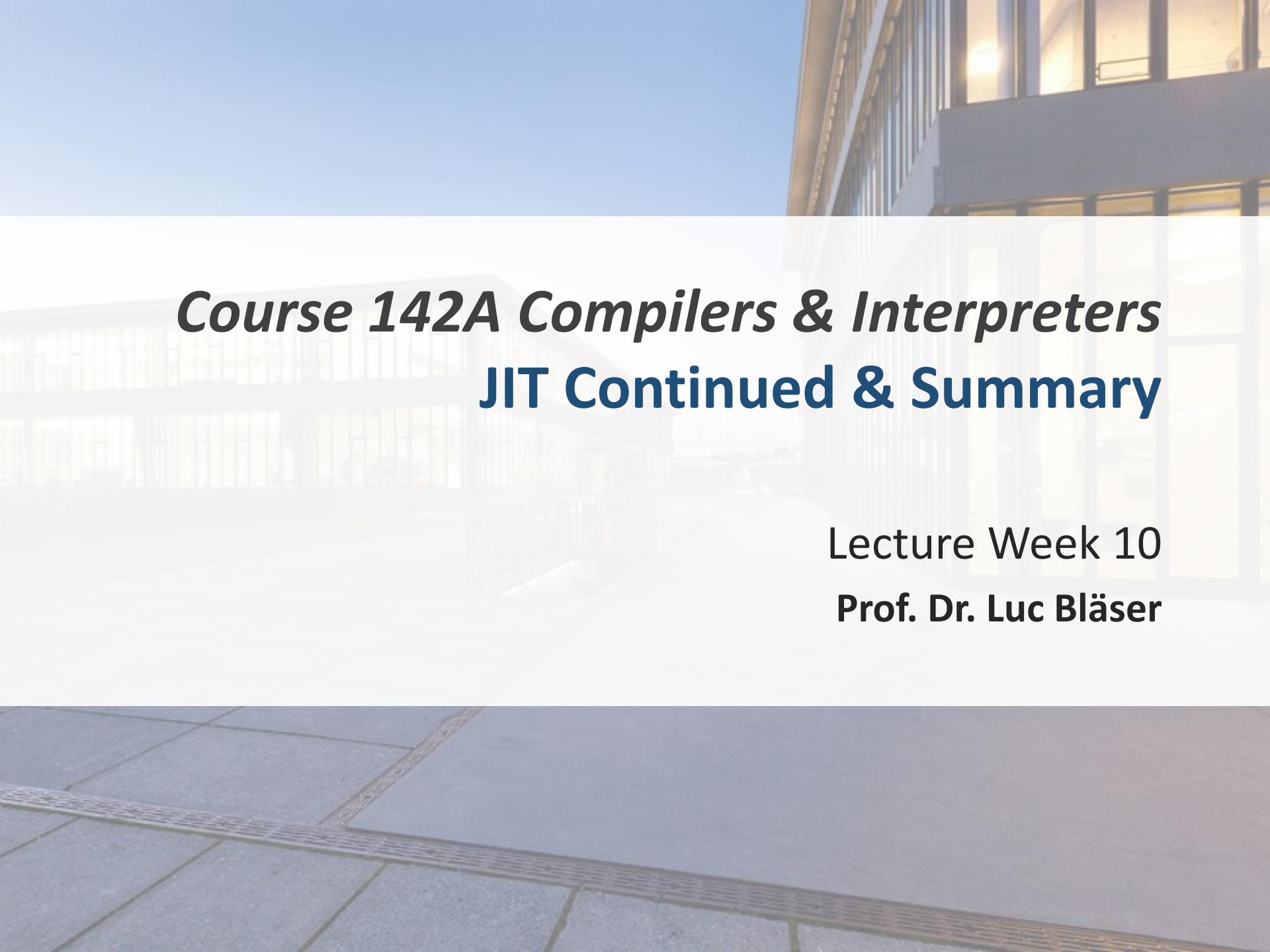


Course 142A Compilers & Interpreters
JIT Continued & Summary

Lecture Week 10

Prof. Dr. Luc Bläser

Quiz – Last Lecture

Which aspects did we leave open in the JIT-compiler?

Open Aspects

- Branches: Consistent allocations
- Register pressure
- Native code execution

Today's Agenda

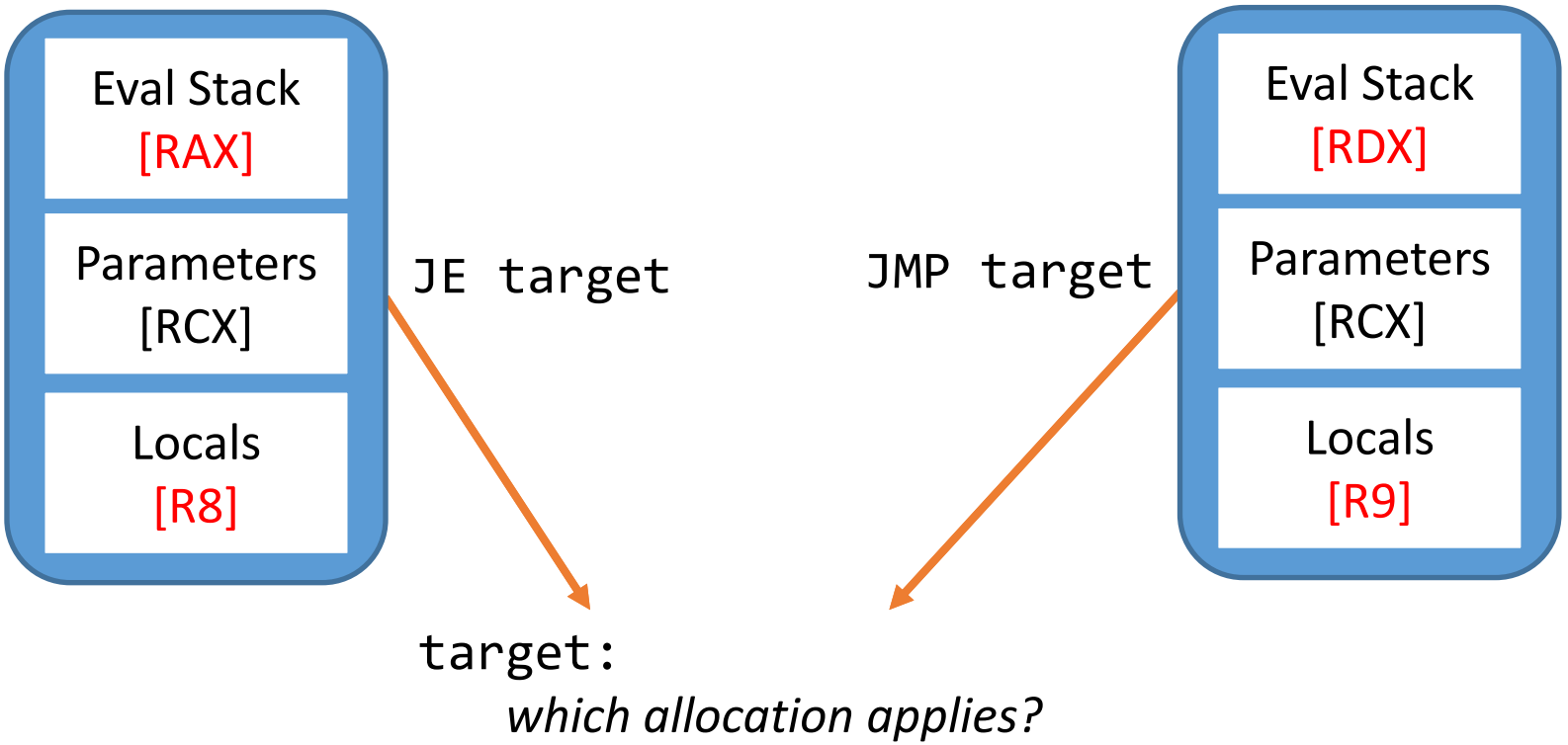
- JIT-Compiler continued
- Course review & summary

