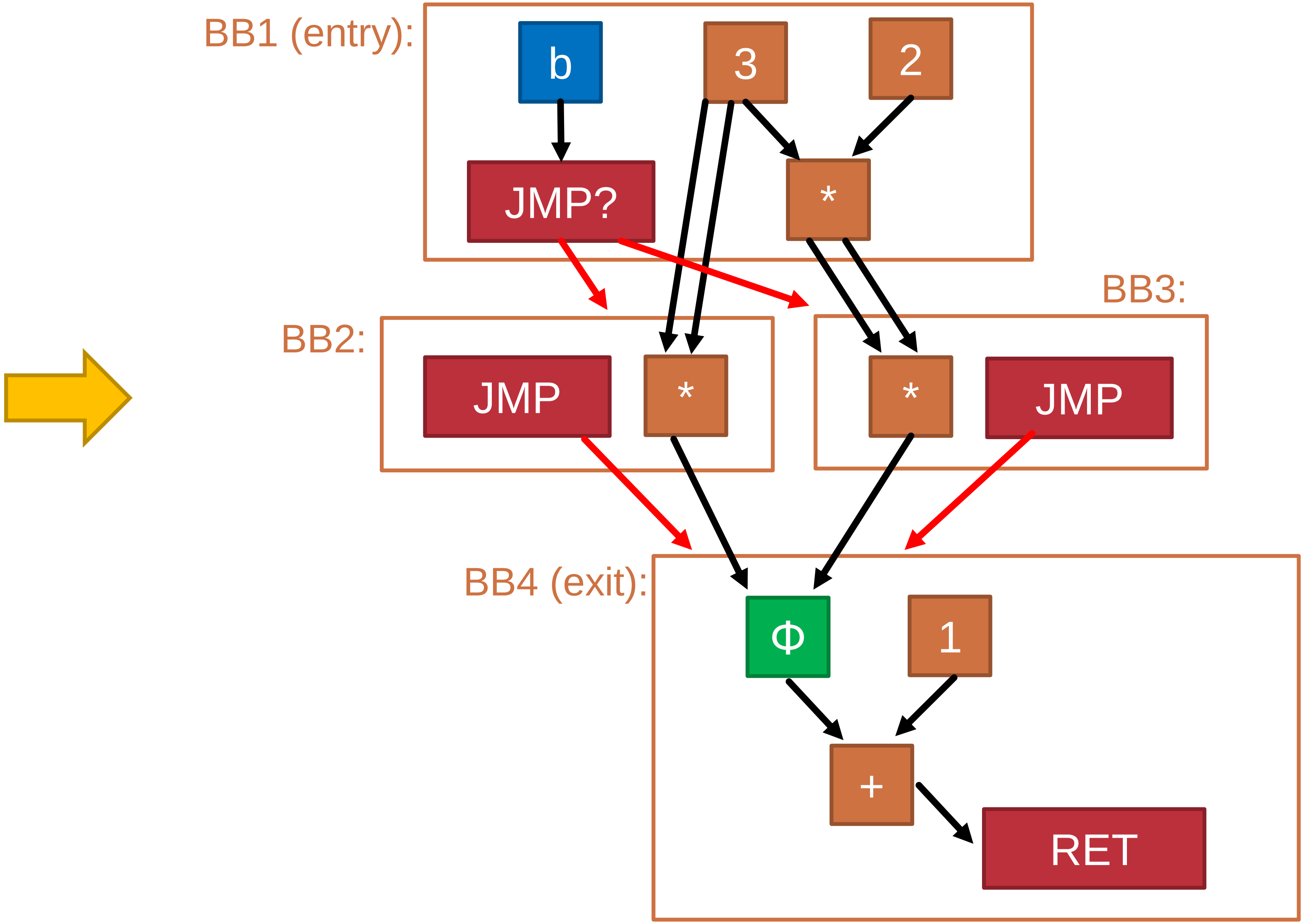
Control-flow graph example

```
int foo(bool b)
{
  int x = 3;
  int y = x*2;
  int z;
  if (b) z = x*x;
  else z = y*y;
  x = z+1;
  return x;
}
```



Control-flow and data-flow graphs

```
int foo(bool b)
{
  int x = 3;
  int y = x*2;
  int z;
  if (b) z = x*x;
  else z = y*y;
  x = z+1;
  return x;
}
```



Mapping of statements

- Arithmetic and logical expressions: mapped as is
  - some languages require overflow checking (expensive!)
- Branches, loops, exceptions: basic blocks and control edges
- Memory access: pointer arithmetics (base address + relative address)
  - read, write
  - some languages require index checking (expensive!)
- Type conversion: mapped to the corresponding operations
- Function call:
  - save state, reserve stack, pass control, free stack, restore state

Arithmetic and logical expressions

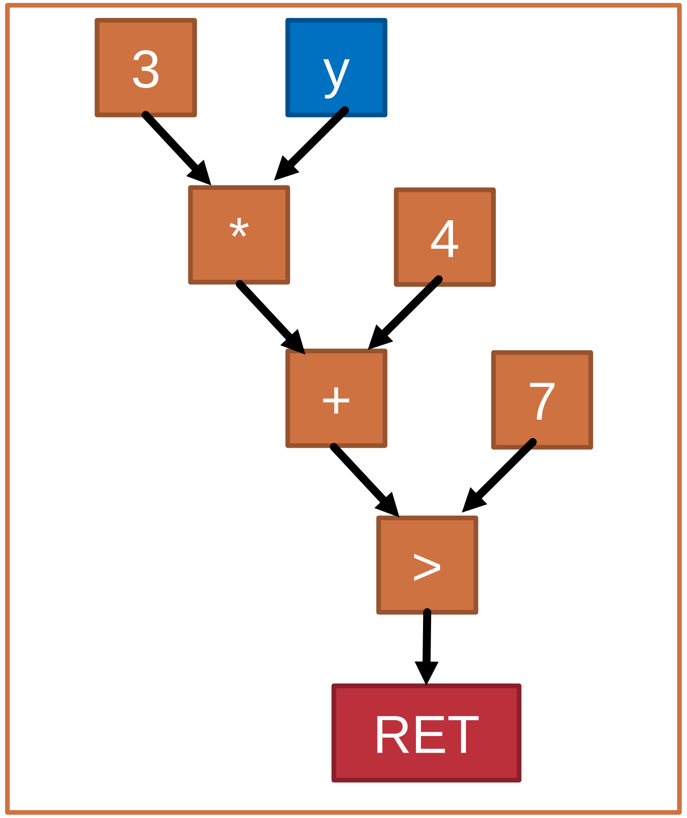
Program code:

```
bool foo(int y)
{
  return 3 * y + 4 > 7;
}
```

SSA:

```
t1: 3 * y
t2: t1 + 4
t3: t2 > 7
RET t3
```

Graph:

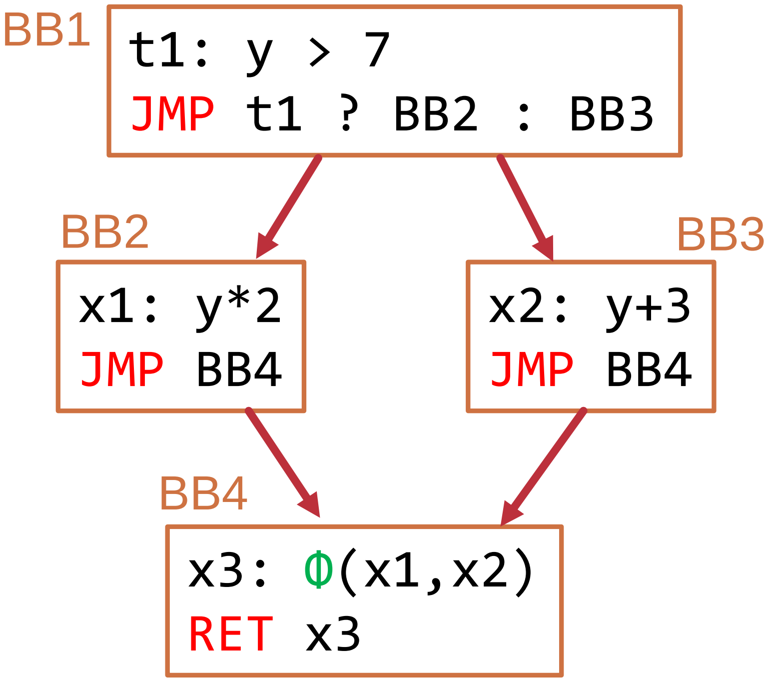


Branching: if

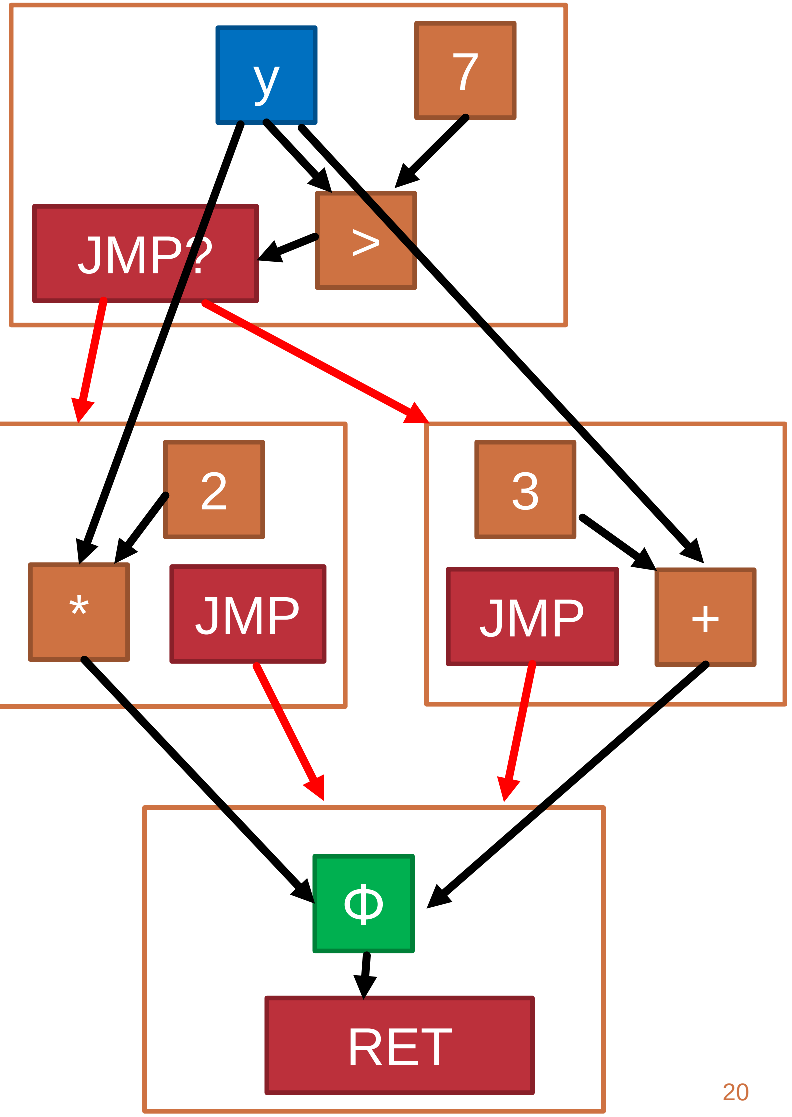
Program code:

```
int foo(int y)
{
   int x;
   if (y > 7)
   {
      x = y*2;
   }
   else
   {
      x = y+3;
   }
   return x;
}
```

