Course 142A Compilers & Interpreters

Code Generation

Lecture Week 4

Prof. Dr. Luc Bläser

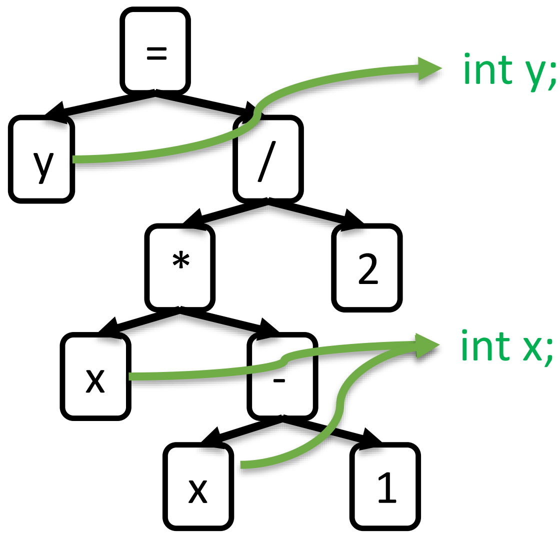
Last Lecture - Quiz

```
y = x * (x - 1) / 2;
```

Which semantic checks are to be applied?
How does the intermediate representation look like?

Intermediate Representation

- Abstract Syntax Tree
  - Designators x and y resolved
  - Type int resolved for all expressions
- Symbol Table
  - Class, Method
  - Variables x and y
  - Of inbuilt type int



Next Step

- Intermediate language code for the runtime system

```
load 1        // load x
load 1        // load x
ldc 1         // const 1
isub          // x - 1
imul          // x * (x - 1)
ldc 2         // const 2
idiv          // x * (x - 1) / 2
store 1       // store y
```

UCI-Java Bytecode

Analogous to real Java bytecode

Today's Topics

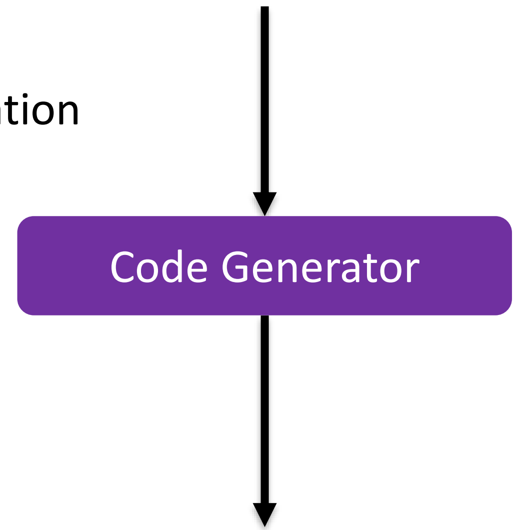
- Intermediate Language Code
- Virtual Stack Machine
- Instruction Selection
- Template-Based Approach

Learning Goals

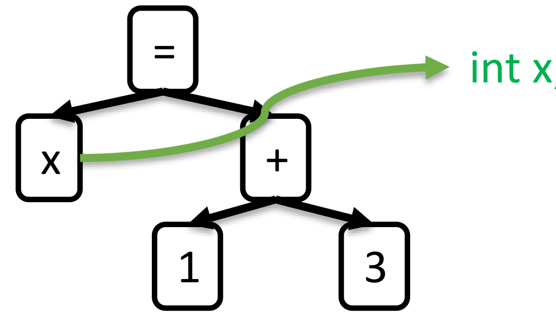
- Know the properties of modern intermediate language, such as Java bytecode
- Understand template-based code generation for a stack-based intermediate language

Compiler Backend

Intermediate Representation



(Virtual) Machine Code




```
ldc 1
ldc 3
iadd
store 1
```

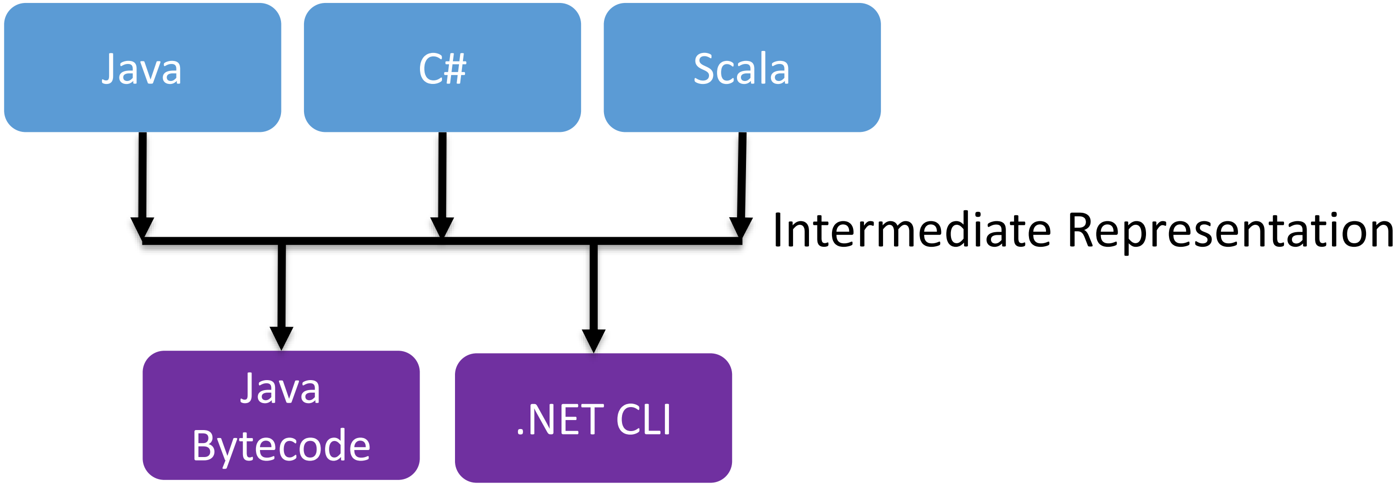
Task of a Code Generator

- Generate executable machine code
  - Input: Intermediate representation (symbol table + AST)
  - Output: Machine code
- Possible target machines
  - Real machine, e.g. Intel 64, ARM processor
  - Virtual machine, e.g. Java VM, .NET CLI

Our focus today
Code generator for VM

Compiler Frontend/Backend

- Separation allows multi-languages + multi-platforms



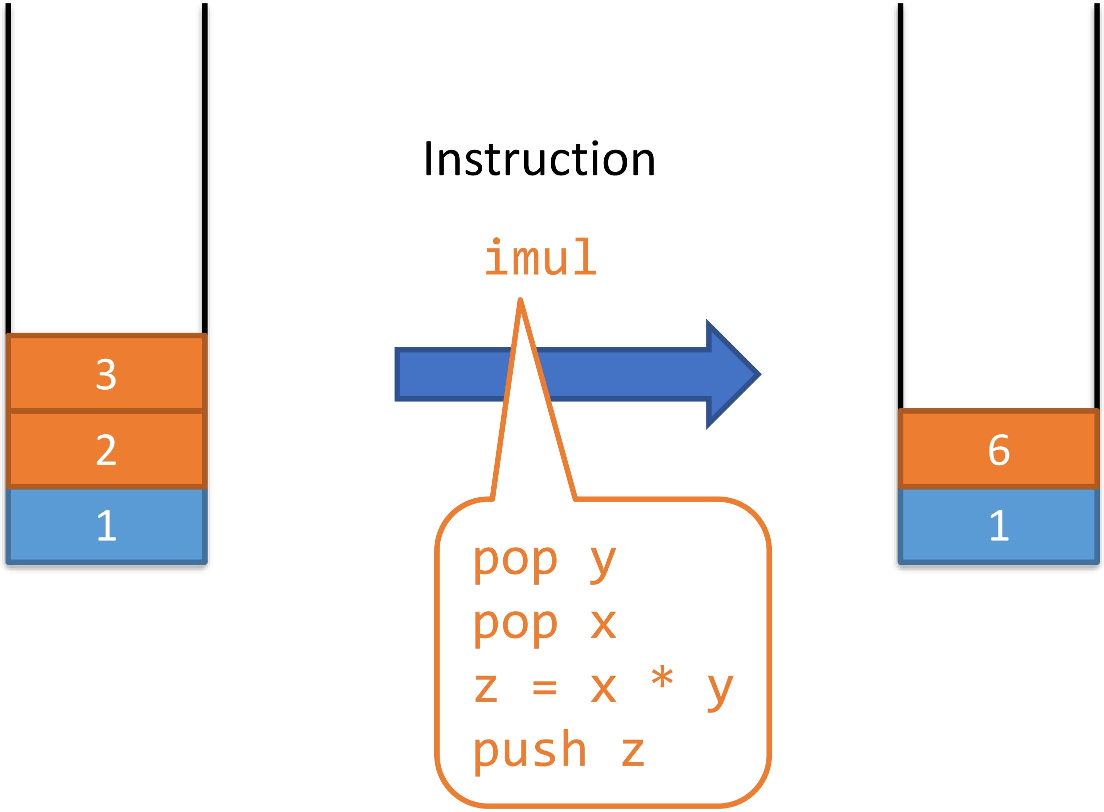
Our Target Machine

- Own VM with own intermediate language
- Oriented on Java Bytecode
  - .NET intermediate language is also similar

- Core concepts
  - Virtual stack processor
  - Branch instructions (“goto”)
  - Metadata

Stack Processor

- Instructions use an evaluation stack
  – No registers in contrast to real processors



Evaluation Stack

- Each instruction has a defined number of
  - Pop calls
  - Push calls
- Own stack per method call
  - Empty at the method entry and exit
- Stack is conceptually unbound
  - Supports arbitrarily complex evaluations

Instruction Set (Extract)