Learning Goals

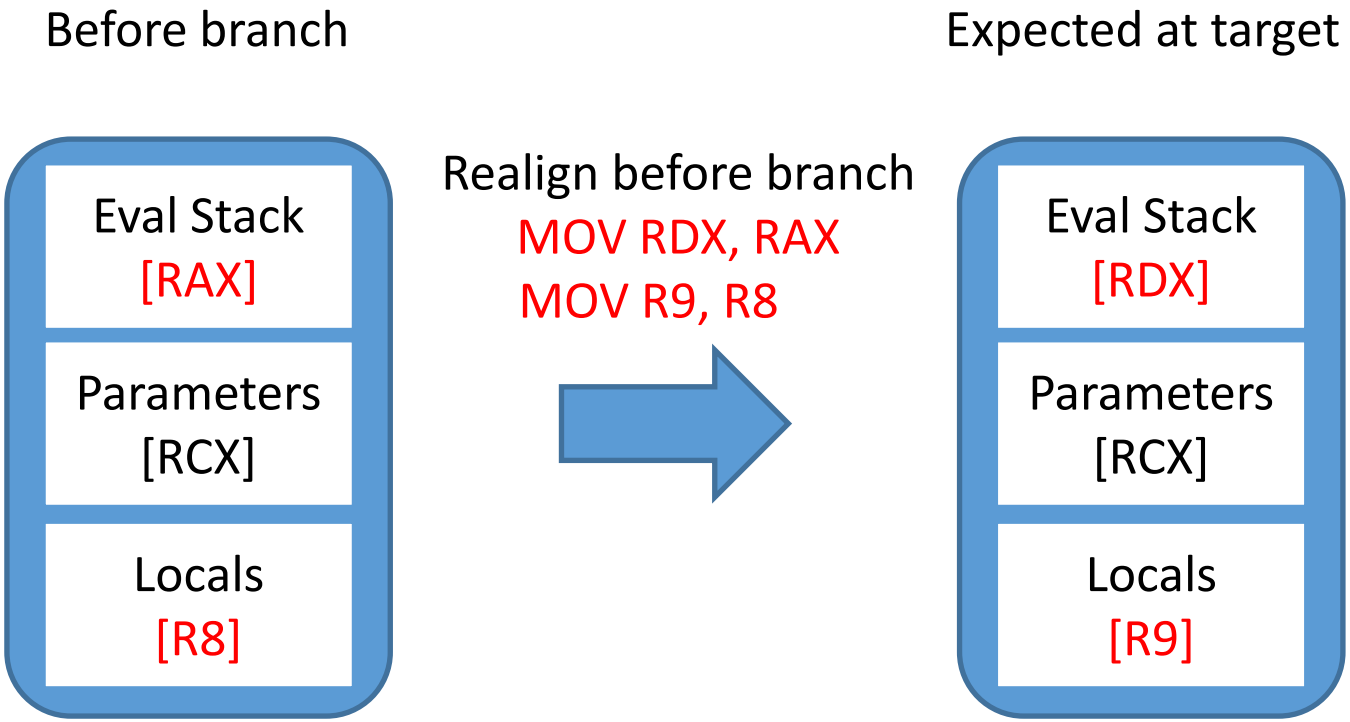
- Know remaining aspects of a JIT-compiler
- Be able to implement the full JIT-compiler for your virtual machine

Branches: Inconsistent Allocations

- Different allocations could be used for branch target



Match Allocations on Branches



JIT Compiler Code (if_true)

```
switch (opCode) {
  case if_true:
    var offset = (int)instruction.getOperand();
    var target = code[position + 1 + offset];
    var label = labels.get(target);
    matchAllocation(label);
    if (previous == CMPEQ) {
       assembler.JE_Rel(label);
    } …
    break;
  …
}
```

