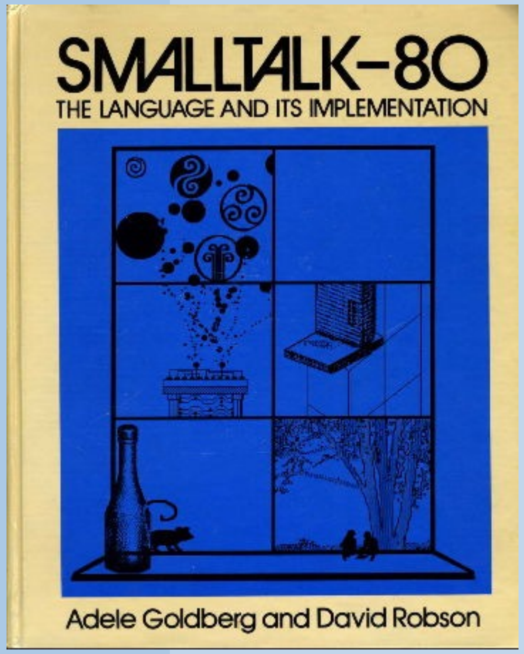
UNIVERSITÄT
BERN

8. Bytecode and Virtual Machines

Oscar Nierstrasz

Original material prepared by
Adrian Lienhard and Marcus Denker



Roadmap

- Introduction
- Bytecode
- The heap
- Interpreter
- Automatic memory management
- Threading System
- Optimizations

References

- *Virtual Machines*, Iain D. Craig, Springer, 2006
- *Back to the Future – The Story of Squeak, A Practical Smalltalk Written in Itself*, Ingalls et al, OOPSLA ’97
- *Smalltalk-80, the Language and Its Implementation* (AKA “the Blue Book”), Goldberg, Robson, Addison-Wesley, ’83
  - http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf
- *The Java Virtual Machine Specification*, Second Edition
  - http://java.sun.com/docs/books/jvms/
- *Stacking them up: a Comparison of Virtual Machines*, Gough, IEEE’01
- *Virtual Machine Showdown: Stack Versus Registers*, Shi, Gregg, Beatty, Ertl, VEE’05

Birds-eye view

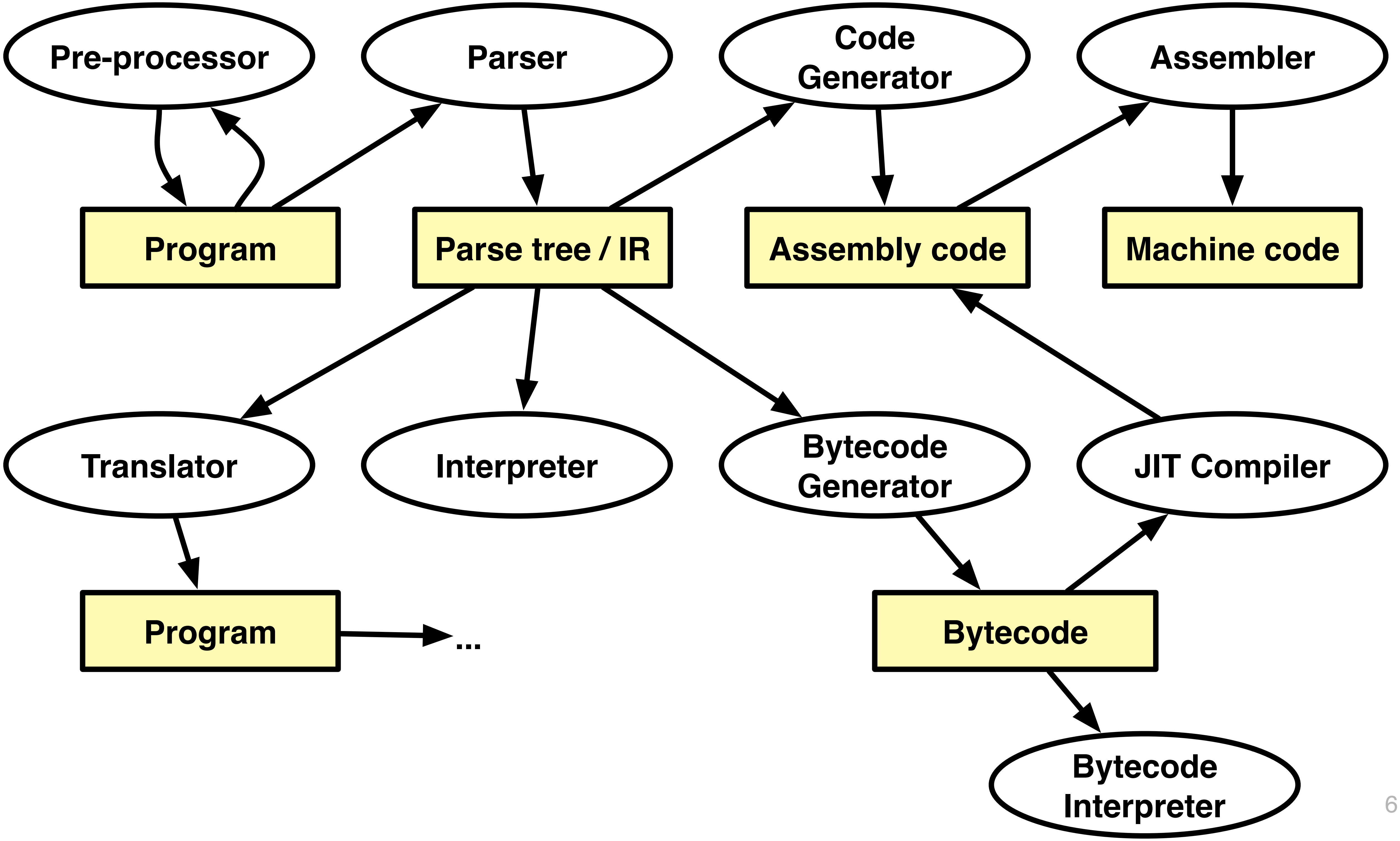
A virtual machine is an abstract computing architecture supporting a programming language in a hardware-independent fashion

Z1, 1938

Roadmap

- **Introduction**
- Bytecode
- The heap
- Interpreter
- Automatic memory management
- Threading System
- Optimizations

Implementing a Programming Language



Pre-processor

Parser

Code Generator

Assembler

Program

Parse tree / IR

Assembly code

Machine code

Translator

Interpreter

Bytecode Generator

JIT Compiler

Program

...

Bytecode

Bytecode Interpreter

How are VMs implemented?

Typically using an *efficient and portable language* such as C, C++, or assembly code

Pharo VM platform-independent part written in *Slang:*

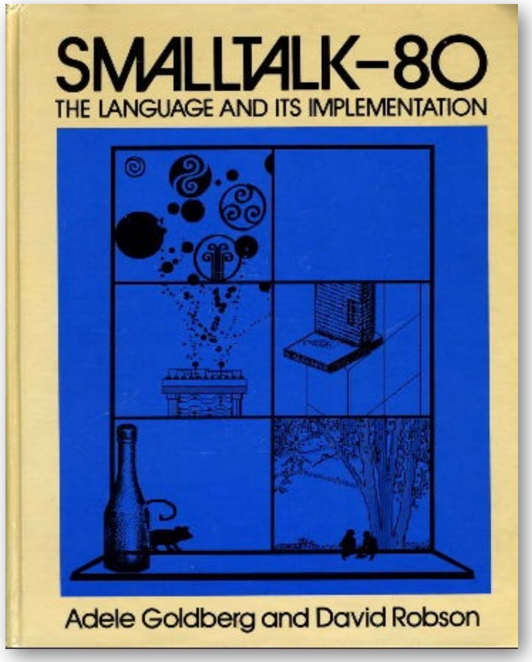
- subset of Smalltalk, translated to C
- core: 600 methods or 8k LOC in Slang
- Slang allows one to simulate VM in Smalltalk

In this lecture we will look at the VM of Pharo Smalltalk, as it is based closely on the original Smalltalk-80 VM. On the one hand it is simpler than the Java VM, and on the other hand it heavily influenced VM technology that followed.