Instruction	Meaning	Evaluation Stack
`ldc <const>`	Load constant (int, boolean, string)	1 push
`iadd`	Integer add	2 pop, 1 push
`isub`	Subtract	2 pop, 1 push
`imul`	Multiply	2 pop, 1 push
`idiv`	Divide	2 pop, 1 push
`irem`	Remainder	2 pop, 1 push
`ineg`	Integer negate	1 pop, 1 push
`load <num>`	Load parameter or local var (numbered)	1 push
`store <num>`	Store parameter or local var (numbered)	1 pop

Load/Store Numbering

- «this» reference: index 0 (virtual method)
- Then, n parameters: index 1..n
- Then, m locals: index n+1..n+m

load/store 1 load/store 2 load/store 3

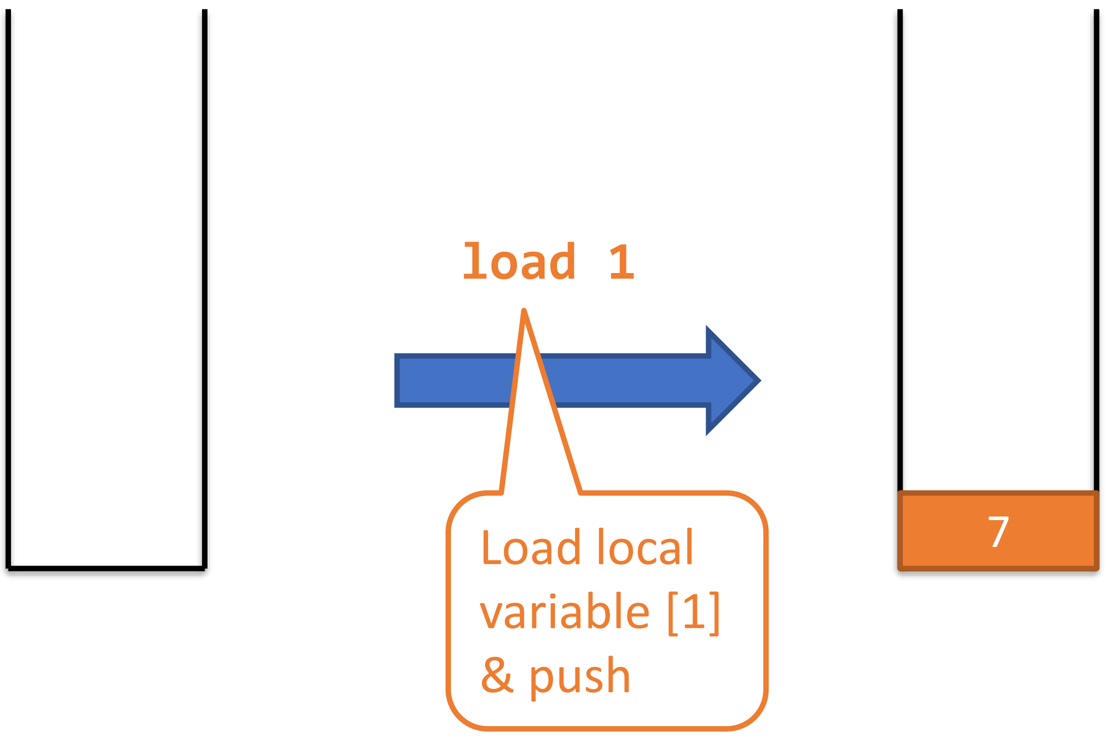
```
int calculate(int a, int b, int c) {
   int d;
   int e;
   …
}
```