SSA:



Graph:

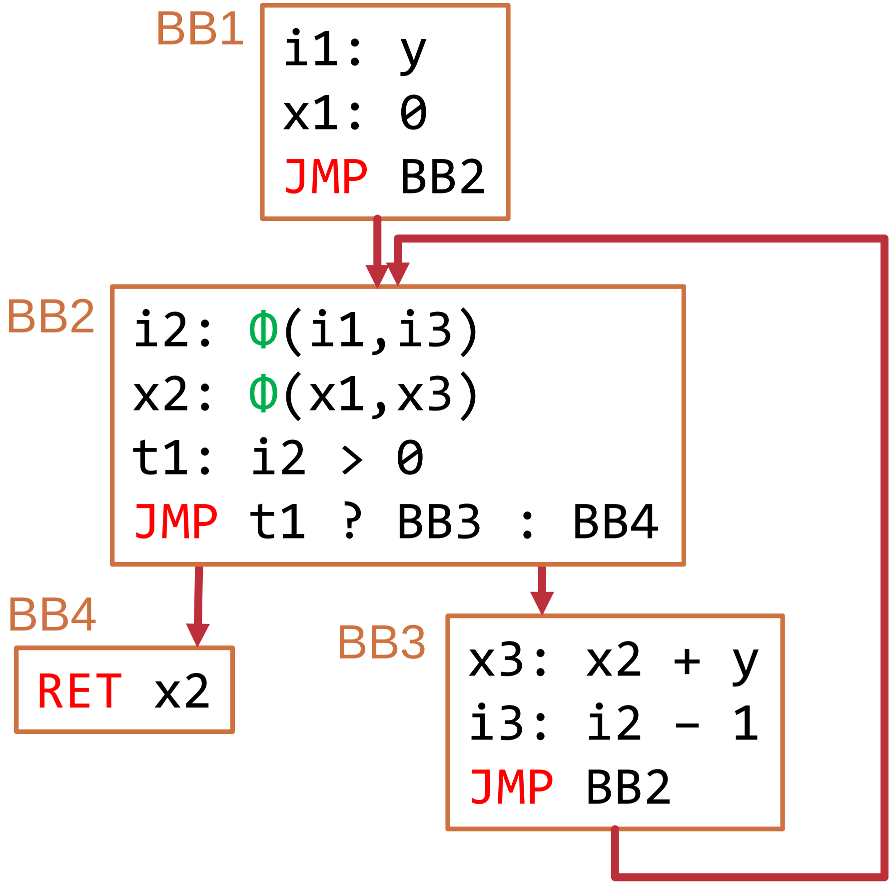


Loop: while

Program code:

```
int foo(int y)
{
  int i = y;
  int x = 0;
  while (i > 0)
  {
    x += y;
    --i;
  }
  return x;
}
```

SSA:

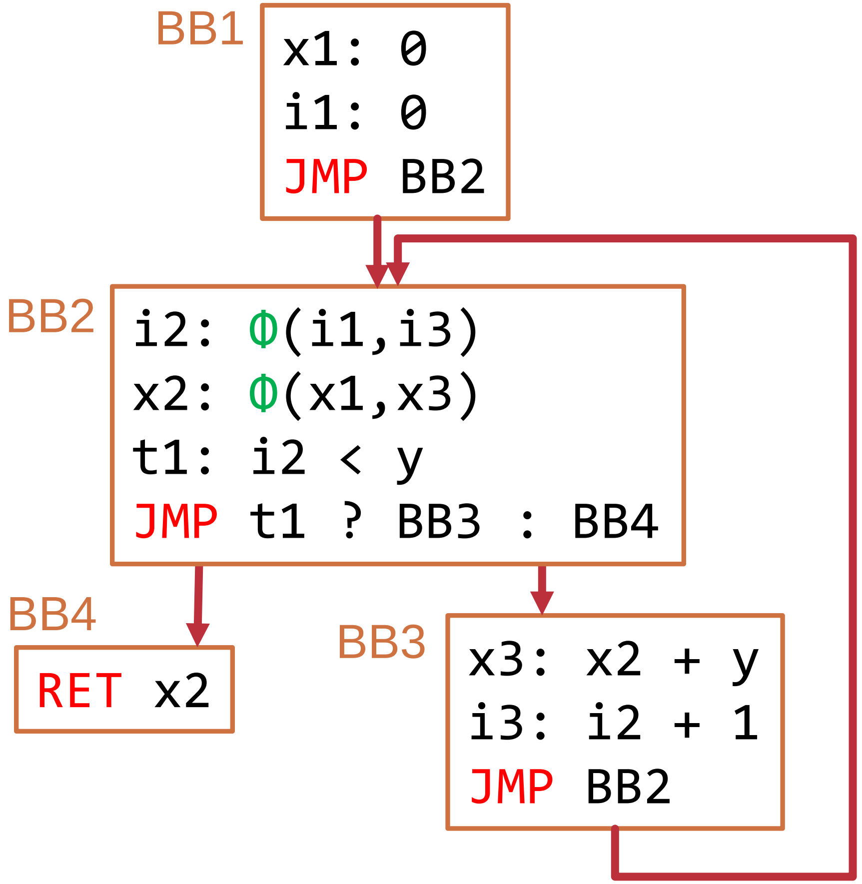


Loop: for

Program code:

```
int foo(int y)
{
  int x = 0;
  for (int i = 0; i < y; ++i)
  {
    x += y;
  }
  return x;
}
```

SSA:

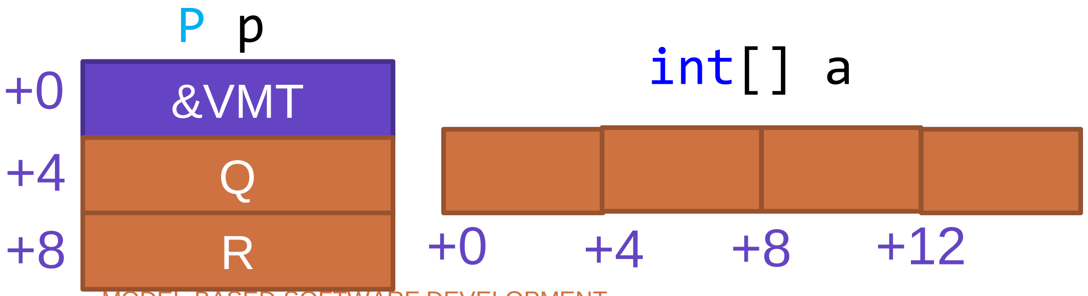


Memory: read (LD = load), write (ST = store)

Address: base + offset

Program code:

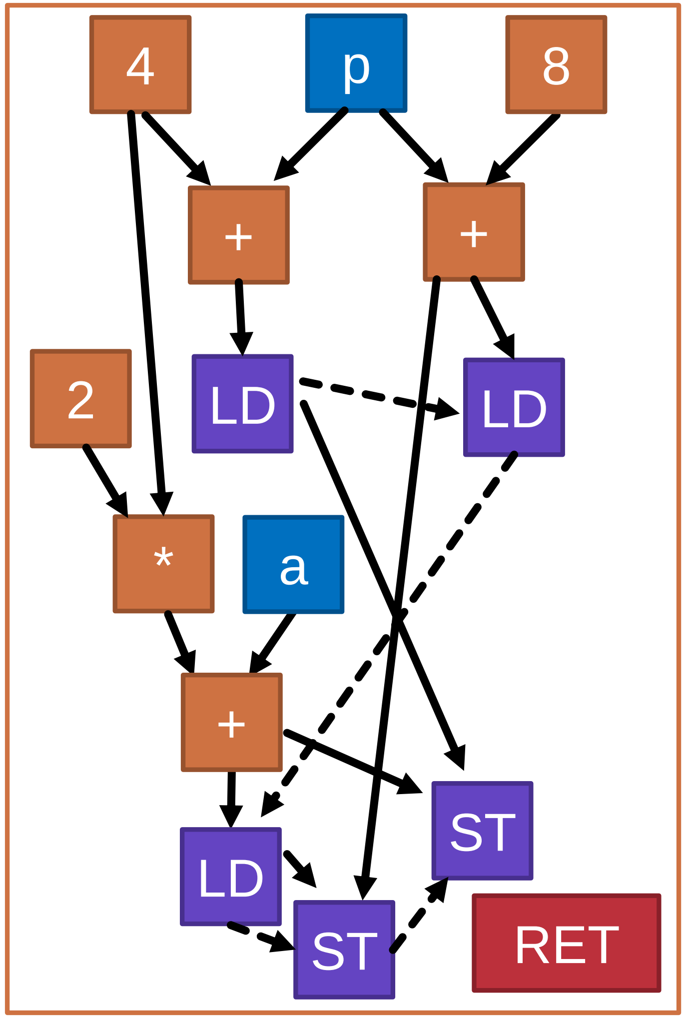
```
class P { int Q; int R; }

void foo(P p, int[] a)
{
   int q = p.Q;
   int r = p.R;
   int s = a[2];
   p.R = s;
   a[2] = q;
}
```



SSA:

```
t1: p + 4
q1: LD t1
t2: p + 8
r1: LD t2
t3: 4 * 2
t4: a + t3
s1: LD t4
ST t2: s1
ST t4: q1
RET
```

Graph:



Function call

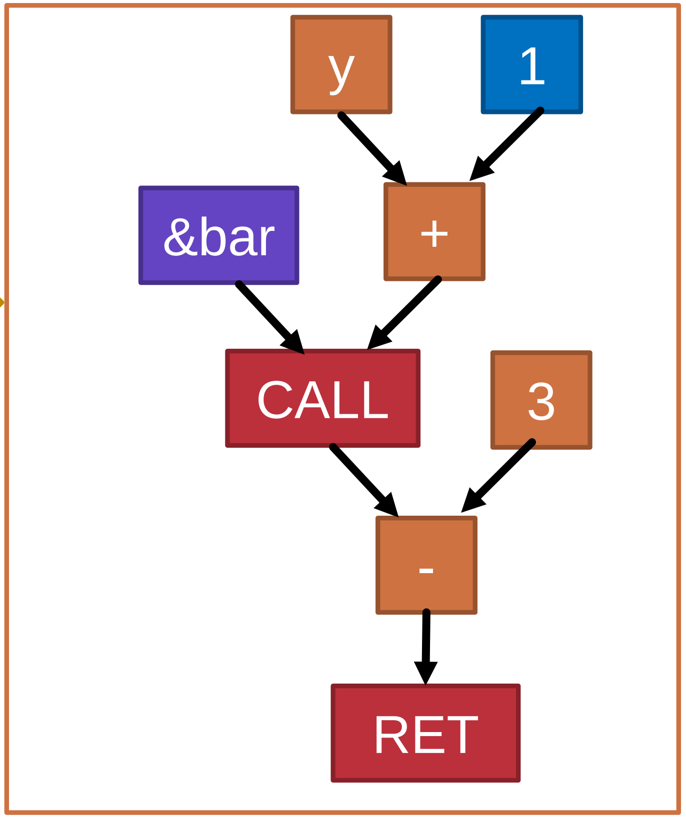
Program code:

```
bool foo(int y)
{
  return bar(y+1)-3;
}

int bar(int x)
{
  return x*2;
}
```

