USANDO METAPROGRAMAÇÃO PARA INSTRUMENTAR PYTHON COM PYTHON

• • •

Marcel Rodrigues

2015 - PYTHON VALE

FATOS

1. Nenhum programa é perfeito.
2. Depurar código é mais difícil do que escrever código.

• • •

IDEIA

ESCREVER CÓDIGO PARA DEPURAR CÓDIGO!

SOLUÇÕES PRONTAS

- Análise Estática: PyLint, Pyflakes, ...
- Cobertura de Código: coverage.py
- Profiler: cProfile, line_profiler, ...
- Depurador: pdb, pdb++, ipdb, ...
- Outros: pycallgraph, ...

SOLUÇÃO "FAÇA VOCÊ MESMO"

...

VANTAGENS

- Menos dependência
- Mais flexibilidade
- Mais diversão!

SOLUÇÃO "FAÇA VOCÊ MESMO"

...

TÉCNICAS

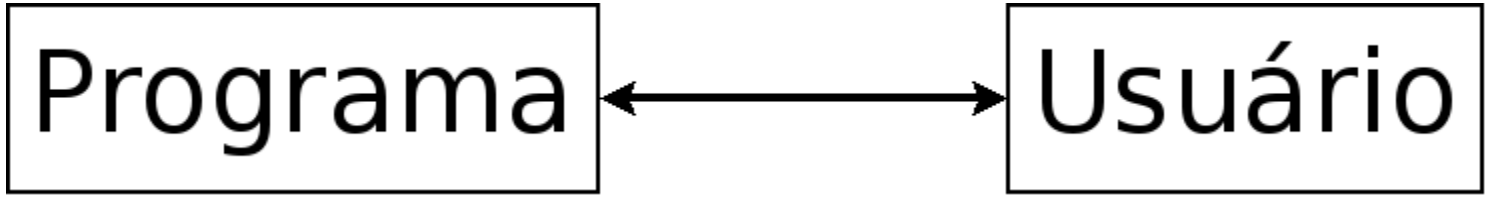
- Metaprogramação
- Instrumentação

METAPROGRAMAÇÃO

Criação de programas que têm outros programas como entrada/saída.

"Code as Data"

PROGRAMAÇÃO

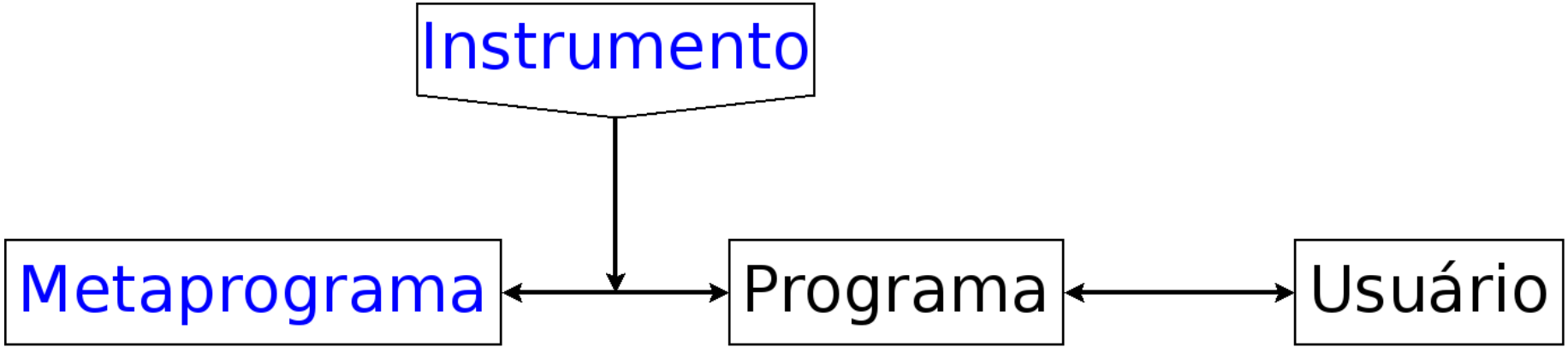


METAPROGRAMAÇÃO

INSTRUMENTAÇÃO

Monitoramento da execução de um programa para coleta de informações.