Get label for target

Realign allocation for target

Jump requires label

Register Pressure

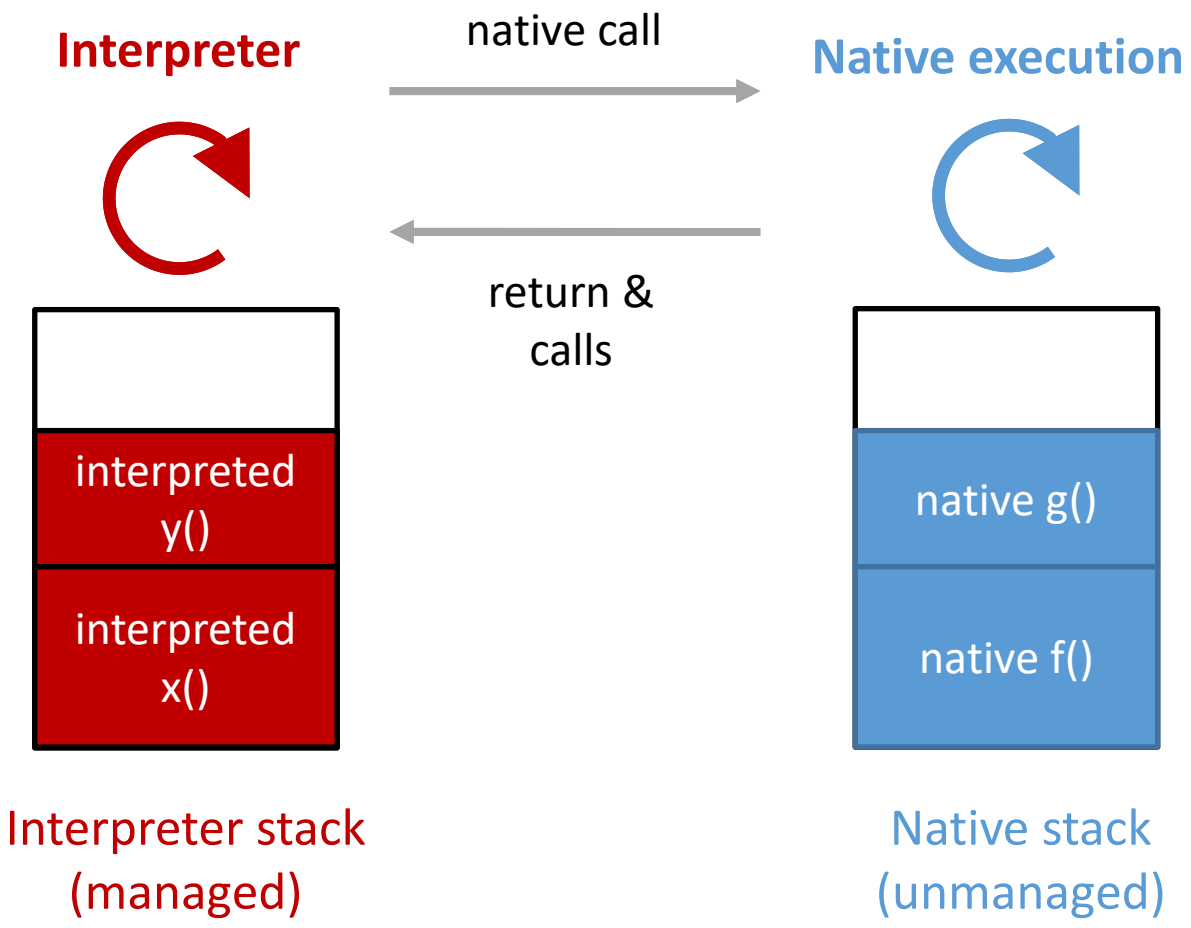
- Limited amount of registers (14 general purpose)
- JIT may go out of registers
  - Too deep evaluation (local allocation)
  - Too many locals/params (global allocation)
- Solutions
  - Stack spilling: push temporary values on call stack
  - Store locals/params on call stack

Not done in our JIT (simplification)

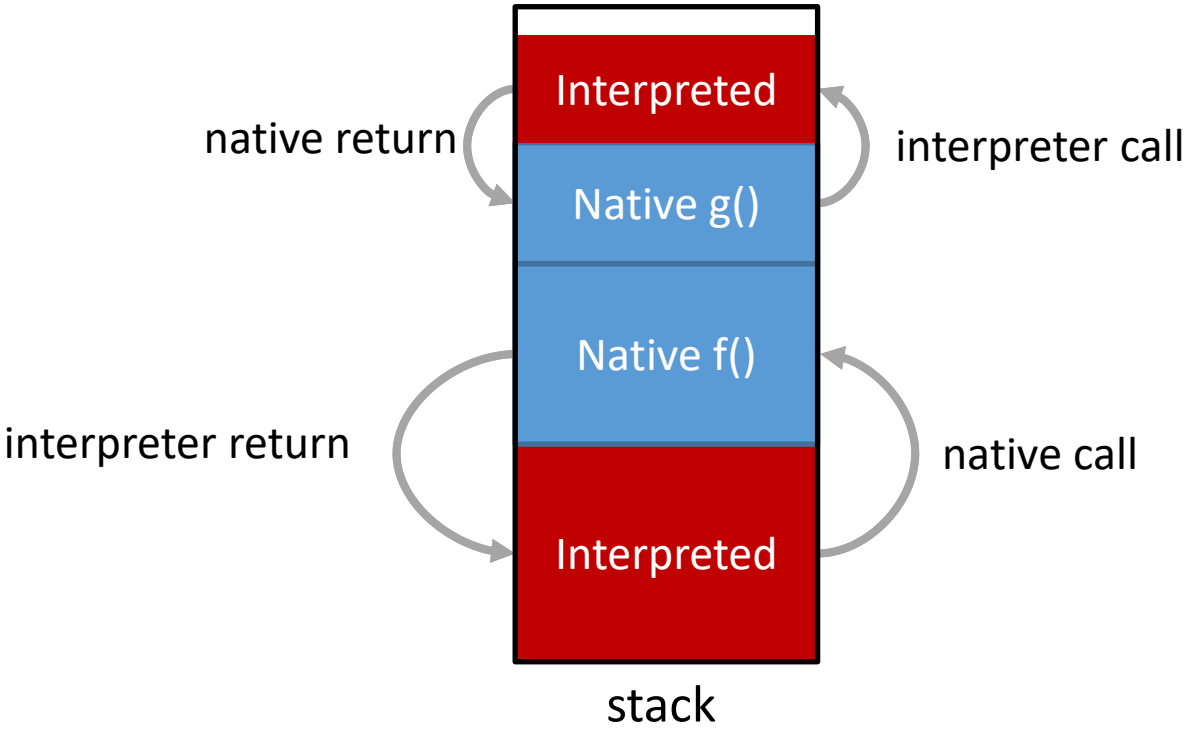
Assembler and Linker

- Assembler => binary instruction encoding
  - According to Intel 64 specification
  - Builder design pattern
- Linker => patch addresses in instructions
  - Link to static variables
  - Method call targets
  - Known at JIT time
  - Directly insert in JIT-compilation