A VM is typically implemented in C. The Pharo VM is written in a subset of Smalltalk that can either be directly interpreted as Smalltalk code (useful for debugging), or translated to C.

Smalltalk

The Smalltalk language and its VM implementation are specified in the “Blue Book”.



In this lecture we won’t attempt to introduce Smalltalk itself, except to say that it is a classical dynamically-typed object-oriented language. The key difference to more recent languages like Python or Ruby, is that it supports *live programming*. Smalltalk programs are always running in a live environment, which is incrementally modified by adding or modifying classes and methods.

Many modern Smalltalk implementations are based on or inspired by the Blue Book.

Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*, Addison Wesley, Reading, Mass., May 1983.

http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf

Main Components of a VM

The interpreter

The threading System

The heap

Automatic memory management

A Virtual Machine must typically include:

- a bytecode interpreter
- a threading system to manage processes of the interpreted language
- heap storage for running programs
- a garbage collector to detect and free unused heap storage

Pros and Cons of the VM Approach

Pros

- Platform independence of application code
  “Write once, run anywhere”
- Simpler programming model
- Security
- Optimizations for different hardware architectures

Cons

- Execution overhead
- Not suitable for system programming

Roadmap

- > Introduction
- > **Bytecode**
- > The heap
- > Interpreter
- > Automatic memory management
- > Threading System
- > Optimizations

The Pharo Virtual Machine

- Virtual machine provides a virtual processor
  - Bytecode: The “machine-code” of the virtual machine

- Smalltalk (like Java): Stack machine
  - easy to implement interpreters for different processors
  - most hardware processors are register machines

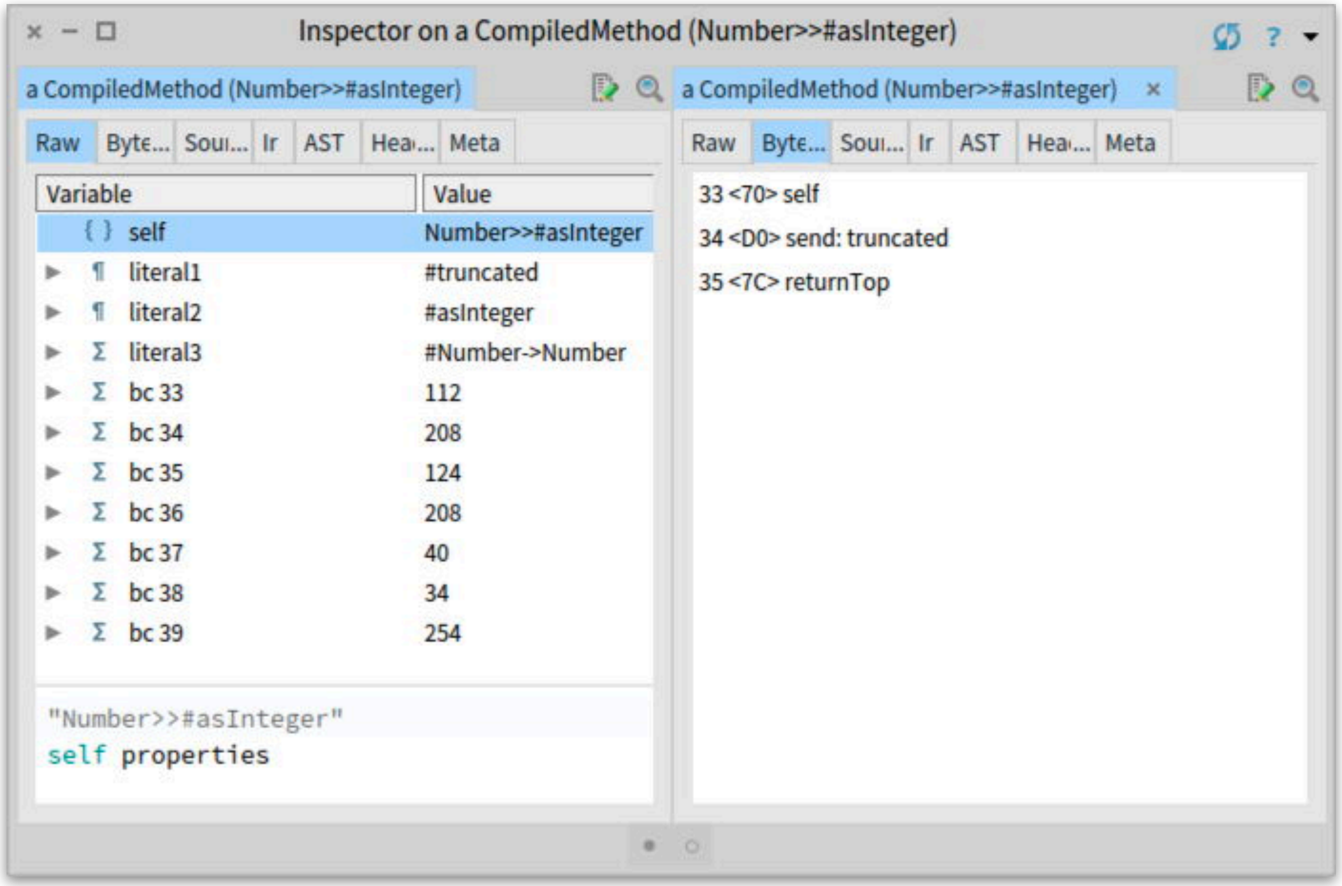
- Pharo VM: Implemented in *Slang*
  - Slang: Subset of Smalltalk. (“C with Smalltalk Syntax”)
  - Translated to C

Bytecode is analogous to assembler, except it targets a virtual machine rather than a physical one. Many VMs are stack machines: the generated bytecode pushes values onto a stack, and executes operations that consume one or more values on the top of the stack, replacing them with results.

Bytecode in the CompiledMethod

- CompiledMethod format:

Header	Number of temps, literals...
Literals	Array of all Literal Objects
Bytecode	
Trailer	Pointer to Source



Variable	Value
self	Number>>#asInteger
literal1	#truncated
literal2	#asInteger
literal3	#Number->Number
bc 33	112
bc 34	208
bc 35	124
bc 36	208
bc 37	40
bc 38	34
bc 39	254

```
"Number>>#asInteger"
self properties
```

```
33 <70> self
34 <D0> send: truncated
35 <7C> returnTop
```

```
(Number>>#asInteger) inspect
```

```
(Number methodDict at: #asInteger) inspect
```

Smalltalk is unusual in that everything is consistently represented as objects down to a very low level, including compiled methods.

We exploit this here to inspect the `asInteger` method of the `Number` class.

A compiled method consists of:

- a header that provides bookkeeping information (number of local variables etc.)
- an array of literal objects (constants) used within the method
- the actual bytecode to be executed
- a trailer that points back to the source code (useful for tools)

The code in the slide asks the `Number` class for its `asInteger` method (which is looked up in its method dictionary), and then sends it the `inspect` method to pop up an object inspector on the actual compiled method.

NB: the last four “bytecodes” are actually the source pointer.

Bytecodes: Single or multibyte

- Different forms of bytecodes:
  - Single bytecodes:
    - *Example: 112: push self*
  - Groups of similar bytecodes
    - *16: push temp 1*
    - *17: push temp 2*
    - *up to 31*
  - Multibyte bytecodes
    - *Problem: 4 bit offset may be too small*
    - *Solution: Use the following byte as offset*
    - *Example: Jumps need to encode large jump offsets*

Type	***Offset***
4 bits	4 bits

Smalltalk bytecodes are encoded in 8 bits, so there are up to 256 of them, but some of these are actually groups of bytecodes in which the 4 bits represent an offset.

Example: Number>>asInteger

- Smalltalk code:

```
Number>>asInteger
   "Answer an Integer nearest
   the receiver toward zero."

   ^self truncated
```

- Symbolic Bytecode

```
17 <70> self
18 <D0> send: truncated
19 <7C> returnTop
```

In Smalltalk code, the asInteger method of a number just sends the message “`truncated`” to `self` (AKA “`this`”), and returns the result.