load/store 4

load/store 5

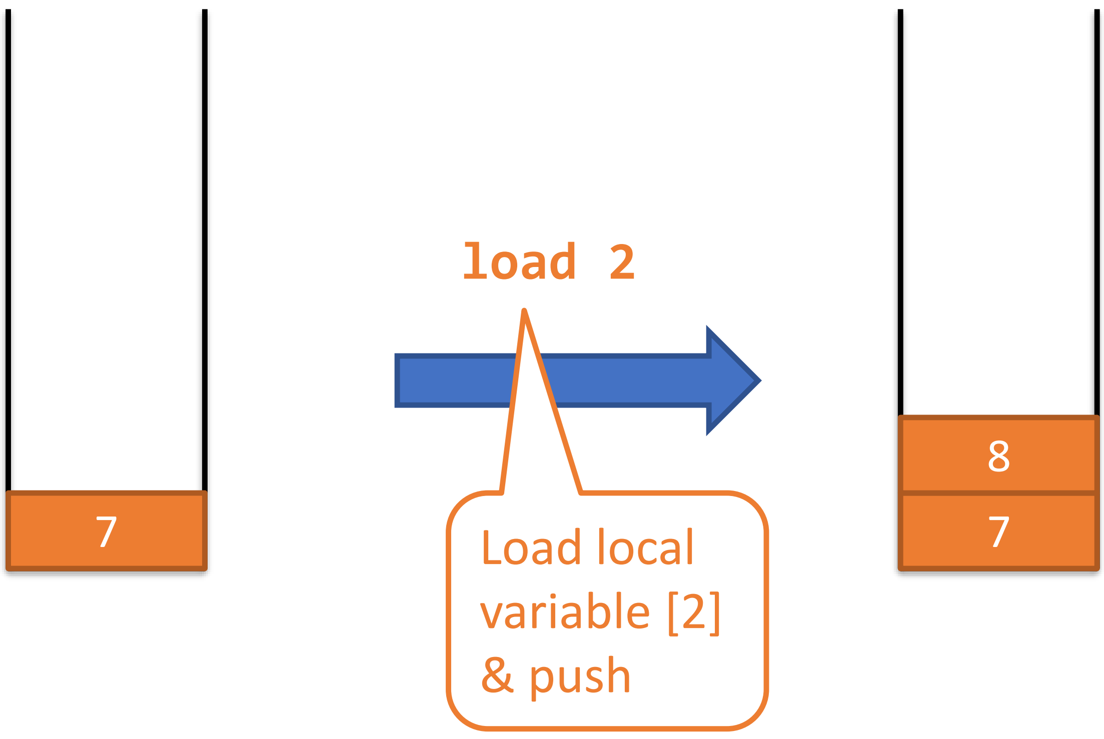
Evaluation Example

Local variables
  [1] int value 7
  [2] int value 8



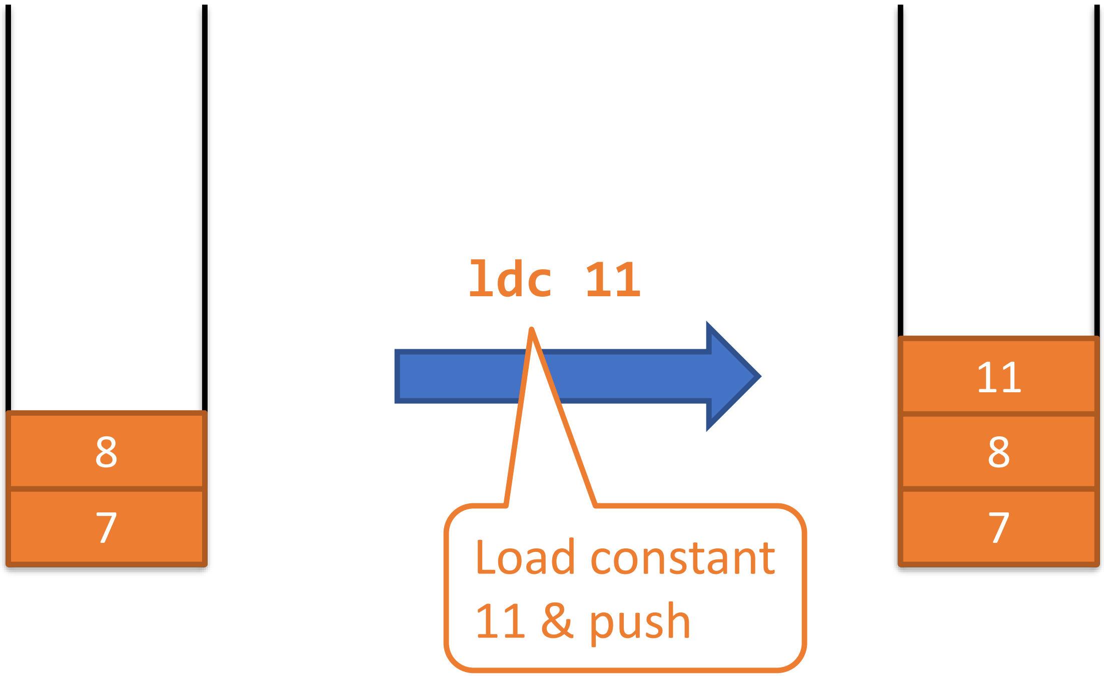
Evaluation Example

Local variables
  [1] int value 7
  [2] int value 8



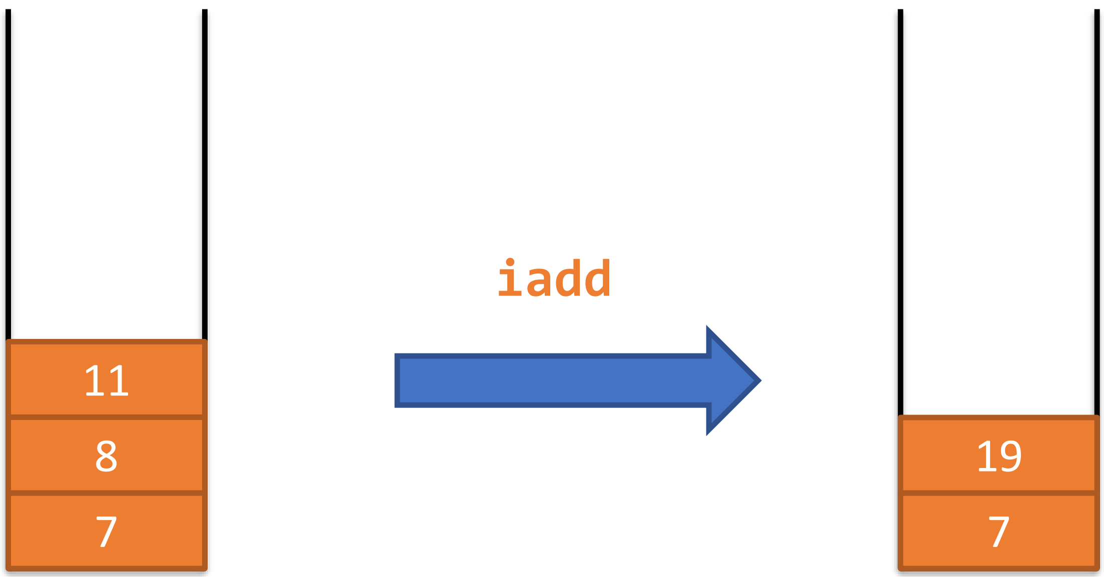
Evaluation Example

Local variables
[1] int value 7
[2] int value 8



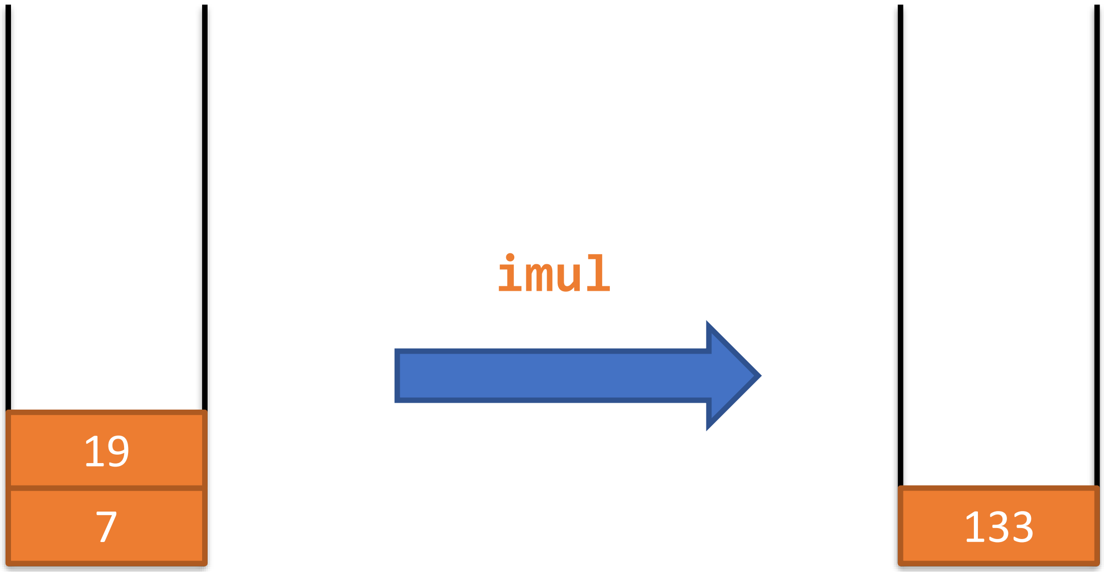
Evaluation Example

Local variables
[1] int value 7
[2] int value 8



Evaluation Example

Local variables
  [1] int value 7
  [2] int value 8

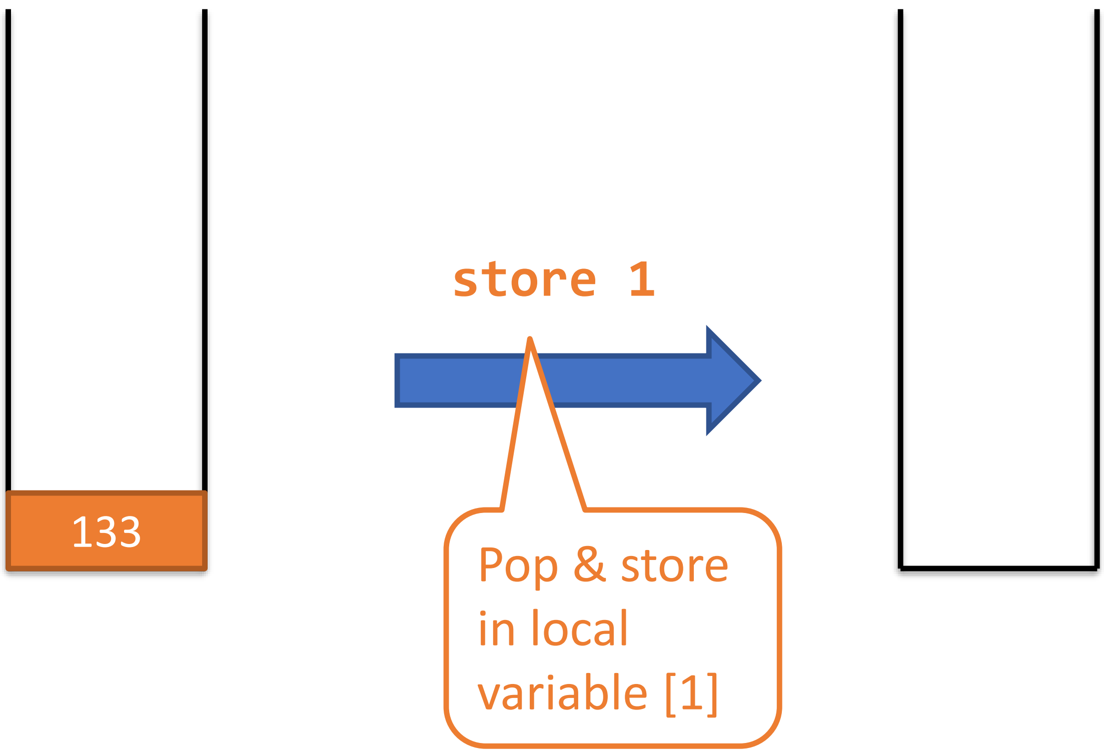


Evaluation Example

Local variables

[1] int value 133

[2] int value 8



Boolean Values

- UCI Java Bytecode: Stored as boolean on stack
- Real Java Bytecode: Stored as int (0 = false)

Instruction	Meaning	Evaluation Stack
bneg	Boolean negate (not)	1 Pop, 1 Push

Why do we have no “and” / “or” instructions?

Example: Boolean

Local variables
 [1] boolean value
 [2] boolean value

```
load 1
bneg
store 2
```

Which statement corresponds to this code?

Control Flow

- Control statements translated to branch-based code

```
while (x < 10) {
  x = x + 1;
}
```

```
begin: load 1
       ldc 10
       icmplt
       if_false end
       load 1
       ldc 1
       iadd
       store 1
       goto begin
end:
```

Compare Instructions

Instruction	Meaning
`cmpeq`	Compare equal (various types)
`cmpne`	Compare not equal (various types)
`icmpgt`	Integer compare greater than
`icmpge`	Integer compare greater equal
`icmplt`	Integer compare less than
`icmple`	Integer compare less equal

pop right, pop left, push boolean

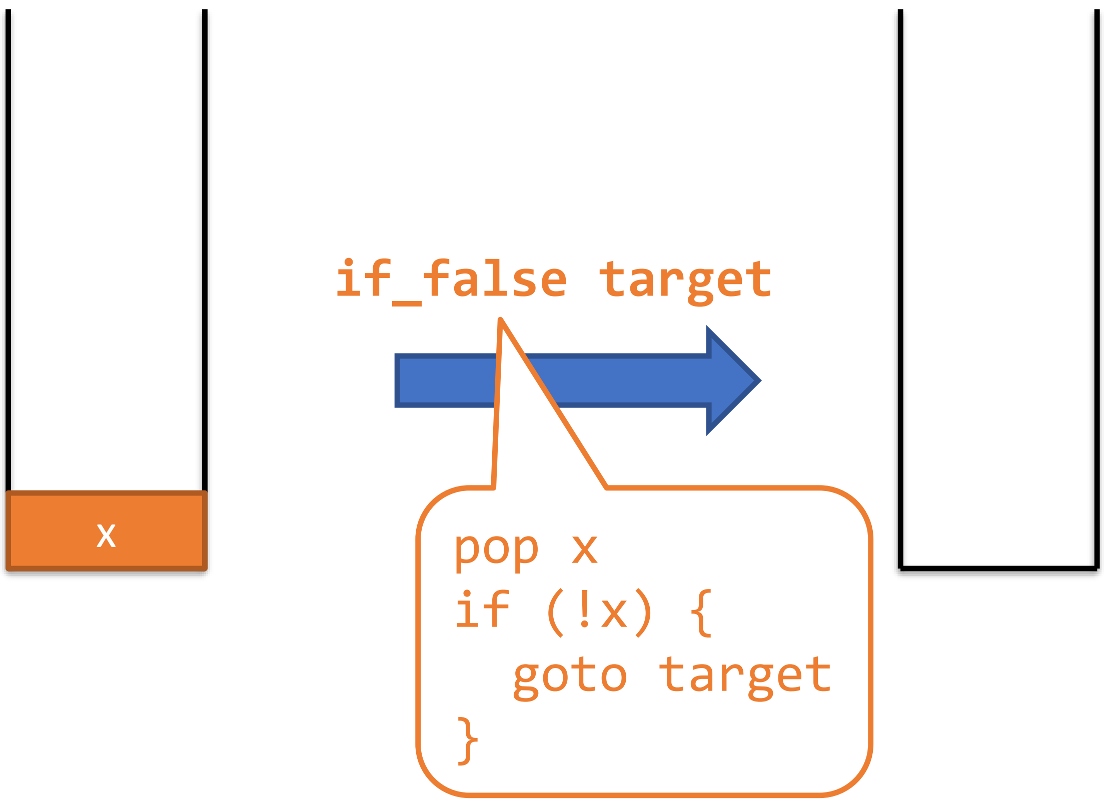
Branch Instructions

Instruction	Meaning	Evaluation Stack
`goto <label>`	Branch (unconditional)	-
`if_true <label>`	Branch if true	1 pop
`if_false <label>`	Branch if false	1 pop

Difference to Java bytecode: compare & branch combined, e.g `if_icmplt` (branch if integer compare less than)

Conditional Branch

- Pop boolean and decide
- Jump to label, if condition is fulfilled



Metadata

- Intermediate language knows all information about
  - Classes (names & types of fields and methods)
  - Methods (parameter types and return type)
  - Local variables (types)
- No immediate memory layout prescribed
- Not contained
  - Names of local variables and parameters
  - These are only numbered

Why is this information not needed in the VM?