HW Execution



HW Execution



Native Code

- Through Java Native Access (JNA)
  - Using small C library (platform-dependent)
- Security:
  - Executable code must be in special virtual page (executable flag)

```
VirtualProtect(code, length, PAGE_EXECUTE, …);
```

- Native call:
  - Push IP, branch to native method code

```
((func*)code)(arguments)
```

Other Aspects

- System calls in native code
  - Heap allocation, virtual method calls, type tests
- Garbage collection
  - Native stack frames & registers belong to root set

Simplification: We only JIT-compile methods with certain instructions

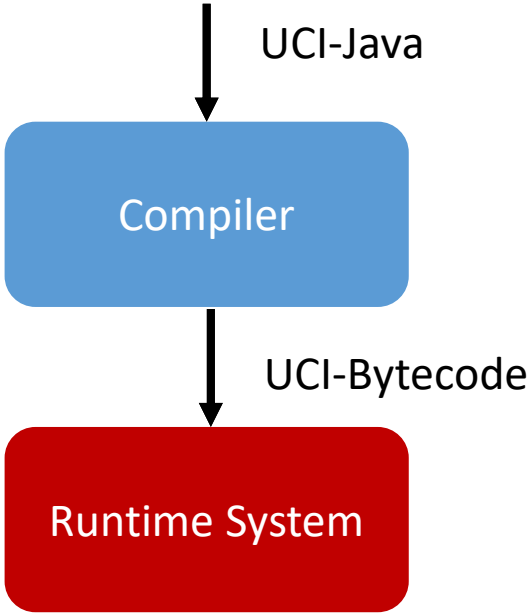
Learning Goals

- ✓ Know remaining aspects of a JIT-compiler
- ✓ Be able to implement the full JIT-compiler for your virtual machine

Course Summary & Review

Review: Covered Topics

- Design of full compiler and runtime system
- For a realistic modern OO programming language



Lexer
Parser
Semantic Checker
Code Generator
Code Analysis & Optimization

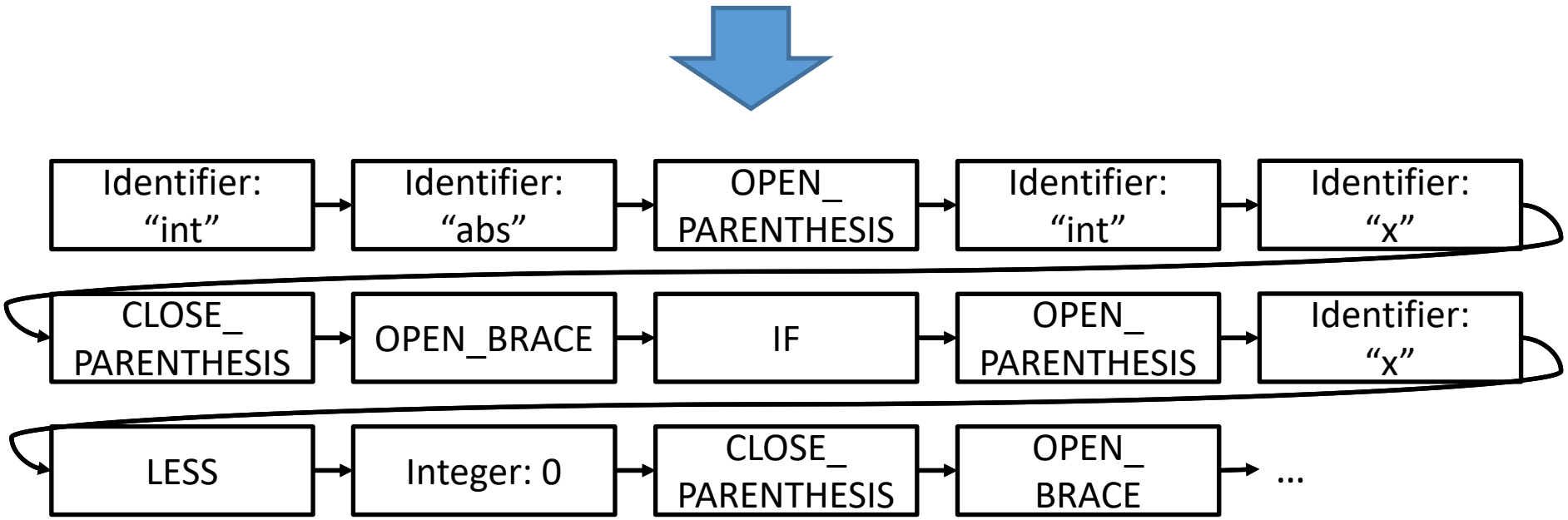
VM Interpreter
Stack, Heap, Metadata
OO Support & Type Polymorphism
GC
JIT-Compiler

Summary Lexer

- Combine characters to tokens
  - Keywords, operators, punctuation, identifiers, literals (int, string)
- Processes regular language
  - expressible as EBNF without recursion
- Eliminates white spaces and comments
  - As far as irrelevant for language
- Goal: Ease parsing (constant-size lookaheads)

Example Lexer

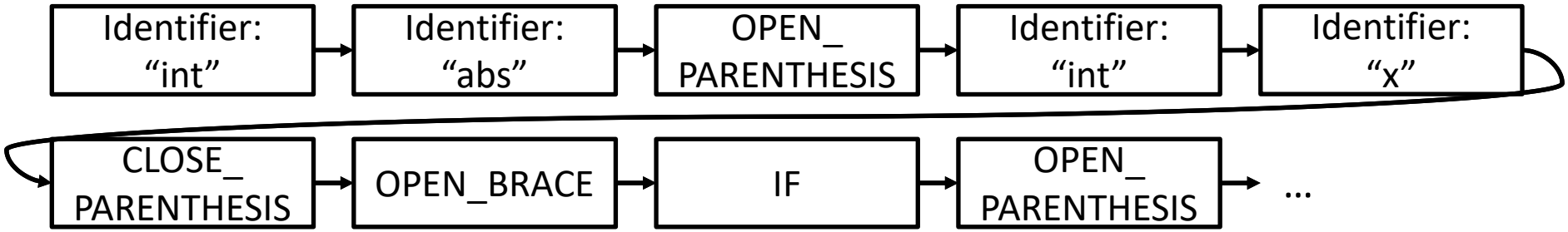
```
int abs(int x) {
  if (x < 0) {
    x = -x;
  }
  return x;
}
```