The corresponding bytecode:

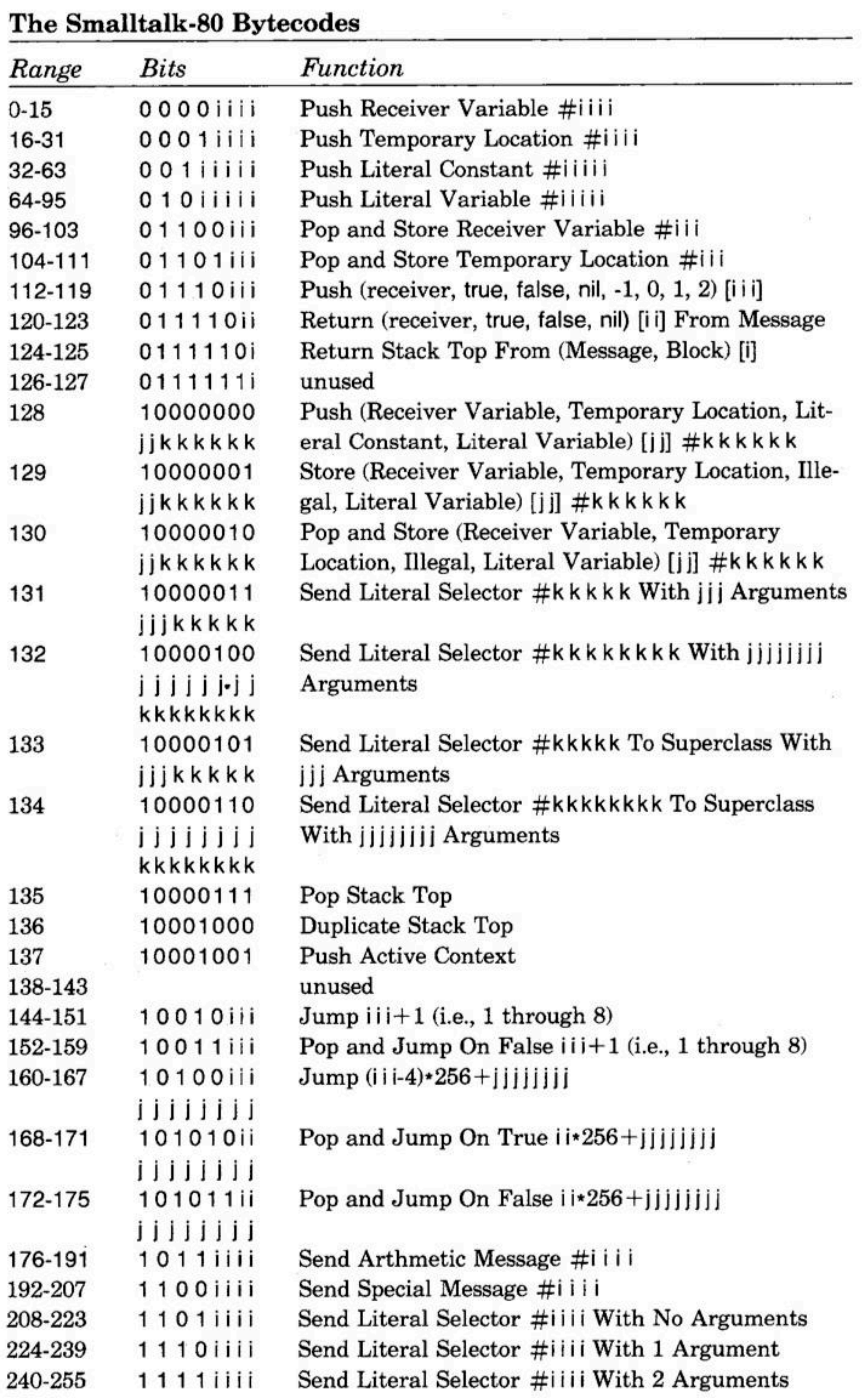
1. pushes `self` onto the stack
2. sends the message “`truncated`” (stored as a literal in the compiled method) — this causes self to be popped and the result of the `truncated` method to be left on the stack
3. (pops and) returns the top of the stack

Example: Step by Step

- `17 <70> self`
  - Byte code 112: the receiver (self) is pushed on the stack
- `18 <D0> send: truncated`
  - Bytecode 208: send literal selector 1
  - Get the selector from the first literal
  - start message lookup in the class of the object that is on top of the stack
  - result is pushed on the stack
- `19 <7C> returnTop`
  - Byte code 124: return the object on top of the stack to the calling method

Pharo Bytecode

- 256 Bytecodes, four groups:
  - Stack Bytecodes
    - *Stack manipulation: push / pop / dup*
  - Send Bytecodes
    - *Invoke Methods*
  - Return Bytecodes
    - *Return to caller*
  - Jump Bytecodes
    - *Control flow inside a method*



The Smalltalk-80 Bytecodes

Range	*Bits*	*Function*
0-15	0000iiii	Push Receiver Variable #iiii
16-31	0001iiii	Push Temporary Location #iiii
32-63	001iiiii	Push Literal Constant #iiiii
64-95	010iiiii	Push Literal Variable #iiiii
96-103	01100iii	Pop and Store Receiver Variable #iii
104-111	01101iii	Pop and Store Temporary Location #iii
112-119	01110iii	Push (receiver, true, false, nil, -1, 0, 1, 2) [iii]
120-123	011110ii	Return (receiver, true, false, nil) [ii] From Message
124-125	0111110i	Return Stack Top From (Message, Block) [i]
126-127	0111111i	unused
128	10000000 jjkkkkkk	Push (Receiver Variable, Temporary Location, Literal Constant, Literal Variable) [jj] #kkkkkk
129	10000001 jjkkkkkk	Store (Receiver Variable, Temporary Location, Illegal, Literal Variable) [jj] #kkkkkk
130	10000010 jjkkkkkk	Pop and Store (Receiver Variable, Temporary Location, Illegal, Literal Variable) [jj] #kkkkkk
131	10000011 jjjkkkkk	Send Literal Selector #kkkkk With jjj Arguments
132	10000100 jjjjjjjj kkkkkkkk	Send Literal Selector #kkkkkkkk With jjjjjjjj Arguments
133	10000101 jjjkkkkk	Send Literal Selector #kkkkk To Superclass With jjj Arguments
134	10000110 jjjjjjjj kkkkkkkk	Send Literal Selector #kkkkkkkk To Superclass With jjjjjjjj Arguments
135	10000111	Pop Stack Top
136	10001000	Duplicate Stack Top
137	10001001	Push Active Context
138-143		unused
144-151	10010iii	Jump iii+1 (i.e., 1 through 8)
152-159	10011iii	Pop and Jump On False iii+1 (i.e., 1 through 8)
160-167	10100iii jjjjjjjj	Jump (iii-4)*256+jjjjjjjj
168-171	101010ii jjjjjjjj	Pop and Jump On True ii*256+jjjjjjjj
172-175	101011ii jjjjjjjj	Pop and Jump On False ii*256+jjjjjjjj
176-191	1011iiii	Send Arthmetic Message #iiii
192-207	1100iiii	Send Special Message #iiii
208-223	1101iiii	Send Literal Selector #iiii With No Arguments
224-239	1110iiii	Send Literal Selector #iiii With 1 Argument
240-255	1111iiii	Send Literal Selector #iiii With 2 Arguments

The table is from page 596 of the “Blue book.”

Stack Bytecodes

- Push values on the stack
  - e.g., temps, instVars, literals
  - e.g: 16 - 31: push instance variable
- Push Constants
  - False/True/Nil/1/0/2/-1
- Push `self`, `thisContext`
- Duplicate top of stack
- Pop

Sends and Returns

- Sends: receiver is on top of stack
  - Normal send
  - Super Sends
  - Hard-coded sends for efficiency, e.g. `+`, `-`
- Returns
  - Return top of stack to the sender
  - Return from a block
  - Special bytecodes for return `self`, `nil`, `true`, `false` (for efficiency)

Jump Bytecodes

- Control Flow inside one method
  - Used to implement control-flow efficiently
  - Example:

```
^ 1<2 ifTrue: ['true']
```

```
17 <76> pushConstant: 1
18 <77> pushConstant: 2
19 <B2> send: <
20 <99> jumpFalse: 23
21 <20> pushConstant: 'true'
22 <90> jumpTo: 24
23 <73> pushConstant: nil
24 <7C> returnTop
```

The example Smalltalk code sends “<2” to 1, resulting in a Boolean object. This object is sent the keyword message `ifTrue:` with a block as an argument. Only if the result is true, will the block (in square brackets) be evaluated to the string `'true'`. The result is returned (“`^`” symbol in Smalltalk).

