Compiling with inlining

Druid + Opal = DrOpal ❤️

Nahuel PALUMBO

ESUG 2024 - Lille

Inlines in Pharo

Current state

```
run

   5 timesRepeat: [ 'Repeat' traceCr ].
   'end' traceCr
```

We have a simple method using **timesRepeat:** to log

After execution

Transcript

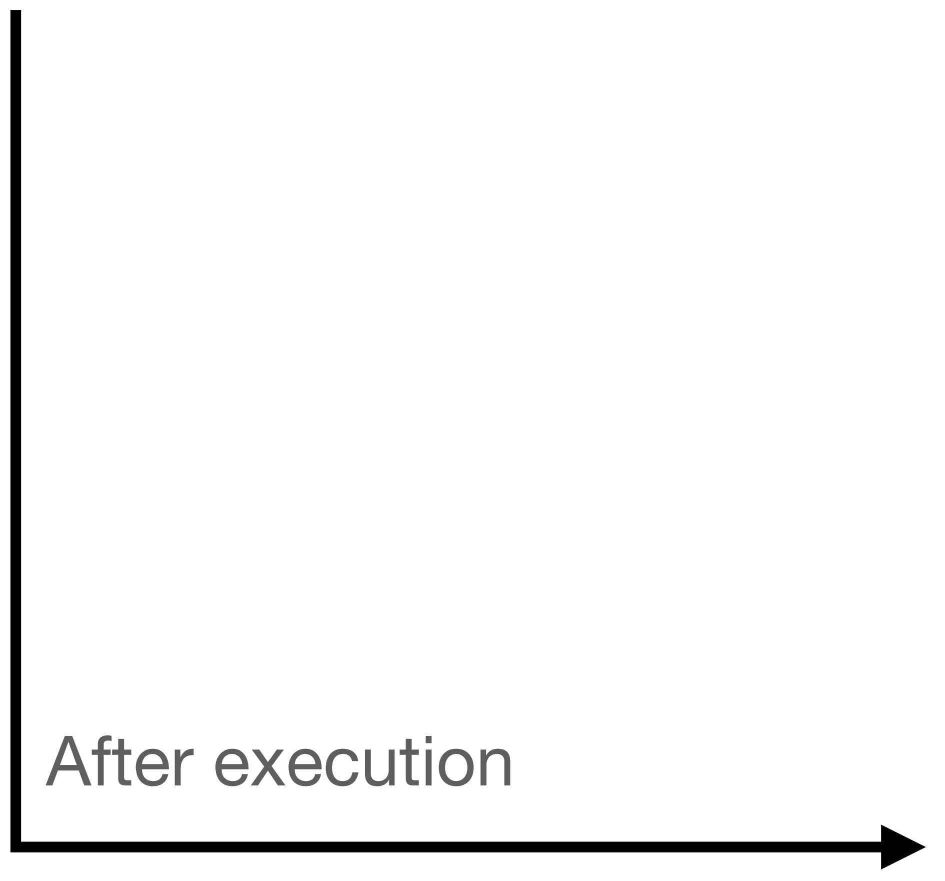
```
World!
World!
World!
World!
World!
end
```

Inlines in Pharo

Current state

```
run

   5 timesRepeat: [ 'Repeat' traceCr ].
   'end' traceCr
```

