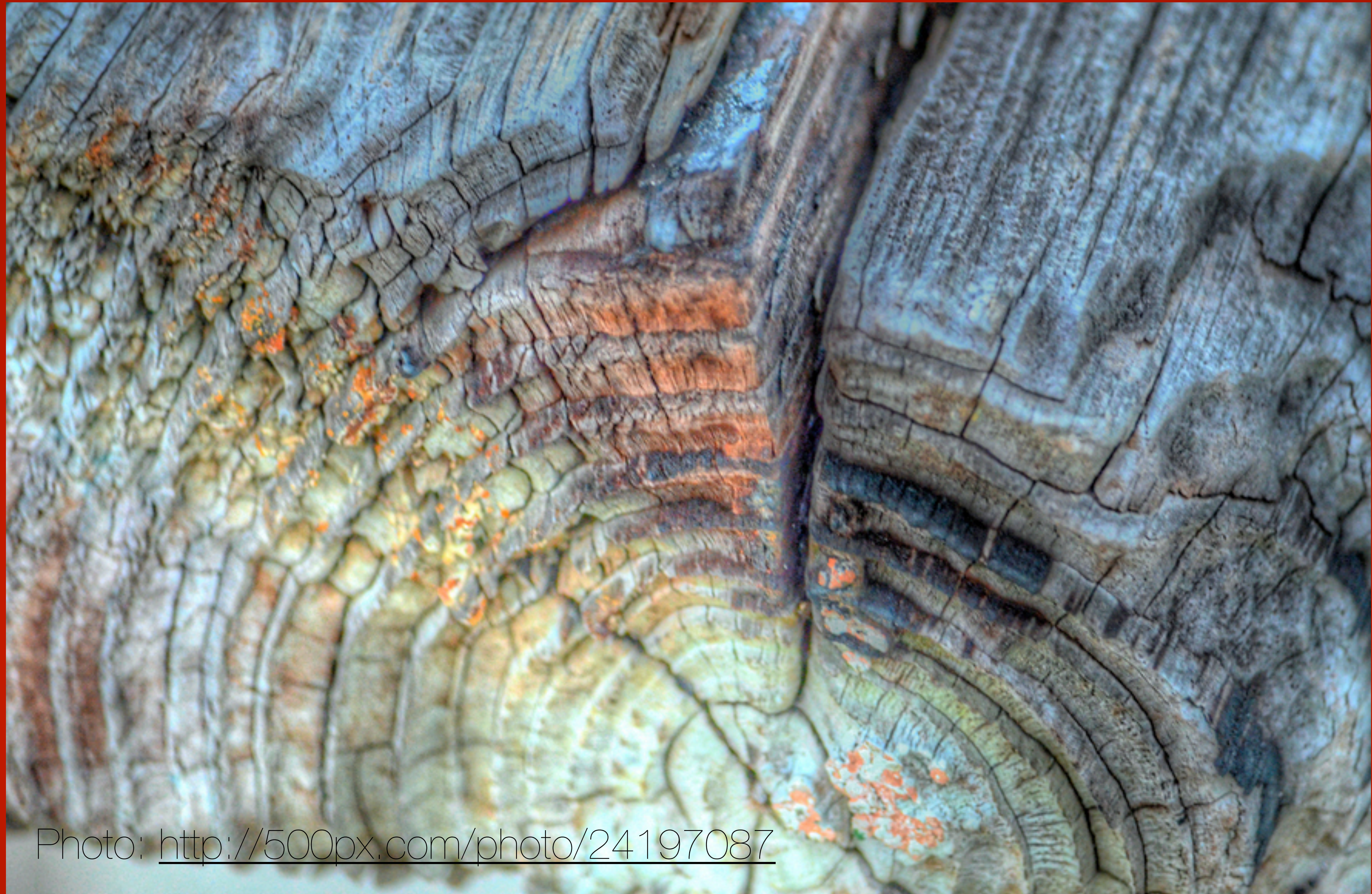
RUBY INTERNALS

Photo: http://500px.com/photo/25805131

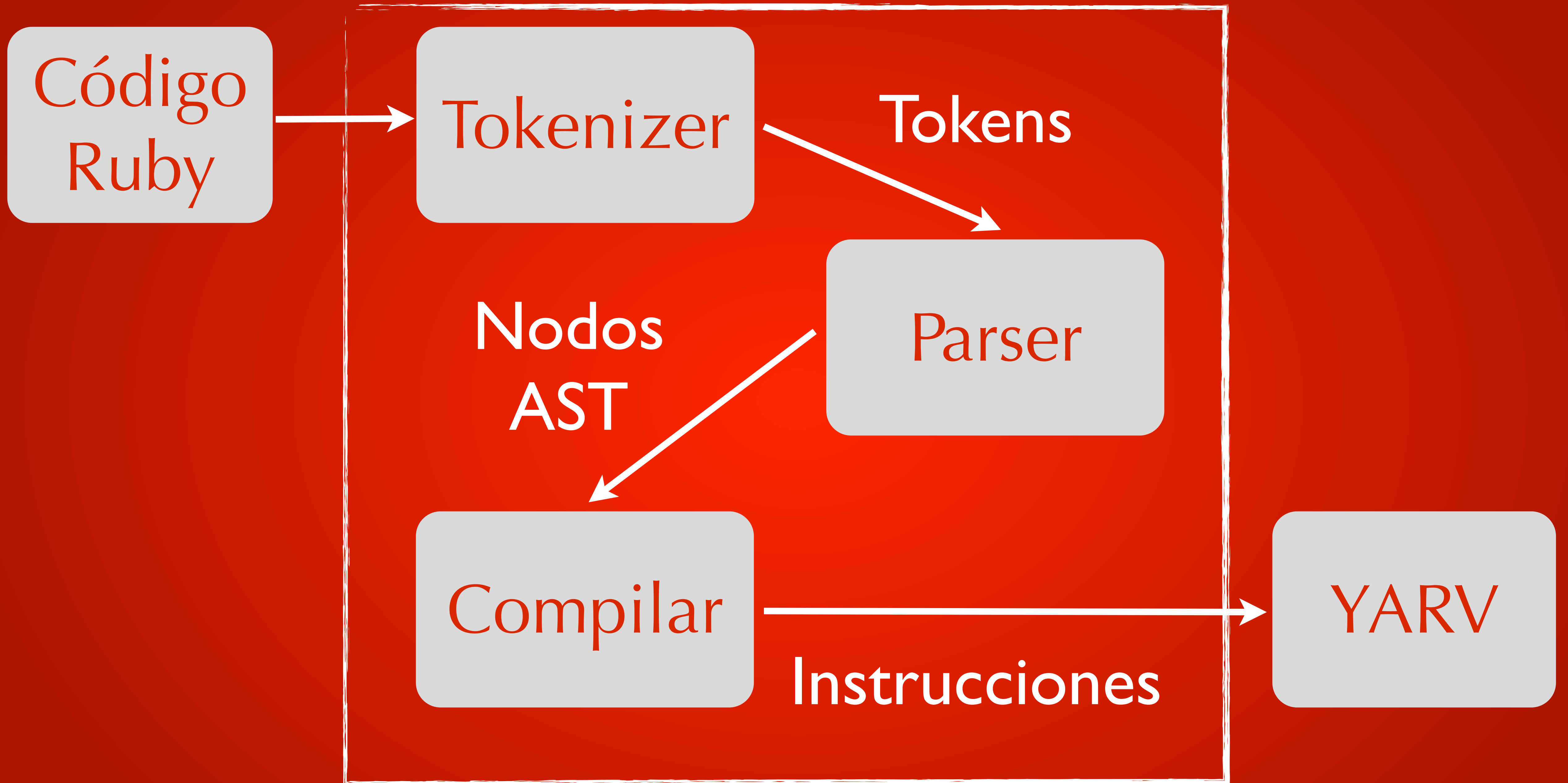
Mario Alberto Chávez
@mario_chavez



Viaje al centro de Ruby

Exploraremos la versión C de Ruby

(Sí Ruby está escrito en C)