The bytecode achieves this by:

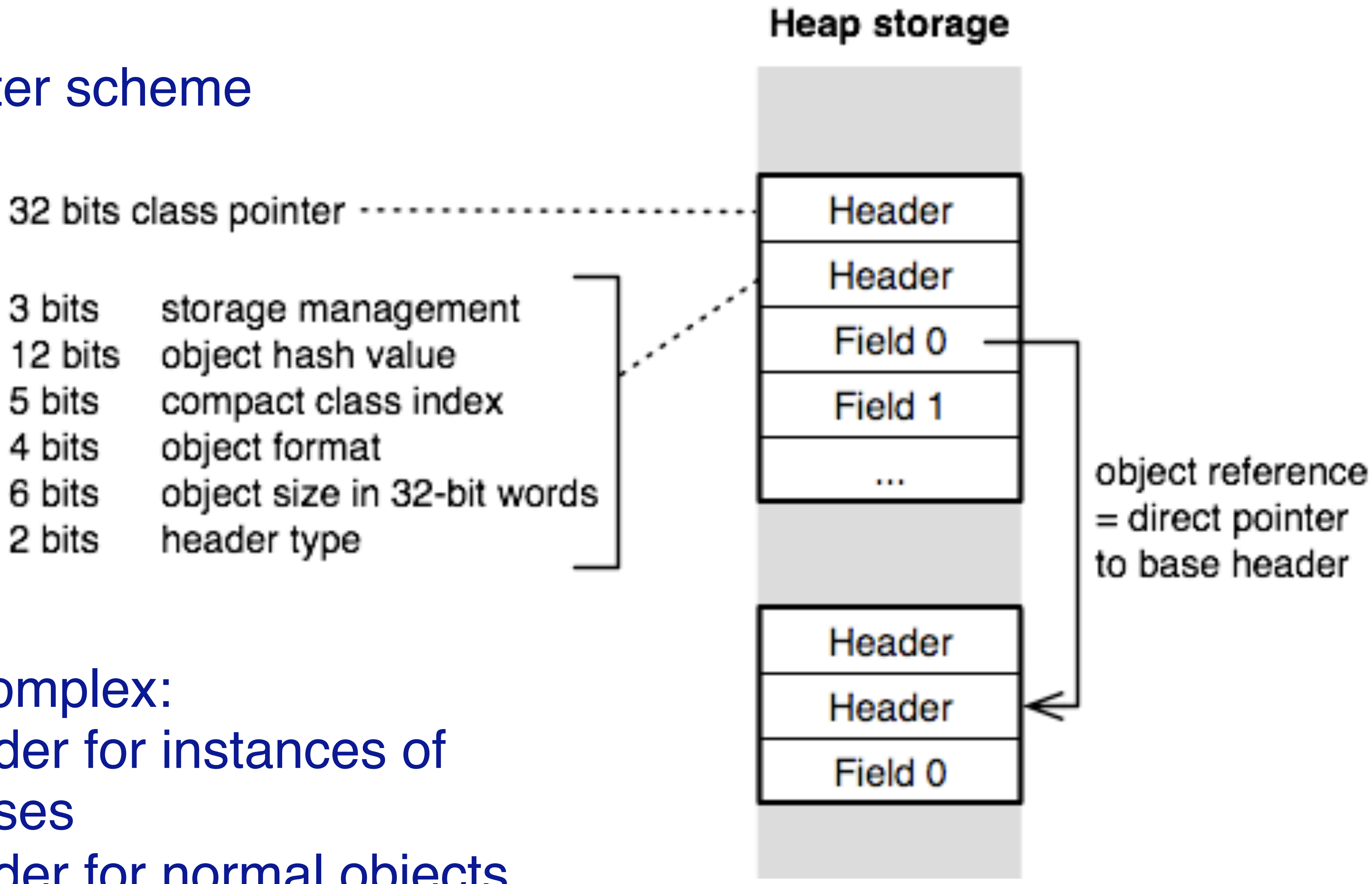
- pushing 1 and 2 onto the stack
- sending the message <, thus consuming these values and leaving a Boolean on top
- the ifTrue: “method” is inlined as a jump, causing either the 'true' string or nil to be pushed and returned

Roadmap

- Introduction
- Bytecode
- **The heap**
- Interpreter
- Automatic memory management
- Threading System
- Optimizations

Object Memory Layout

32-bit direct-pointer scheme



Heap storage

32 bits class pointer

3 bits	storage management
12 bits	object hash value
5 bits	compact class index
4 bits	object format
6 bits	object size in 32-bit words
2 bits	header type

object reference = direct pointer to base header

Reality is more complex:

- 1-word header for instances of compact classes
- 2-word header for normal objects
- 3-word header for large objects

Objects in Smalltalk reside in heap storage. (A snapshot of the heap can be saved as an “*image*” file that can be restarted at a later time.)

Until very recently, objects were identified by 32-bit direct pointers (as of 2016, 64-bit pointers are supported).

The storage for a regular object consist of two headers and a series of fields. The first header identifies the class of the object (classes are also objects, so this is just an object pointer). The second header contains bookkeeping information. Each of the fields is an object pointer.

Different Object Formats

- fixed pointer fields
- indexable types:
  - indexable pointer fields (e.g., Array)
  - indexable weak pointer fields (e.g., WeakArray)
  - indexable word fields (e.g., Bitmap)
  - indexable byte fields (e.g., ByteString)

Object format (4bit)

0	no fields
1	fixed fields only
2	indexable pointer fields only
3	both fixed and indexable pointer fields
4	both fixed and indexable weak fields
6	indexable word fields only
8-11	indexable byte fields only
12-15	...

In practice, there are several different kinds of object formats are supported.

The first bit of an object pointer indicates if the object is a `SmallInteger` (the remaining bits encode the number). Otherwise it is a “regular” object.

Regular objects might have named fields, or possibly indexed fields (i.e., Array-like objects).

Iterating Over All Objects in Memory

```
"Answer the first object on the heap"
anObject someObject

"Answer the next object on the heap"
anObject nextObject
```

Excludes small integers!

```
SystemNavigation>>allObjectsDo: aBlock
   | object endMarker |
   object := self someObject.
   endMarker := Object new.
   [endMarker == object]
      whileFalse: [aBlock value: object.
         object := object nextObject]
```

```
|count|
count := 0.
SystemNavigation default allObjectsDo:
   [:anObject | count := count + 1].
count
```

```
529468
```

Roadmap

- > Introduction
- > Bytecode
- > The heap
- > **Interpreter**
- > Automatic memory management
- > Threading System
- > Optimizations

Stack vs. Register VMs

The VM provides a virtual processor that interprets bytecode instructions

Stack machines

- Smalltalk, Java and most other VMs
- Simple to implement for different hardware architectures
- Very compact code

Register machines

- Potentially faster than stack machines
- Only a few register VMs exist, e.g., Parrot VM (Perl6)