We have a simple method using **timesRepeat:** to log



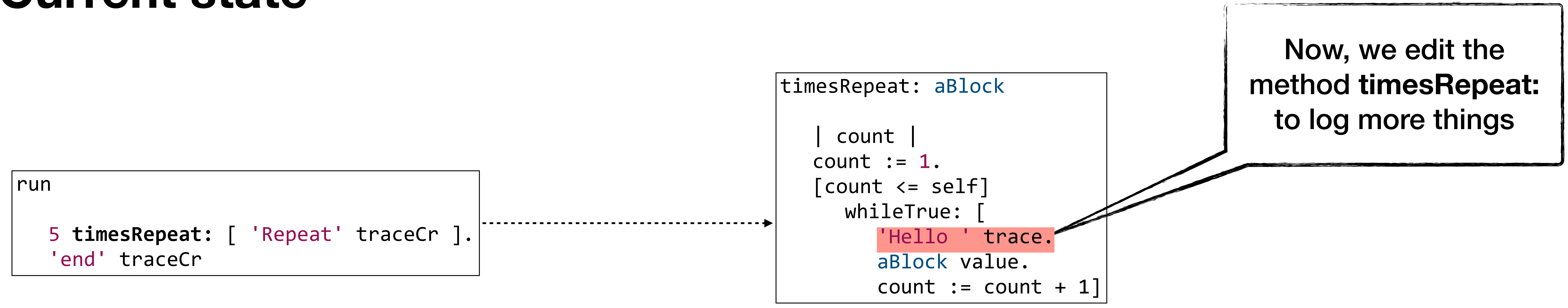
Transcript

```
World!
World!
World!
World!
World!
end
```

✓

Inlines in Pharo

Current state



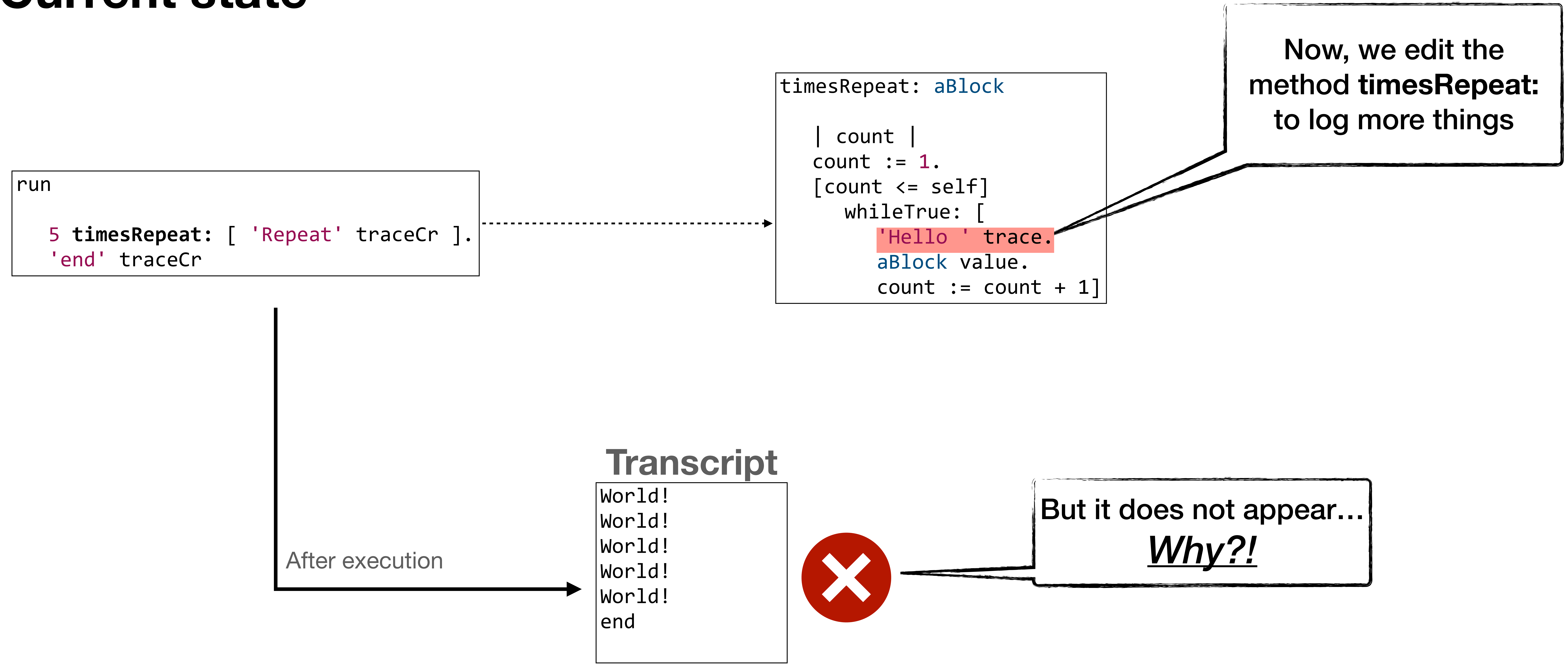
```
run

   5 timesRepeat: [ 'Repeat' traceCr ].
   'end' traceCr
```

```
timesRepeat: aBlock

   | count |
   count := 1.
   [count <= self]
      whileTrue: [
         'Hello ' trace.
         aBlock value.
         count := count + 1]
```