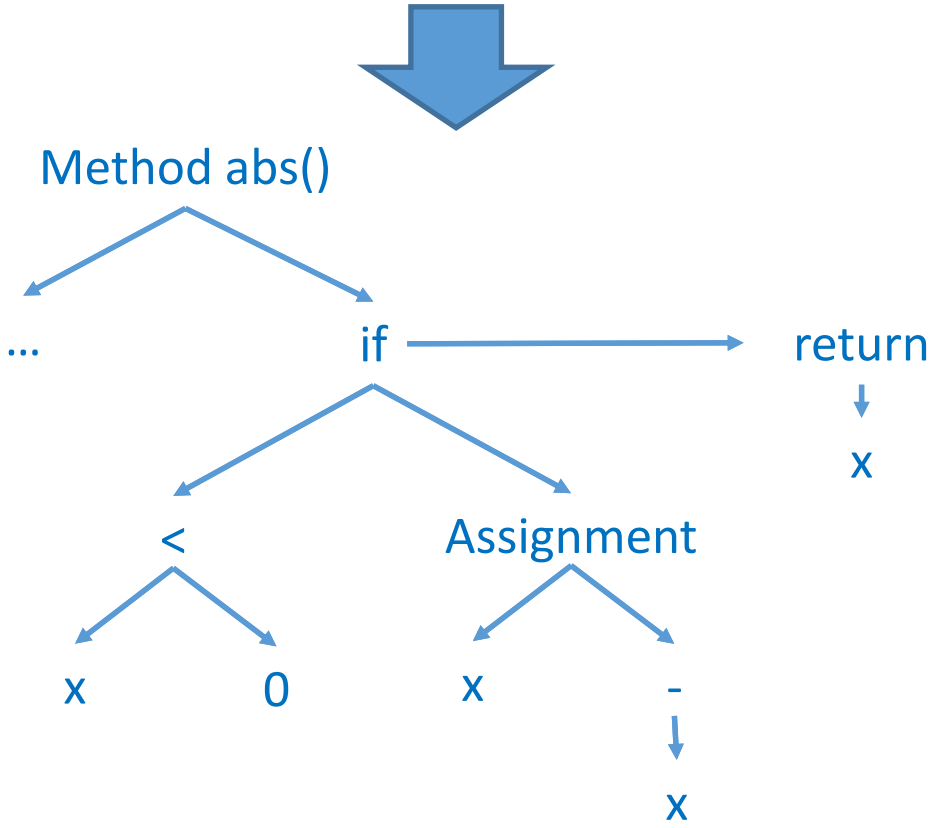


Summary Parser

- Analyzes syntax
  - Top-down (LL-k) or bottom-up (LR-k)
- Creates syntax tree
  - Concrete or abstract
- Context-free grammar
  - EBNF (some limitations with certain parsers)
- We built a predictive recursive descent parser
  - But there exist also parser generators