Code Generation

- Traverse symbol table
  - Generate bytecode metadata
- Traverse AST per method (visitor)
  - Generate instructions via bytecode assembler
- Serialize to output file
  - Our case: Java serialization (convenience)
  - Java/.NET: proprietary binary format

Bytecode Assembler

```
var assembler = new Assembler(…);

var begin = assembler.createLabel();
var end = assembler.createLabel();
assembler.setLabel(begin);
assembler.emit(LOAD, 1);
assembler.emit(LDC, 10);
assembler.emit(ICMPLT);
assembler.emit(IF_FALSE, end);
assembler.emit(LOAD, 1);
assembler.emit(LDC, 1);
assembler.emit(IADD);
assembler.emit(STORE, 1);
assembler.emit(GOTO, begin);
assembler.setLabel(end);

assembler.complete();
```

```
begin:
      load 1
      ldc 10
      icmplt
      if_false end
      load 1
      ldc 1
      iadd
      store 1
      goto begin
end:
```

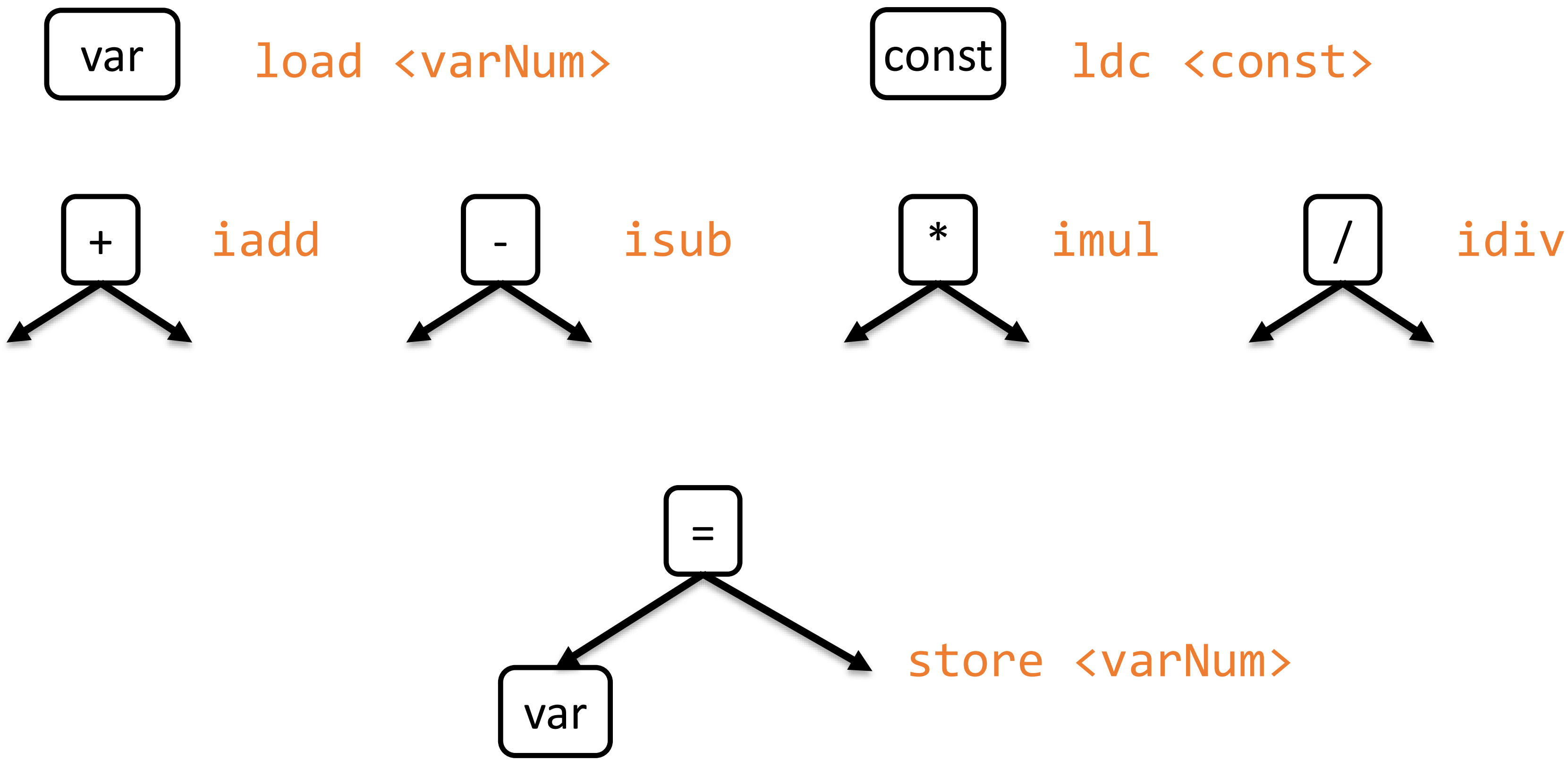
Backpatching

- Jump target resolution, e.g. at end of assembling
- Label => relative instruction offset at the end of current branch instruction

```
load 1
ldc 10
icmplt
if_false <5>
load 1
ldc 1
iadd
store 1
br <-9>
```

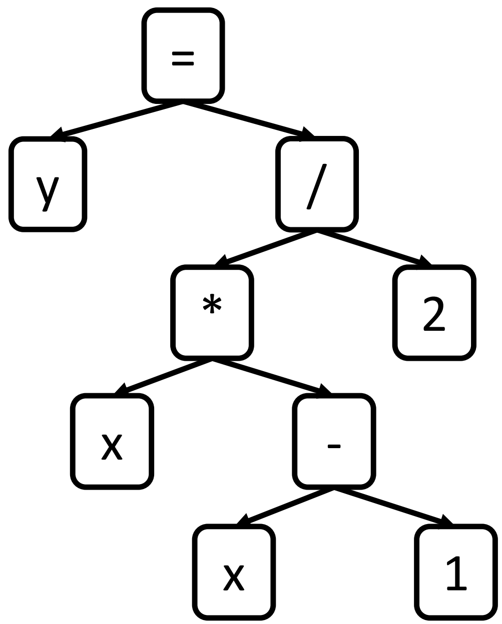
Tree Templates

- Code template per AST-node or subtree pattern



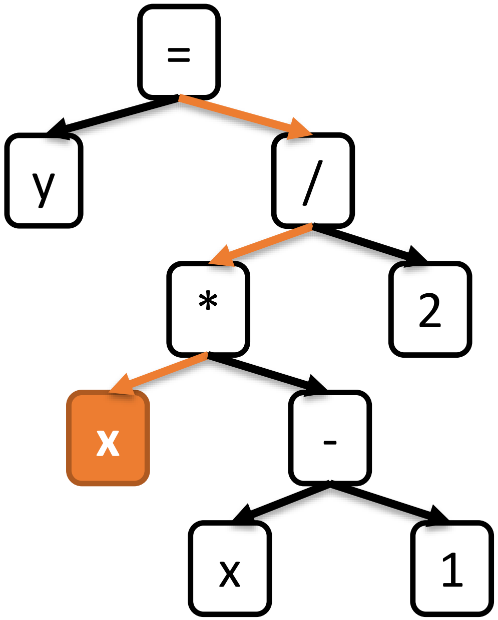
Template-Based Code Generation

- Post-order traversal: Visit children first
- Apply code template per detected subtree pattern



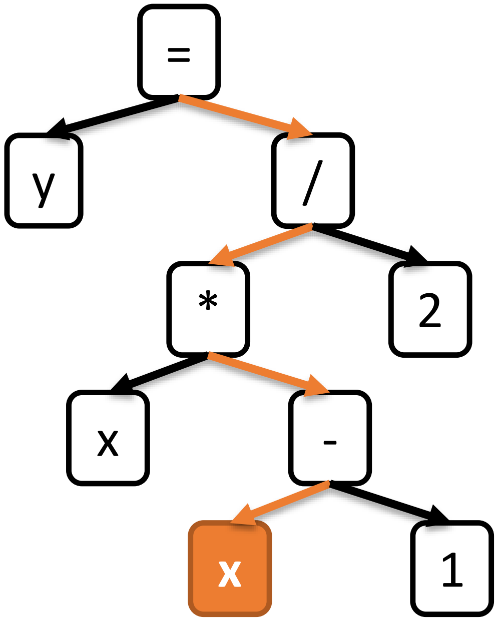
Local variables
int x; int y;

