

# NativeBoost

## today and tomorrow

‘Igor Stasenko’

at: ‘Smalltalks 2012 conference’

location: ‘Puerto Madryn, Argentina’

- Abstract
- Philosophy
- What we have today
- Where we heading

# I trying to DRY

- Google for:

03-24-ESUG-NativeBoost.pdf

- or just: 'NativeBoost pdf'

# What is NativeBoost?

- A framework for **using** native code:
  - generating
  - executing
  - managing

# To make it clear:

`(NativeBoost == FFI) == false`

`(NativeBoost includes: FFI) == true`

# NB is infrastructure

- it is not an “end-user” ready-made solution.
- you build on top of it

# A philosophy

- No barriers
- No magic
- **ALL** interesting stuff should happen at language side (and better it has to be Smalltalk)

Turtles all the way down



# Turtles all the way down

But wait ... whatIsAPrimitive ?

## whatIsAPrimitive

"Some messages in the system are responded to primitively. A primitive response is performed directly by the interpreter rather than by evaluating expressions in a method. The methods for these messages indicate the presence of a primitive response by including <primitive:xx> before the first expression in the method.

Primitives exist for several reasons. Certain basic or 'primitive' operations cannot be performed in any other way. Smalltalk without primitives can move values from one variable to another; but cannot add two SmallIntegers together. Many methods for arithmetic and comparison between numbers are primitives. Some primitives allow Smalltalk to communicate with I/O devices such as the disk, the display, and the keyboard. Some primitives exist only to make the system run faster; each does the same thing as a certain Smalltalk method, and its implementation as a primitive is optional.

When the Smalltalk interpreter begins to execute a method which specifies a primitive response, it tries to perform the primitive action and to return a result. If the routine in the interpreter for this primitive is successful, it will return a value and the expressions in the method will not be evaluated. If the primitive routine is not successful, the primitive 'fails', and the Smalltalk expressions in the method are executed instead. These expressions are evaluated as though the primitive routine had not been called.

The Smalltalk code that is evaluated when a primitive fails usually anticipates why that primitive might fail. If the primitive is optional, the expressions in the method do exactly what the primitive would have done (See Number @). If the primitive only works on certain classes of arguments, the Smalltalk code tries to coerce the argument or appeals to a superclass to find a more general way of doing the operation (see SmallInteger +). If the primitive is never supposed to fail, the expressions signal an error (see SmallInteger asFloat).

Each method that specifies a primitive has a comment in it. If the primitive is optional, the comment will say 'Optional'. An optional primitive that is not implemented always fails, and the Smalltalk expressions do the work instead.

If a primitive is not optional, the comment will say, 'Essential'. Some methods will have the comment, 'No Lookup'. See Object howToModifyPrimitives for an explanation of special selectors which are not looked up.

For the primitives for +, -, \*, and bitShift: in SmallInteger; and truncated in Float, the primitive constructs and returns a 16-bit LargePositiveInteger when the result warrants it. Returning 16-bit LargePositiveIntegers from these primitives instead of failing is optional in the same sense that the LargePositiveInteger arithmetic primitives are optional. The comments in the SmallInteger primitives say,

Turtles all the way  
down ... to C ?

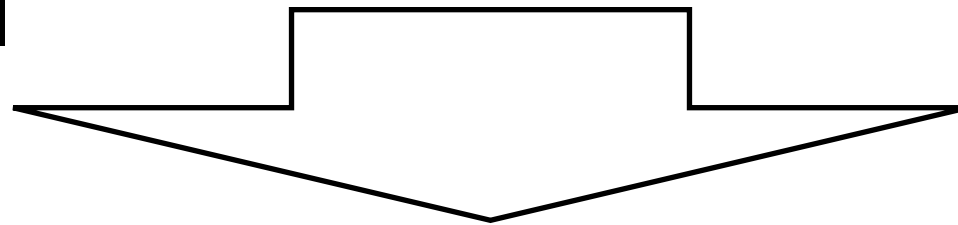
Turtles all the way  
down ... to C ?

But C is not nearly as deep down  
as we need! Hardware is!