Example Parser

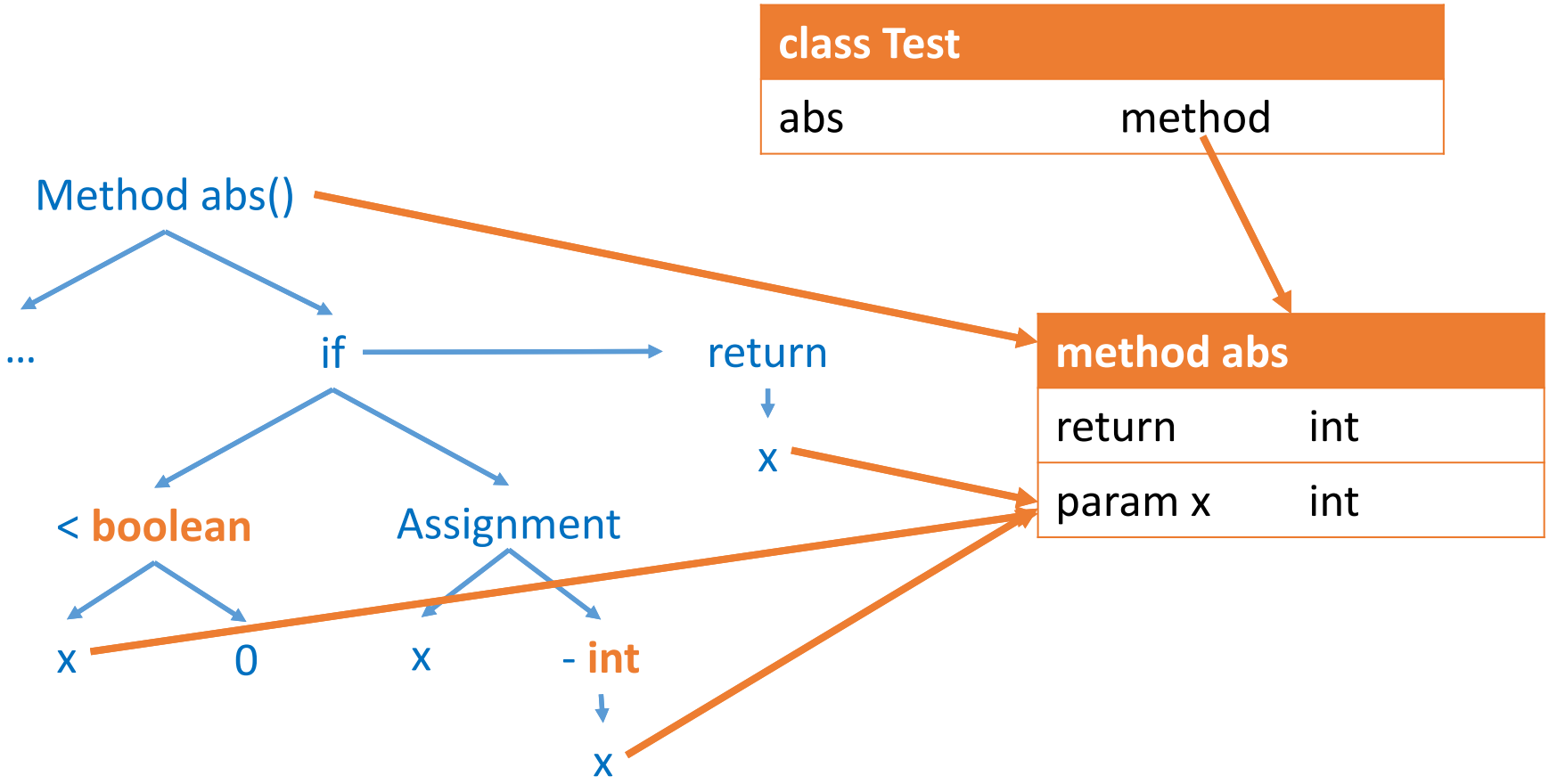




Summary Semantic Checker

- Build a symbol table
  - Database of all declarations (scoped)
- Resolve declarations & types
  - For all designators and expressions
- Type checking
  - Assignment, parameter passing, return values
- Other semantic checks
  - Main-method, boolean if/while-conditions etc.

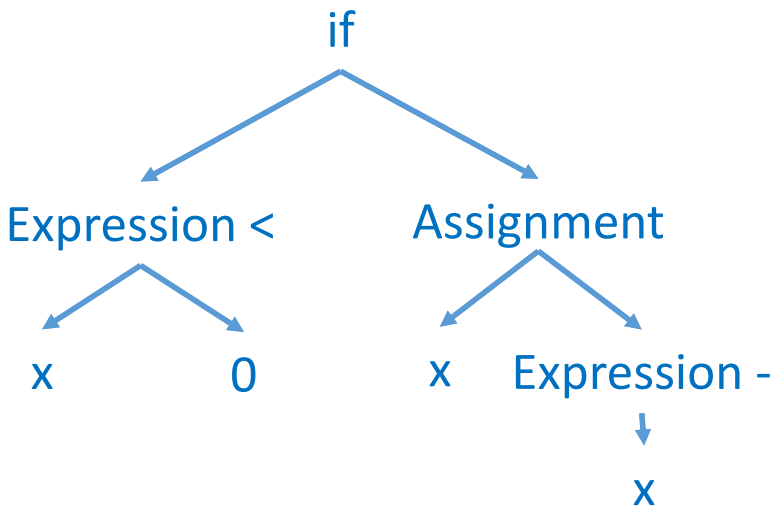
Example Semantic Checker



Summary Code Generator

- Generate machine code
  - VM code (stack processor) or HW code (register processor)
- Template-based code generation
  - Map AST pattern to instructions
  - Custom traversal (visitor pattern)
    - Expressions: Bottom-up
    - While/if: With branches

Example Code Generation

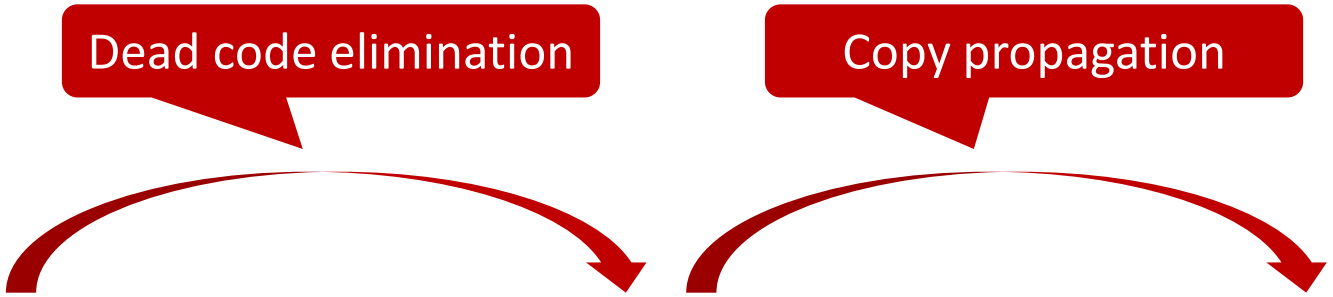



```
  load 1
  ldc 0
  icmplt
  br_false label1
  load 1
  ineg
  store 1
label1:
  …
```

Summary Code Optimization

- Optional step in between
  - Transform IR into more efficient IR
  - Before code gen, on intermediate code, in JIT-compiler etc.
- Different optimizations e.g.
  - Algebraic simplification
  - Common subexpression elimination
  - Dead code elimination
  - Copy propagation
  - Constant propagation

Example Code Optimization




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

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

```
writeInt(
    readInt());
```