Código Ruby
Tokenizer
Tokens
Parser
Nodos AST
Compilar
Instrucciones
YARV

Tokenize: Convertir cadena de texto en elementos que Ruby comprenda

```ruby
10.times do |i|
  puts i
end
```

```
10.times do |i| puts i end
```

tInteger 10

.

tIndetifier "times"

keyword_do

|

tIndetifier "i"

|

tIndetifier "puts"

tIndetifier "i"

keyword_end

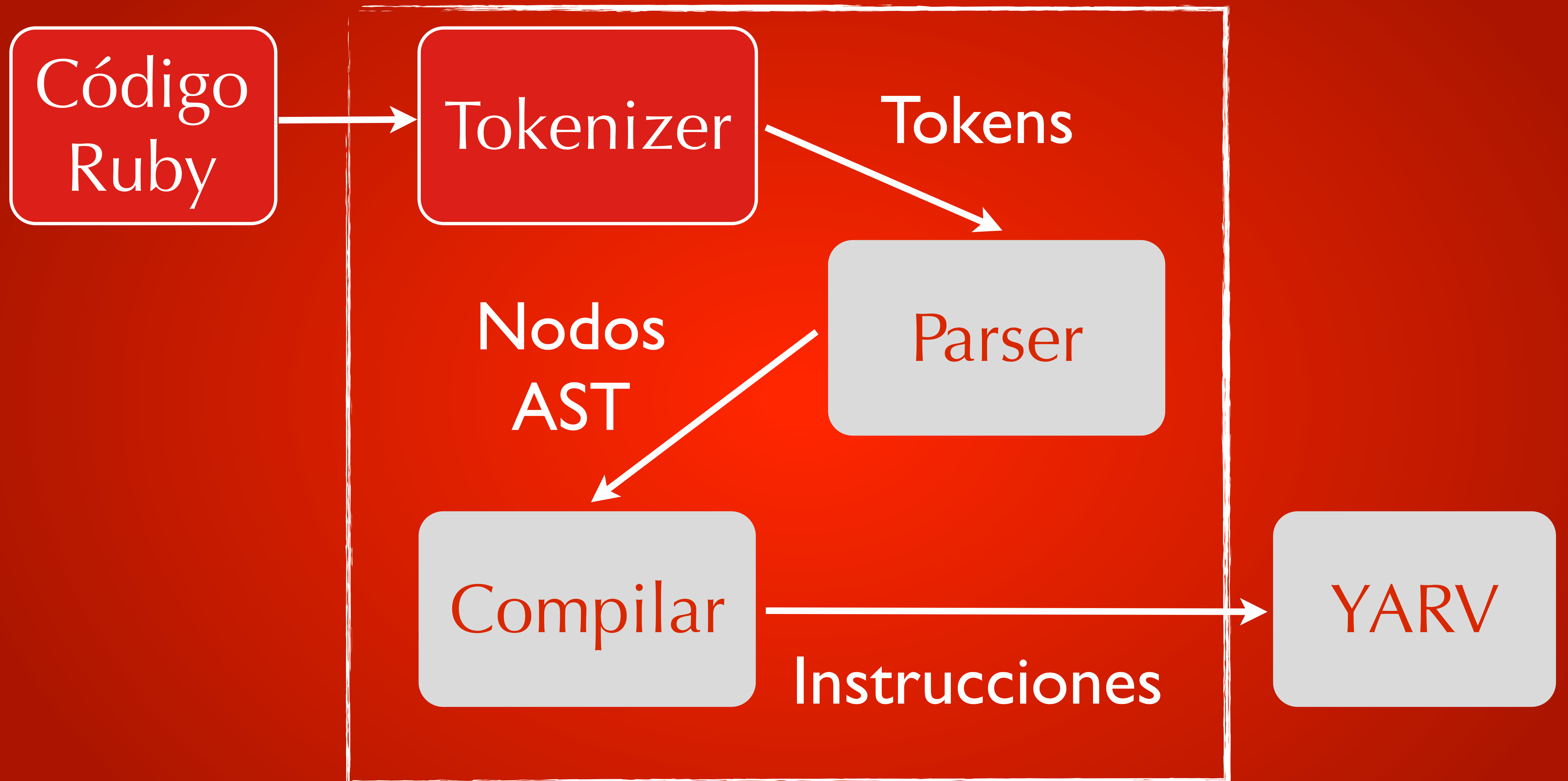
```
require ‘ripper’
require ‘awesome_print’

code = <<STR
10.times do |i|
  puts i
end
STR

ap Ripper.lex(code)
```

```
[[[1, 0], :on_int, "10"],
 [[1, 2], :on_period, "."],
 [[1, 3], :on_ident, "times"],
 [[1, 8], :on_sp, " "],
 [[1, 9], :on_kw, "do"],
 [[1, 11], :on_sp, " "],
 [[1, 12], :on_op, "|"],
 [[1, 13], :on_ident, "i"],
 [[1, 14], :on_op, "|"],
 [[1, 15], :on_ignored_nl, "\n"],
 [[2, 0], :on_sp, "  "],
 [[2, 2], :on_ident, "puts"],
 [[2, 6], :on_sp, " "],
 [[2, 7], :on_ident, "i"],
 [[2, 8], :on_nl, "\n"],
 [[3, 0], :on_kw, "end"],
 [[3, 3], :on_nl, "\n"]]
```