Interpreter State and Loop

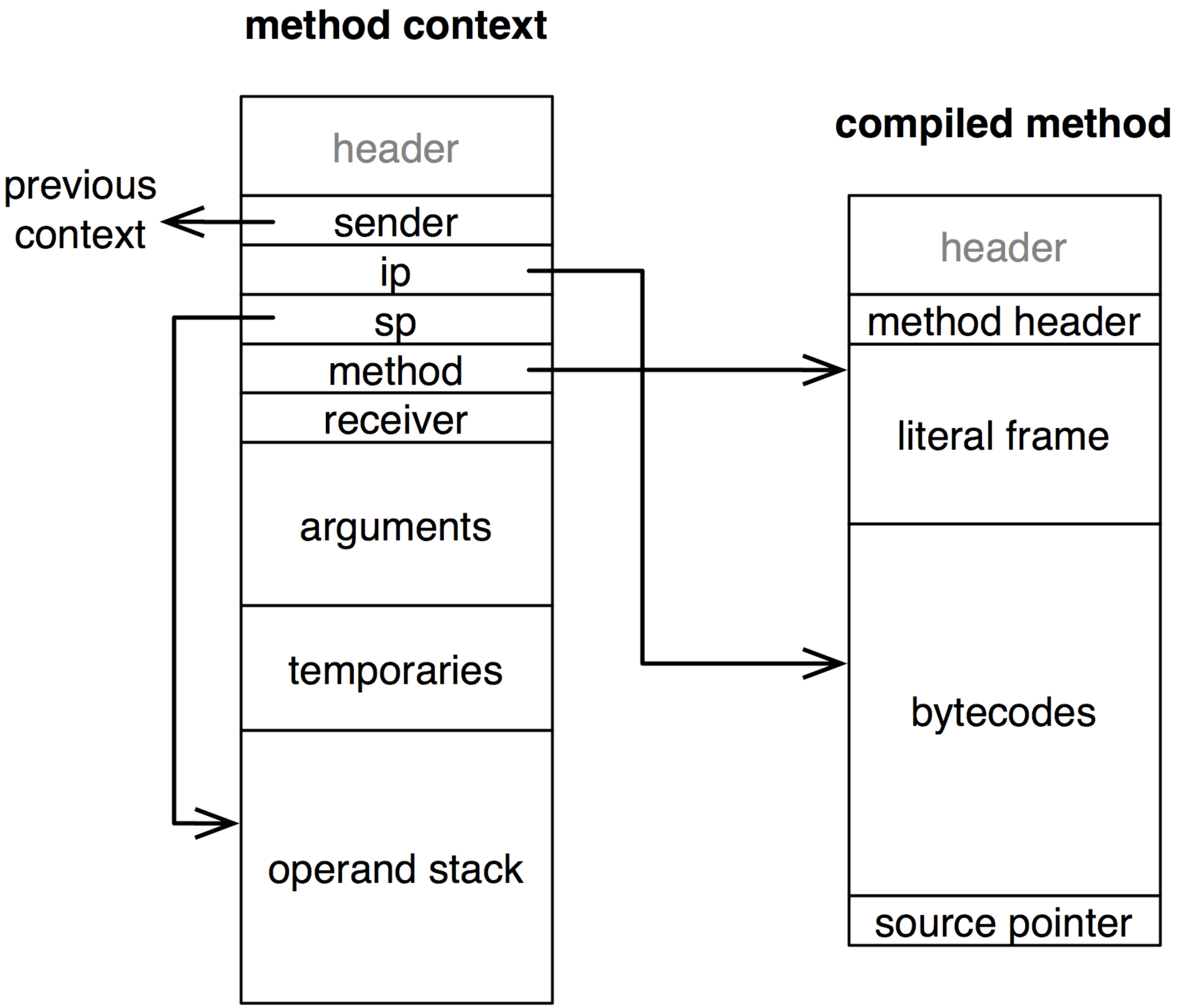
Interpreter state

- instruction pointer (ip): points to current bytecode
- stack pointer (sp): topmost item in the operand stack
- current active method or block context
- current active receiver and method

Interpreter loop

1. branch to appropriate bytecode routine
2. fetch next bytecode
3. increment instruction pointer
4. execute the bytecode routine
5. return to 1.

Method Contexts



method header:
- primitive index
- number of args
- number of temps
- large context flag
- number of literals

A method context is an object that represents an activation record (stack frame) for a method invocation. Each method context points to the context of its caller, thus constituting a stack of contexts.

Note how the ip points to a bytecode in the compiled method, while the sp points to a location in the operand stack.

Stack Manipulating Bytecode Routine

Example: bytecode `<70> self`

```
Interpreter>>pushReceiverBytecode
  self fetchNextBytecode.
  self push: receiver

Interpreter>>push: anObject
  sp := sp + BytesPerWord.
  self longAt: sp put: anObject
```

This bytecode pushes `self` on the stack. Note that `self` refers to the receiver of the message, while `self` in the `pushReceiverBytecode` method refers to the `Interpreter` instance. (We want to push the receiver, not the bytecode interpreter!)

The `push:` method increments the sp by enough bytes to hold an object pointer (i.e., one word), and then writes that word.

Stack Manipulating Bytecode Routine

Example: bytecode `<01> pushRcvr: 1`

```
Interpreter>>pushReceiverVariableBytecode
  self fetchNextBytecode.
  self pushReceiverVariable: (currentBytecode bitAnd: 16rF)

Interpreter>>pushReceiverVariable: fieldIndex
  self push: (self fetchPointer: fieldIndex ofObject: receiver)

Interpreter>>fetchPointer: fieldIndex ofObject: oop
  ^ self longAt: oop + BaseHeaderSize + (fieldIndex * BytesPerWord)
```

The first 16 bytecodes are all implemented by the same method, `pushReceiverVariableBytecode`. The bottom 4 bits of the bytecode encode which of the first 16 instance variables to push. (The `bitAnd:` method will extract these bits.)

We then fetch the instance variable by computing the offset in memory starting after the receiver's header.

See the Blue book p 598.

Message Sending Bytecode Routine

Example: bytecode `<E0> send: hello`

1. find selector, receiver and its class
2. lookup message in the method dictionary of the class
3. if method not found, repeat this lookup in successive superclasses; if superclass is nil, instead send #doesNotUnderstand:
4. create a new method context and set it up
5. activate the context and start executing the instructions in the new method

Message Sending Bytecode Routine

Example: bytecode `<E0> send: hello`

```
Interpreter>>sendLiteralSelectorBytecode
  selector := self literal: (currentBytecode bitAnd: 16rF).
  argumentCount := ((currentBytecode >> 4) bitAnd: 3) - 1.
  rcvr := self stackValue: argumentCount.
  class := self fetchClassOf: rcvr.
  self findNewMethod.
  self executeNewMethod.
  self fetchNewBytecode
```

This routine (bytecodes 208-255) can use any of the first 16 literals and pass up to 2 arguments

```
E0(hex) = 224(dec)
   = 1110 0000(bin)

E0 AND F = 0
   => literal frame at 0

((E0 >> 4) AND 3) - 1 = 1
   => 1 argument
```

Here too we have 48 bytecodes with the same implementation. The bottom 4 bits encode which literal (stored in the compiled method's literal frame) to send. An additional 2 bits encode 0, 1 or 2 arguments to pass.

Primitives

Primitive methods trigger a VM routine and are executed without a new method context unless they fail

```
ProtoObject>>nextObject
  <primitive: 139>
  self primitiveFailed
```

- Improve performance (arithmetics, at:, at:put:, ...)
- Do work that can only be done in VM (new object creation, process manipulation, become, ...)
- Interface with outside world (keyboard input, networking, ...)
- Interact with VM plugins (named primitives)

The bodies of primitive methods start with a “pragma” (in angle brackets) indicating the VM method to invoke. If the primitive fails, the code following the pragma will be executed.

Roadmap

- Introduction
- Bytecode
- The heap
- Interpreter
- **Automatic memory management**
- Threading System
- Optimizations

Automatic Memory Management

Tell when an object is no longer used and then recycle the memory

Challenges

- Fast allocation
- Fast program execution
- Small predictable pauses
- Scalable to large heaps
- Minimal space usage

Instead of requiring application developers to specify when memory allocated to objects can be safely recycled, automatic memory management makes use of an additional “garbage collection” process that periodically checks which objects are no longer accessible and recycles them automatically.

Garbage collection was first conceived by John McCarthy in 1959 for Lisp.

https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

Main Approaches

- 1. Reference Counting
- 2. Mark and Sweep

There are two basic approaches to garbage collection.

Reference counting requires that a count be maintained for each object of all the references pointing to it. Reference counts must be updated every time a reference is created or dropped. Objects that are no longer referenced can be recycled.

Mark and sweep, on the other hand, requires a full pass over memory. In the “mark” phase, objects that are referenced at least once are “marked”. Any object not marked is not referenced and is recycled in a second “sweep” phase.