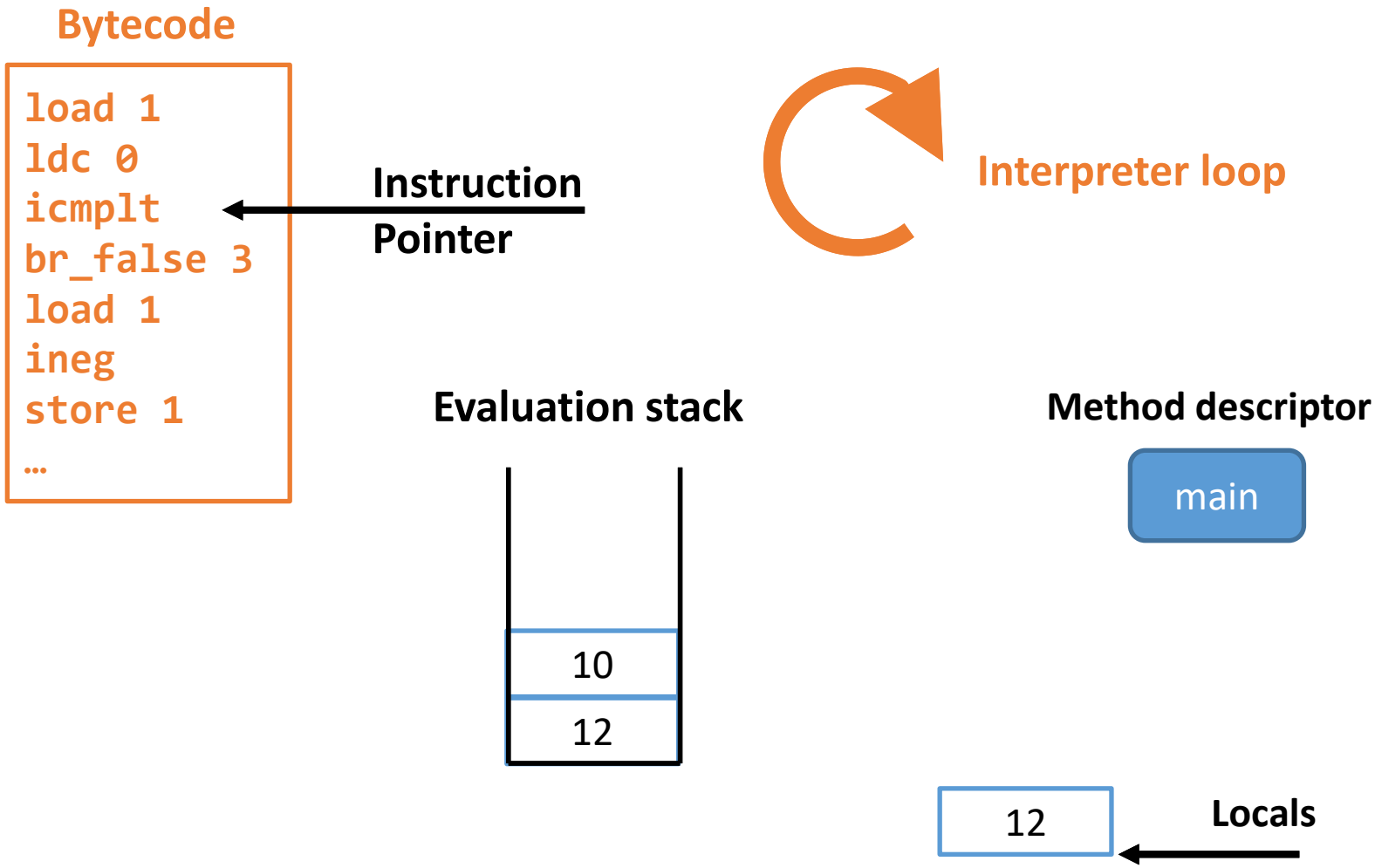
Summary Code Analysis

- Control Flow Graph (CFG)
  - Basic blocks as nodes
  - Branches as edges
- Dataflow Analysis as generic tool
  - Fixpoint iteration over CFG
  - Configuration: State, Transfer and Join
  - Transfer with Gen- and Kill-set
  - Forward or backward direction

Summary VM Interpretation

- Emulate IL instructions
  - Instruction pointer
  - Evaluation stack
- Runtime structures
  - Call stack (method activation frames)
  - Heap (object allocations)
  - Metadata (type/method descriptors)
- Verifier (static or dynamic)
  - For safety and security

Interpreter

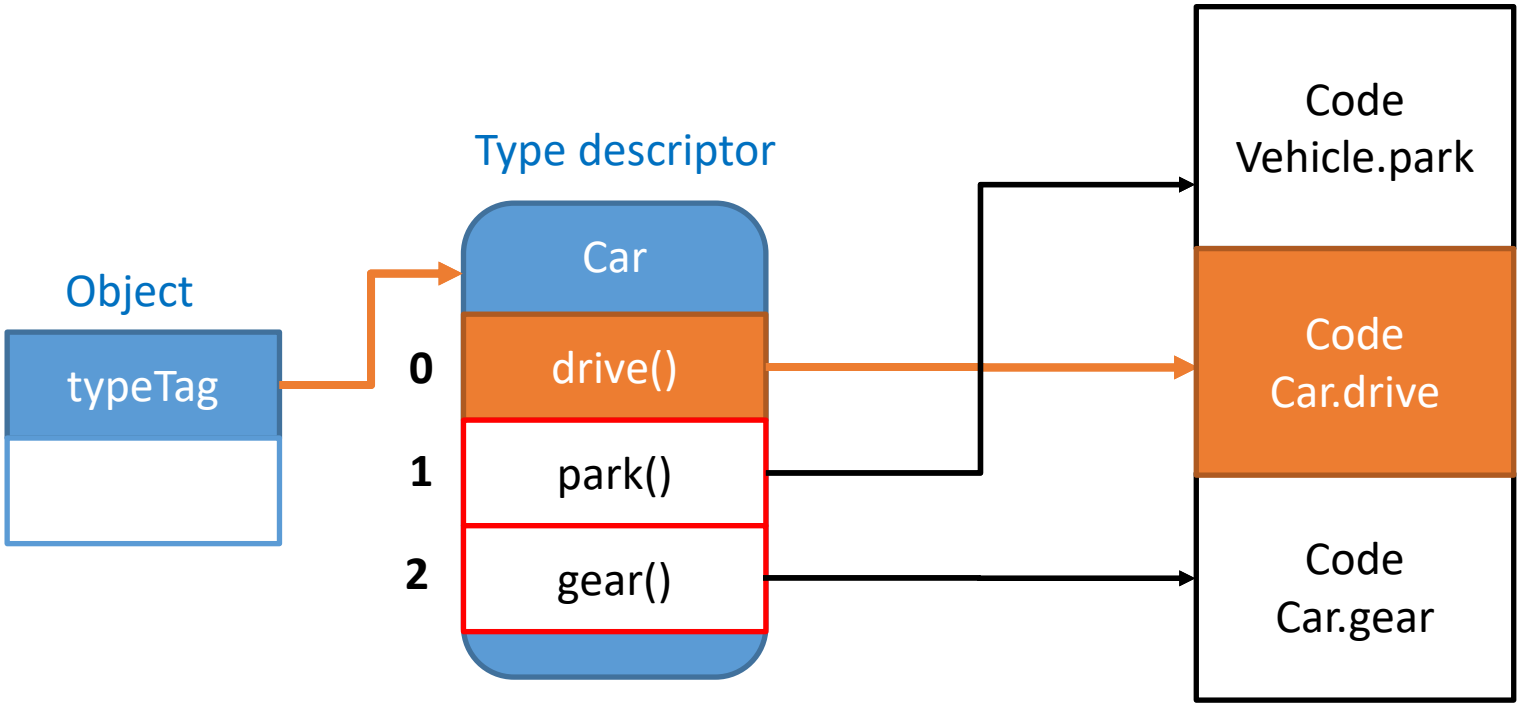


Summary OO Runtime Support

- Heap
  - Linear address space for object allocations
  - Deallocation by garbage collector
  - Free list
- Single inheritance
  - Linear extension of object layout
- Type polymorphism
  - Ancestor table for constant-time type tests/casts
  - Virtual table for constant-time virtual method calls