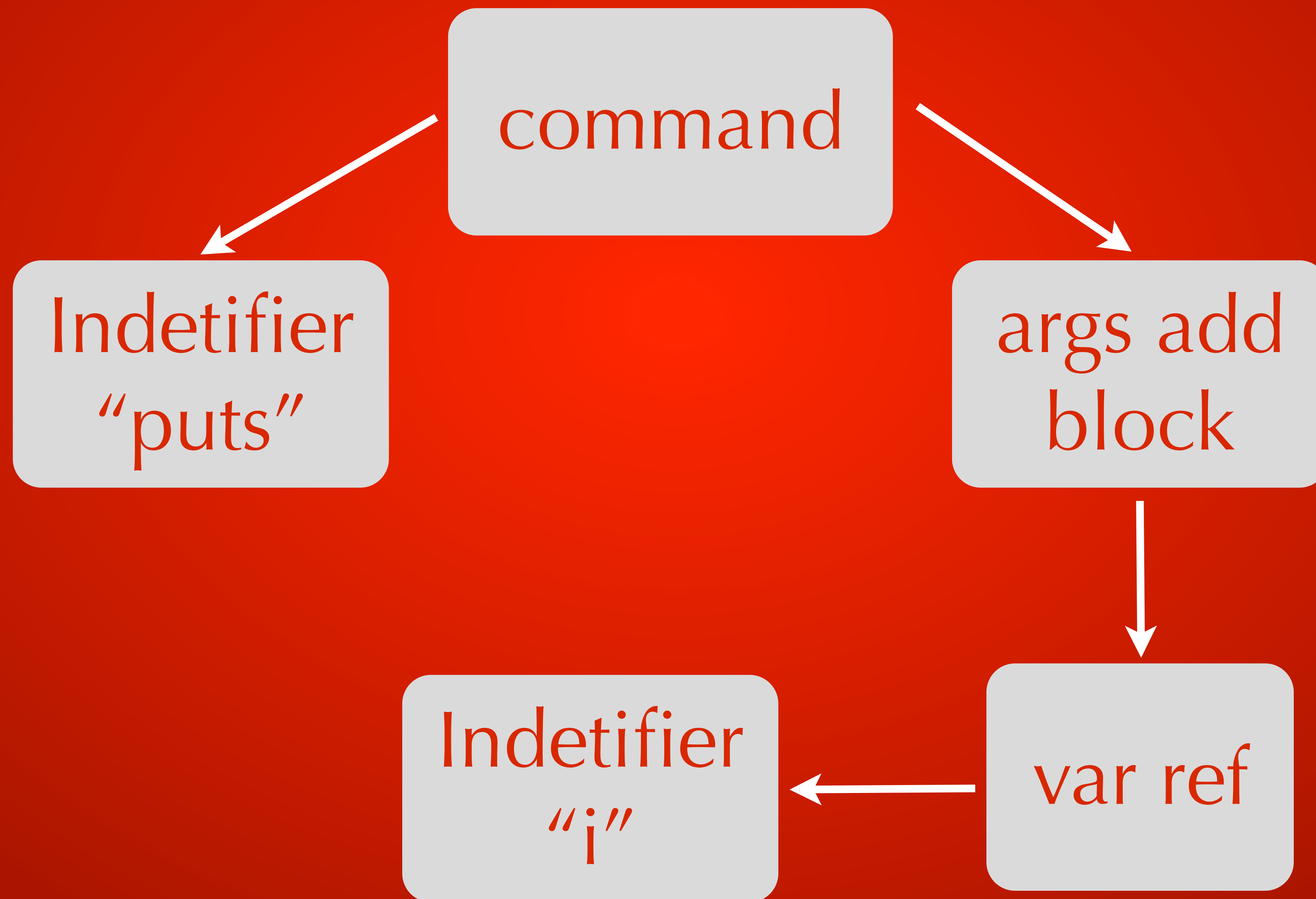
- Ripper es un tokenizer de Ruby
- Ripper esta presente en Ruby 1.9 y 2.0
- Ruby no utiliza LEX, Matz escribió su propio lexer
- parser.y contiene las reglas del lenguaje


Código
Ruby
Tokenizer
Tokens
Parser
Nodos
AST
Compilar
Instrucciones
YARV

Parse: Agrupa los tokens en frases que Ruby comprenda

```ruby
10.times do |i|
  puts i
end
```

10.times do |i| puts i end



```
require ‘ripper’
require ‘awesome_print’

code = <<STR
10.times do |i|
  puts i
end
STR

ap Ripper.sexp(code)
```

```
[:program,
 [[:method_add_block,
   [:call, [:@int, "10", [1, 0]], :".",
[:@ident, "times", [1, 3]]],
   [:do_block,
    [:block_var,
     [:params, [[:@ident, "i", [1, 13]]],
nil, nil, nil, nil],
     nil],
    [[:command,
      [:@ident, "puts", [2, 0]],
      [:args_add_block, [[:var_ref,
[:@ident, "i", [2, 5]]]], false]]]]]]
```

- Ruby usa un LALR parser generator (Look Ahead LR)
- Usa el programa GNU Bison
- La reglas gramiticales siguen notacion “Backus-Naur Form” (BNF)
- parser.y contiene las reglas del lenguaje

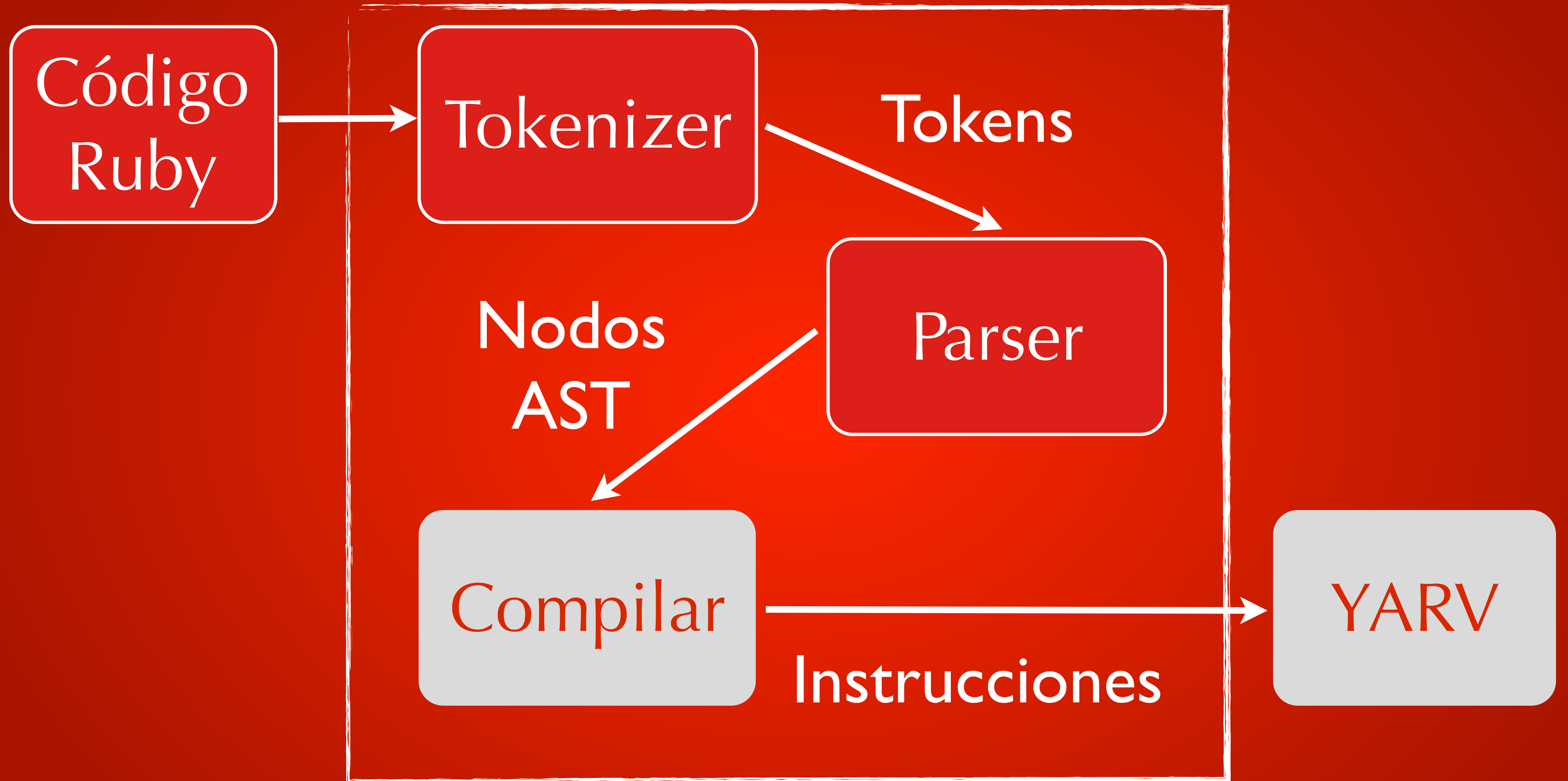
- El parser genera AST (Abstract Syntax Tree)
- Con la informacion de debug podemos ver otro formato de AST

```
$ ruby dump --parsetree my_program.rb
```

```
###########################################################
## Do NOT use this node dump for any purpose other than  ##
## debug and research.  Compatibility is not guaranteed. ##
###########################################################

# @ NODE_SCOPE (line: 3)
# +- nd_tbl: (empty)
# +- nd_args:
# |   (null node)
# +- nd_body:
#     @ NODE_ITER (line: 1)
#     +- nd_iter:
#     |   @ NODE_CALL (line: 1)
#     |   +- nd_mid: :times
#     |   +- nd_recv:
#     |   |   @ NODE_LIT (line: 1)
#     |   |   +- nd_lit: 10
#     |   +- nd_args:
#     |       (null node)
#     +- nd_body:
#         @ NODE_SCOPE (line: 3)
#         +- nd_tbl: :i
#         +- nd_args:
#         |   @ NODE_ARGS (line: 1)
#         |   +- nd_frml: 1
#         |   +- nd_next:
#         |   |   @ NODE_ARGS_AUX (line: 1)
#         |   |   +- nd_rest: (null)
#         |   |   +- nd_body: (null)
#         |   |   +- nd_next:
#         |   |       (null node)
#         |   +- nd_opt:
#         |       (null node)
#         +- nd_body:
#             @ NODE_FCALL (line: 2)
#             +- nd_mid: :puts
#             +- nd_args:
#                 @ NODE_ARRAY (line: 2)
#                 +- nd_alen: 1
#                 +- nd_head:
#                 |   @ NODE_DVAR (line: 2)
#                 |   +- nd_vid: :i
#                 +- nd_next:
#                     (null node)
```