SSA:

```
t1: y + 1
t2: CALL bar(t1)
t3: t2 - 3
RET t3
```

Graph:



Function call

- 1. Save state (registers)
- 2. Reserve stack for the function (Stack Frame)
  - return address, return value, parameters, local variables
- 3. Write return address to stack (there are 2 return addresses if there is exception handling!)
- 4. Write arguments to the stack
- 5. Set Program Counter to the address of the function
- 6. Execute the body of the function
- 7. Reset Program Counter to the return address
- 8. Read return value from the stack
- 9. Free the reserved Stack Frame
- 10. Restore state (registers)

SSA computation

- Create basic blocks and the control flow graph
- For each block: number the values for each statement/expression
- Φ-functions can only be computed, when all preceding blocks are finished
  - until then: temporary Φ' function
- Φ-function can be inserted into a preceding block, too!
- A Φ' function may not necessarily result in a Φ-function

Rules for the Φ-function

- Φ-functions always appear at the beginning of a basic block which has multiple preceding blocks
- A Φ-function has exactly as many operands as the number of preceding basic blocks
- The value of the Φ-function is the value of the operand coming from the actual preceding block at runtime
- All Φ-functions must be evaluated simultaneously inside a single block

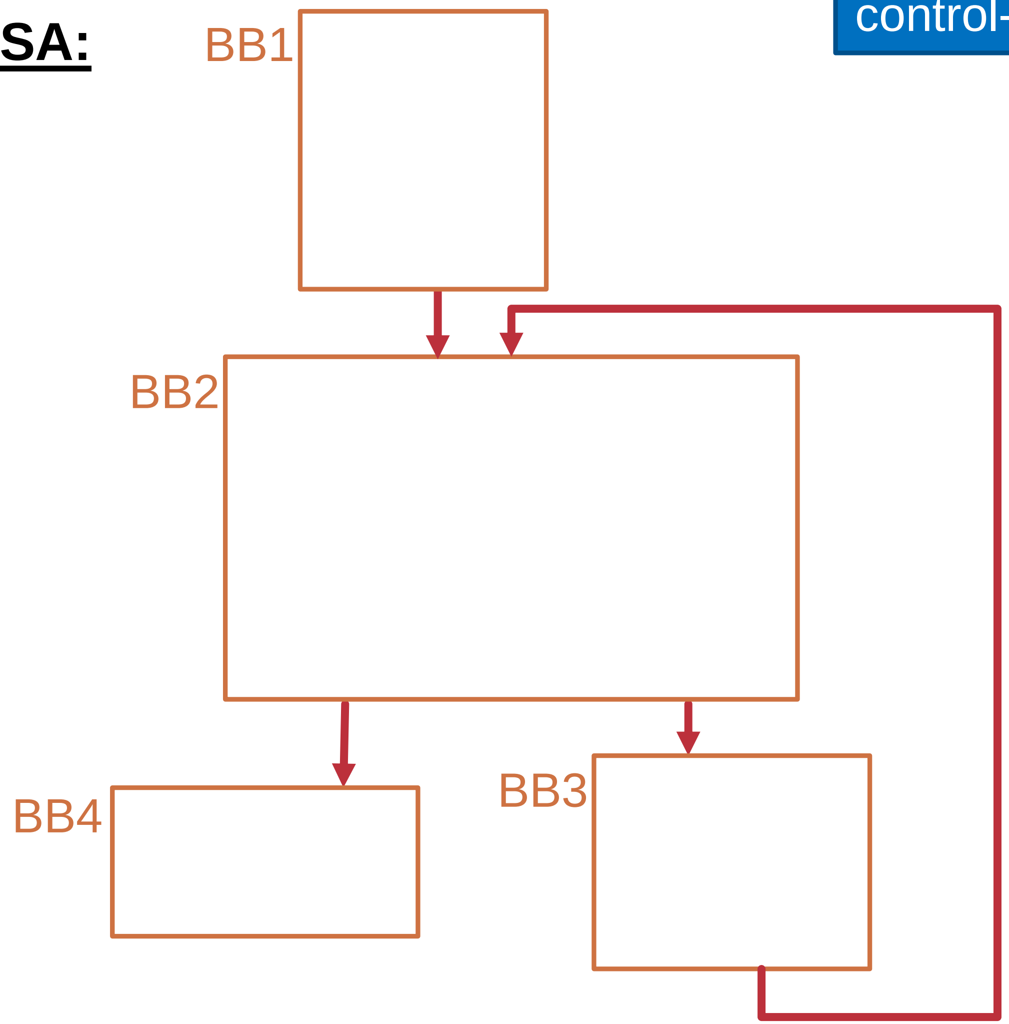
SSA computation example (1/6)

Create basic blocks and control-flow graph.

Program code:

```
int foo(int y)
{
  int x = 0;
  int z = 2;
  for (int i = 0; i < y; ++i)
  {
    x += y;
  }
  return x + z;
}
```

SSA:



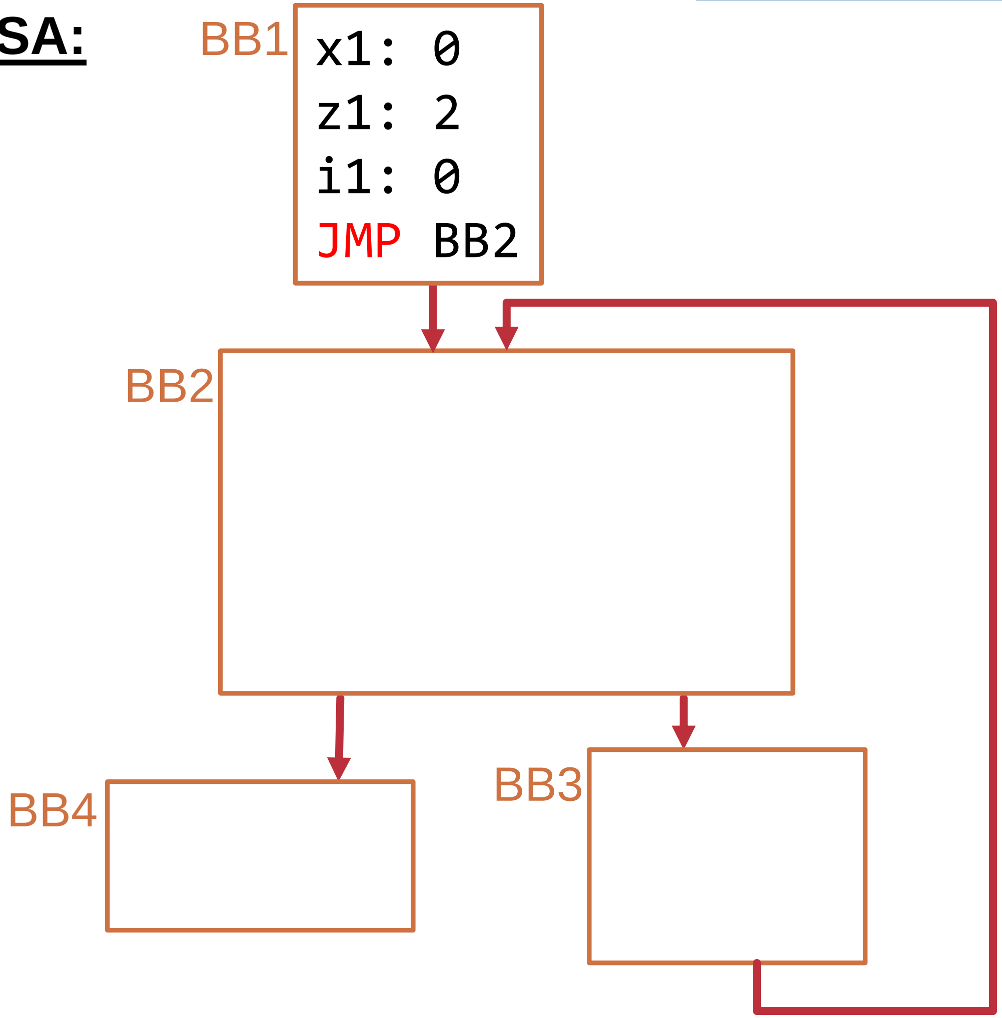
SSA computation example (2/6)

Number values in BB1.

Program code:

```
int foo(int y)
{
  int x = 0;
  int z = 2;
  for (int i = 0; i < y; ++i)
  {
    x += y;
  }
  return x + z;
}
```

SSA:



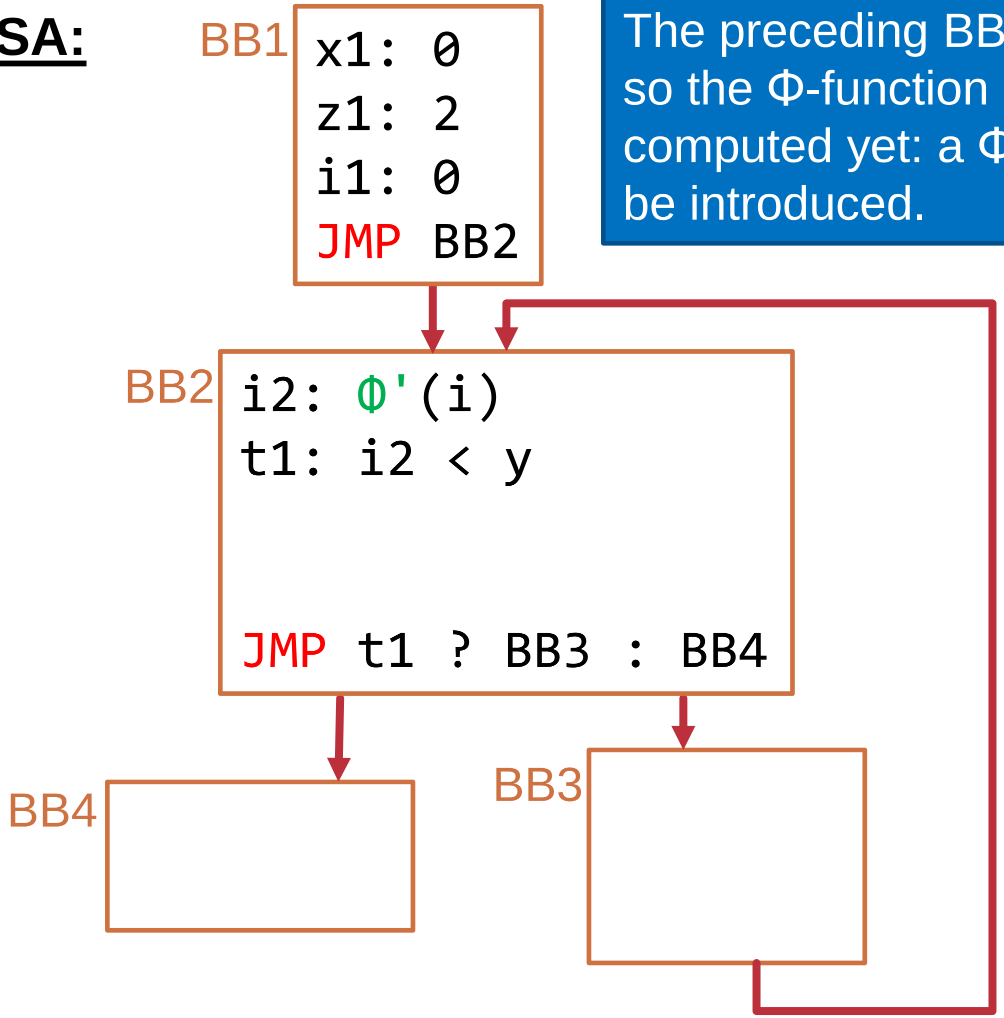
SSA computation example (3/6)

Number values in BB2.
The preceding BB3 is not yet finished, so the Φ-function of i cannot be computed yet: a Φ’(i) function must be introduced.