INSTRUMENTAÇÃO



SOLUÇÃO "FAÇA VOCÊ MESMO"

...

FERRAMENTAS

- Python
- e só!

FERRAMENTAS DO PYTHON

O módulo `sys` fornece:

- Instrumentação via callbacks.
- Dois níveis de instrumentação:
  - nível de função;
  - nível de linha.

INSTRUMENTAÇÃO DE FUNÇÃO

`sys.setprofile(callback)`

Define uma função a ser chamada nos seguintes eventos:

- `call`
- `return`
- `c_call`
- `c_return`
- `c_exception`

INSTRUMENTAÇÃO DE LINHA

`sys.settrace(callback)`

Define uma função a ser chamada nos seguintes eventos:

- `call`
- `return`
- `line`
- `exception`

FUNÇÃO CALLBACK

Callbacks do tipo "trace" são substituídas pelo seu valor de retorno nos seguintes eventos:

- `call`
- `line`
- `exception`

FUNÇÃO CALLBACK

`callback(frame, event, arg)`

- `frame`: informações sobre o frame em execução;
- `event`: string indicando o tipo do evento;
- `arg`: informação adicional sobre o evento.

FUNÇÃO CALLBACK

event	arg
"call"	None
"return"	Object (*)
"line"	None
"exception"	(exception, value, traceback)
"c_call"	CFunction
"c_return"	CFunction
"c_exception"	CFunction

(*) Valor a ser retornado.

OBJETOS FRAME

Atributo	Tipo	Alterável
`f_back`	Frame	Não
`f_locals`	Dict	Não
`f_code`	Code	Não
`f_lasti`	Int	Não
`f_lineno`	Int	Sim
…	…	…

OBJETOS CODE

Atributo	Tipo	Alterável
co_name	String	Não
co_filename	String	Não
co_firstlineno	Int	Não
co_argcount	Int	Não
co_varnames	(String)	Não
co_code	String	Não
...	...	...

CASOS DE USO

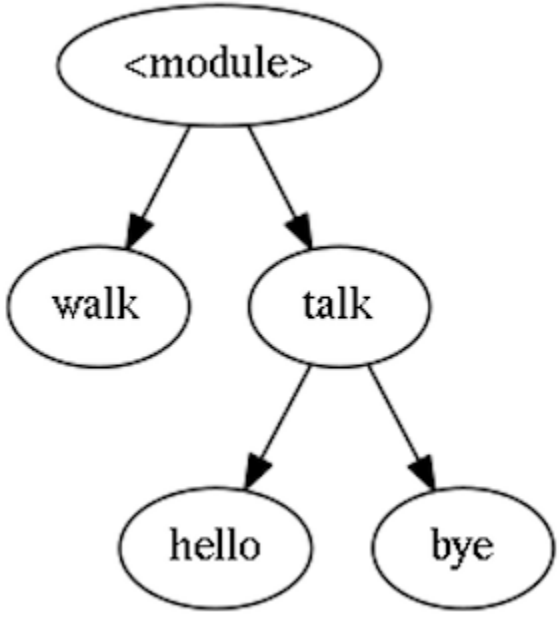
- **Grafo de Chamadas** ⇦
- Análise de Cobertura
- Profiler (perfilador)
- Depurador

GRAFO DE CHAMADAS

...

OBJETIVO: visualizar de onde as funções são chamadas.

```
def talk():
  hello()
  bye()

walk()
talk()
```