Now, we edit the method **timesRepeat:** to log more things

Inlines in Pharo

Current state



```
run

   5 timesRepeat: [ 'Repeat' traceCr ].
   'end' traceCr
```

```
timesRepeat: aBlock

   | count |
   count := 1.
   [count <= self]
      whileTrue: [
         'Hello ' trace.
         aBlock value.
         count := count + 1]
```

Now, we edit the method **timesRepeat:** to log more things

After execution

Transcript

```
World!
World!
World!
World!
World!
end
```

But it does not appear…
Why?!

Inlines in Pharo

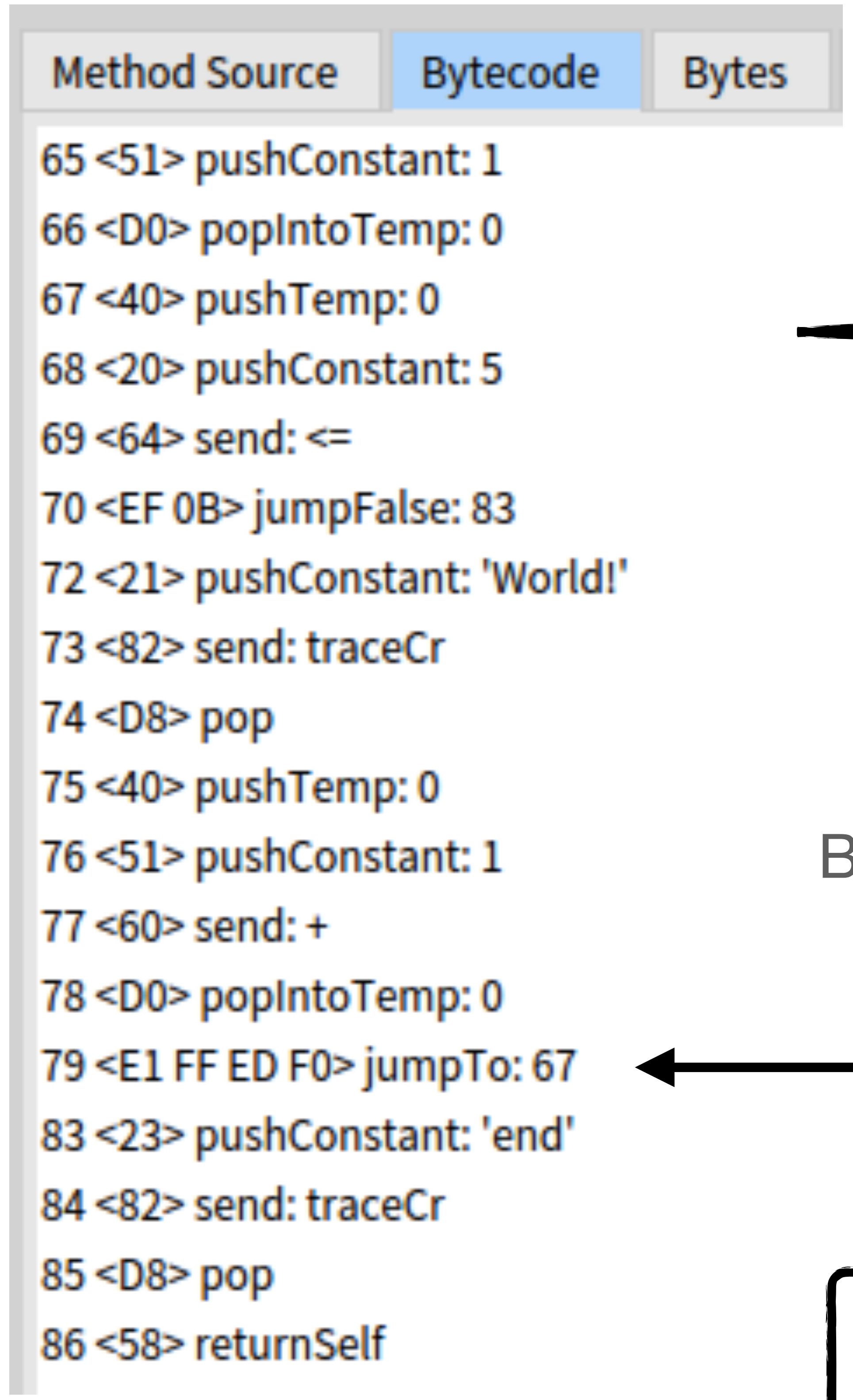
Current state

```
timesRepeat: aBlock

   | count |
   count := 1.
   [count <= self]
      whileTrue: [
         'Hello ' trace.
         aBlock value.
         count := count + 1]
```