Program code:

```
int foo(int y)
{
  int x = 0;
  int z = 2;
  for (int i = 0; i < y; ++i)
  {
    x += y;
  }
  return x + z;
}
```

SSA:

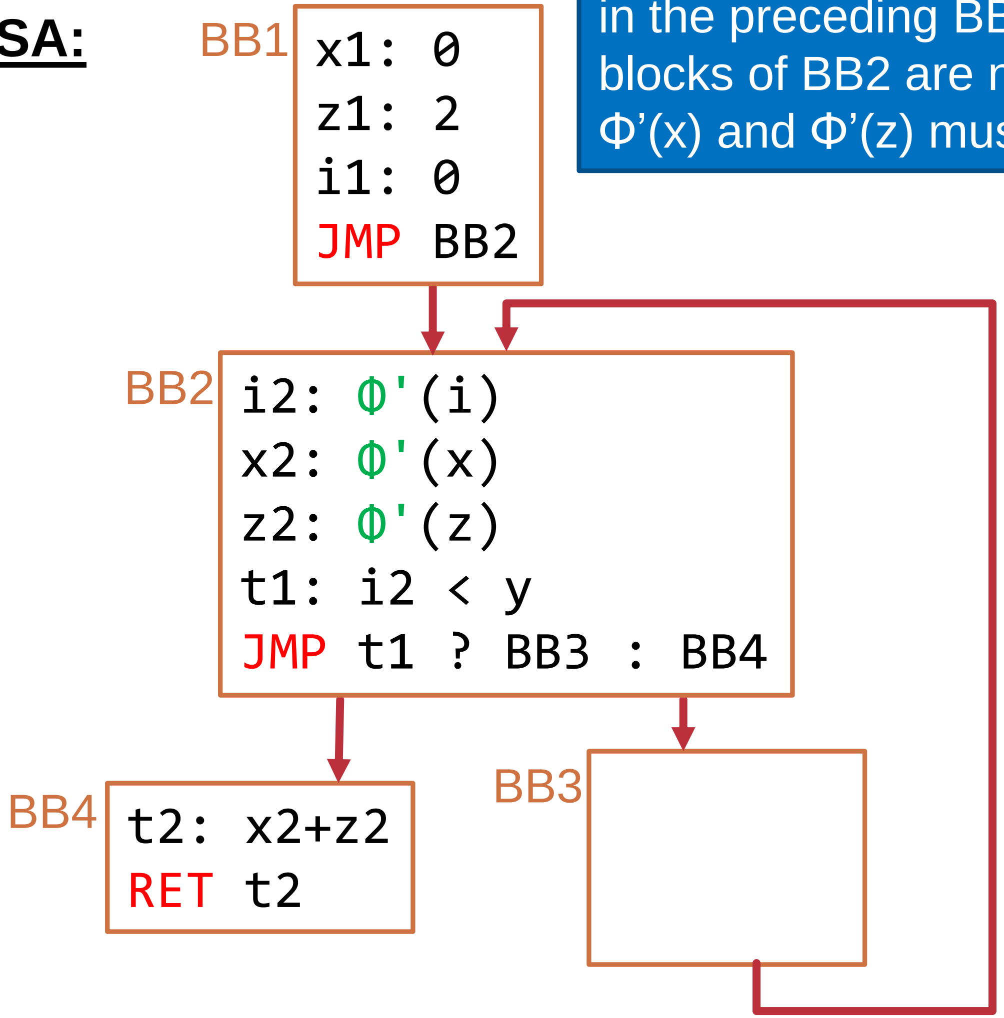


SSA computation example (4/6)

Program code:

```
int foo(int y)
{
  int x = 0;
  int z = 2;
  for (int i = 0; i < y; ++i)
  {
    x += y;
  }
  return x + z;
}
```

SSA:



BB4 uses x and z. They don’t appear in the preceding BB2. The preceding blocks of BB2 are not yet finished: Φ’(x) and Φ’(z) must be introduced.

SSA computation example (5/6)

Program code:

```
int foo(int y)
{
  int x = 0;
  int z = 2;
  for (int i = 0; i < y; ++i)
  {
    x += y;
  }
  return x + z;
}
```

SSA:

BB1
```
x1: 0
z1: 2
i1: 0
JMP BB2
```

BB2
```
i2: Φ'(i)
x2: Φ'(x)
z2: Φ'(z)
t1: i2 < y
JMP t1 ? BB3 : BB4
```

BB3
```
x3: x2+y
i3: i2+1
JMP BB2
```

BB4
```
t2: x2+z2
RET t2
```

BB3 can be computed easily, since both x and i appear in BB2.

SSA computation example (6/6)

Program code:

```
int foo(int y)
{
  int x = 0;
  int z = 2;
  for (int i = 0; i < y; ++i)
  {
    x += y;
  }
  return x + z;
}
```

SSA:

BB1
```
x1: 0
z1: 2
i1: 0
JMP BB2
```

BB2
```
i2: Φ(i1,i3)
x2: Φ(x1,x3)
z2: z1
t1: i2 < y
JMP t1 ? BB3 : BB4
```

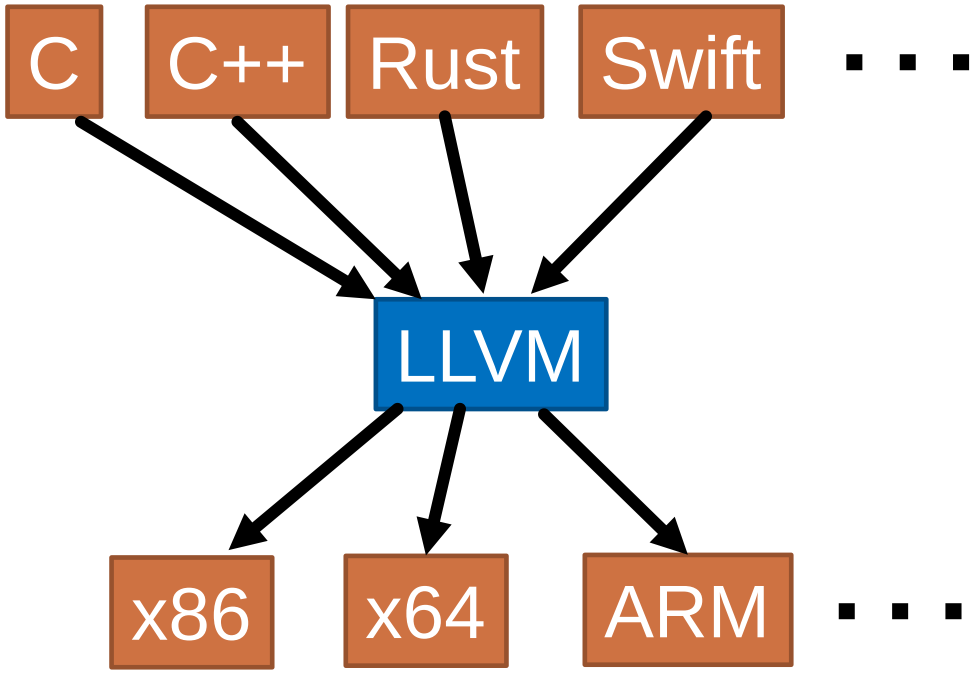
BB3
```
x3: x2+y
i3: i2+1
JMP BB2
```

BB4
```
t2: x2+z2
RET t2
```

Transformation of Φ' in BB2:
x and i have different numbers in the preceding blocks, the Φ-function is necessary. The numbers of z are the same in both preceding blocks, hence Φ' will not lead to a Φ-function.

LLVM

- Compiler tool
  - front end (transformation into SSA form)
  - optimization
  - back end (machine code generation)
- LLVM Intermediate Representation (IR) independent of programming languages
  - type system, statements and expressions
- We can use it for our own compiler!
- Alternative for LLVM:
  - we can generate C, Java, C#, etc. code, and compile it with the regular compiler



This lecture: Transformation, Optimization

- I. Transformation
- II. Type mapping
- III. Statement mapping: SSA
- **IV. Optimization**
- V. Optimization techniques

Optimization

- Goal: more efficient execution without changing the observable behavior
  - smaller code, faster execution, less memory, lower consumption, etc.
- There is no optimal program, only optimized!
  - otherwise, we would have and algorithm for the undecidable halting problem
- Some examples for optimization techniques:
  - Compile time evaluation of constant expressions, Dead code elimination, Operator simplification, Moving loop-invariant code, Eliminating partial redundancies, Moving code, Elimination of index checking, Inlining functions, Simplification of execution branches, Loop unrolling/splitting, Register assignment, Replace right recursion with loop

Optimization order

- Question: Which optimizations to apply? In what order?
  - no exact answer
- Some optimizations are similar to each other, or subsets of each other
- For numerical programs:
  - operator simplification: usually results in at least 2x faster code
  - cache optimization: usually results in at least 2-5x faster code
- Other optimizations:
  - first optimization usually results in 15% faster code
  - each further optimization brings less than 5% improvement
- Source: http://www.info.uni-karlsruhe.de/lehre/2007WS/uebau1/folien/10-SSA_v2.pdf

Levels of optimizations

- Local
  - within a single basic block, isolated from the others
- Global (intraprocedural)
  - on a series of basic blocks, withing a control-flow graph
- Interprocedural
  - analyses all of a program’s code
  - takes into account the context of the function calls
  - rarely supported by compilers

Optimization using SSA

- Many optimizations are simpler in SSA form
- Two main transformations: normalization and optimization
- Normalization: making different expressions comparable with each other
  - makes optimization easier
  - uses algebraic identities: commutativity, associativity, distributivity
  - can be limited by exceptions and prescribed evaluation order (e.g., Java)
- Optimization: making program execution more efficient

Dataflow analysis using SSA

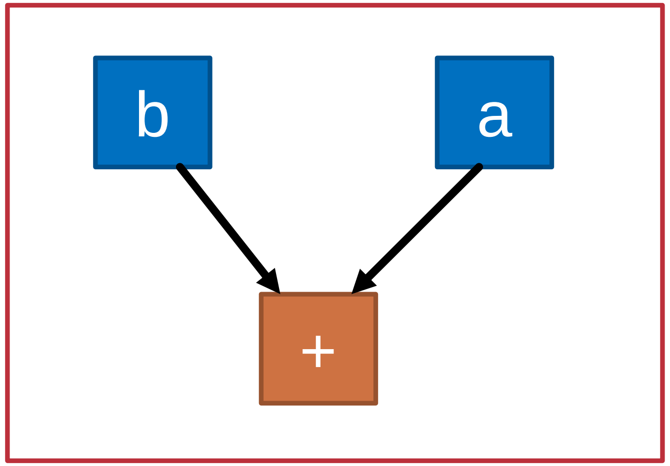
- Some optimizations may change the execution order of statements
- Dataflow analysis: finding data dependencies
  - definition-use of variables, dependencies of writes and reads
  - important in the code generation phase for the correct ordering of operations
- Dataflow analysis can also be performed without SSA, however, usually it is easier with SSA
- Useful also for semantic analysis
  - recognizing uninitialized variables
  - reporting unused variables
  - finding live (potentially used later) variables
  - recognizing that a function does not return on all branches
  - etc.