Code Generation Example



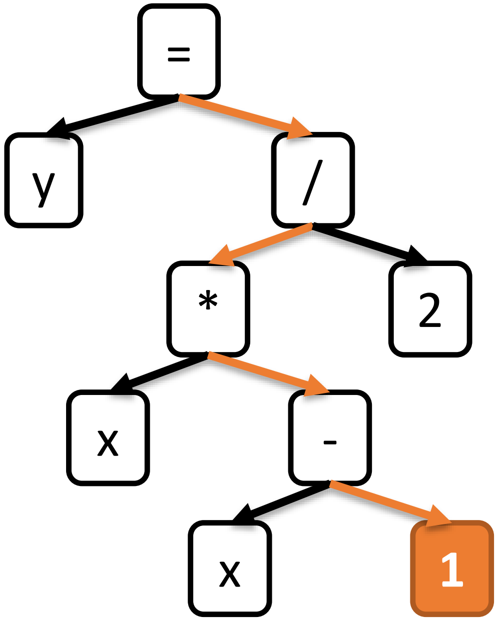

```
load 1
```

Code Generation Example



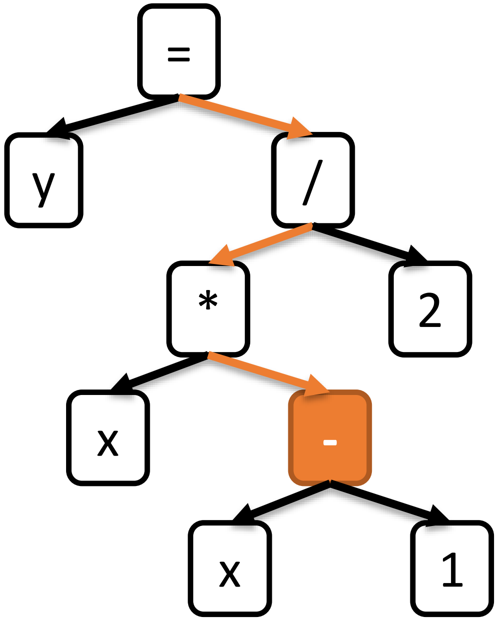

```
load 1
load 1
```

Code Generation Example



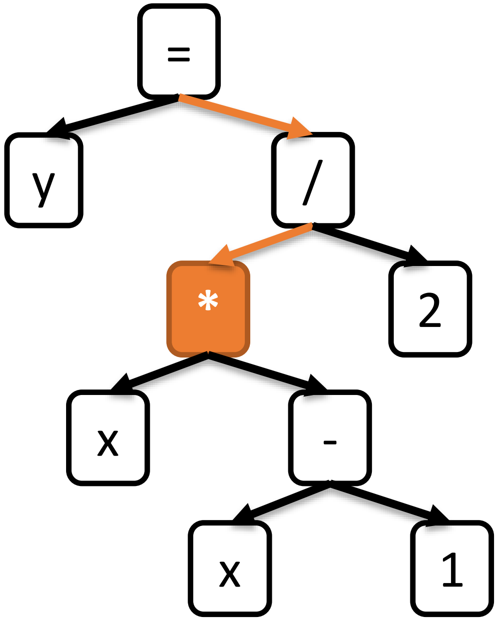

```
load 1
load 1
ldc 1
```

Code Generation Example



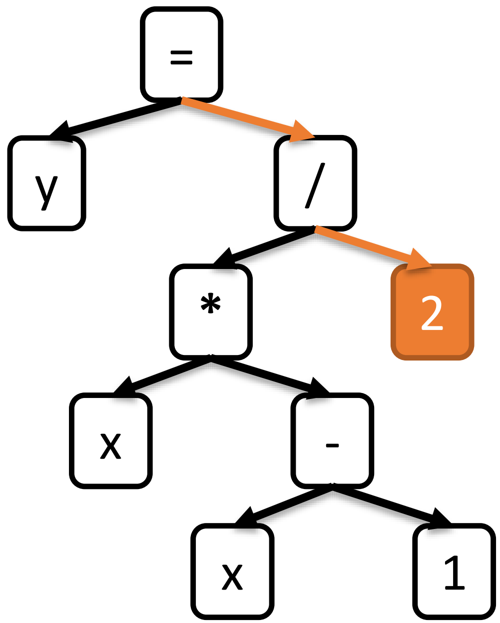

```
load 1
load 1
ldc 1
isub
```

Code Generation Example



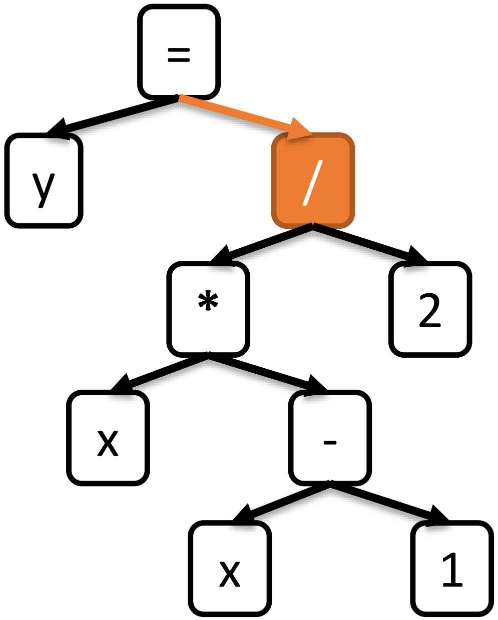

```
load 1
load 1
ldc 1
isub
imul
```

Code Generation Example



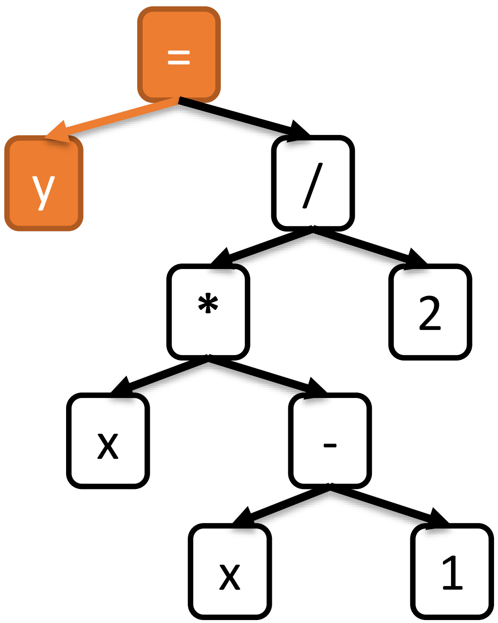

```
load 1
load 1
ldc 1
isub
imul
ldc 2
```

Code Generation Example




```
load 1
load 1
ldc 1
isub
imul
ldc 2
idiv
```

Code Generation Example

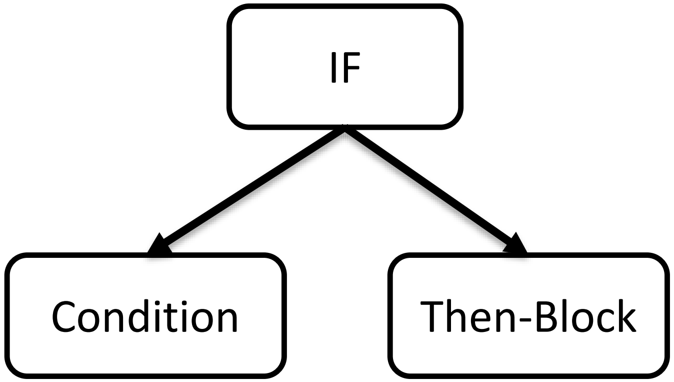



```
load 1
load 1
ldc 1
isub
imul
ldc 2
idiv
store 2
```

Traversal Order

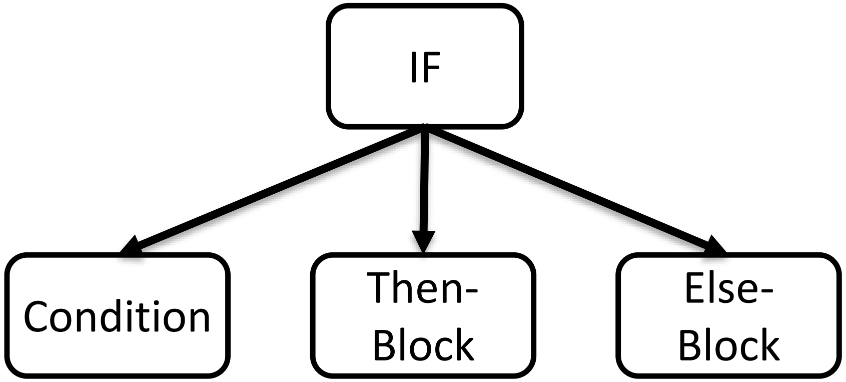
- For expressions: Always post order
- For statements: Depending on code template
  - Assignment: Right-hand side first, then template
  - If, If-Else, While etc. are more complex

If-Statement



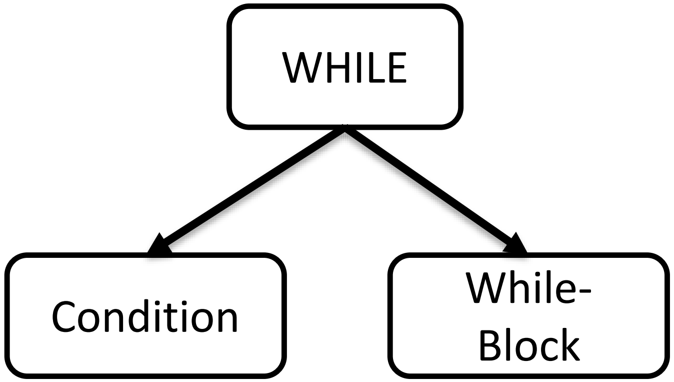

```
        Condition
        if_false target
        Then-Block
target:
```

If-Else Statement




```
           Condition
           if_false target0
           Then-Block
           goto target1
target0:   Else-Block
target1:
```

While-Statement




```
target0: Condition
         if_false target1
         While-Block
         goto target0
target1:
```

Visitor Implementation

```
@Override
public void visit(WhileStatementNode node) {
  var beginLabel = assembler.createLabel();
  var endLabel = assembler.createLabel();
  assembler.setLabel(beginLabel);
  node.getCondition().accept(this);
  assembler.emit(IF_FALSE, endLabel);
  node.getBody().accept(this);
  assembler.emit(GOTO, beginLabel);
  assembler.setLabel(endLabel);
}
```

accept() for code generation of subtrees

Conditional Evaluation

- Short-circuit logic

```
a && b
```

```
if a then b
else false
```

```
a || b
```

```
if !a then b
else true
```