Example: Virtual Table

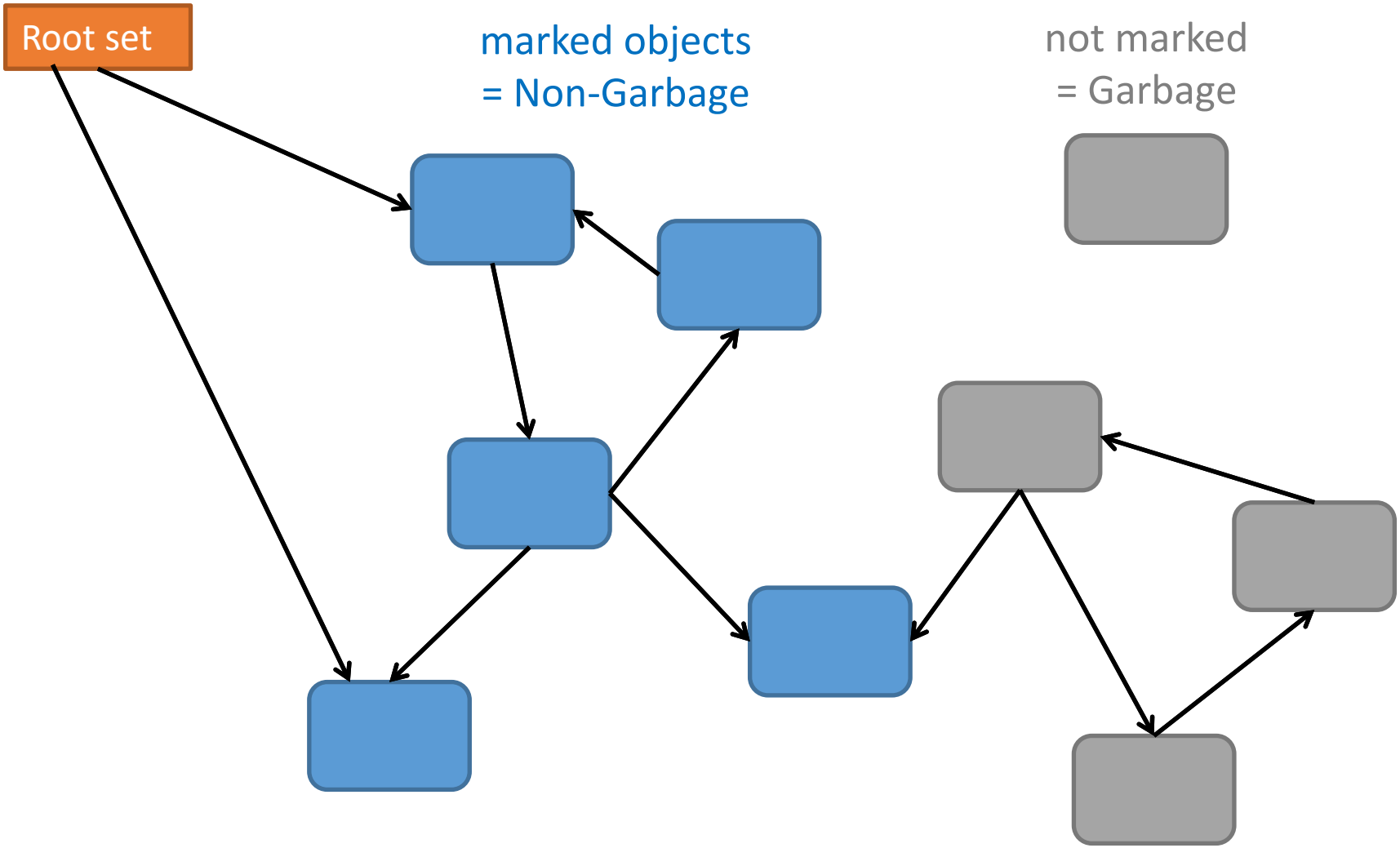
```
v.drive();
```
// method pos 0



Summary Garbage Collection

- For memory safety
  - No dangling pointers
  - No memory leaks
- Automatic delayed memory reclamation
  - Based on reachability from root set
  - Root set: call stack, possibly also statics, registers
- Mark & sweep
  - Transitively mark reachable objects
  - Scan over heap, free unmarked objects

Example: Mark & Sweep GC



Summary JIT-Compilation

- Much faster than interpretation
- Profiling: Detect hot spots
- Compile to native code
  - Template-based code generation
- Run on hardware
  - Code in executable memory page
- Local register allocation
  - Instead of evaluation stack
- Global register allocation
  - Store (frequently accessed) variables in registers

Example: JIT Compilation

```
  load 1
  ldc 0
  icmplt
  br_false label1
  load 1
  ineg
  store 1
label1:
  load 1
  return
```

```
  MOV RAX, 0
  CMP RCX, RAX
  JGE label1
  NEG RCX
label1:
  MOV RAX, RCX
  RET
```

Out-of-Scope Topics

- Advanced code optimizations and analyses
- Advanced language concepts
- Debuggers, multi-assembly loader
- And more…

Advanced Language Concepts

Static concepts (mainly compiler, metadata)

- Overloading
- Generics
- Lambdas/delegates

Dynamic concepts (runtime systems)

- Exception handling
- Concurrency
- Reflection

Conclusions

- We have seen the most important concepts
  - Compiler & runtime system
- Goal: In-depth understanding
  - Theory but also by doing (implementation)

- Of course, there is more to learn about compilers & runtime systems

Review: Overall Learning Goals

- ✓ Understand the fundamental architecture, concepts and techniques of compilers and runtime systems
- ✓ Implement own pieces of a compiler and runtime system for a modern OO language on a state-of-the-art platform
- ✓ Become familiar with syntax and semantic specifications of programming languages