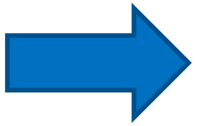
This lecture: Transformation, Optimization

- I. Transformation
- II. Type mapping
- III. Statement mapping: SSA
- IV. Optimization
- V. Optimization techniques

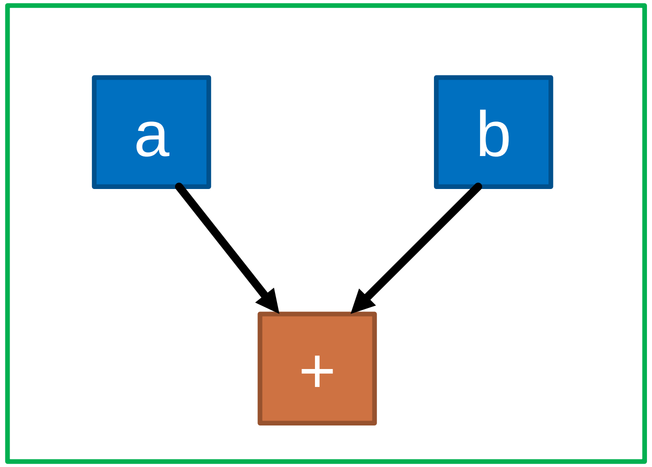
Normalization: using commutativity

`t1: b+a`





`t1: a+b`



Common subexpressions can be recognized more easily

Optimization: elimination of common subexpressions

```
x = b+a;
y = (a+b)*2;
```

SSA

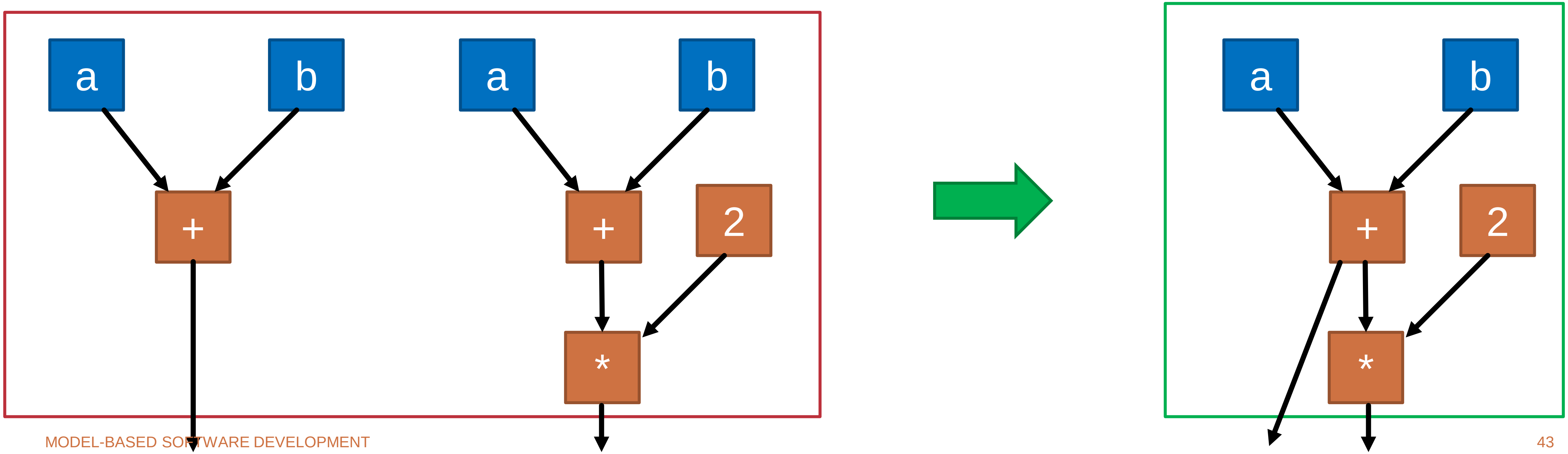
```
x1: b+a
t2: a+b
y1: t2*2
```

Norm.

```
x1: a+b
t2: a+b
y1: t2*2
```

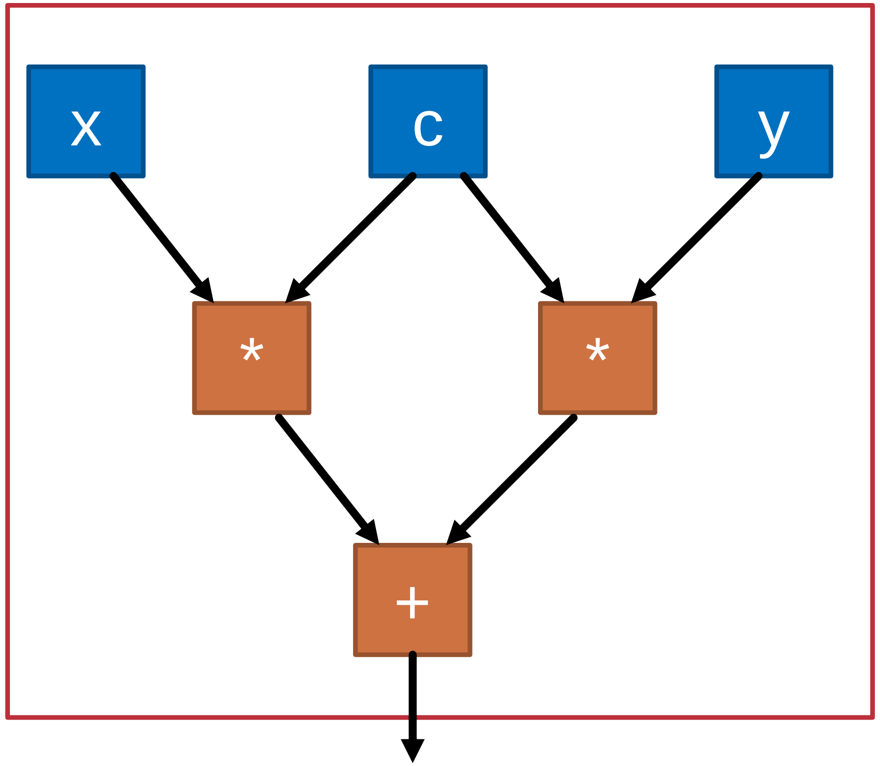
Opt.

```
x1: a+b

y1: x1*2
```

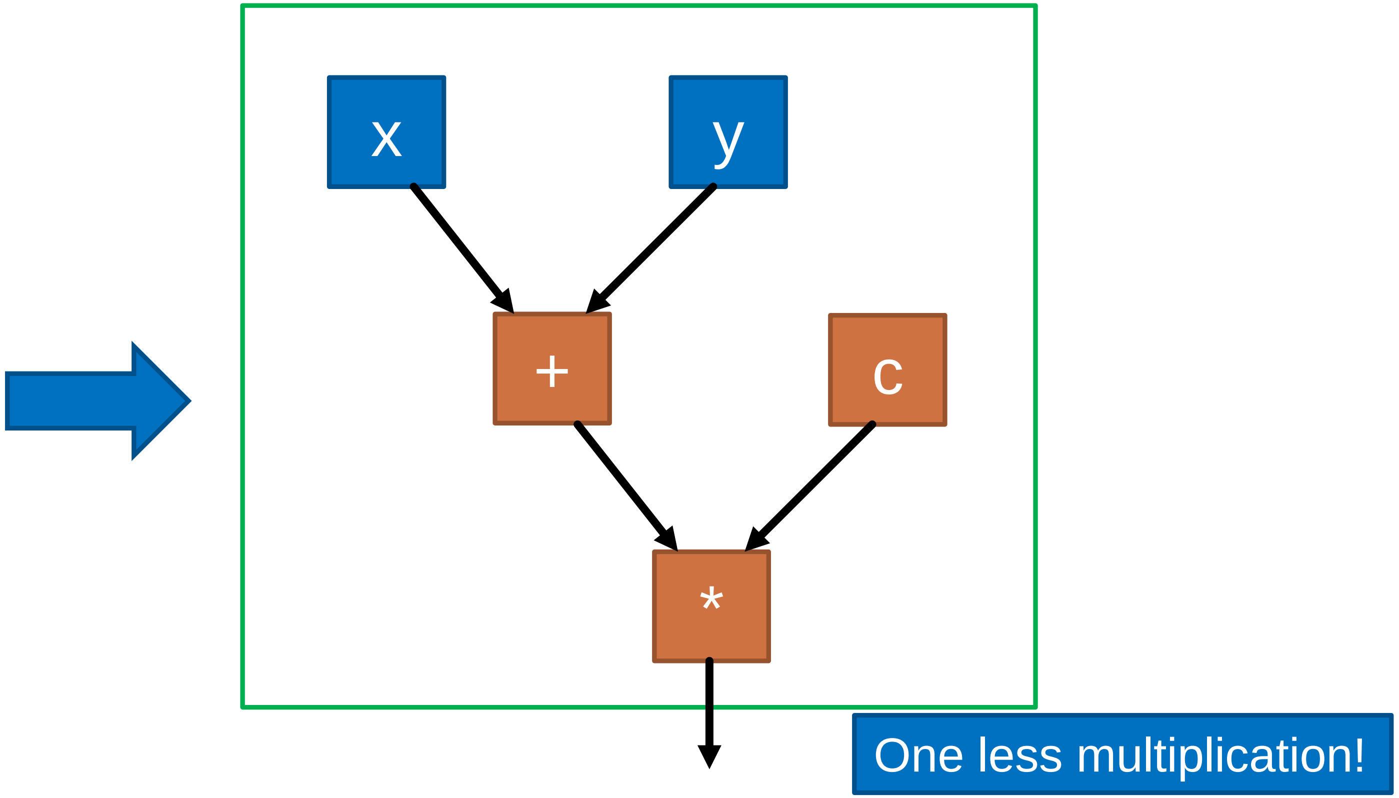


Normalization: using distributivity

```
t1: x*c
t2: y*c
t3: t1+t2
```




```
t1: x+y
t2: t1*c
```