This method is never invoked!

```
run

   5 timesRepeat: [ 'Repeat' traceCr ].
   'end' traceCr
```

Method's bytecode



```
65 <51> pushConstant: 1
66 <D0> popIntoTemp: 0
67 <40> pushTemp: 0
68 <20> pushConstant: 5
69 <64> send: <=
70 <EF 0B> jumpFalse: 83
72 <21> pushConstant: 'World!'
73 <82> send: traceCr
74 <D8> pop
75 <40> pushTemp: 0
76 <51> pushConstant: 1
77 <60> send: +
78 <D0> popIntoTemp: 0
79 <E1 FF ED F0> jumpTo: 67
83 <23> pushConstant: 'end'
84 <82> send: traceCr
85 <D8> pop
86 <58> returnSelf
```

There is not `send: timesRepeat:` in the bytecode

Backjump = Loop

timesRepeat: is one of the messages *inlined* by the bytecode compiler *(Opal)*

The loop was *"inlined"*…
But how?!

Inlines in Pharo

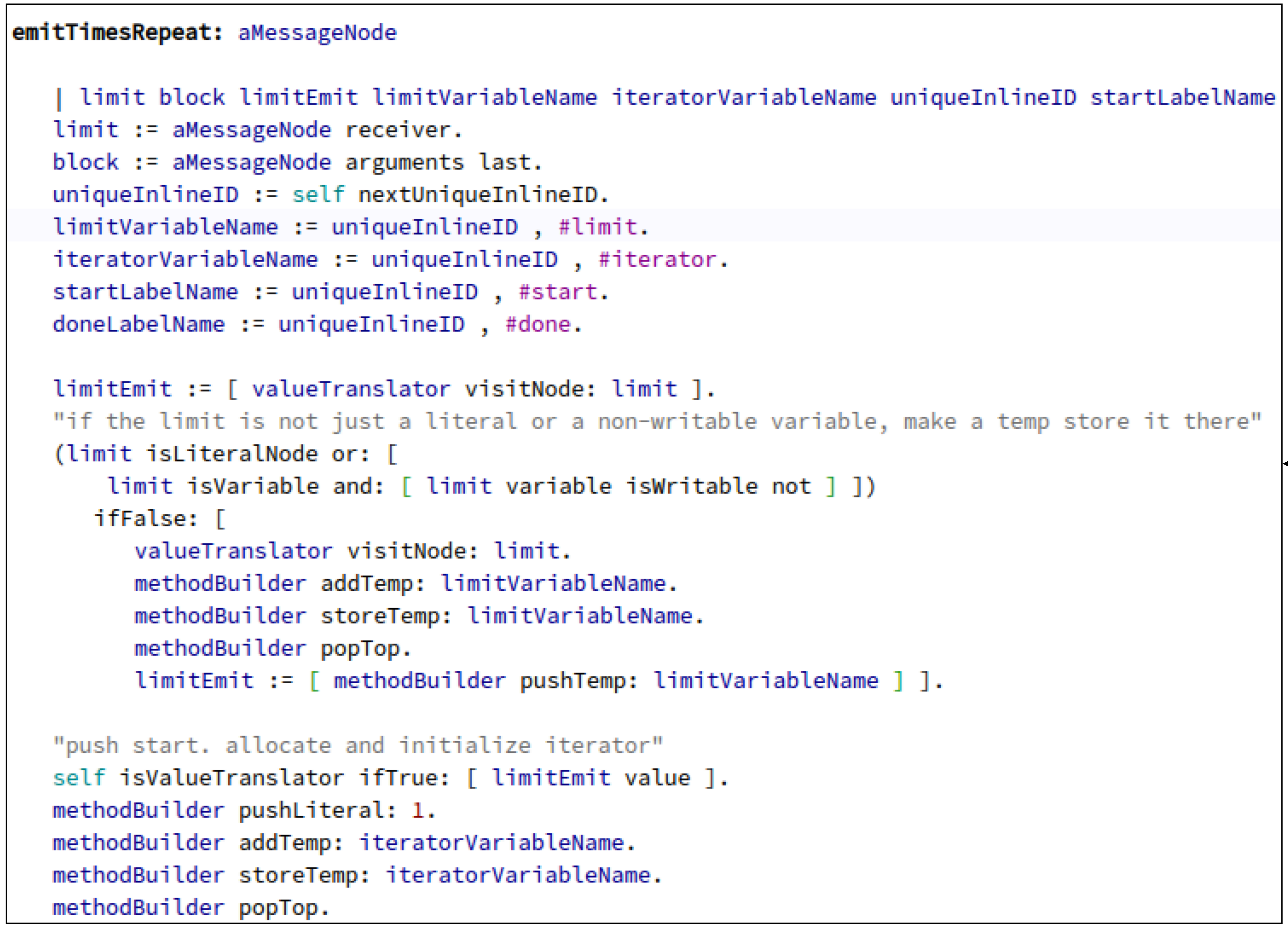
Current state

They are
__not connected__ !

```
timesRepeat: aBlock

   | count |
   count := 1.
   [count <= self]
      whileTrue: [
         'Hello ' trace.
         aBlock value.
         count := count + 1]
```

```
run

   5 timesRepeat: [ 'Repeat' traceCr ].
   'end' traceCr
```

When a **timesRepeat:** is found, it is compiled using a custom definition



```
emitTimesRepeat: aMessageNode

   | limit block limitEmit limitVariableName iteratorVariableName uniqueInlineID startLabelName
   limit := aMessageNode receiver.
   block := aMessageNode arguments last.
   uniqueInlineID := self nextUniqueInlineID.
   limitVariableName := uniqueInlineID , #limit.
   iteratorVariableName := uniqueInlineID , #iterator.
   startLabelName := uniqueInlineID , #start.
   doneLabelName := uniqueInlineID , #done.

   limitEmit := [ valueTranslator visitNode: limit ].
   "if the limit is not just a literal or a non-writable variable, make a temp store it there"
   (limit isLiteralNode or: [
       limit isVariable and: [ limit variable isWritable not ] ])
      ifFalse: [
         valueTranslator visitNode: limit.
         methodBuilder addTemp: limitVariableName.
         methodBuilder storeTemp: limitVariableName.
         methodBuilder popTop.
         limitEmit := [ methodBuilder pushTemp: limitVariableName ] ].

   "push start. allocate and initialize iterator"
   self isValueTranslator ifTrue: [ limitEmit value ].
   methodBuilder pushLiteral: 1.
   methodBuilder addTemp: iteratorVariableName.
   methodBuilder storeTemp: iteratorVariableName.
   methodBuilder popTop.
```