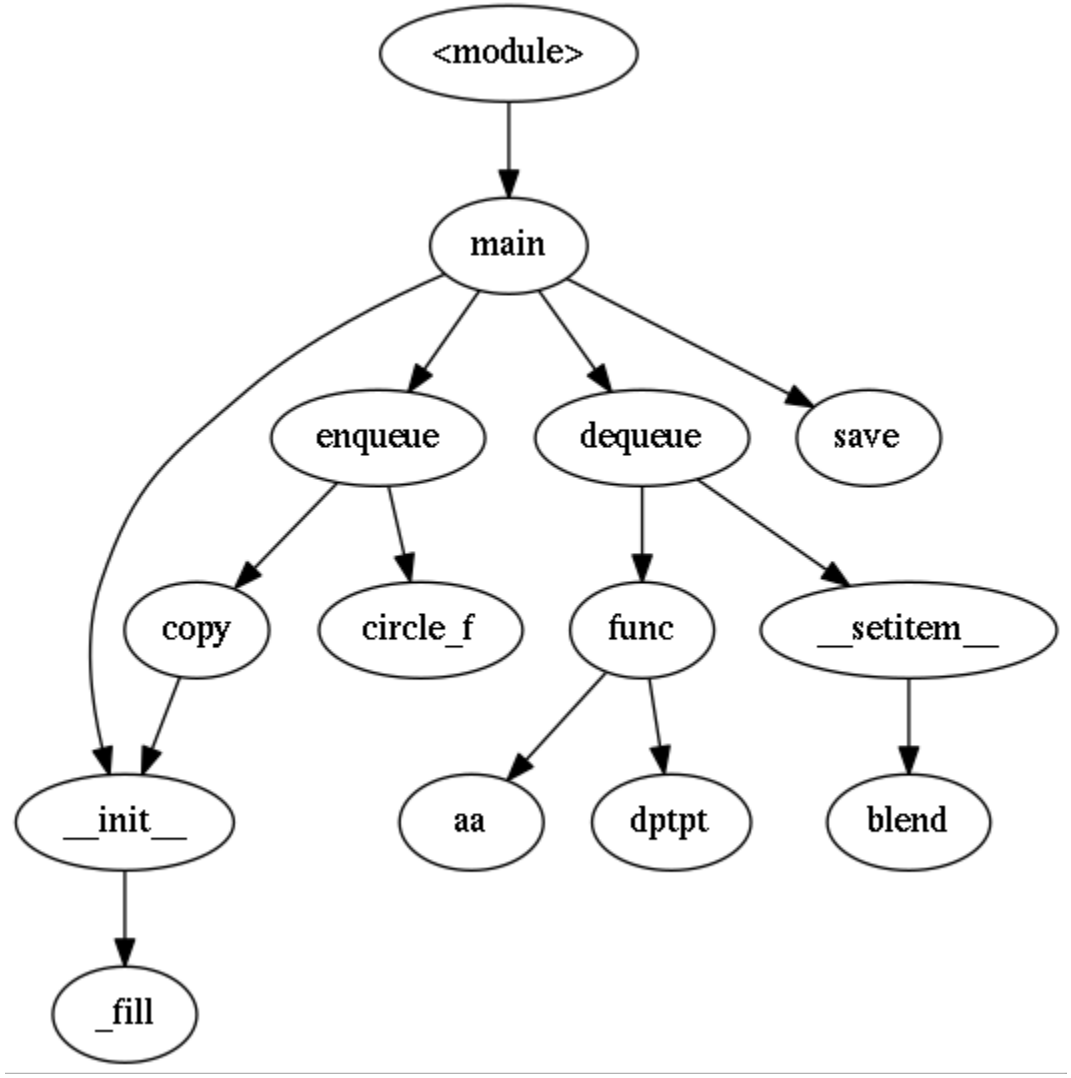
CONTEXTO PARA INSTRUMENTAR FUNÇÕES

```
def __enter__(self):
  self.old_cb = sys.getprofile()
  sys.setprofile(self.callback)
  return self

def __exit__(self, *args):
  sys.setprofile(self.old_cb)
```

COLETANDO CHAMADAS

```
def __init__(self):
  self.calls = set()

def callback(self, frame, event, arg):
  if event == 'call':
    caller = frame.f_back.f_code.co_name
    callee = frame.f_code.co_name
    self.calls.add((caller, callee))
```