Optimization: evaluation of constant expressions

```
x = 2+4;
y = x*3;
```

SSA →

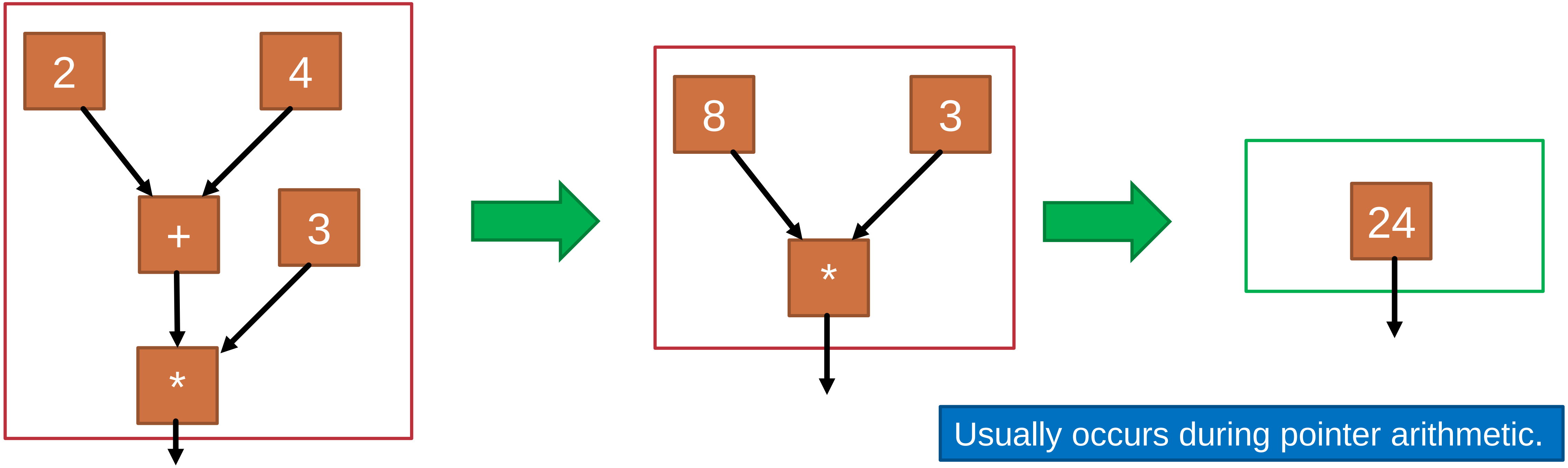
```
x1: 2+4
y1: x1*3
```

Opt. →

```
x1: 8
y1: x1*3
```

Opt. →

```
y1: 24
```



Usually occurs during pointer arithmetic.