Casi llegamos al final



Código Ruby
Tokenizer
Tokens
Parser
Nodos AST
Compilar
Instrucciones
YARV

Ruby 1.9/2.0 introduce "Yet Another Virtual Machine" (YARV)

- YARV usa el AST para:
  - Generar codigo intermedio (YARV Instructions)
  - Es posible aplicar micro-optimización

VM Stack: Asume que parámetros y valores de retorno estan en el stack

puts 2 + 2

Instrucciones YARV

NODE_SCOPE
table: [ ]
args: [ ]

NODE_FCALL
method_id: “puts”

NODE_CALL
method_id: “+”

NODE_LITERAL
“2”

NODE_LITERAL
“2”

puts 2 + 2

NODE_SCOPE
table: [ ]
args: [ ]

NODE_FCALL
method_id: “puts”

NODE_CALL
method_id: “+”

NODE_LITERAL
“2”

NODE_LITERAL
“2”

Instrucciones YARV

```
putself
```

puts 2 + 2

NODE_SCOPE
table: [ ]
args: [ ]

NODE_FCALL
method_id: “puts”

NODE_CALL
method_id: “+”

NODE_LITERAL
“2”

NODE_LITERAL
“2”

Instrucciones YARV

```
putself
putobject 2
```

puts 2 + 2

Instrucciones YARV

```
putself
putobject 2
putobject 2
```

NODE_SCOPE
table: [ ]
args: [ ]

NODE_FCALL
method_id: “puts”

NODE_CALL
method_id: “+”

NODE_LITERAL
“2”

NODE_LITERAL
“2”

puts 2 + 2

NODE_SCOPE
table: [ ]
args: [ ]

NODE_FCALL
method_id: “puts”

NODE_CALL
method_id: “+”

NODE_LITERAL
“2”

NODE_LITERAL
“2”

Instrucciones YARV

```
putself
putobject 2
putobject 2
send :+, 1
```

puts 2 + 2

NODE_SCOPE
table: [ ]
args: [ ]

NODE_FCALL
method_id: “puts”

NODE_CALL
method_id: “+”

NODE_LITERAL
“2”

NODE_LITERAL
“2”

Instrucciones YARV

```
putself
putobject 2
putobject 2
send :+, 1
send :puts, 1
```

puts 2 + 2

NODE_SCOPE
table: [ ]
args: [ ]

NODE_FCALL
method_id: "puts"

NODE_CALL
method_id: "+"

NODE_LITERAL
"2"

NODE_LITERAL
"2"

Instrucciones YARV

```
putself
putobject 2
putobject 2
opt_plus
send :puts, 1
```

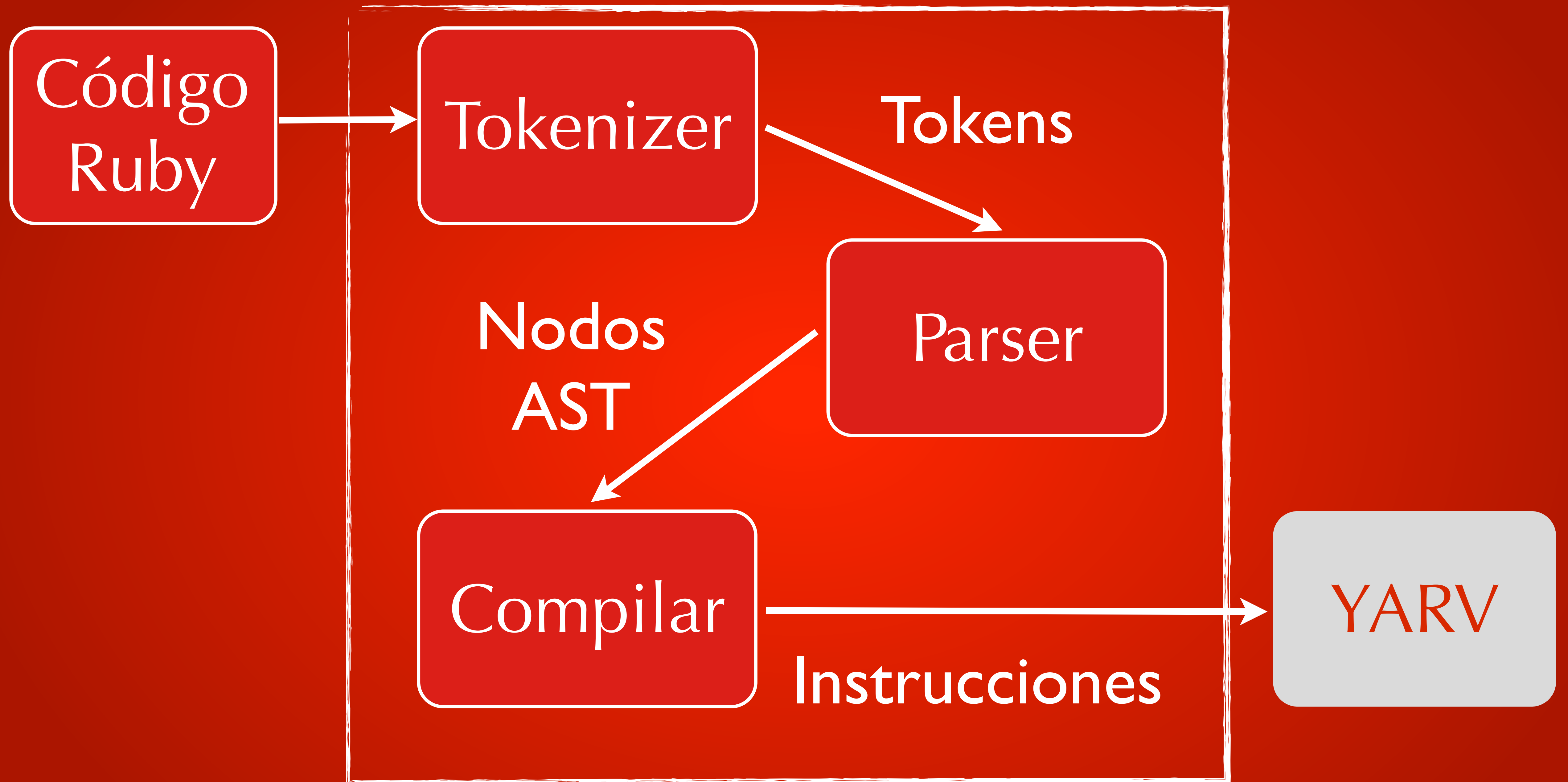
- YARV usa el AST para:
  - Generar codigo intermedio (YARV Instructions)
  - Es posible aplicar micro-optimización
  - YARV VM orientada a Stack:
    - Empujar receptor, empujar argumentos , llamar a la función

```
require ‘ripper’
require ‘awesome_print’

code = <<STR
2 + 2
STR

ap
RubyVM::InstructionSeque
nce.compile(code).disasm
```

```
== disasm:
<RubyVM::InstructionSequence:<compiled>@<com
piled>>==========
0000 trace            1
(   1)
0002 putself
0003 putobject        2
0005 putobject        2
0007 opt_plus         <ic:2>
0009 send             :puts, 1, nil, 8, <ic:
1>
0015 leave
=> nil
```