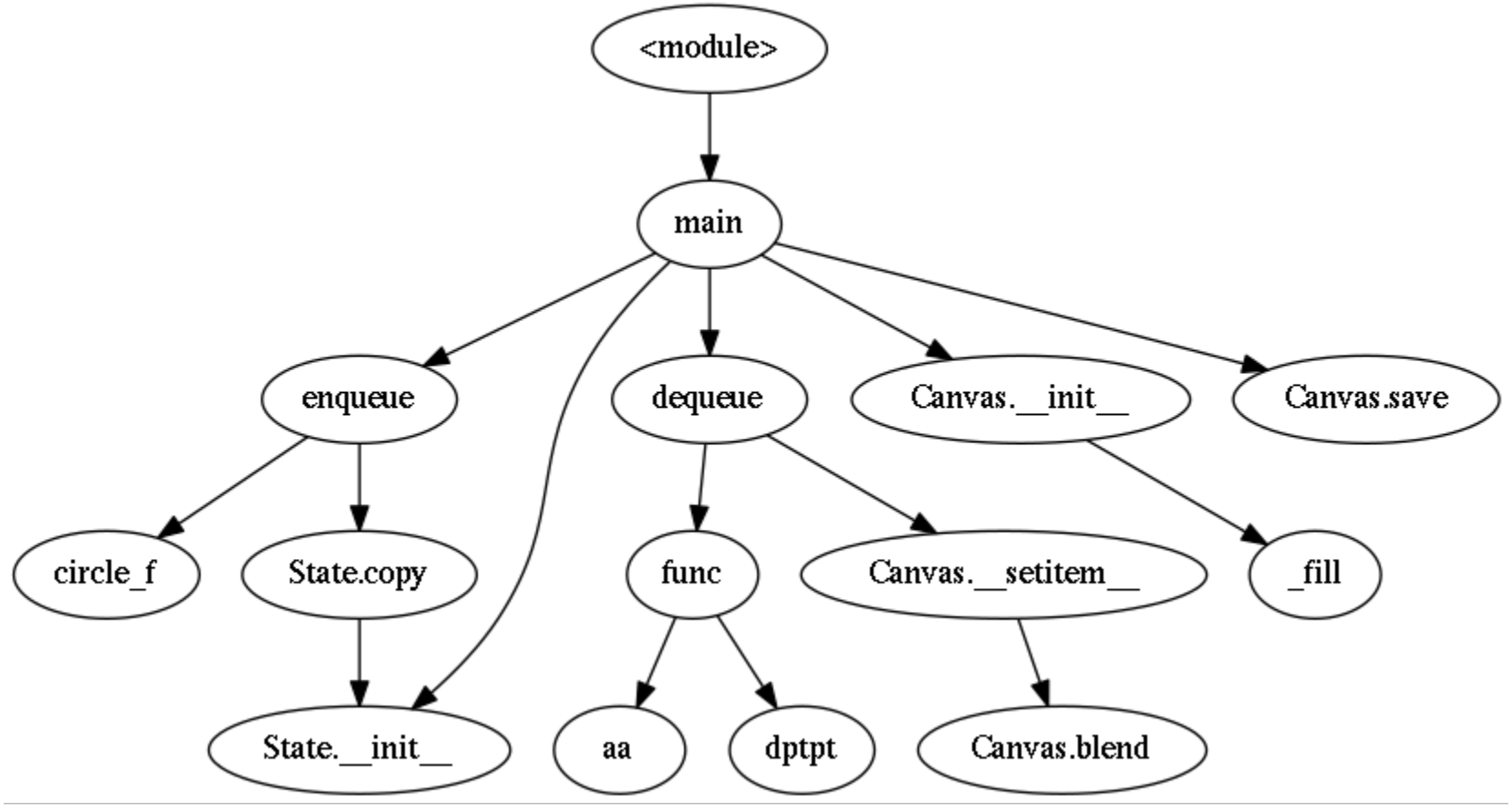
RESULTADO PRELIMINAR



IDENTIFICANDO MÉTODOS

```
def frame2name(frame):
  code = frame.f_code
  name = code.co_name
  if code.co_argcount:
    arg0 = code.co_varnames[0]
    self = frame.f_locals[arg0]
    mtd = getattr(self, name, None)
    if hasattr(mtd, "__func__"):
      if mtd.__func__.__code__ is code:
        cls = type(self).__name__
        name = "{}.{}".format(cls, name)
  return name
```

RESULTADO


<module>
main
enqueue
dequeue
Canvas.__init__
Canvas.save
circle_f
State.copy
func
Canvas.__setitem__
_fill
State.__init__
aa
dptpt
Canvas.blend

CASOS DE USO

- Grafo de Chamadas ✔
- **Análise de Cobertura** ⇦
- Profiler (perfilador)
- Depurador

ANÁLISE DE COBERTURA

...