Optimization: removing inverse operator

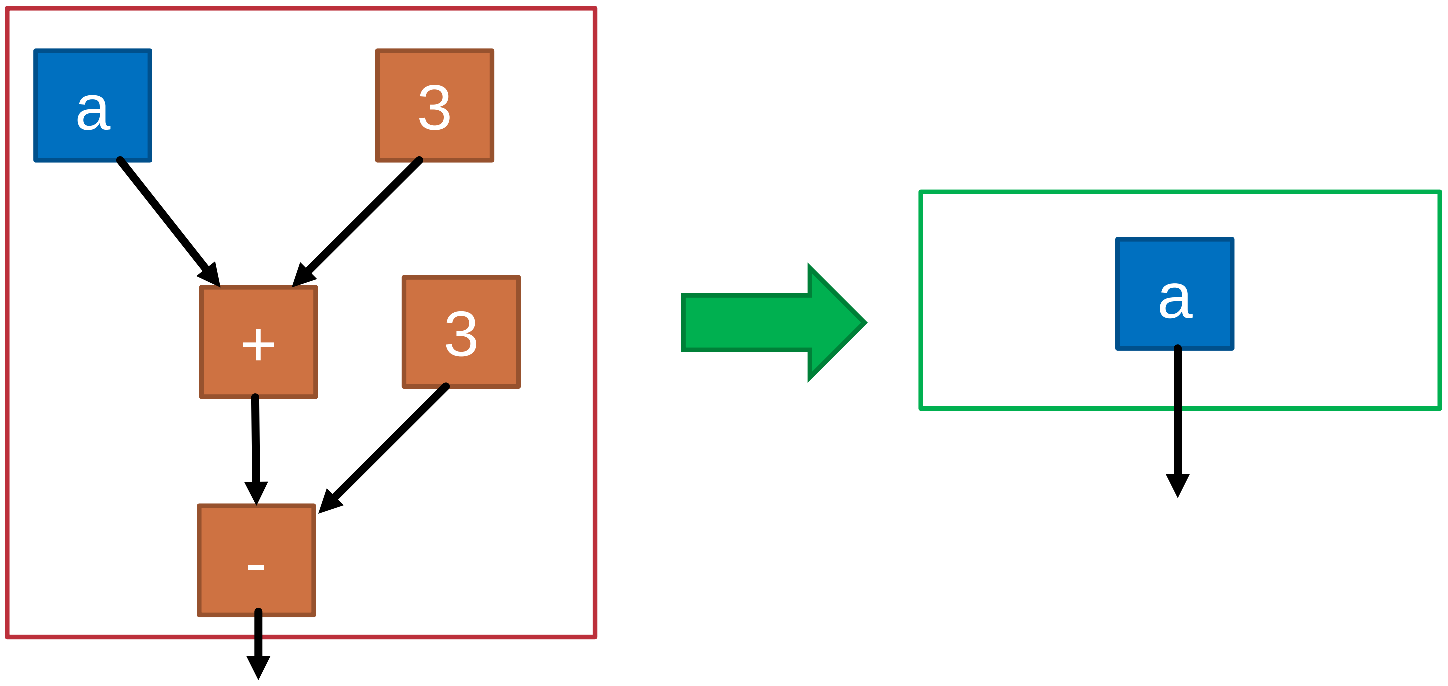
```
x = a+3;
y = x-3;
```

SSA

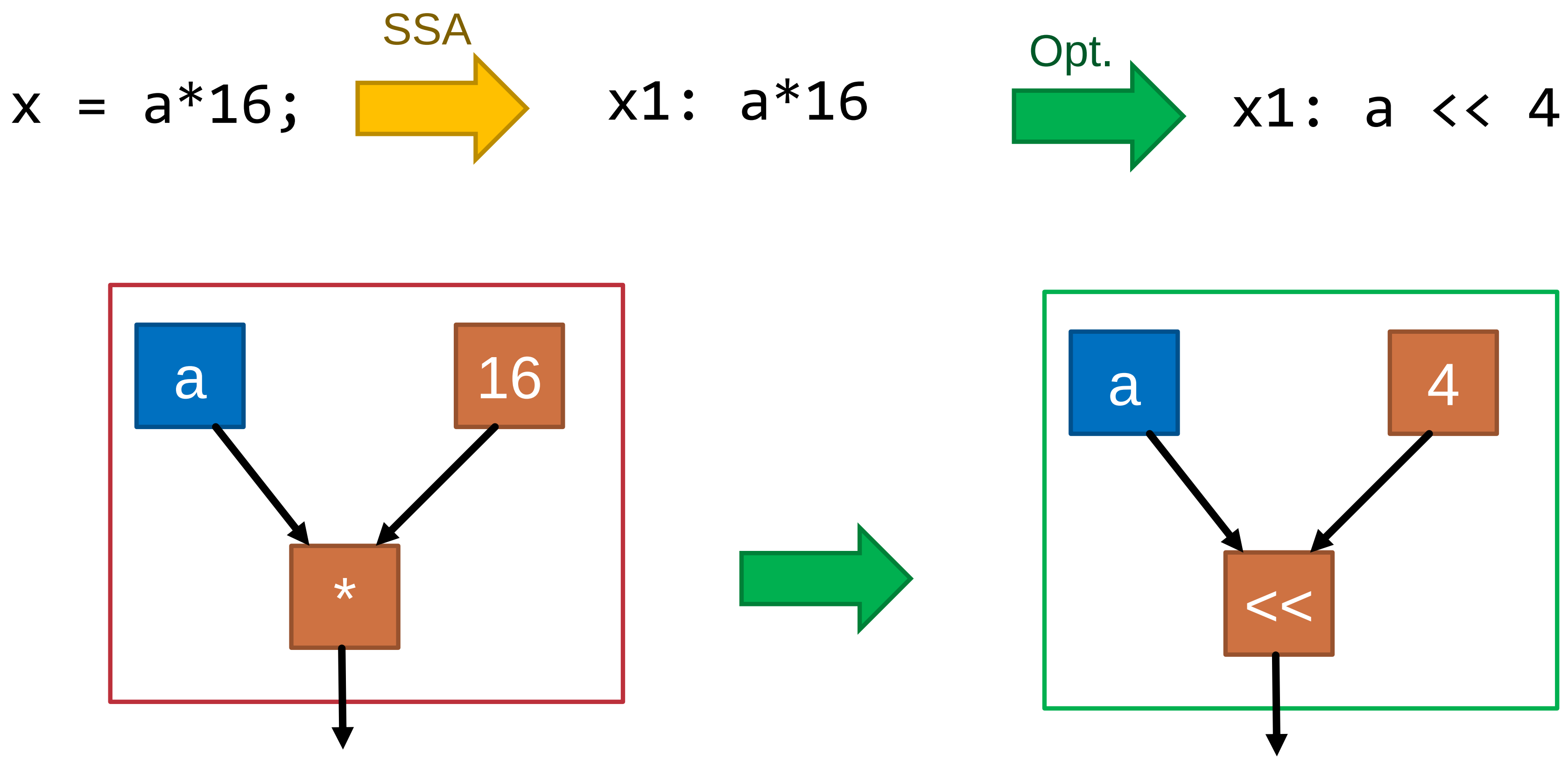
```
x1: a+3
y1: x1-3
```

Opt.

```
y1: a
```



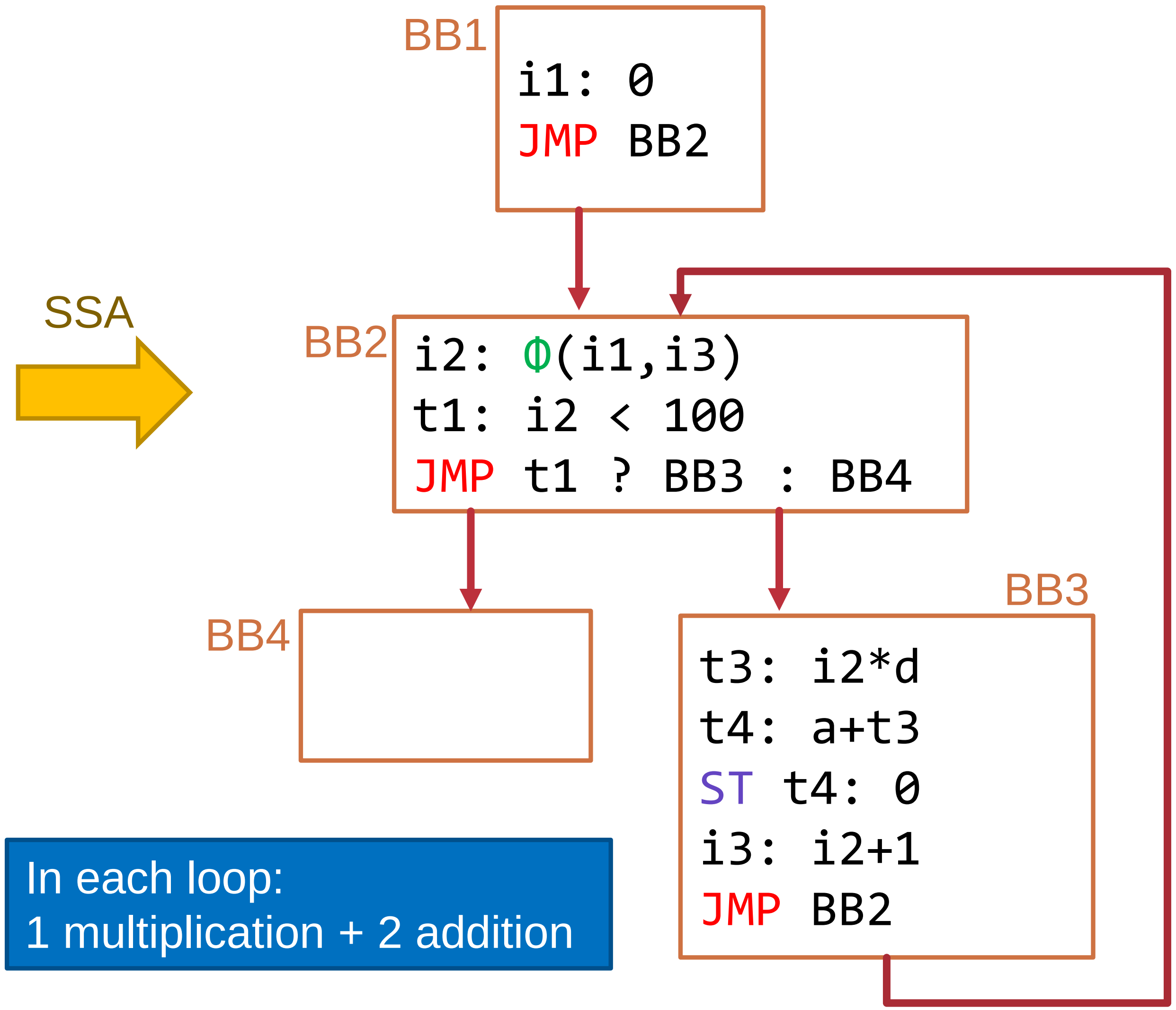
Optimization: operator simplification



shift operator instead of multiplication with a power of 2

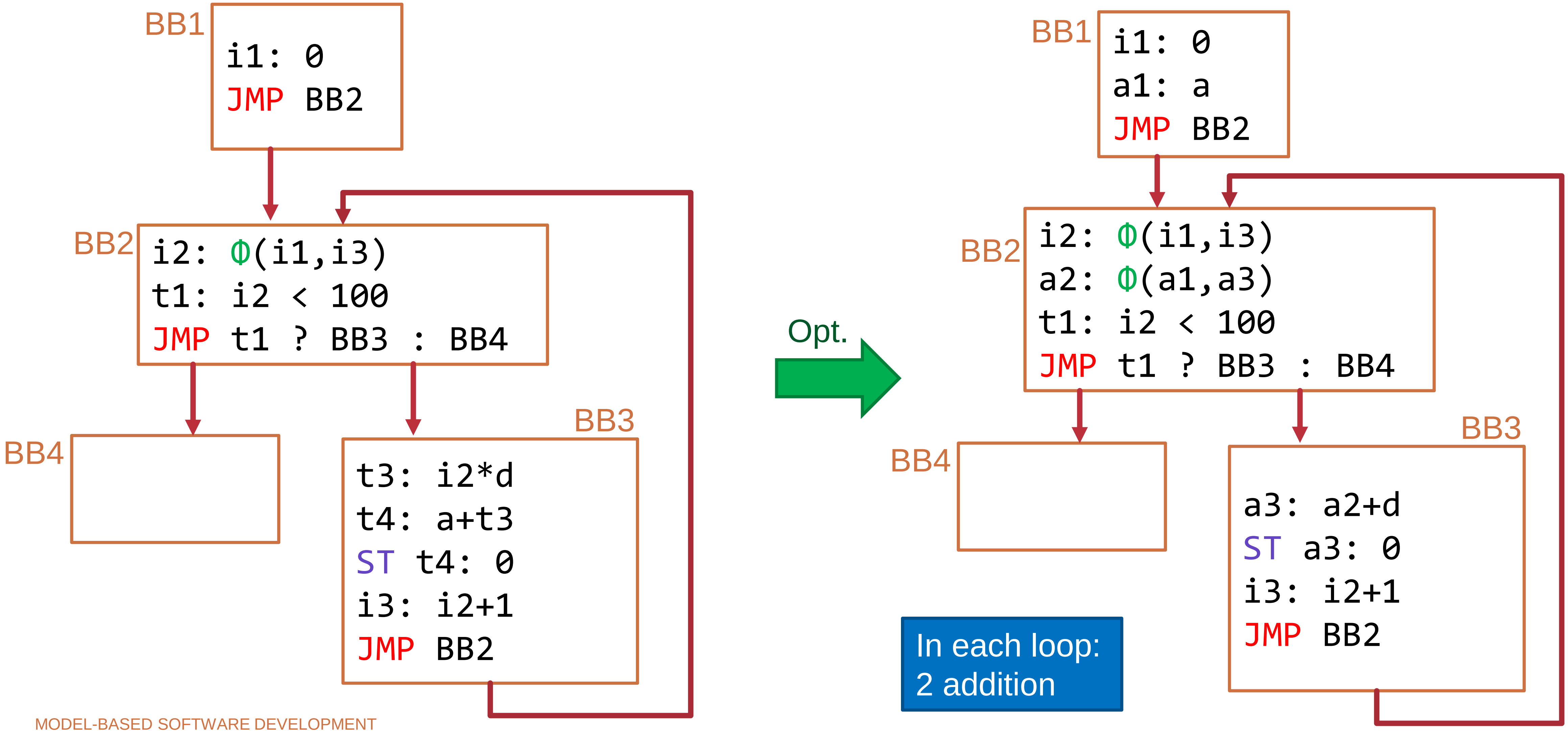
Optimization: simplifying loops (1/3)

```
for (int i = 0; i < 100; ++i)
{
    a[i] = 0;
}
```

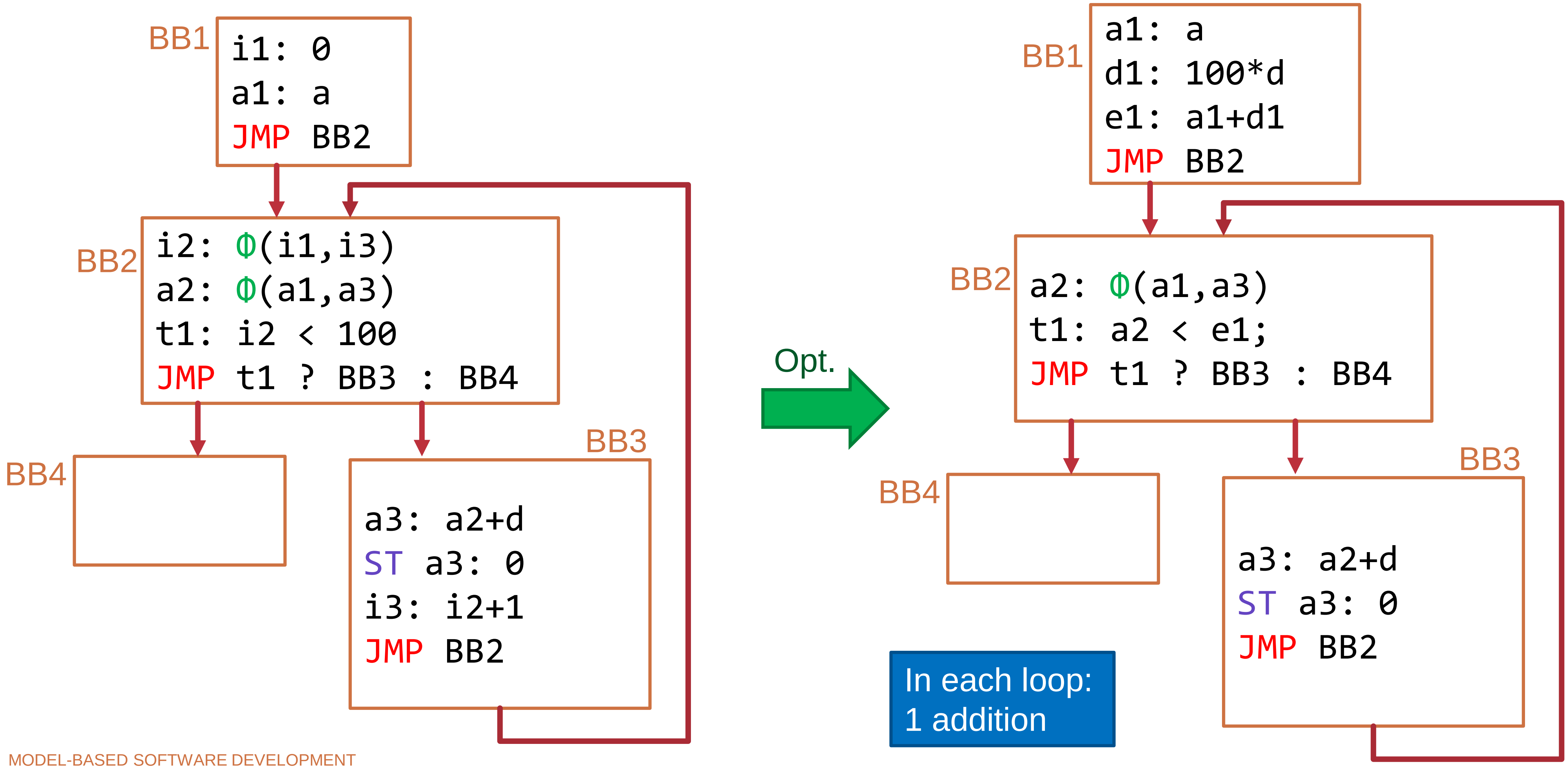


In each loop:
1 multiplication + 2 addition

Optimization: simplifying loops (2/3)



Optimization: simplifying loops (3/3)