Reference Counting GC

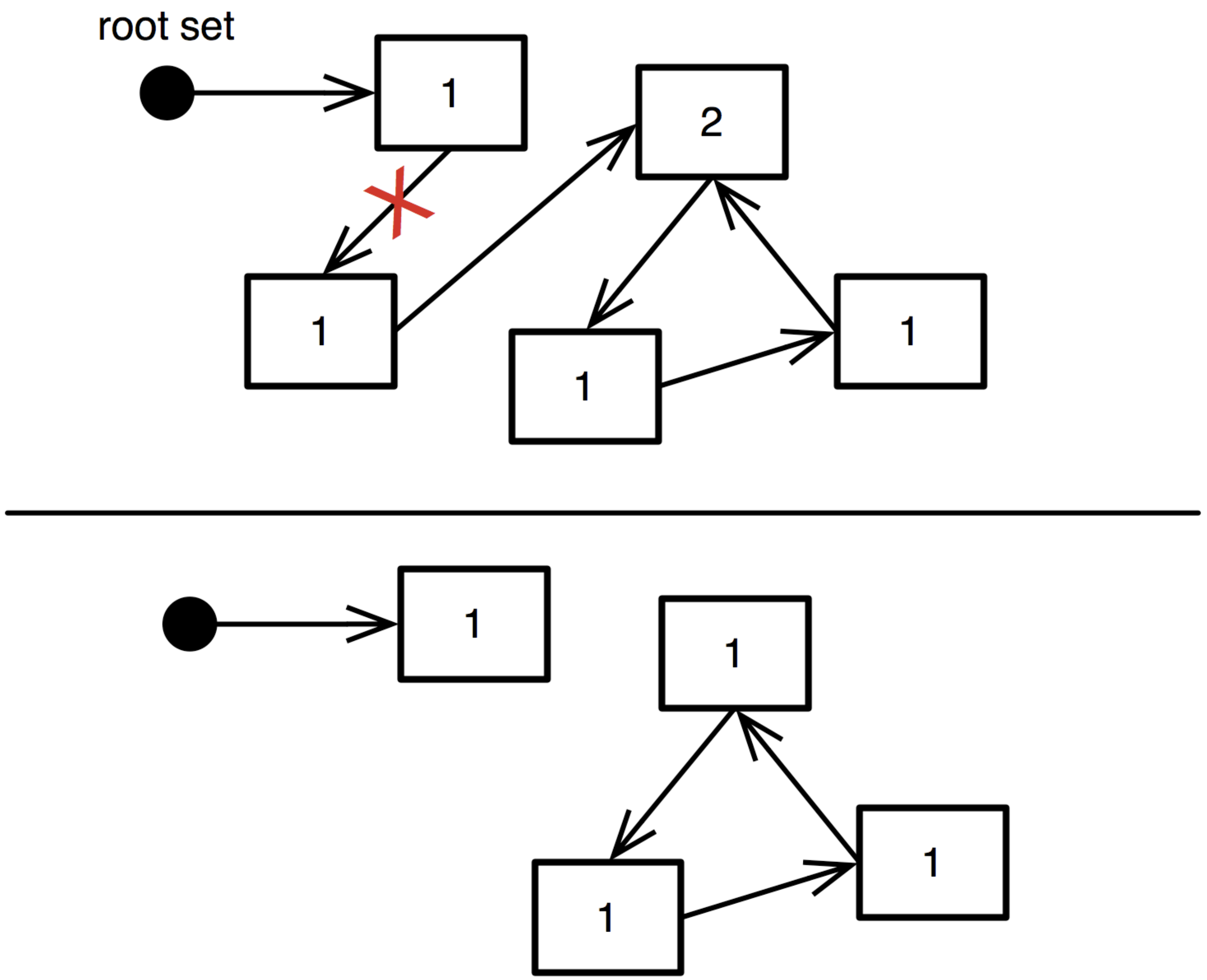
Idea

- For each store operation increment count field in header of newly stored object
- Decrement if object is overwritten
- If count is 0, collect object and decrement the counter of each object it pointed to

Problems

- Run-time overhead of counting (particularly on stack)
- Inability to detect cycles (need additional GC technique)

Reference Counting GC



With naive reference counting, it is possible for disconnected cycles of objects to be left undetected as “garbage”.

Mark and Sweep GC

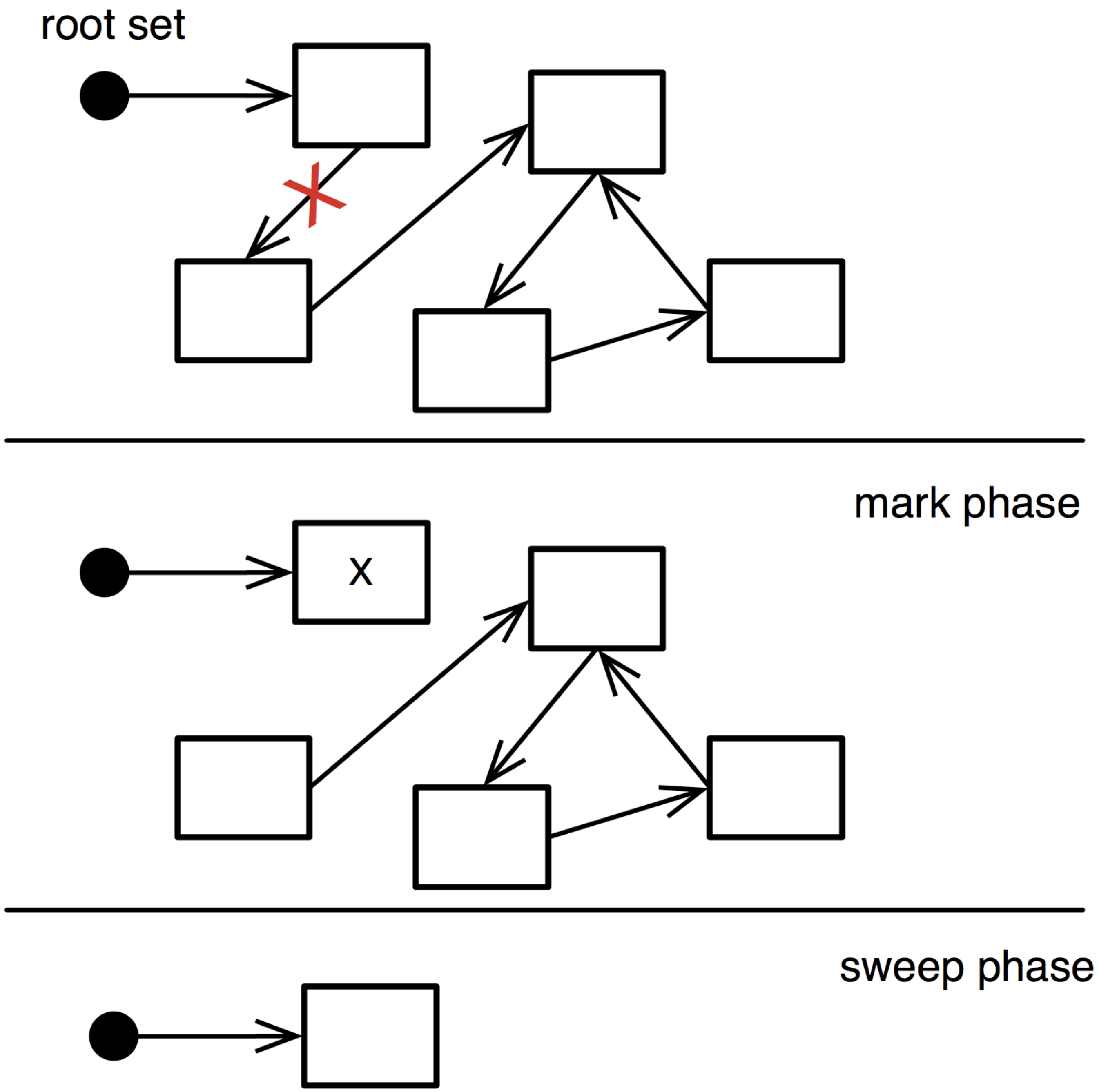
Idea

- Suspend current process
- Mark phase: trace each accessible object leaving a mark in the object header (start at known root objects)
- Sweep phase: all objects with no mark are collected
- Remove all marks and resume current process

Problems

- Need to “stop the world”
- Slow for large heaps ➔generational collectors
- Fragmentation ➔compacting collectors