OBJETIVO: visualizar quais linhas são executadas.

CONTEXTO PARA INSTRUMENTAR LINHAS

```
def __enter__(self):
  self.old_cb = sys.gettrace()
  sys.settrace(self.callback)
  return self

def __exit__(self, *args):
  sys.settrace(self.old_cb)
```

COLETANDO LINHAS EXECUTADAS

```
def __init__(self):
  self.covered = {}

def callback(self, frame, event, arg):
  if event == 'line':
    fname = frame.f_code.co_filename
    lineno = frame.f_lineno
    s = self.covered.get(fname, set())
    s.add(lineno)
    self.covered[fname] = s
  return self.callback
```

RESULTADO

```
elif token == "setwidth":
  state.width = stack.pop()
elif token == "setalpha":
  state.alpha = stack.pop()
elif token == "setfill":
  state.fill = tuple(stack[-3:])
  stack[-3:] = []
elif token == "setstroke":
  state.stroke = tuple(stack[-3:])
  stack[-3:] = []
elif token == "line":
  x0, y0, x1, y1 = stack[-4:]
```

CASOS DE USO

- Grafo de Chamadas ✔
- Análise de Cobertura ✔
- **Profiler (perfilador)** ⇦
- Depurador

PROFILER

...

OBJETIVO: medir o desempenho de cada parte do código.

COLETANDO DESEMPENHO DE FUNÇÕES

```
def __init__(self, instrument):
  self.instrument = instrument
  self.stack = []
  self.traces = {}
```

COLETANDO DESEMPENHO DE FUNÇÕES

```
def callback(self, frame, event, arg):
  file = frame.f_code.co_filename
  if file == __file__ or file[0] in "</":
    return
  if event == 'call':
    self.push()
  elif event == 'return':
    func = frame2name(frame)
    key = (file, func)
    self.pop(key)
```

COLETANDO DESEMPENHO DE FUNÇÕES

```
def push(self):
    before = self.instrument()
    self.stack.append(before)

def pop(self, key):
    after = self.instrument()
    before = self.stack.pop()
    delta = after - before
    trace = self.traces.get(key, [])
    trace.append(delta)
    self.traces[key] = trace
```

RESULTADO

```
file:function                          value
app.py:<module>                        7.81
app.py:main                            7.80
app.py:dequeue                         6.07
app.py:func                            3.10
app.py:Canvas.__setitem__              2.06
app.py:Canvas.__init__                 1.73
app.py:_fill                           1.00
app.py:Canvas.blend                    0.82
app.py:dptpt                           0.75
app.py:aa                              0.75
```

CASOS DE USO

- Grafo de Chamadas ✔
- Análise de Cobertura ✔
- Profiler (perfilador) ✔
- **Depurador** ⇦

DEPURADOR

...

OBJETIVO: inspecionar a execução do código passo-a-passo.

LINHA DE COMANDO

```
def prompt(self, globs=None, locs=None):
  while True:
    args = input("> ").split() or args
    cmd = args.pop(0)
    if "break".startswith(cmd):
      self.breaks.add(int(args[0]))
    elif "continue".startswith(cmd):
      self.state = "cont"
      break
    ...
    else:
      print("tente outra vez")
```

CALLBACK PARA EVENTO "TRACE"

```
def callback(self, frame, event, arg):
  ...
  if event == 'line':
    if self.state == "cont":
      if lineno in self.breaks:
        print(lineno, line.rstrip())
        self.prompt(globs, locs)
    elif self.state == "step":
      print(lineno, line.rstrip())
      self.prompt(globs, locs)
  return self.callback
```

RESULTADO

```
$ rlwrap python debug.py test.py
> break 5
> continue
5             i += 1
> print e
2
> c
5             i += 1
> p e
2.5
```

CASOS DE USO

- Grafo de Chamadas ✔
- Análise de Cobertura ✔
- Profiler (perfilador) ✔
- Depurador ✔

GOTO CONSIDERADO POSSÍVEL

...