Apply corresponding code templates

Method Calls

- Static
  - Predefined methods: readInt(), writeInt() etc.
- Otherwise all virtual (dynamic)
  - Bound to an object, e.g. `x.run()`, `this.run()`

Instruction	Meaning
`invokestatic`	Static call
`invokevirtual`	Virtual call

`load 0` for «this» reference

Static Method Call

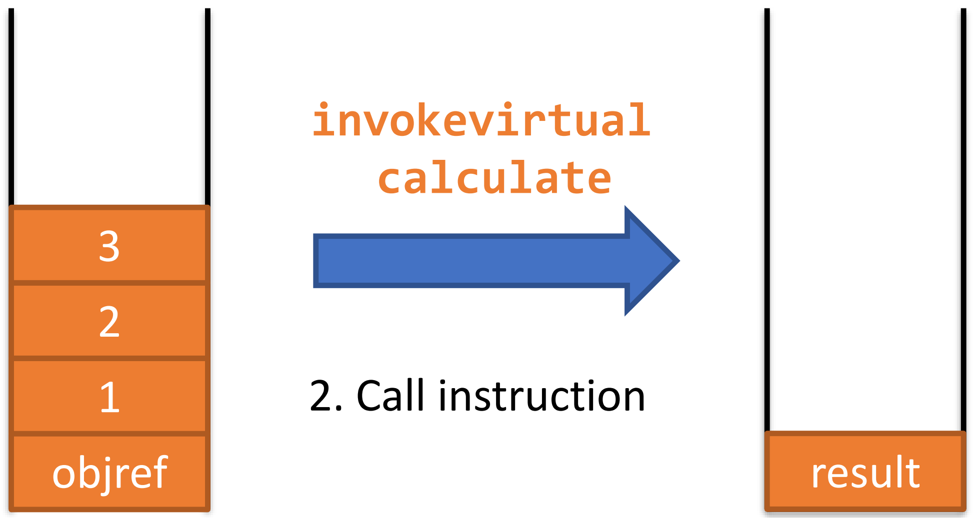
`writeInt(1)`

`readInt()`

Virtual Method Call

```
objref.calculate(1, 2, 3);
```

1. Arguments on stack (last on top),
   this reference at bottom



3. Call removes arguments & object reference, pushes result on stack (unless void)

Parameter & Return

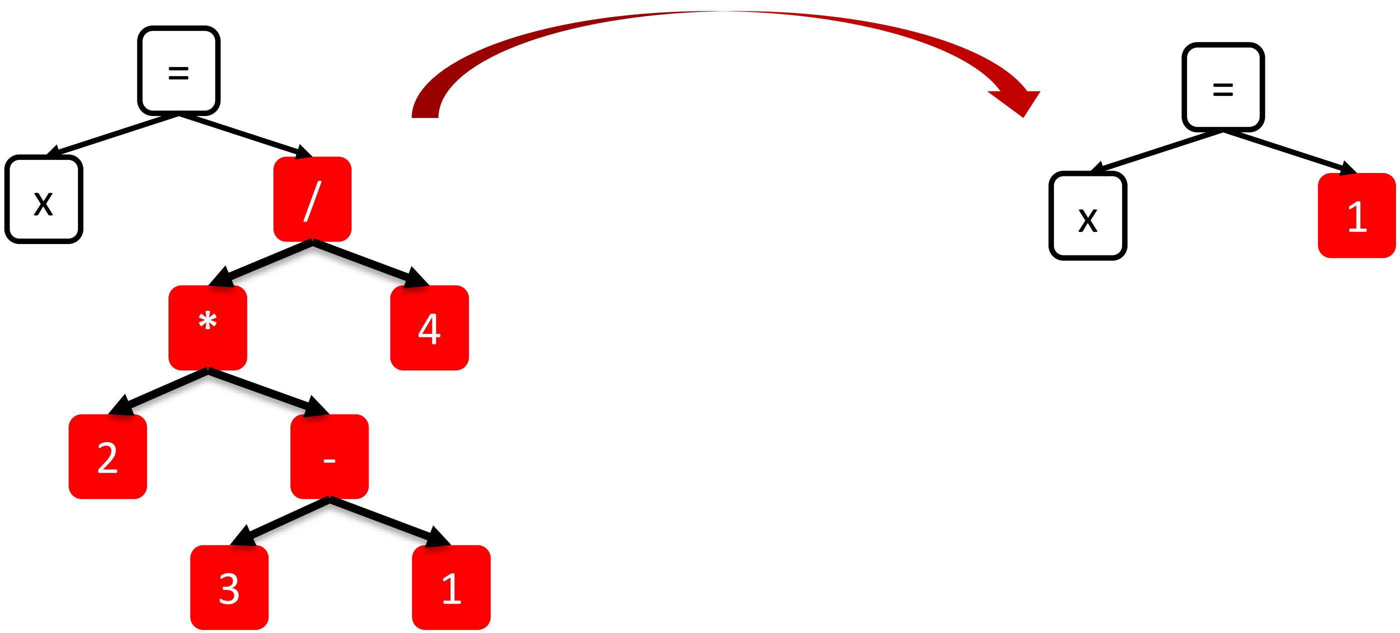
```
int sum(int x, int y) {
  return x + y;
}
```

```
load 1  // load parameter x
load 2  // load parameter y
iadd
ret       // return from method
```

ret is also required for void methods

Additional Possible Optimization

- Precompute constant sub-expression
- AST-rewriting before code generation



More optimization: Next lecture

Other Code Generation

- Direct machine code
  - Register allocation
  - Covered later (JIT compilation)
- Non-stack intermediate code
  - For certain machine-independent optimizations
  - E.g. Three Address Code

```
t1 = x  - 1
t2 = x  * t1
y  = t2 / 2
```

Review: Learning Goals

- ✓ Know the properties of modern intermediate language, such as Java bytecode
- ✓ Understand template-based code generation for a stack-based intermediate language

Further Reading

- Dragon Book, Code Generation
  - Section 8.9 (Template-based code gen/tree rewriting)
  - Section 6.6, 6.7, 6.9 (Control statements, calls)
- Optional, if interested
  - The Java Virtual Machine Instruction Set https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-6.html
  - Three Address Code: Dragon book, section 6.2

Course 142A Compilers & Interpreters

Code Optimization

Lecture Week 4, Wednesday

Prof. Dr. Luc Bläser

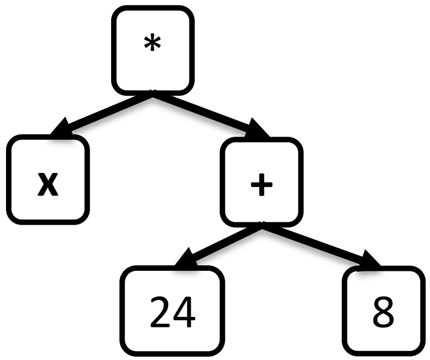
Last Lecture - Quiz

```
load 1
ldc 24
ldc 8
iadd
imul
```

Which expression could this bytecode origin from?

Disassembling

```
load 1
ldc 24
ldc 8
iadd
imul
```




```
x * (24 + 8)
```

How could this be optimized?

Optimization

`x * (24 + 8)`

Constant expression

`x * 32`

Left shift is faster

`x << 5`

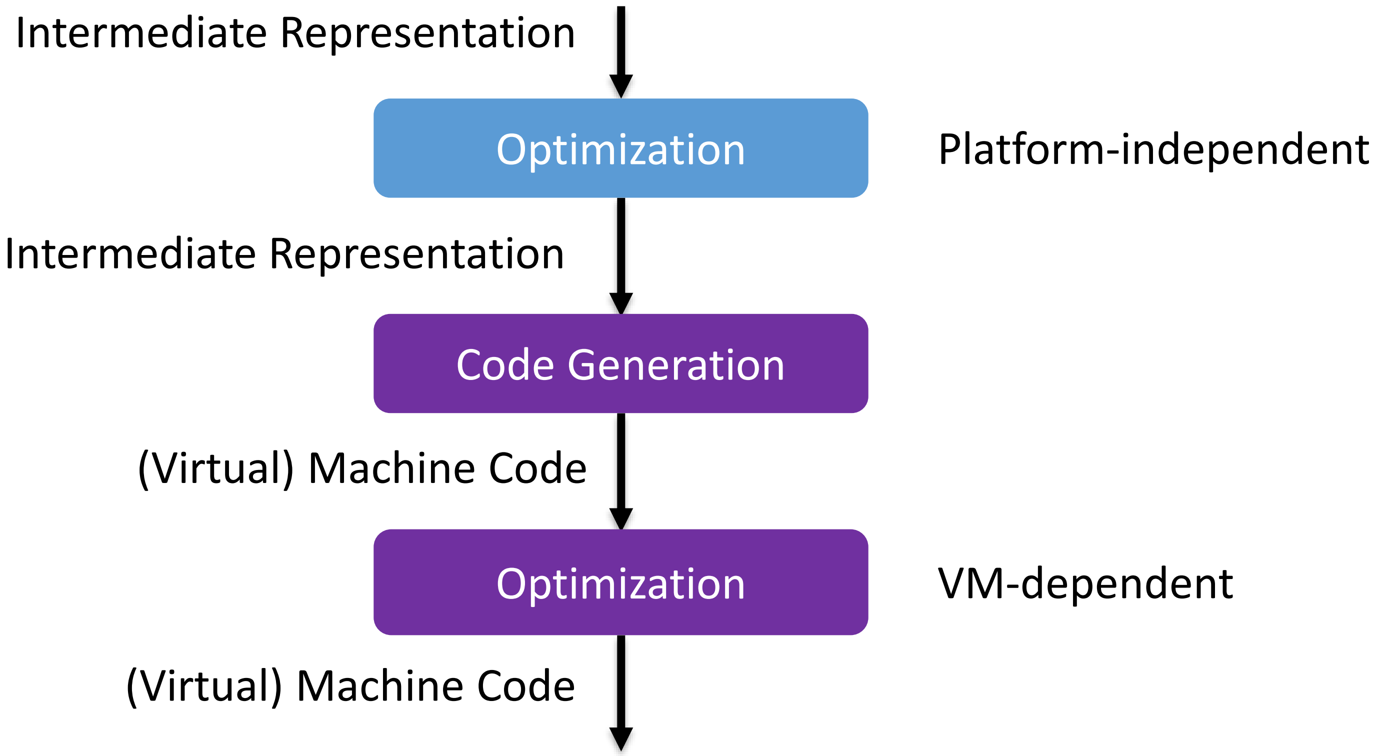
Today's Topics

- Various optimizations
  - Optimized arithmetic
  - Algebraic simplification
  - Loop-invariant code motion
  - Common subexpression elimination
  - Dead code elimination
  - Copy propagation
  - Constant propagation
  - Partial redundancy elimination
- And their facilitating analyses