Optimization: store-load

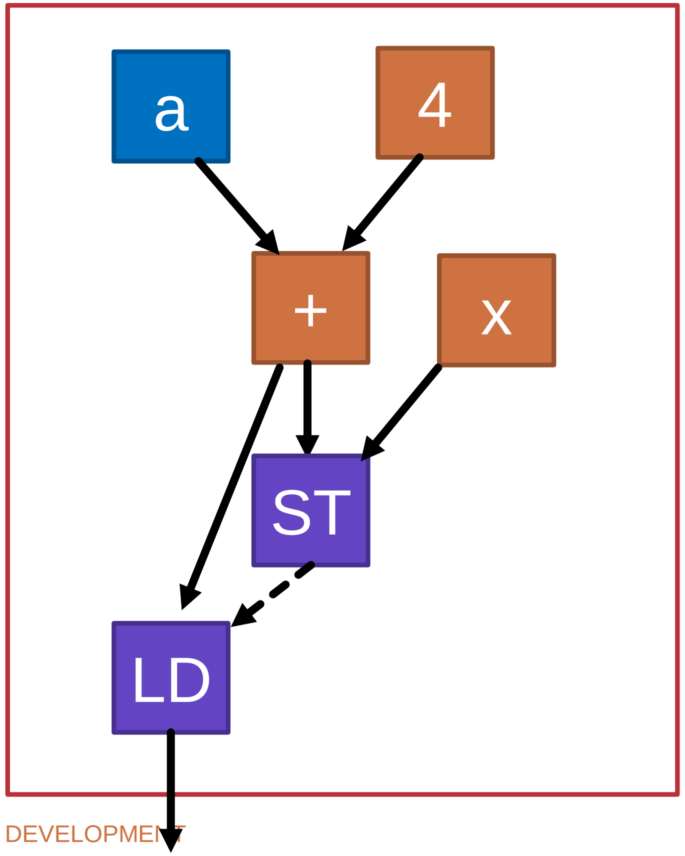
```
a[1] = x;
y = a[1];
```

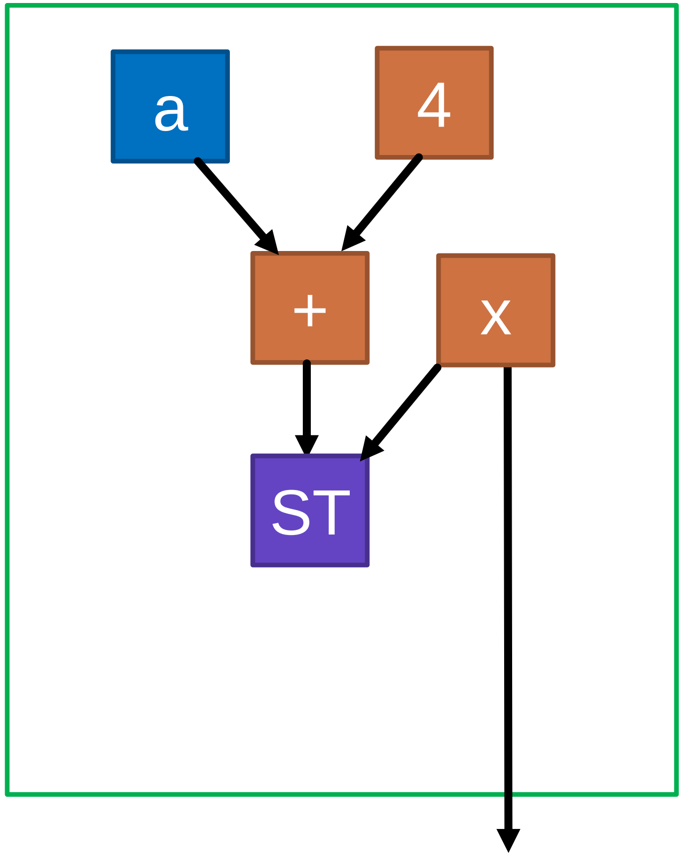
SSA

```
t1: a+4
ST t1: x
y1: LD t1
```

Opt.

```
t1: a+4
ST t1: x
y1: x
```





Optimization: load-load

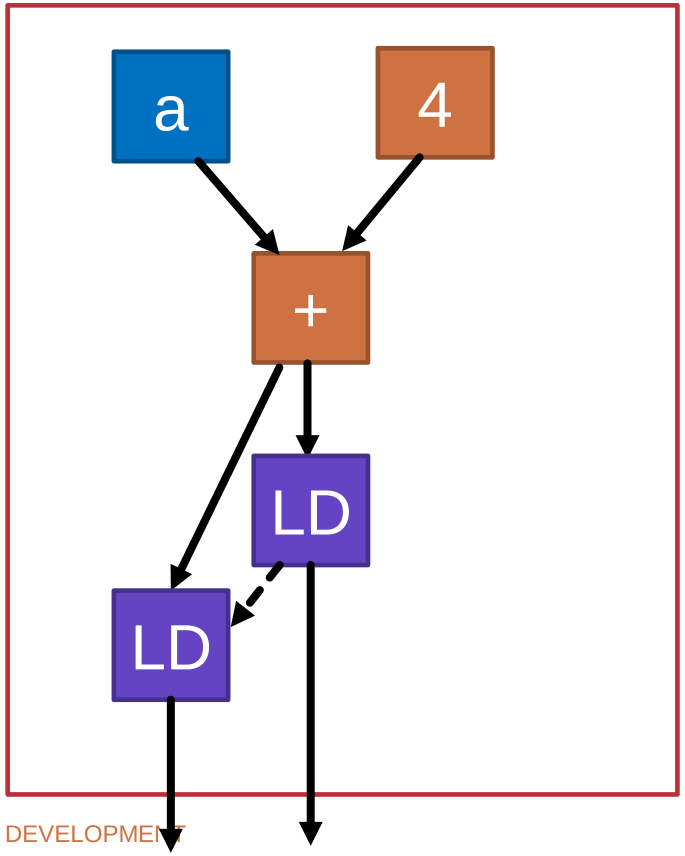
```
x = a[1];
y = a[1];
```

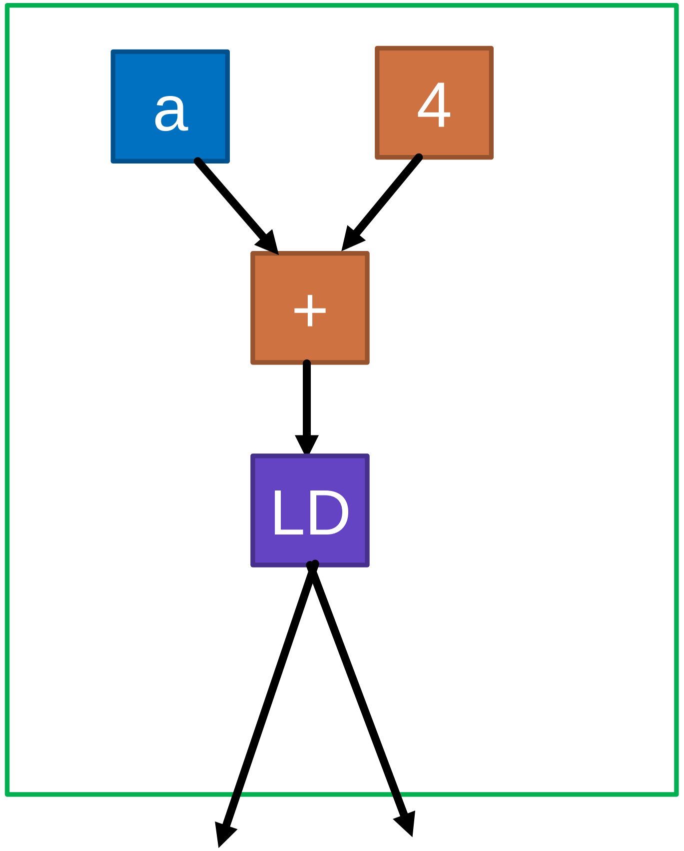
SSA

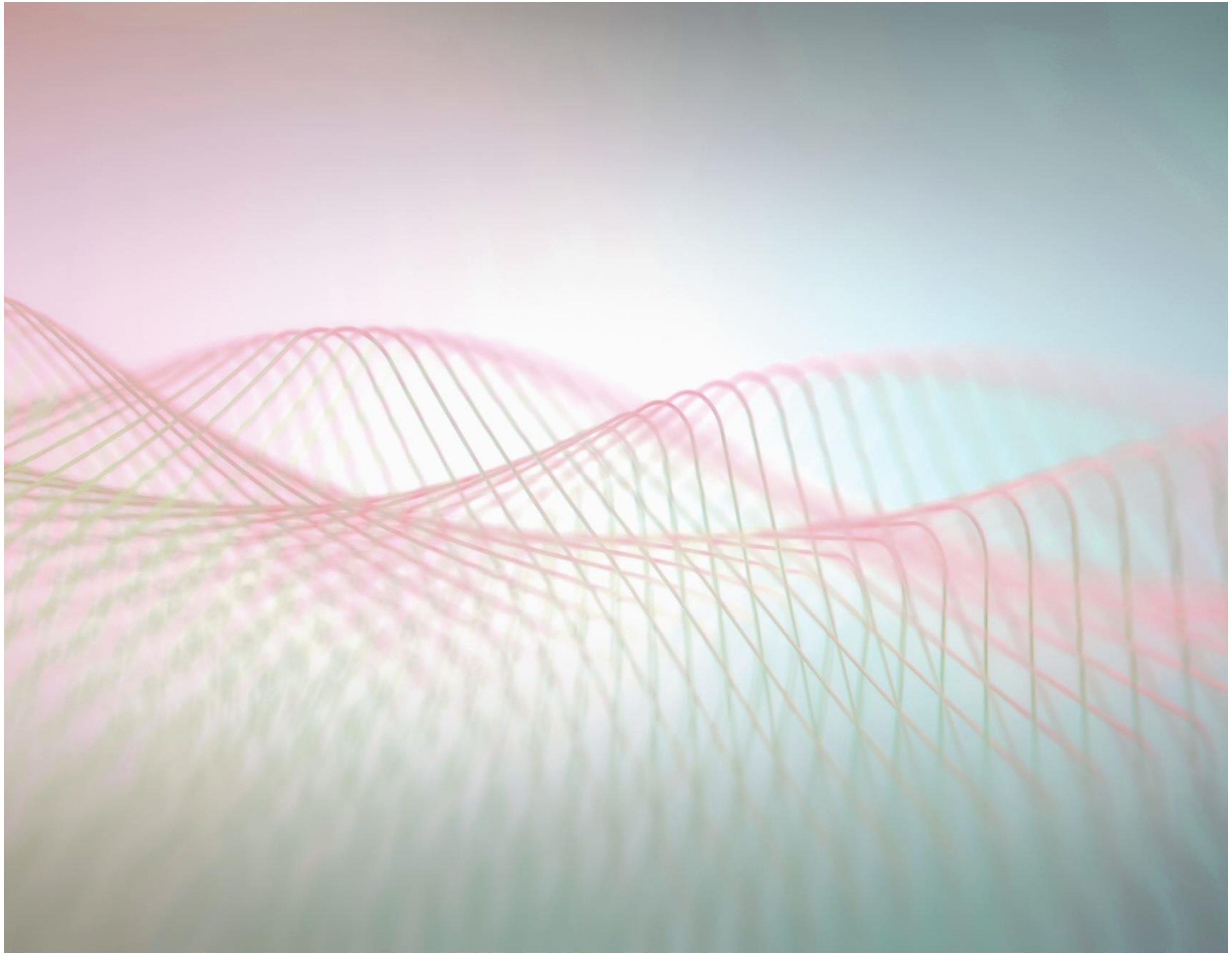
```
t1: a+4
x1: LD t1
y1: LD t1
```

Opt.

```
t1: a+4
x1: LD t1
y1: x1
```







Practice 3

Coming up next…

- Topic: IDE support for textual DSLs
- Eclipse environment – Java language
- Xtext – describing and processing DSLs, IDE support
- Xcore – meta-modeling (AST description at the same time)
- Xtend – template-based code generation
- IDE functions:
  - syntax highlighting, validation, error markers, content assist, hyperlinking, outline, automatic code formatting, automatic build / code generation

Thank you!