C stands for C[rutches],  
not C[ar]

# Not a rocket science

- machines can run only machine code
- compilers is made to help us to abstract from gory details
- but often at a price being unable to harness the power of hardware at full potential

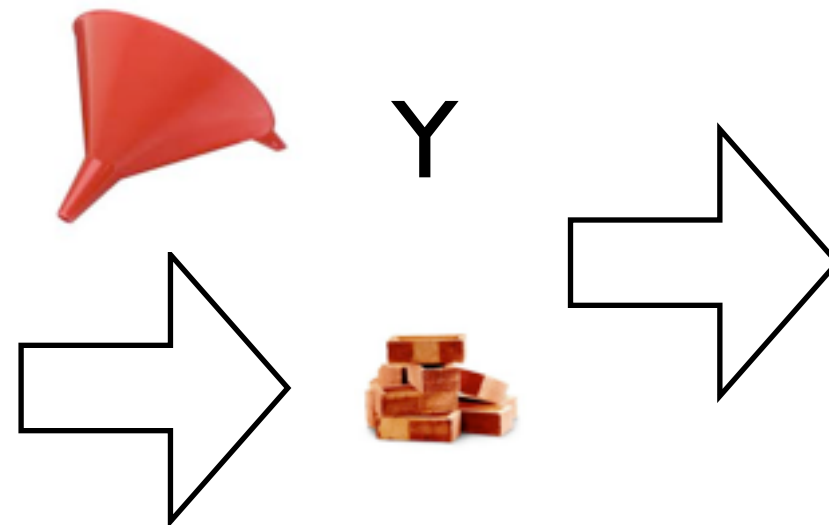


Can't go fast using crutches

# Compiler is a funnel

Supported semantics: X,Y,Z

X



Y

Z



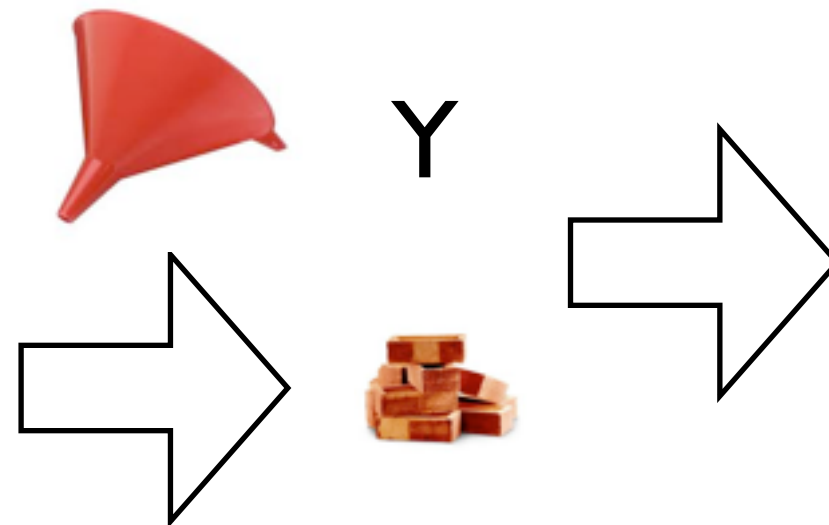
H  
a  
r  
d  
w  
a  
r  
e

Metaphor by: Marcus Denker

# Compiler is a funnel

Supported semantics: X,Y,Z

X



Y

Z



Compiler maps  $X \rightarrow Y \rightarrow Z(y)$

But we want  $X \rightarrow Z$ , simply because:

$$Z(y) < Z$$

Metaphor by: Marcus Denker

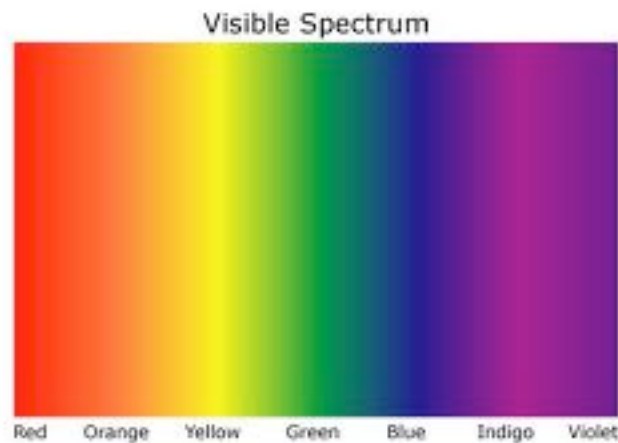
L  
a  
n  
g  
u  
a  
g  
e

H  
a  
r  
d  
w  
a  
r  
e



# When compiler fits:

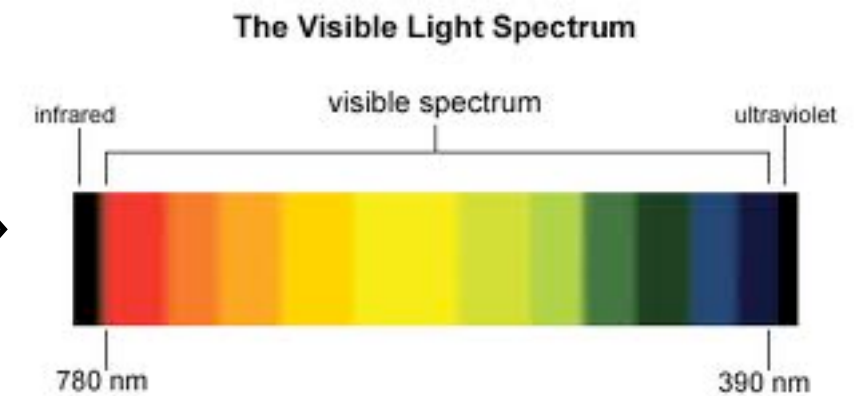
## Language potential



## Translator (Compiler)



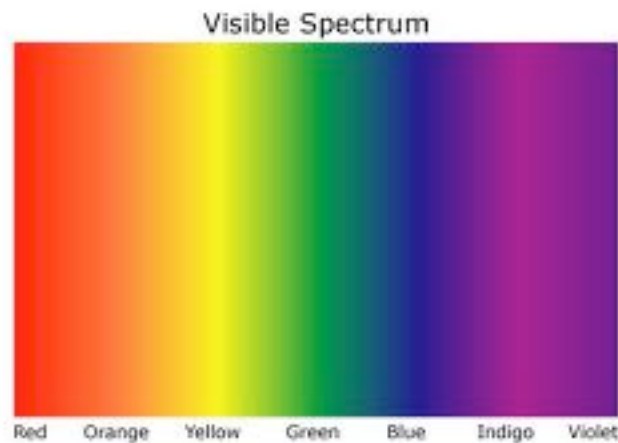
## Hardware potential





# And when it's not:

## Language potential

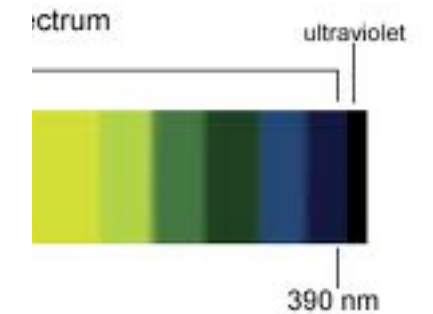


## Translator (Compiler)



## Hardware potential

The Visible Light Spectrum



And when it's not:

Language  
potential

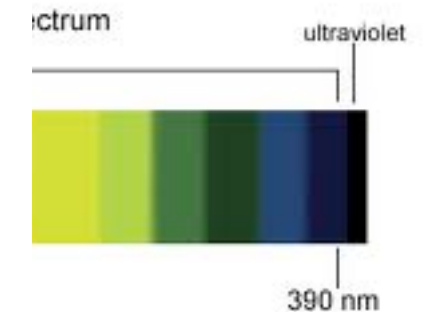


Translator  
(Compiler)



Hardware potential

The Visible Light Spectrum



But i want that part. And i mean it

## Example: Arithmetic Overflow

multiply: a with: b

$\wedge$  (a \* b) onArithmeticOverflow: [  
self primitiveFail ].