Mark and Sweep GC



Generational Collectors

Most new objects live very short lives;
most older objects live forever [Ungar 87]

Idea

- Partition objects into generations
- Create objects in young generation
- Tenuring: move live objects from young to old generation
- Incremental GC: frequently collect young generation (very fast)
- Full GC: infrequently collect young+old generation (slow)

Difficulty

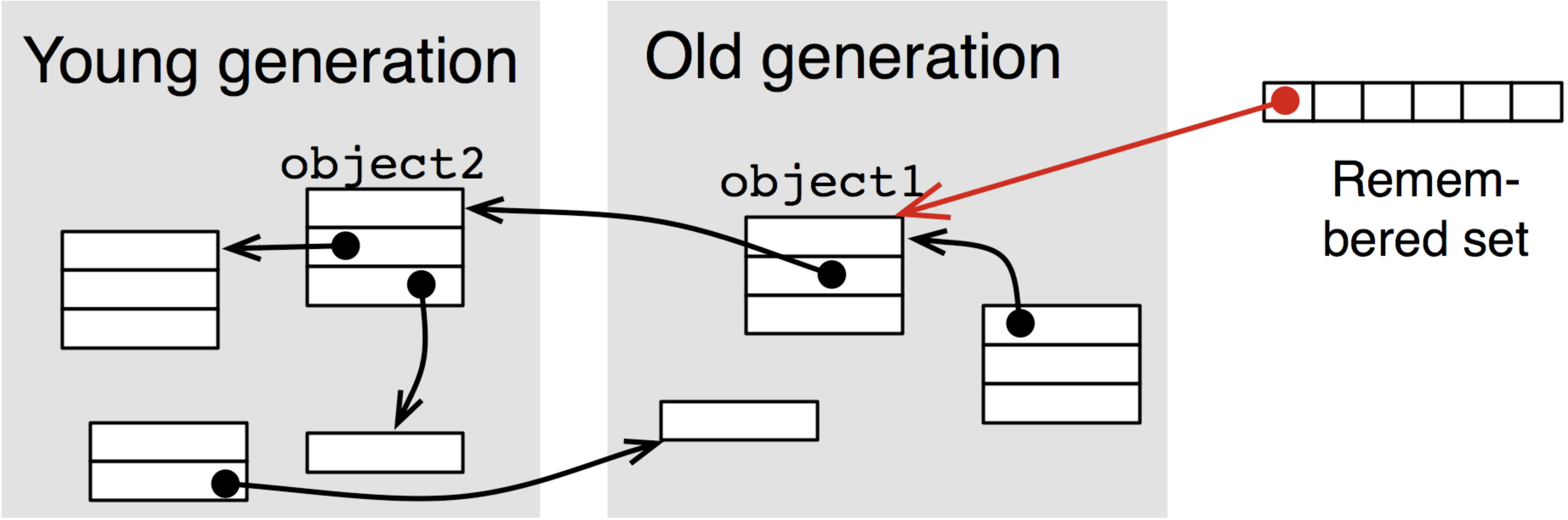
- Need to track pointers from old to new space

Generational Collectors: Remembered Set

Write barrier: remember objects with old-young pointers:

- On each store check whether stored object (object2) is young and storer (object1) is old
- If true, add storer to remembered set
- When marking young generation, use objects in remembered set as additional roots

```
object1.f := object2
```



Compacting Collectors

Idea

- During the sweep phase all live objects are packed to the beginning of the heap
- Simplifies allocation since free space is in one contiguous block

Challenge

- Adjust all pointers of moved objects
  - object references on the heap
  - pointer variables of the interpreter!

The Pharo GC

Pharo: mark and sweep compacting collector with two generations

- Cooperative, i.e., not concurrent
- Single threaded

When Does the GC Run?

- Incremental GC on allocation count or memory needs
- Full GC on memory needs
- Tenure objects if survivor threshold exceeded

VM Memory Statistics

```
Smalltalk vm statisticsReport
```

```
uptime                 0h9m47s
memory                 103,424,000 bytes
          old          94,471,968 bytes (91.30000000000001%)
          young        771,840 bytes (0.7000000000000001%)
          used         69,751,384 bytes (67.4%)
          free         25,492,424 bytes (24.6%)
GCs                    688 (854ms between GCs)
          full         1 totalling 69ms (0.0% uptime), avg 69.0ms
          incr         687 totalling 264ms (0.0% uptime), avg 0.4ms
          tenures      153,132 (avg 0 GCs/tenure)
```

Memory System API

```
“Force GC”
Smalltalk garbageCollectMost.
Smalltalk garbageCollect.

“Is object young?”
Smalltalk isYoung: anObject.

“Various settings and statistics”
Smalltalk vm getParameters.

”Grow/shrink headroom”
Smalltalk vm parameterAt: 25 put: 4*1024*1024.
Smalltalk vm parameterAt: 24 put: 8*1024*1024.
```

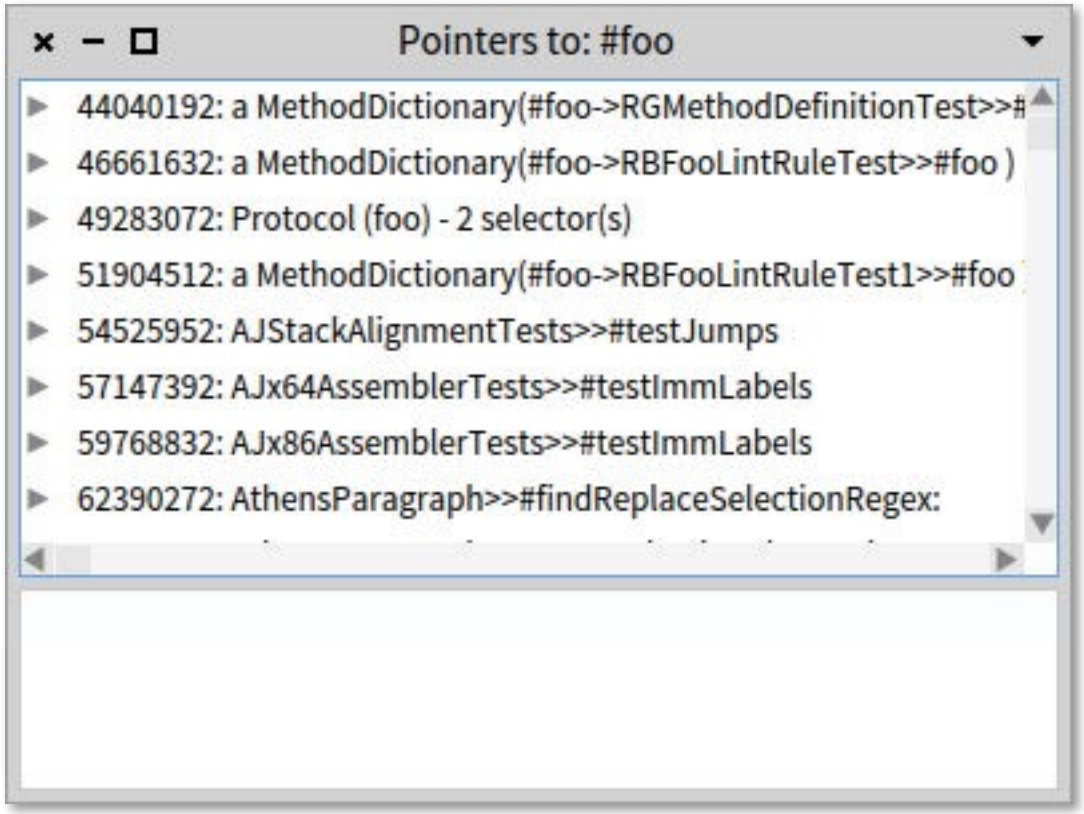
Finding Memory Leaks

I have objects that do not get collected. What's wrong?