Learning Goals

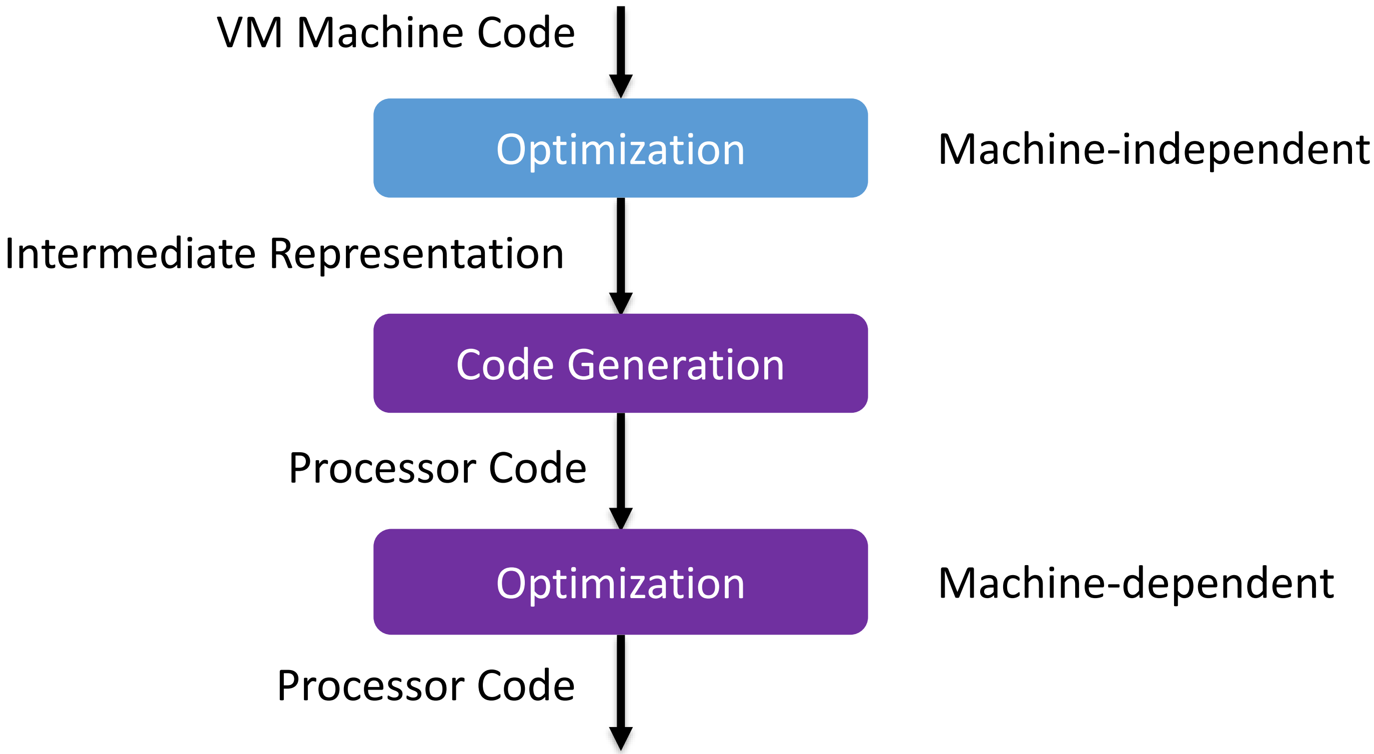
- Know different common code optimizations
- Gain an overview of analyses for optimization

Compiler Steps



(possible additional optimizations steps via intermediate code)

JIT-Compiler Steps



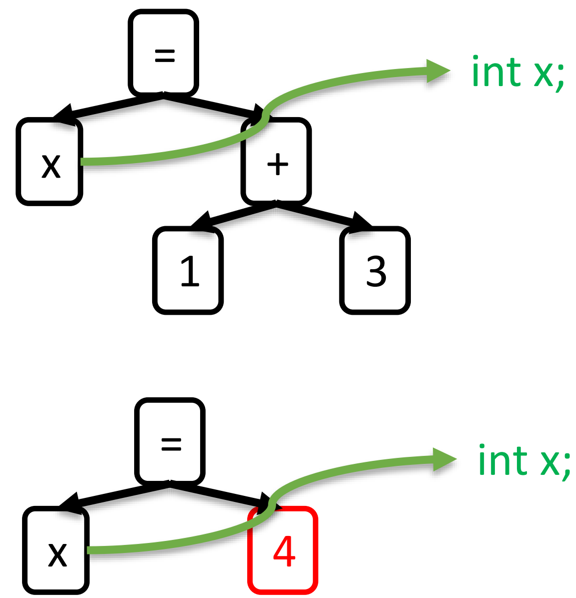
JIT compiler covered later in course

Our Focus: Optimization

Intermediate Representation

Optimization

Intermediate Representation



Task of the Optimization

- Transformation of intermediate representation / machine code
  - Into more efficient representation/machine code
- Possible intermediate representations:
  - AST + symbol table
  - Bytecode
  - Other intermediate code (e.g. Three Address Code)
- Series of optimization steps possible

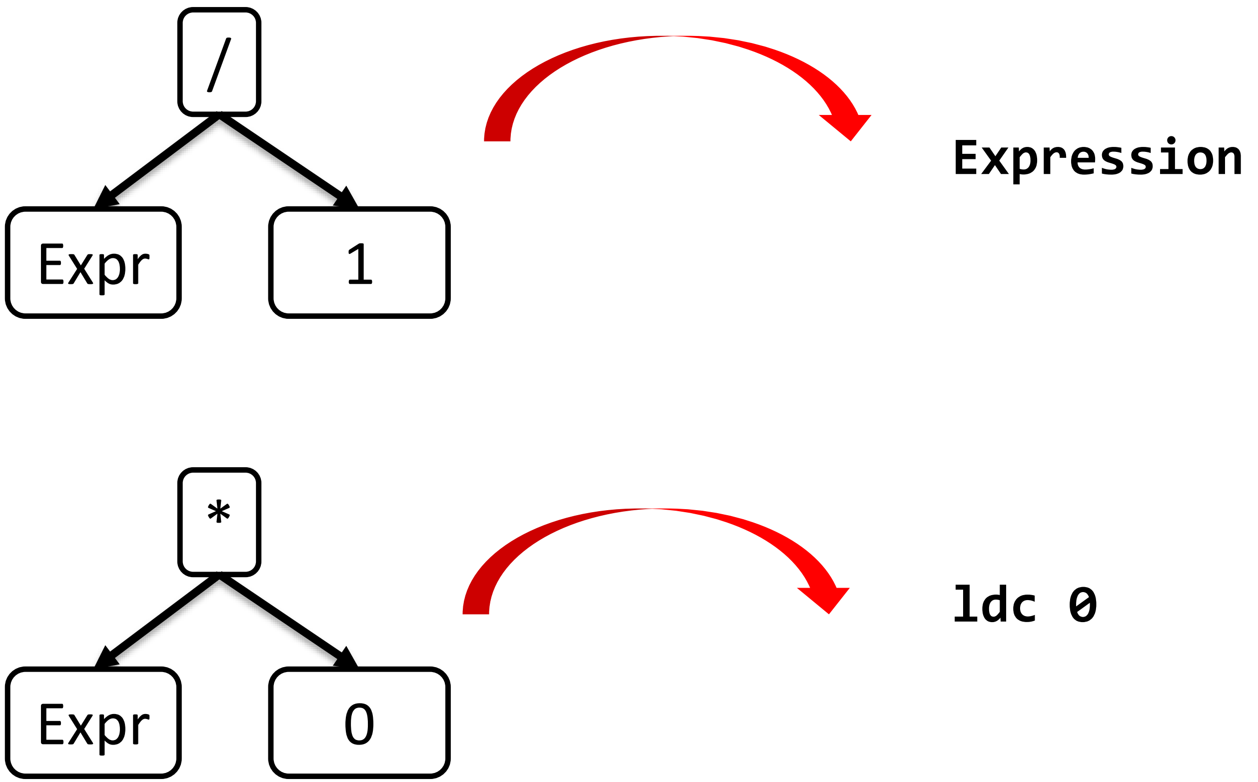
Optimized Arithmetic

- Multiply, divide, remainder by power of 2 constants
  - Cheap bitwise operations can be applied (e.g. JIT compiler)