# Arithmetic Overflow, how it is done

multiply: a with: b

| result |

result := a \* b.

"check for C overflow by seeing if computation is reversible"

```
((b = 0) or: [(result // b) = a])  
  ifTrue: [ ^ result ]  
  ifFalse: [self primitiveFail]]
```

# Arithmetic Overflow, how it is done

multiply: a with: b

| result |

result := a \* b.

"check for C overflow by seeing if computation is reversible"

```
((b = 0) or: [(result // b) = a])  
  ifTrue: [ ^ result ]  
  ifFalse: [self primitiveFail]]
```

**But CPU has a flag to indicate arithmetic overflow!**

Shall we extend C compiler  
to suit better for our language/VM?

Shall we extend C compiler  
to suit better for our language/VM?

Compilers tend to be complex.  
And what language fits better for  
complex things, C or Smalltalk?

**NO**

Should we extend C compiler  
to suit better for our language/VM?



# Arithmetic Overflow, how it can be done with NB:

asm

mul: a with: b;

jumpIfOverflow: #overflow;

ret.

Oh, look! Magic!  
We can test for overflow!

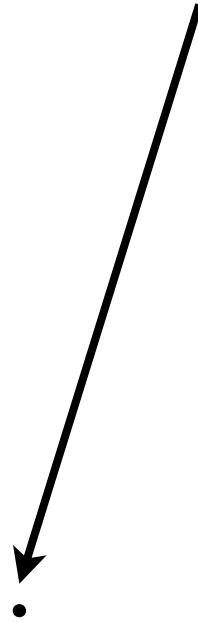


asm label: #overflow.

interpreterProxy primitiveFail.

asm ret.

That's my point



and motivation to  
develop NB

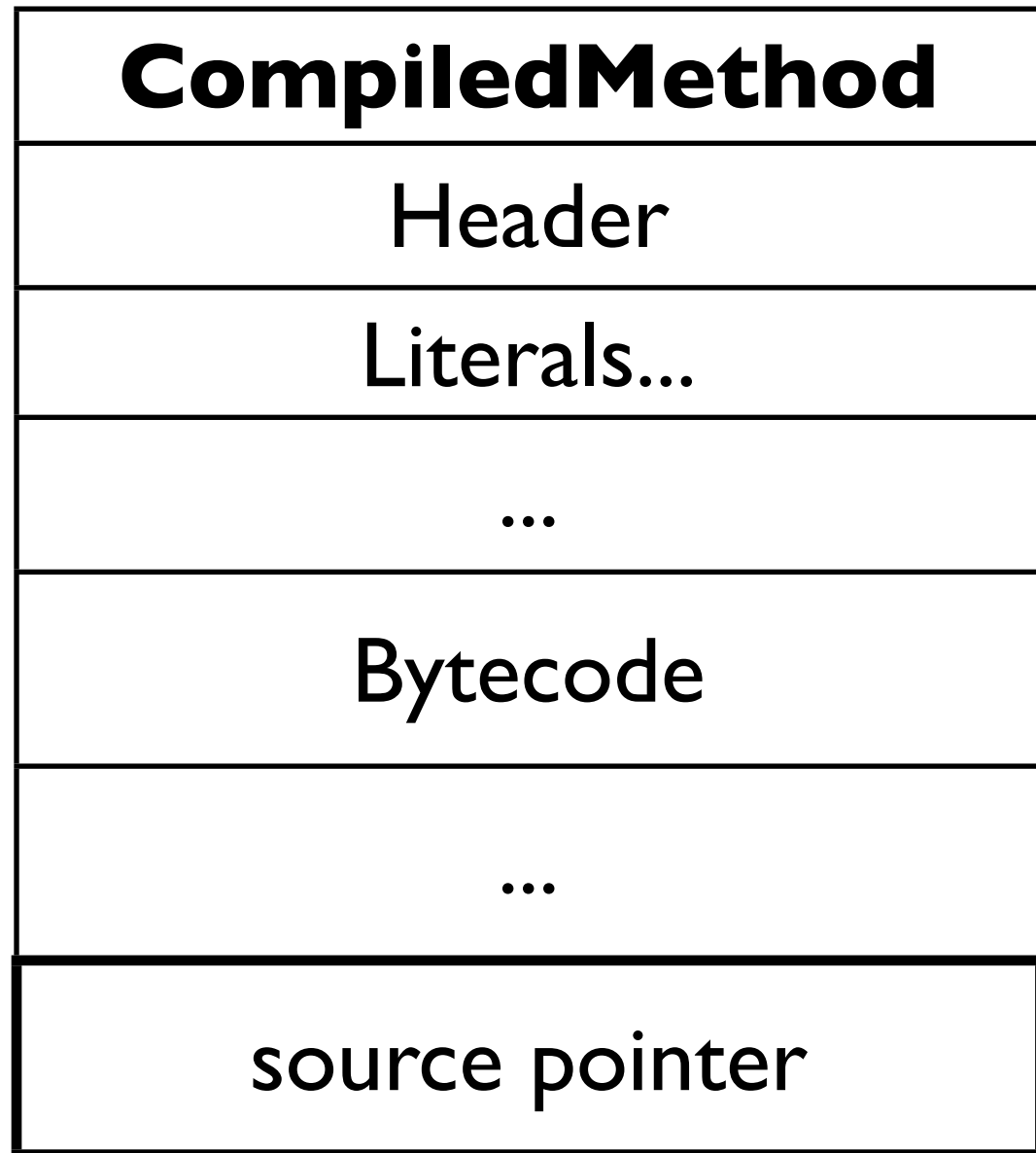
# What we have today:

- x86, x64 assemblers.
- VMs for Linux, Mac and Windows
- VM interface (InterpreterProxy)
- FFI
- Callbacks
- integration with JIT (first results)
- helpers for external resource & session

# Details: Assembler

- direct x86/x64 instruction database (no funneling, remember?)
- you free to use it (and contribute to it) outside NB (see AsmJit on squeaksource)
- you free/welcome to implement more abstract (platform-neutral) assembler/compiler/whatever on top of it
- AsmJIT assemblers will be always platform dependent. see rule #1

# The magic



← Method trailer

# The magic

<b>CompiledMethod</b>
Header
Literals...
...
Bytecode
...
machine code ...
source pointer

← Method trailer

# The magic

CompiledMethod
Header
Literals...
...
Bytecode
...
machine code ...
source pointer

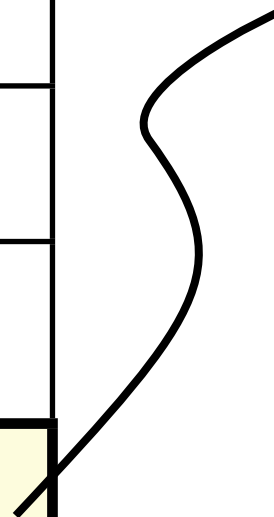
#primitiveNativeCall

...