OBJETIVO: implementar o comando que está faltando no Python.

O COMEÇO

```
def goto(x):
   return x
```

O TRUQUE

```
def localtrace(self, frame, event, arg):
  if event == 'return':
    if frame.f_code.co_name == 'goto':
      self.target = arg
  elif event == 'line':
    if self.target is not None:
      frame.f_lineno = self.target
      self.target = None
  return self.localtrace
```

O TESTE

```
 1: # BASIC MODE
 2: from goto import goto
 3:
 4: def main():
 5:     i = 1
 6:     print(i)
 7:     i += 1
 8:     if i <= 5:
 9:         goto(6)
10:     return
11:
12: main()
```

O RESULTADO

```
$ python basic.py
1
2
3
4
5
$
```

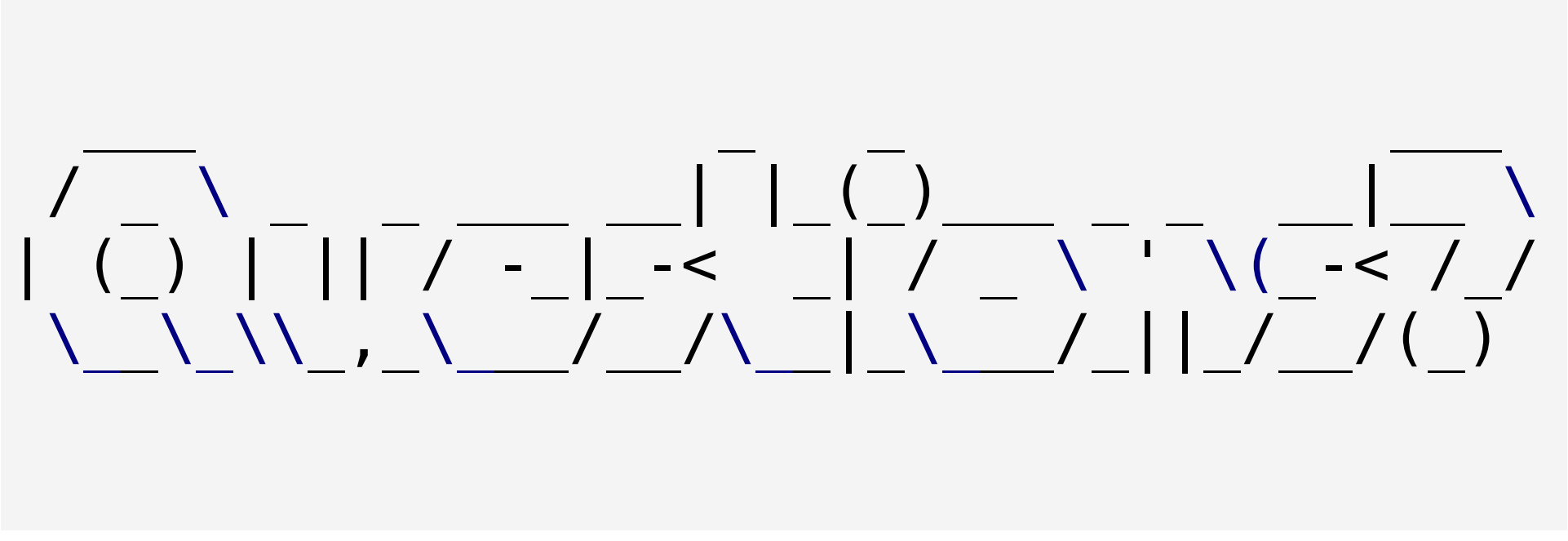
SUCESSO!

- Grafo de Chamadas ✔
- Análise de Cobertura ✔
- Profiler (perfilador) ✔
- Depurador ✔

...

GOTO ✔

FIM



github.com/lecram