Inlines in Pharo

Current state

All these messages are *inlined* by the bytecode compiler using a custom implementation in the compiler

```
timesRepeat: aBlock

   | count |
   count := 1.
   [count <= self]
      whileTrue: [
         'Hello ' trace.
         aBlock value.
         count := count + 1]
```

```
OptimizedMessages := {
  (#caseOf: -> #emitCaseOf:).
  (#caseOf:otherwise: -> #emitCaseOfOtherwise:).
  (#ifFalse: -> #emitIfFalse:).
  (#ifFalse:ifTrue: -> #emitIfFalseIfTrue:).
  (#ifNil: -> #emitIfNil:).
  (#ifNil:ifNotNil: -> #emitIfNilIfNotNil:).
  (#ifNotNil: -> #emitIfNotNil:).
  (#ifNotNil:ifNil: -> #emitIfNotNilIfNil:).
  (#ifTrue: -> #emitIfTrue:).
  (#ifTrue:ifFalse: -> #emitIfTrueIfFalse:).
  (#or: -> #emitOr:).
  (#and: -> #emitAnd:).
  (#timesRepeat: -> #emitTimesRepeat:).
  (#repeat -> #emitRepeat:).
  (#to:by:do: -> #emitToByDo:).
  (#to:do: -> #emitToDo:).
  (#whileFalse: -> #emitWhileFalse:).
  (#whileTrue: -> #emitWhileTrue:).
  (#whileFalse -> #emitWhileFalse:).
  (#whileTrue -> #emitWhileTrue:) } asDictionary
```

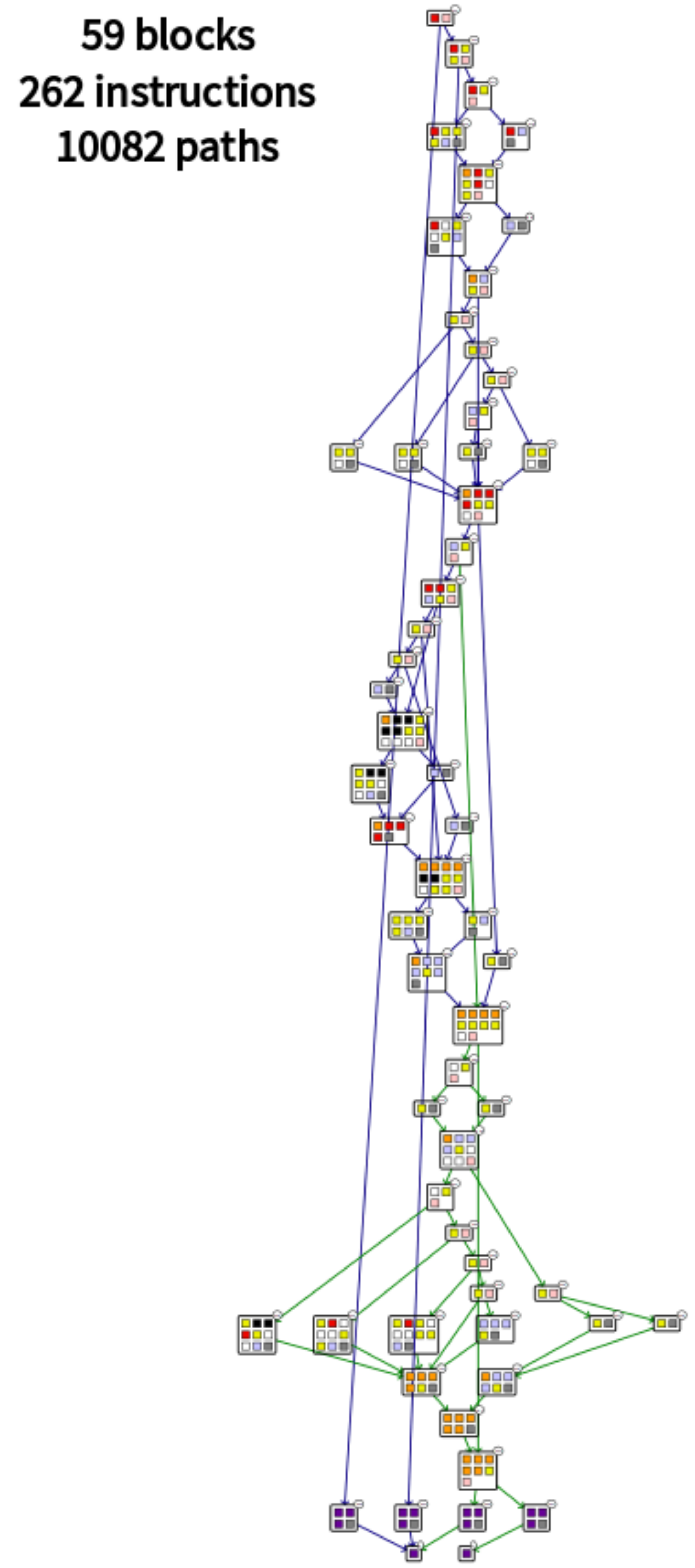
These methods are *almost never executed*.