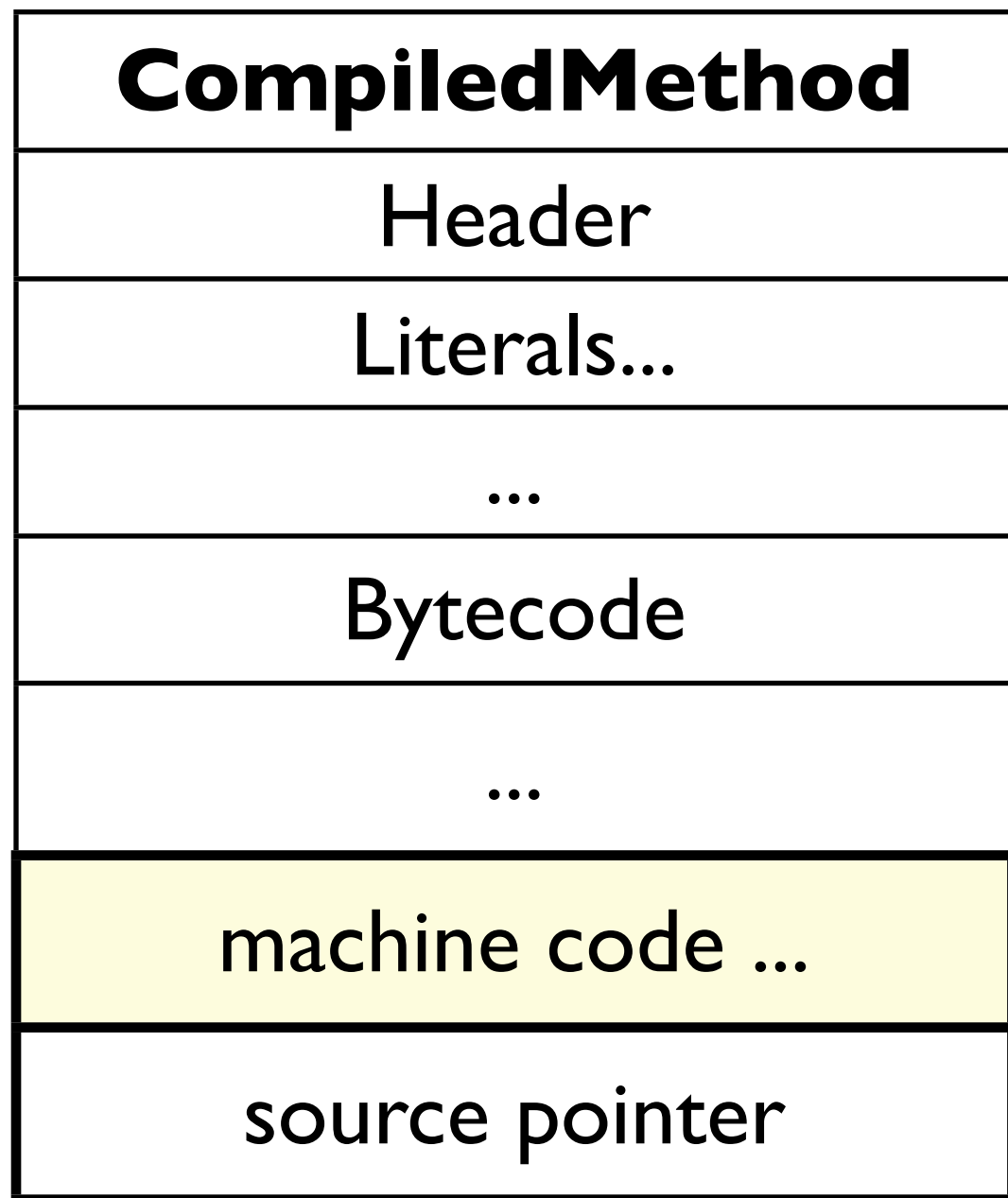
function := pointer to:(machine  
code).

result := function().