Código Ruby
Tokenizer
Tokens
Parser
Nodos AST
Compilar
Instrucciones
YARV

YARV VM orientada a stacks

- YARV usa stack interno: argumentos, valores de retorno e intermedios
- Existen la estructura rb_control_frame_t para almacenar el SP y el PC

puts 2 + 2

rb_control_frame_t

rb_control_frame_t

PC

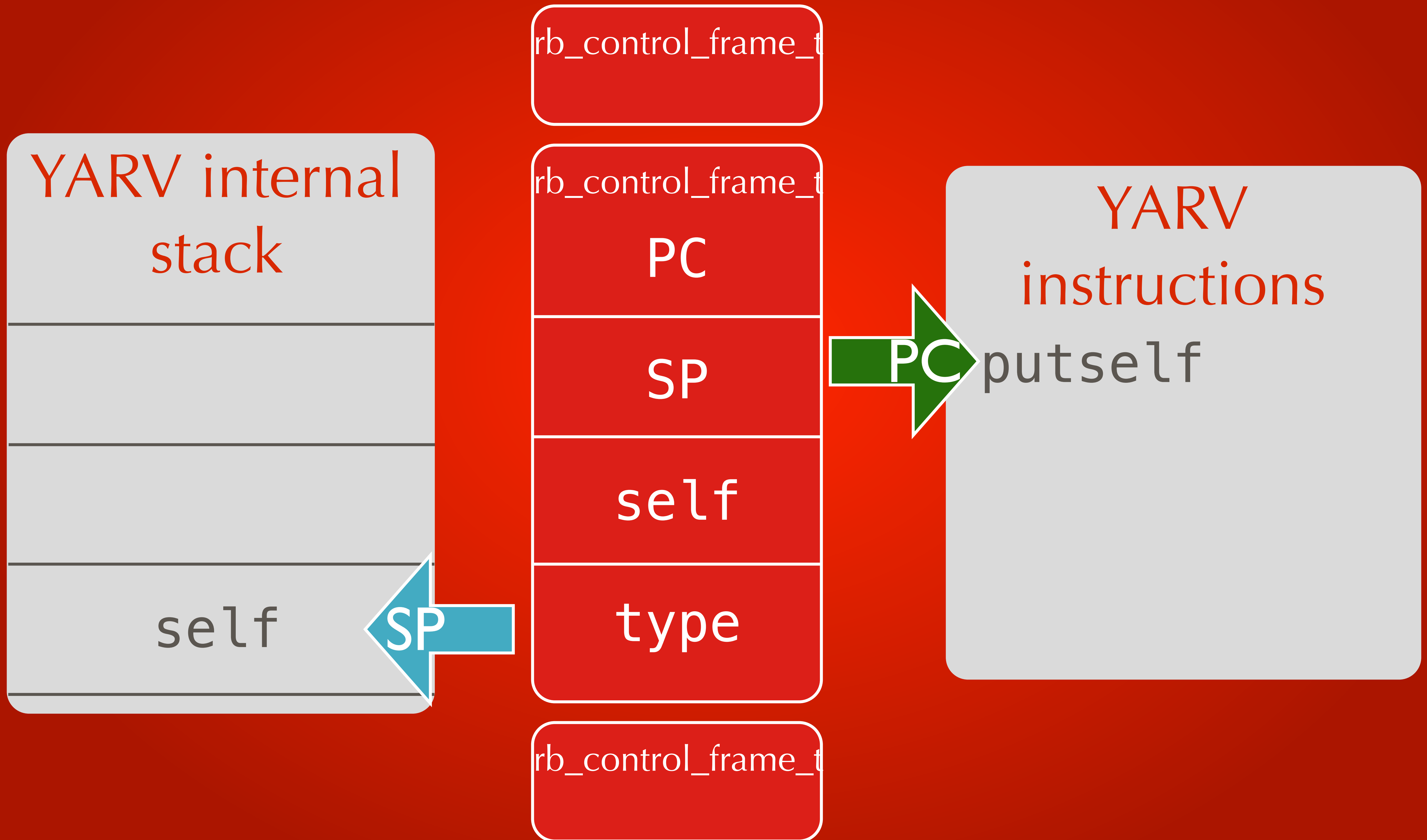
SP

self

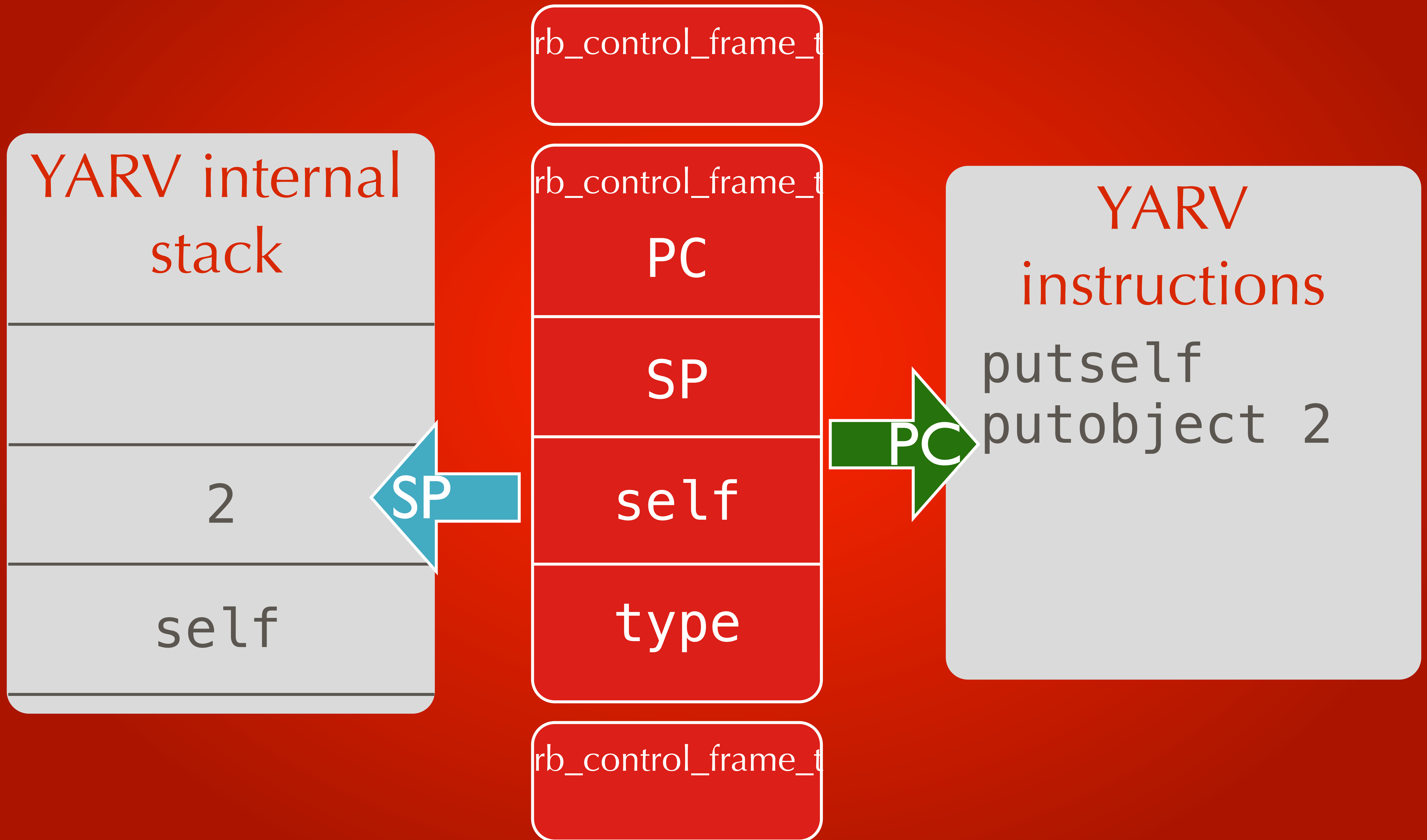
type

rb_control_frame_t