- maybe object is just not GCed *yet* (force a full GC!)
- find the objects and then explore who references them

The pointer finder finds a path from a root to some object

```
EyePointerExplorer openOn: #foo
```



```
44040192: a MethodDictionary(#foo->RGMethodDefinitionTest>>#
46661632: a MethodDictionary(#foo->RBFooLintRuleTest>>#foo )
49283072: Protocol (foo) - 2 selector(s)
51904512: a MethodDictionary(#foo->RBFooLintRuleTest1>>#foo
54525952: AJStackAlignmentTests>>#testJumps
57147392: AJx64AssemblerTests>>#testImmLabels
59768832: AJx86AssemblerTests>>#testImmLabels
62390272: AthensParagraph>>#findReplaceSelectionRegex:
```

Roadmap

> Introduction
> Bytecode
> The heap
> Interpreter
> Automatic memory management
> **Threading System**
> Optimizations

Threading System

Multithreading is the ability to create concurrently running “processes”

Non-native threads (*green threads*)

- Only one native thread used by the VM
- Simpler to implement and easier to port

Native threads

- Using the native thread system provided by the OS
- Potentially higher performance

Pharo: Green Threads

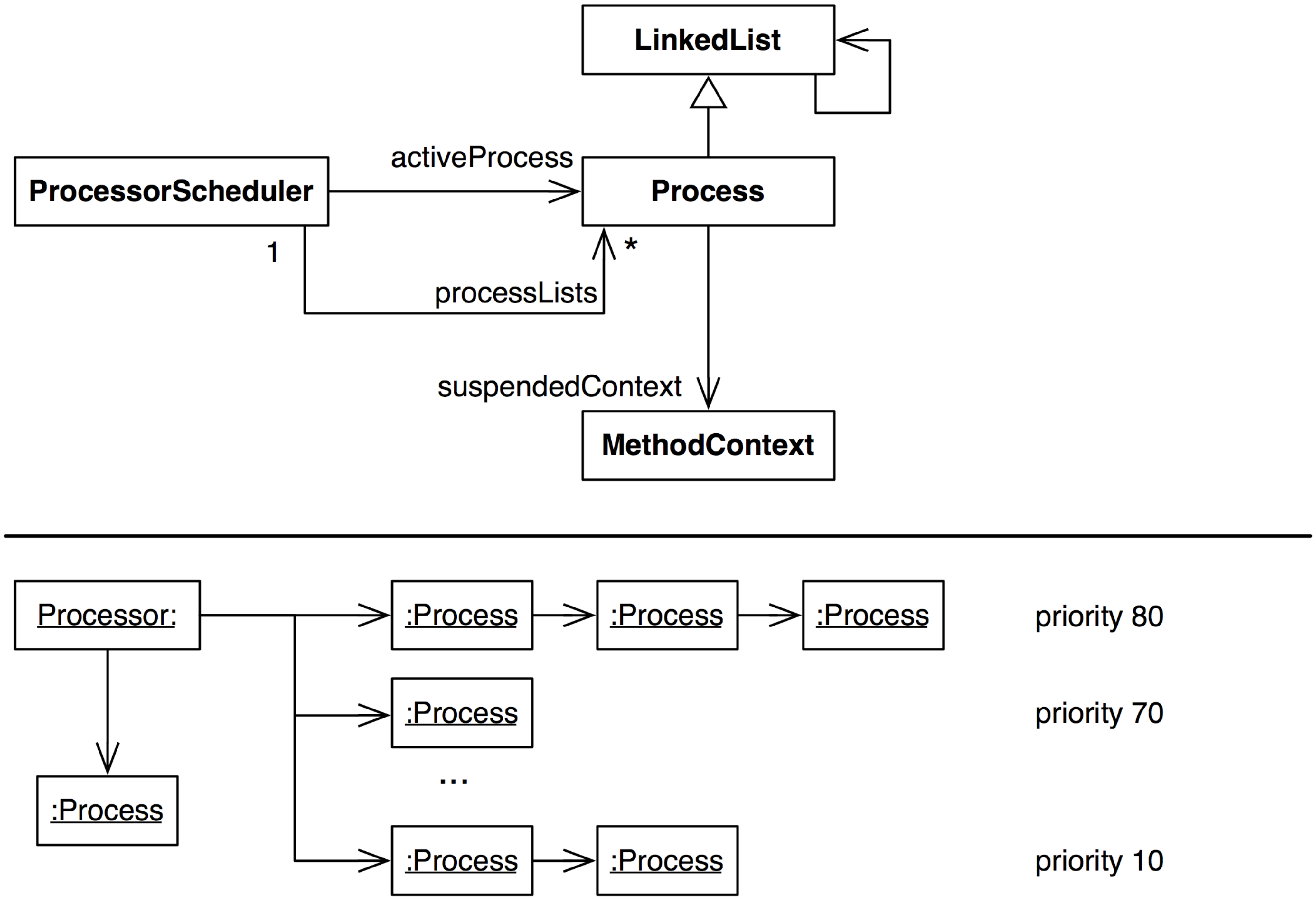
Each process has its own execution stack, ip, sp, ...

There is always one (and only one) running process

Each process behaves as if it owns the entire VM

Each process can be interrupted (➔context switching)

Representing Processes and Run Queues



Context Switching

`Interpreter>>transferTo: newProcess`

1. store the current ip and sp registers to the current context
2. store the current context in the old process’ `suspendedContext`
3. change `Processor` to point to `newProcess`
4. load ip and sp registers from new process’ `suspendedContext`

When you perform a context switch, which process should run next?

Process Scheduler

- *Cooperative* between processes of the same priority
- *Preemptive* between processes of different priorities

Context is switched to the first process with highest priority when:
  - current process waits on a semaphore
  - current process is suspended or terminated
  - `Processor yield` is sent

Context is switched if the following process has a higher priority:
  - process is resumed or created by another process
  - process is resumed from a signaled semaphore

When a process is interrupted, it moves to the back of its run queue

Example: Semaphores and Scheduling

```
here := false.
lock := Semaphore forMutualExclusion.
[lock critical: [here := true]] fork.
lock critical: [
  self assert: here not.
  Processor yield.
  self assert: here not].
Processor yield.
self assert: here
```

When is the forked process activated?

Note that context is only switched when the currently running process voluntary yields. The forked process is at the same priority, but can only execute if it obtains exclusive access to the shared semaphore.

When exactly will the forked process run?

Will either (or both) of these processes terminate?

Roadmap

- > Introduction
- > Bytecode
- > The heap
- > Interpreter
- > Automatic memory management
- > Threading System
- > **Optimizations**

Many Optimizations … (regular VM)

- *Method cache* for faster lookup: receiver’s class + method selector
- *Method context cache* (as much as 80% of objects created are context objects!)

- *Interpreter loop*: 256 way case statement to dispatch bytecodes

- *Quick returns*: methods that simply return a variable or known constant are compiled as a primitive method

- Small integers are *tagged pointers*: value is directly encoded in field references. Pointer is tagged with low-order bit equal to 1. The remaining 31 bits encode the signed integer value.
- ...

A method cache remembers the last method looked up for a given receiver class and method selector. If we see the same class and selector then we don't need to perform the same expensive lookup again.

A method context cache recycles the memory used for method contexts (since they are real objects in the heap instead of on a run-time stack).

Optimization: JIT

Idea: *Just In Time Compilation*

- Translate unit (method, loop, ...) into *native machine code* at runtime
- Store native code in a buffer on the heap

Challenges

- Run-time overhead of compilation
- Machine code takes a lot of space (4-8x compared to bytecode)
- Deoptimization (for debugging) is very tricky

Adaptive compilation: gather statistics to compile only units that are heavily used (*hot spots* — not in Pharo)

What you should know!

- ✎ What is the difference between the operand stack and the execution stack?
- ✎ How do bytecode routines and primitives differ?
- ✎ Why is the object format encoded in a complicated 4bit pattern instead of using regular boolean values?
- ✎ Why is the object address not suitable as a hash value?
- ✎ What happens if an object is only weakly referenced?
- ✎ Why is it hard to build a concurrent mark sweep GC?
- ✎ What does *cooperative multithreading* mean?
- ✎ How do you protect code from concurrent execution?

Can you answer these questions?

- ✎ There is a lot of similarity between VM and OS design. What are the common components?
- ✎ Why is accessing the 16th instance variable of an object more efficient than the 17th?
- ✎ Which disastrous situation could occur if a local C pointer variable exists when a new object is allocated?
- ✎ Why does #allObjectsDo: not include small integers?
- ✎ What is the largest possible small integer?