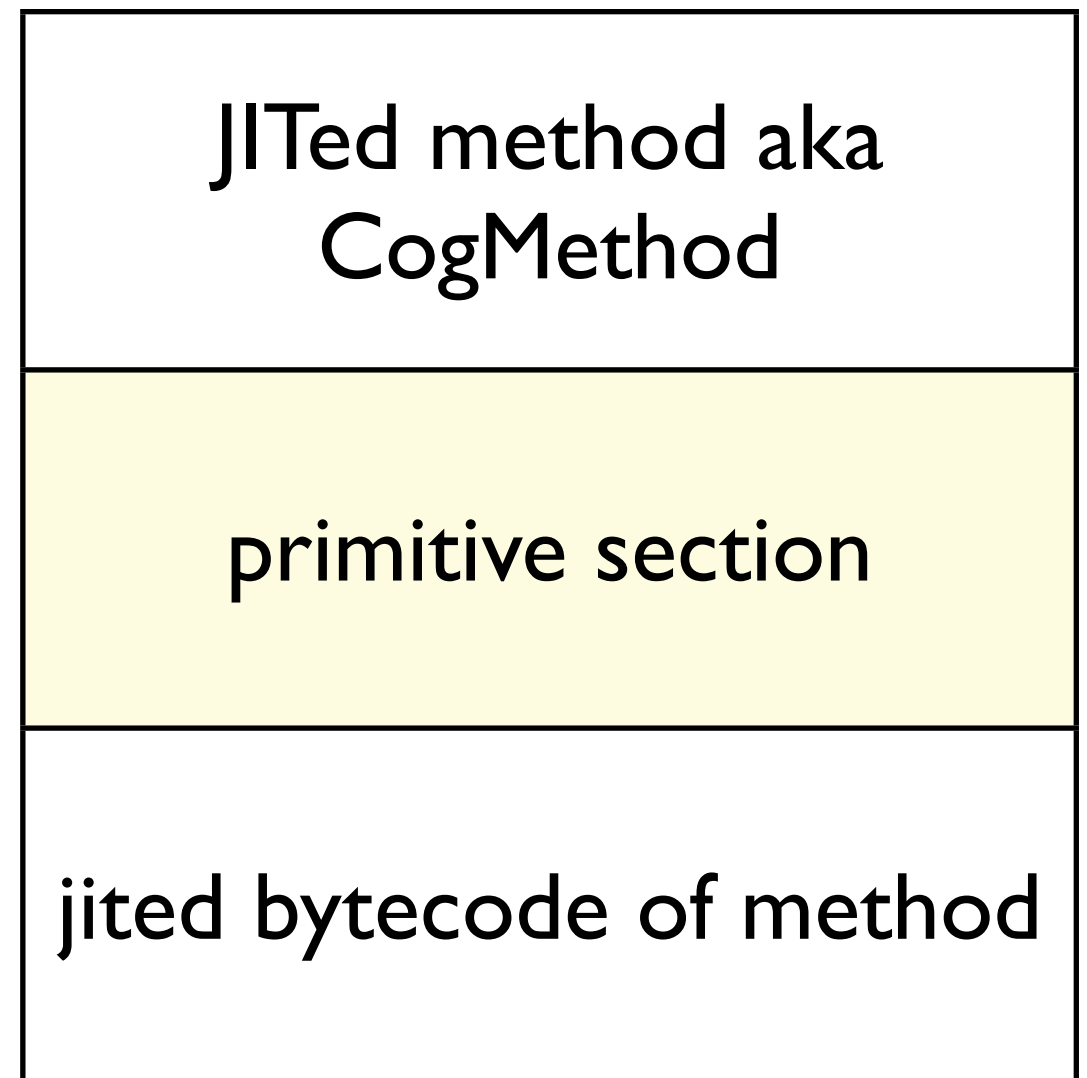
...



# The JIT magic



primitive #220 (voltage)





# Details: VM

- A custom VM with some changes to better support native code
- which includes NativeBoost plugin
- it is decided to be the default VM for future releases of Pharo (nobody asked me about it)

# Details: VM

- built on Jenkins server:

<http://pharo.gforge.inria.fr/ci/vm/nbcog/>

- additionally includes Cairo graphics library, bundled together with VM

# Details: FFI

- using external libraries
- make calls, callbacks
- implement primitives
- fast, flexible, extensible

# Details: FFI

- used by Athens
- NBOpenGL -> SourceCity
- more things is under active development inside and outside our team @Lille

# Types

- support for most basic C types: int, float etc
- C structures (see NBExternalStructure and subclasses)
- values of certain type
- arrays of value types (to be added)

# Examples & Demo

# Future calls:

- unify existing FFI interfaces
- provide solution for platforms with no dynamic code generation allowed (yes it will be limited one)
- threading (aka non-blocking calls)

# Fear Uncertainty Doubt



*Налево пойдешь - коня потеряешь,  
Направо пойдешь - жизнь потеряешь,  
Прямо пойдешь - жив будешь, да себя позабудешь.*

turn left - lose the horse

turn right - lose own life

go straight - you'll keep your life, but lose all your memories







# Focus on what you will win, not lose

- Performance
- Flexibility
- Full control

?

The end