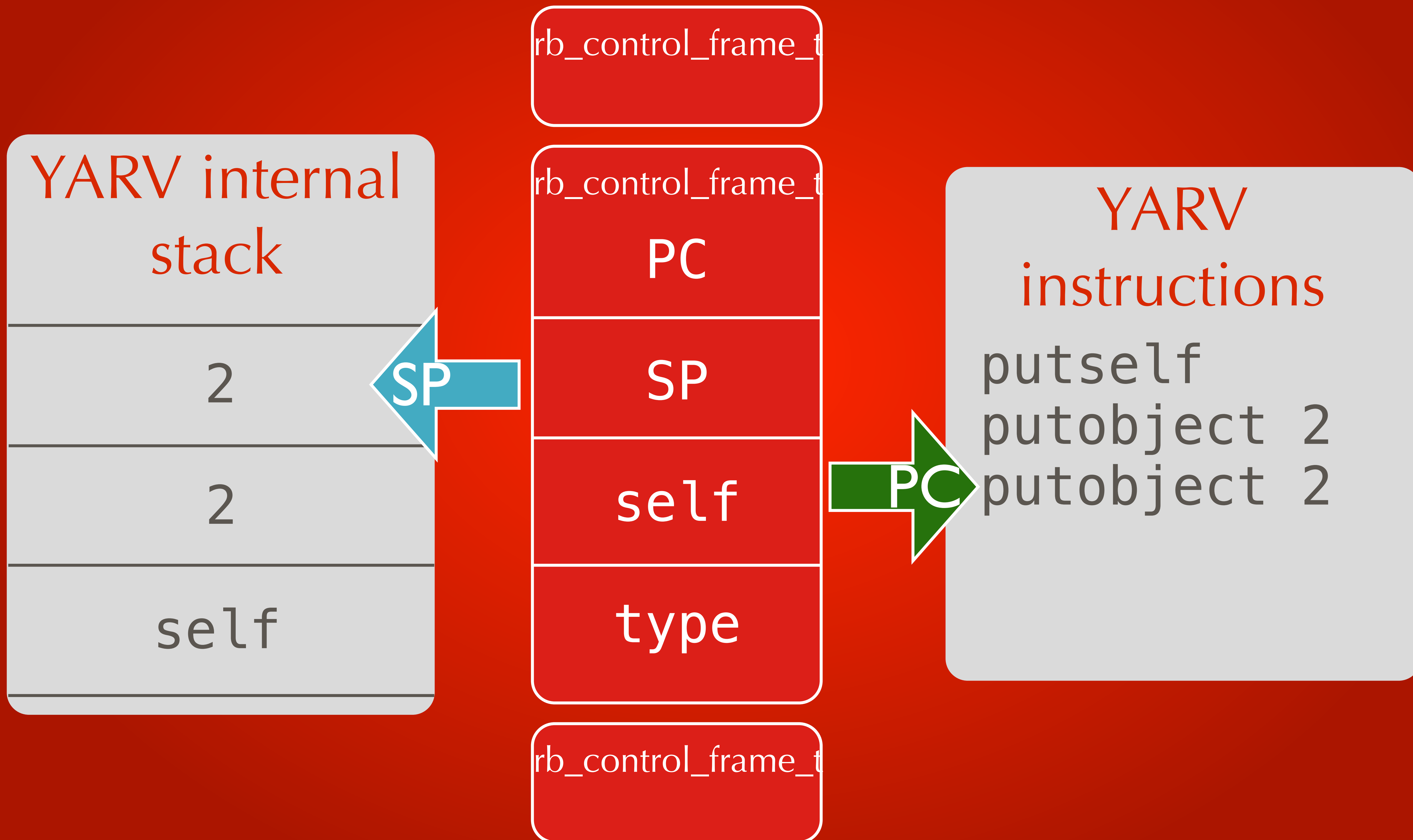
puts 2 + 2



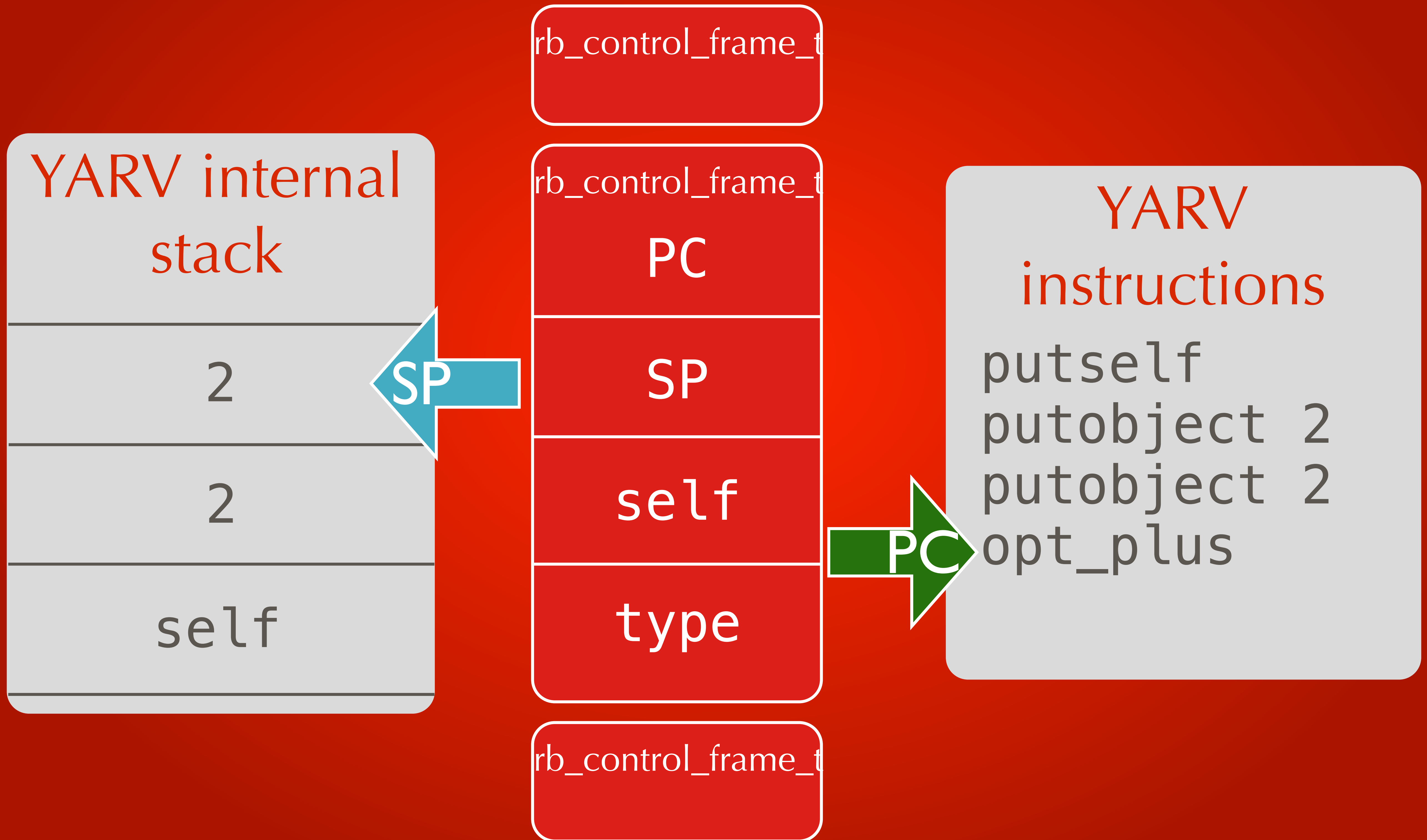
puts 2 + 2



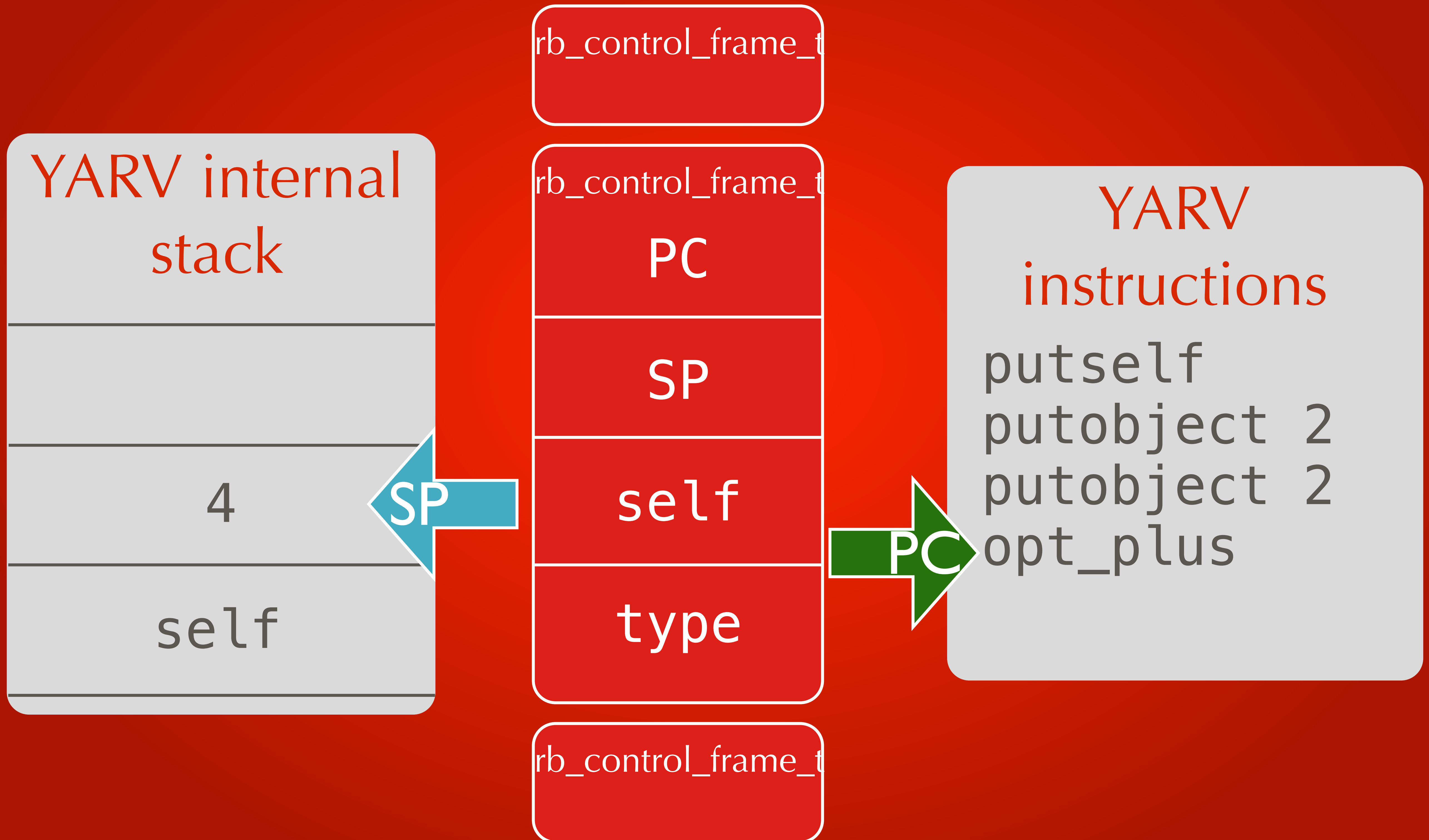
puts 2 + 2



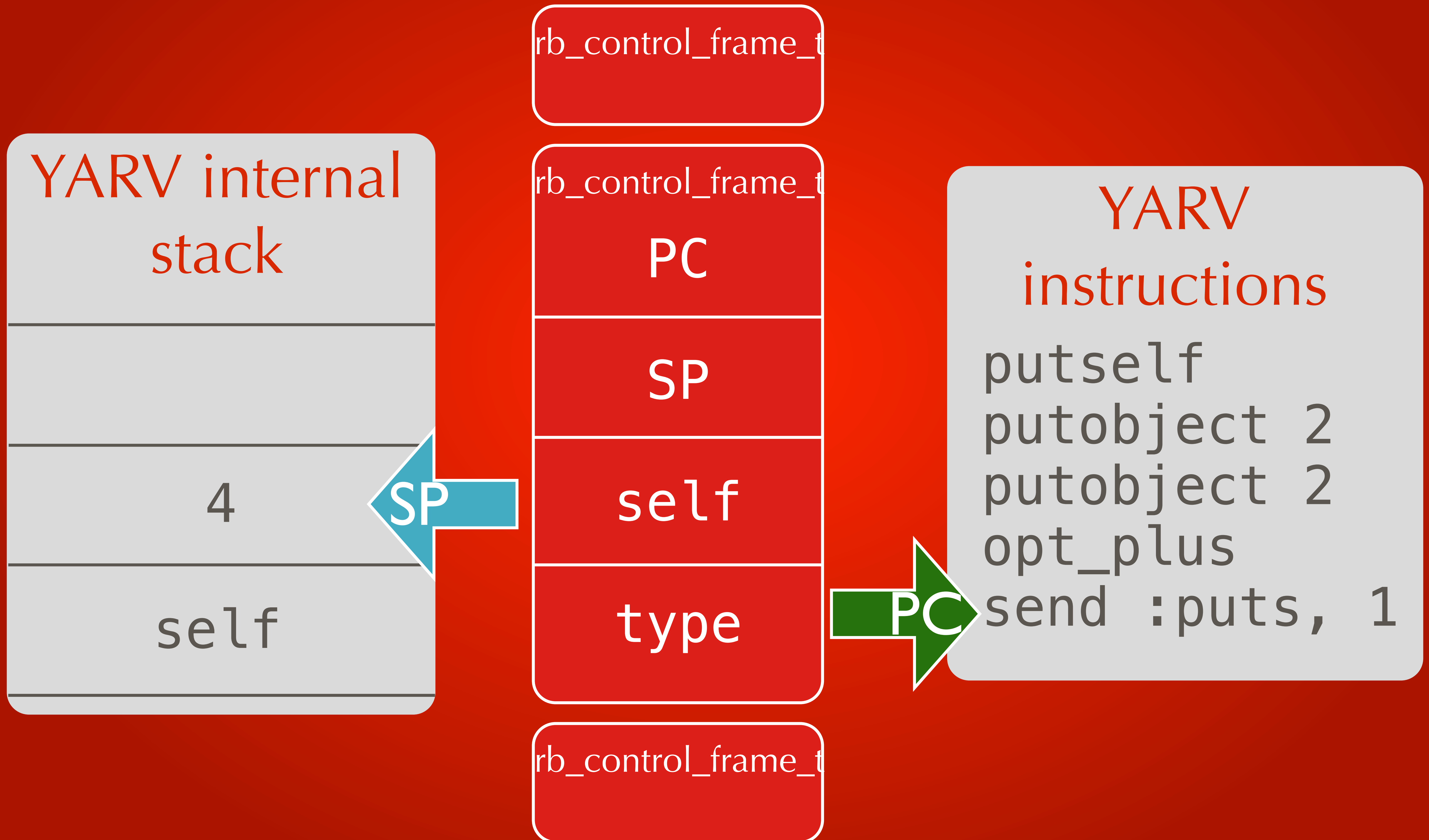
puts 2 + 2



puts 2 + 2



puts 2 + 2



YARV intructions

```
trace
putself
putstring "Hola"
send :puts, 1
leave
```

```
[C Function - "times"]
```

```
trace
putobject 10
send :times, 0, block
leave
```