The emitted bytecode *"simulates"* the work.

Inlines in Pharo

Using Druid

DRUID



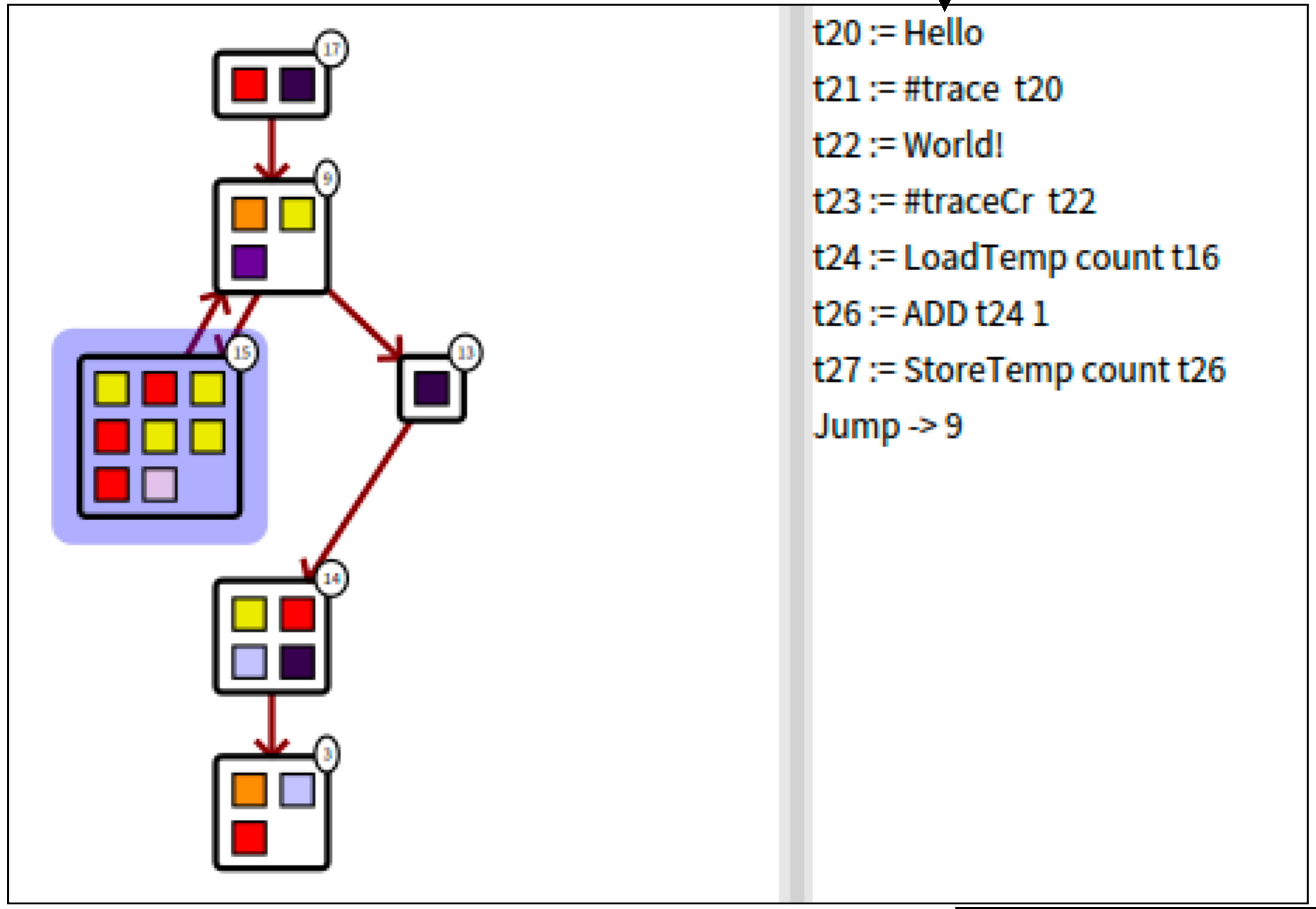
Inlines in Pharo

Using Druid

```
run

   5 timesRepeat: [ 'Repeat' traceCr ].
   'end' traceCr
```

```
timesRepeat: aBlock

   | count |
   count := 1.
   [count <= self]
      whileTrue: [
         'Hello ' trace.
         aBlock value.
         count := count + 1]
```

Inlined



```
t20 := Hello
t21 := #trace t20
t22 := World!
t23 := #traceCr t22
t24 := LoadTemp count t16
t26 := ADD t24 1
t27 := StoreTemp count t26
Jump -> 9
```

```
81 <51> pushConstant: 1
82 <D0> popIntoTemp: 0
83 <40> pushTemp: 0
84 <20> pushConstant: 5
85 <64> send: <=
86 <EF 0E> jumpFalse: 102
88 <21> pushConstant: 'Hello '
89 <82> send: trace
90 <D8> pop
91 <23> pushConstant: 'World!'
92 <84> send: traceCr
93 <D8> pop
94 <40> pushTemp: 0
95 <51> pushConstant: 1
96 <60> send: +
97 <D0> popIntoTemp: 0
98 <E1 FF ED ED> jumpTo: 83
102 <25> pushConstant: 'end'
103 <84> send: traceCr
104 <D8> pop
105 <58> returnSelf
```

We use the same *encoder* and *decompiler*

```
run

   1 to: 5 do: [ :tmp1 |
      'Hello ' trace.
      'World!' traceCr ].
   'end' traceCr
```

I generate an *SSA-form Intermediate Representation* where inlines are performed (from the *original* method!)

DRUID

Inlines in Pharo

Using Druid

What is compiled (and inlined)

```
run

   5 timesRepeat: [ 'Repeat' traceCr ].
   'end' traceCr
```

What you write

```
timesRepeat: aBlock

   | count |
   count := 1.
   [count <= self]
      whileTrue: [
         'Hello ' trace.
         aBlock value.
         count := count + 1]
```

DRUID

```
run

   1 to: 5 do: [ :tmp1 |
      'Hello ' trace.
      'World!' traceCr ].
   'end' traceCr
```

After execution

Transcript

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
end
```

Compiling with inlining

Druid + Opal = DrOpal ❤️

- Druid compiler - https://github.com/Alamvic/druid
- Opal compiler (already in the image) - https://github.com/pharo-project/pharo/tree/Pharo13/src/OpalCompiler-Core

- This is an experimental project yet, it is missing:
  - Support for Blocks compilation
  - Deoptimization