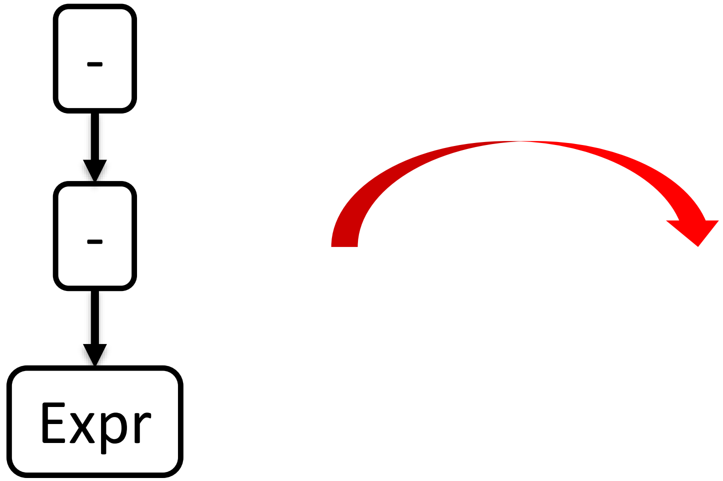
```
x * 32        x << 5

x / 32        x >> 5

x % 32        x & 31
```

Algebraic Simplification (1)

- E.g. by template-based code generation

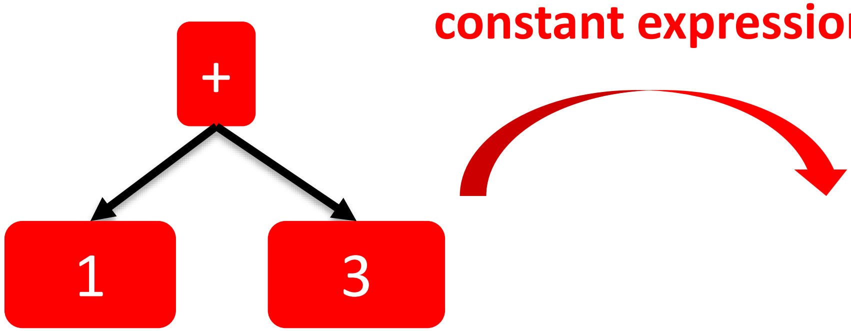


Algebraic Simplification(2)



`Expression`

constant expression



`ldc 4`

Loop-Invariant Code

Never changes during loop

```
while (x < N * M) {
    k = y * M;
    x = x + k;
}
```

Also loop-invariant

How should we optimize this?

Code Motion

```
while (x < N * M) {
    k = y * M;
    x = x + k;
}
```

Move invariant code out of the loop

```
k = y * M;
temp = N * M;
while (x < temp) {
   x = x + k;
}
```

Common Subexpressions

```
x = a * b + c;
…
y = a * b + d;
```

Repeatedly evaluated subexpression

⚠ **a and b must not be altered in between**

Common Subexpression Elimination

```
x = a * b + c;
…
y = a * b + d;
```

```
temp = a * b;
x = temp + c;
…
y = temp + d;
```

Dead Code

```
a = readInt();
b = a + 1;
writeInt(a);
c = b / 2;
```

no subsequent read of c
=> dead code

Dead Code Elimination

```
a = readInt();
b = a + 1;
writeInt(a);
c = b / 2;
```

```
a = readInt();
b = a + 1;
writeInt(a);
```

b is now never used

```
a = readInt();
writeInt(a);
```

Can also be eliminated in one step (discussed next week)

Redundant Loads and Stores

```
t = x + y;
u = t;
writeInt(u);
```

```
writeInt(x + y)
```

Copy Propagation

- Idea: Use last assigned expression for read variable

```
t = x + y;
u = t;
writeInt(u);
```

```
t = x + y;
u = x + y;
writeInt(u);
```

```
t = x + y;
u = x + y;
writeInt(x + y);
```

What have we gained?

Constant Propagation

- Also called constant folding

```
a = 1;
if (…) {
  a = a + 1;
  b = a;
} else {
  b = 2;
}
c = b + 1;
```

b is here constant 2

c is guaranteed to be 3

Detection discussed next week

Constant Propagation

```
a = 1;                    a = 1;
if (…) {                  if (…) {
  a = a + 1;                a = 2;
  b = a;                    b = 2;
} else {                  } else {
  b = 2;                    b = 2;
}                         }
c = b + 1;                c = 3;
```

Can afterwards eliminate dead or duplicate code

Partial Redundancy

```
if (…) {
  y = x + 4;
} else {
  …
}
z = x + 4;
```

On if-path, x + 4 is evaluated twice

On else-path, only once

Partial Redundancy Elimination

```
if (…) {
  y = x + 4;
} else {
  …
}
z = x + 4;
```

```
if (…) {
  t = x + 4;
  y = t;
} else {
  …
  t = x + 4;
}
z = t;
```

x + 4 is now evaluated only once on each path

Detecting Possible Optimizations

Different facilitating techniques

- Static single assignment
- Peephole optimization
- Dataflow analysis

Revisiting Common Subexpression

```
x = a * b + c;
…
…
y = a * b + d;
```

Need to know whether a or b is assigned

⚠ **Only common subexpression if a and b remained the same**

Static Single Assignment (SSA)

- Code transformation easing analysis & optimization

```
x = 1;
x = 2;
y = x;
```

$x_1 = 1;$
$x_2 = 2;$
$y_1 = x_2;$

Every variable is assigned only once in the code

More Complex on Branches

```
if (…) {
  x = 1;
} else {
  x = 2;
}
y = x;
```

```
if (…) {
  x_1 = 1;
} else {
  x_2 = 2;
}
y_1 = x_??;
```

Which version of x?

Phi Function

```
if (…) {
   x₁ = 1;
} else {
   x₂ = 2;
}
y₁ = φ(x₁, x₂);
```

Meaning: If first path (if) is taken, select x_1, otherwise x_2.

SSA Application: Common Subexpressions

- Can be immediately determined in SSA

$x_1 = a_1 * b_1 + c_1;$

…

$y_1 = a_1 * b_1 + d_1;$

…

$z = a_1 * b_2 + d_3;$

Know that b and d changed

Other SSA Application: Dead Code

x_1 never read => dead code

```
x₁ = 1;
x₂ = 2;
y₁ = x₂ + 1;
writeInt(y₁);
```

SSA Computation

- Relatively complicated and expensive
  - Especially deciding where to put which phi (so called dominance frontier)
- Cheaper techniques wanted, especially for JIT
  - E.g. peephole optimization

Peephole Optimization

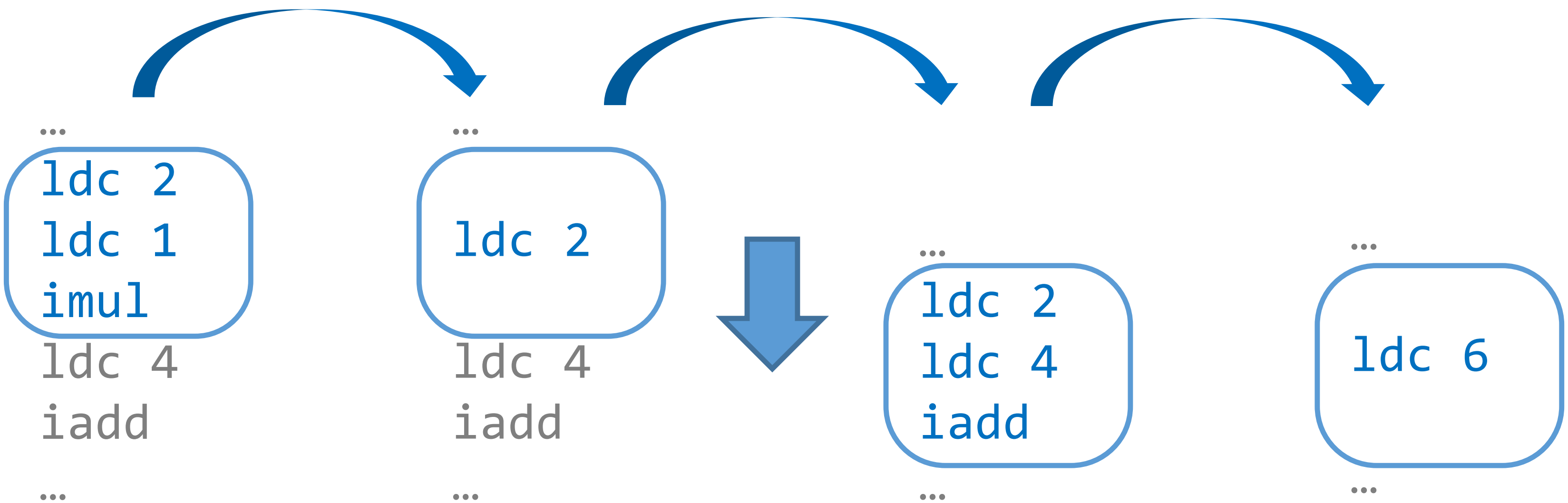
- Optimization over very small amount of instructions
- Used in JIT and on intermediate or machine code

```
…
ldc 2
ldc 1
imul
ldc 4
iadd
…
```

Sliding window of e.g. 3 instructions

Apply optimization pattern inside sliding window

Peephole Optimization



Dataflow Analysis

- Powerful generic code analysis technique
- Useful for many optimizations

Next week's topic

Summary

Optimzation	Facilitating Techniques
Optimized arithmetic	Template-based code gen Peephole optimization
Algebraic simplification	Template-based code gen Peephole optimization
Common subexpression elimination	SSA Dataflow analysis
Dead code elimination	SSA Dataflow analysis
Copy propagation	SSA Dataflow analysis
Constant propagation	SSA Dataflow analysis
Partial redundancy elimination	SSA Dataflow analysis

Review: Learning Goals

- ✓ Know different common code optimizations
- ✓ Gain an overview of analyses for optimization

Further Reading

- Dragon Book, Code Optimization
  - Section 6.2.4: Static single assignment
  - Section 8.7: Peephole optimization
  - Section 9.1-9.1.7: common subexpressions, copy propagation, dead code elimination, code motion
- Optional, if interested
  - Section 6.6.5: Avoiding redundant gotos
  - Section 9.4: Constant propagation (we will revisit it with dataflow analysis)
  - Section 9.5: Partial redundancy elimination (also using dataflow analysis, various steps)