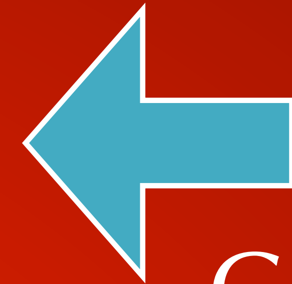
rb_control_frame_t [BLOCK]

rb_control_frame_t [FINISH]

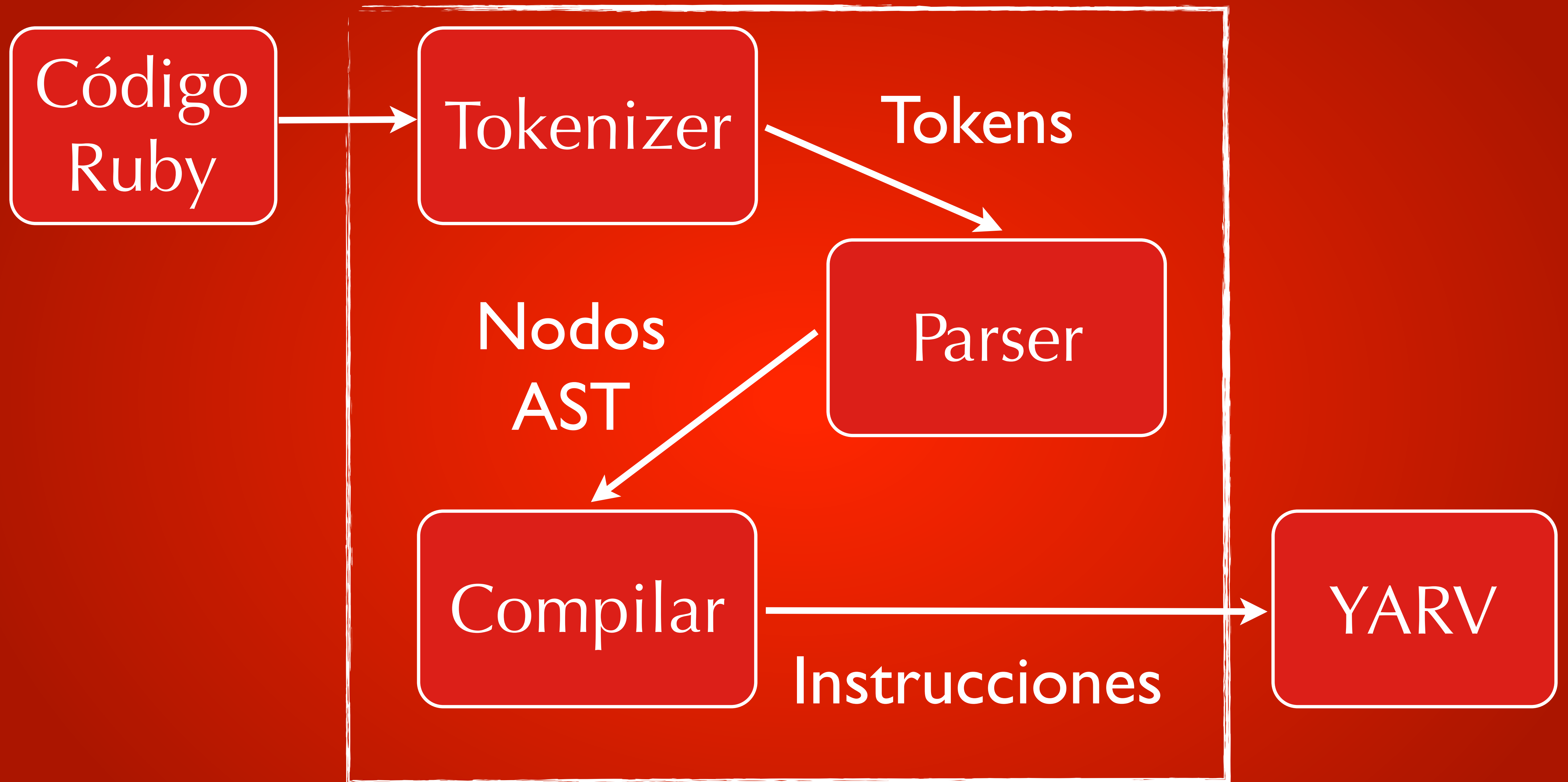
rb_control_frame_t [C_FUNC]

rb_control_frame_t [EVAL]

rb_control_frame_t [FINISH]



Control Frame Pointer (CFP)


Código Ruby
Tokenizer
Tokens
Parser
Nodos AST
Compilar
Instrucciones
YARV

Nuestro código ha sido ejecutado!

- El objetivo de YARV es mejorar el tiempo de ejecución
- YARV tiene un “warmup” mas lento que Ruby 1.8 pero a largo tiempo es mas rápido
- Solo “rascamos” YARV ligeramente

Recursos

- Libro "Ruby Under Microscope" http://patshaughnessy.net/ruby-under-a-microscope
- Blog de Pat Shaughnessy http://patshaughnessy.net/
- Videos de RubyConf 2012 http://confreaks.com/events/rubyconf2012 (Koichi Sasada, Aaron Patterson)

Photo: http://500px.com/photo/25805131

Mario Alberto Chávez
@mario_chavez