Oficina de Linguagens de Programação

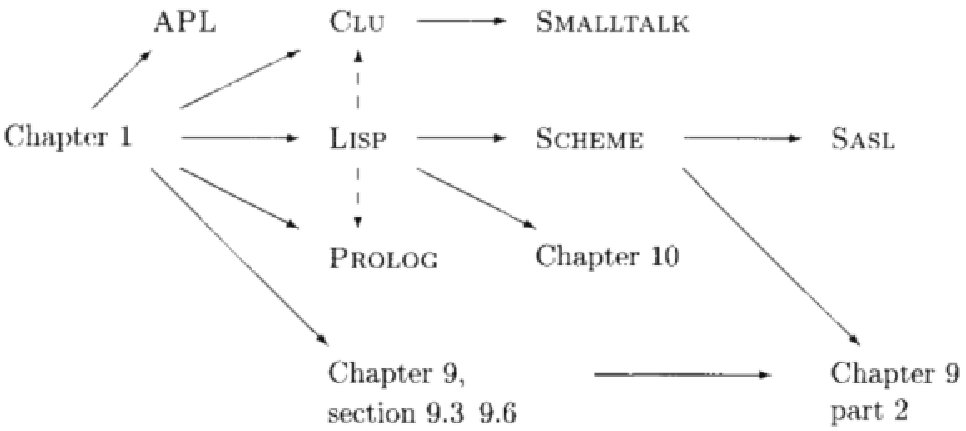
Garoa Hacker Clube
novembro/dezembro de 2018

Facilitação: @ramalhoorg

A ideia desta oficina

“A ideia deste livro é aprender sobre linguagens de programação tanto programando nelas como estudando interpretadores para elas.”

Samuel Kamin, University of Illinois



APL

CLU

SMALLTALK

Chapter 1

LISP

SCHEME

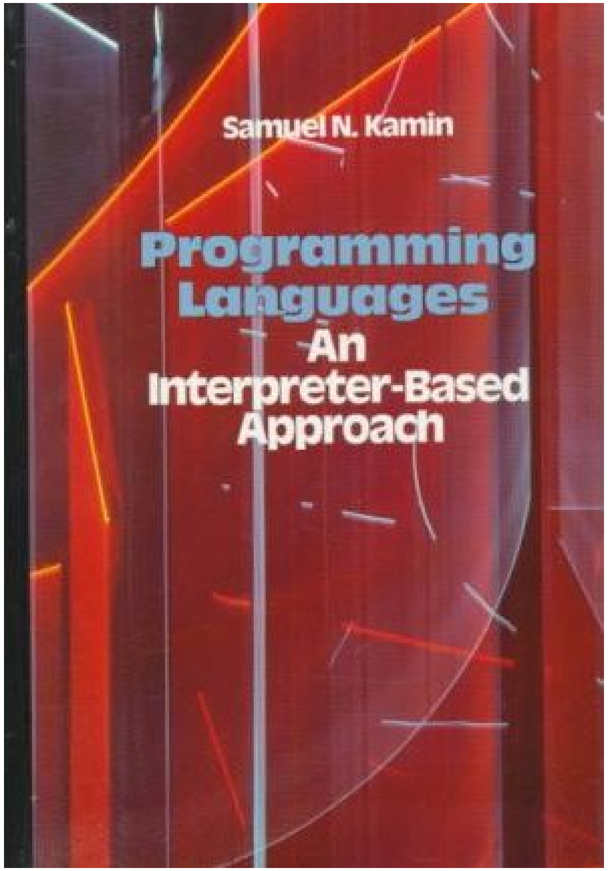
SASL

PROLOG

Chapter 10

Chapter 9,
section 9.3 9.6

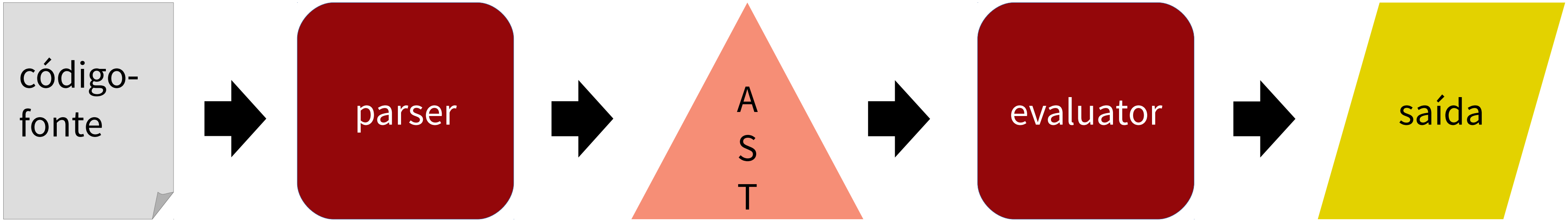
Chapter 9
part 2



Os interpretadores de Kamin

cap.	linguagem	principais características implementadas
1	Pascal	expressoes x comandos, word-at-a-time
2	Lisp	escopo dinâmico, funções de 1ª classe
3	APL	operações vetoriais
4	Scheme	escopo estático, closures
5	SASL	programação funcional, avaliação preguiçosa
6	CLU	call-by-sharing, ADTs
7	Smalltalk	programação orientada a objetos com classes
8	Prolog	programação lógica

Partes essenciais de um interpretador



```
(* 6 (+ 3 4))
```

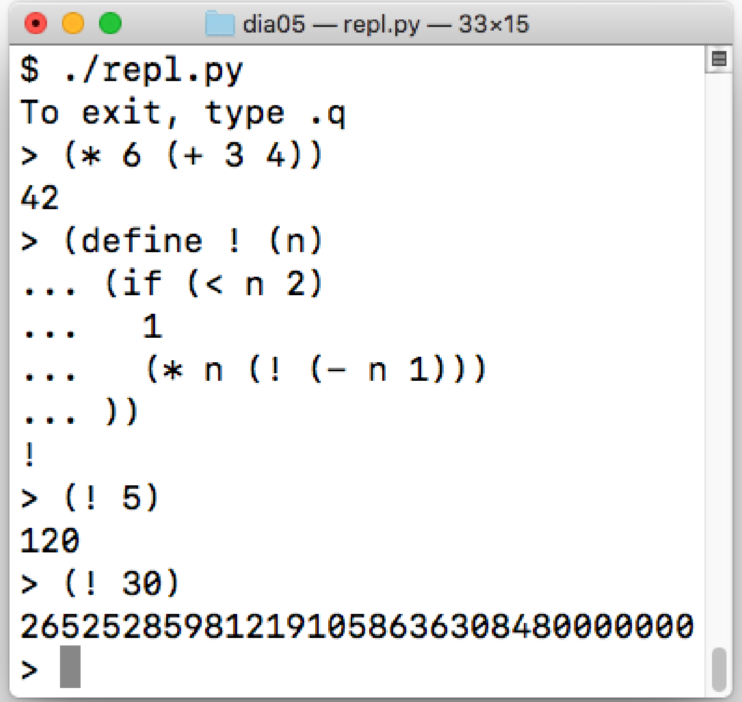
```
['*', 6, ['+', 3, 4]]
```

```
42
```

REPL: Read-Eval-Print-Loop

Laço interativo:

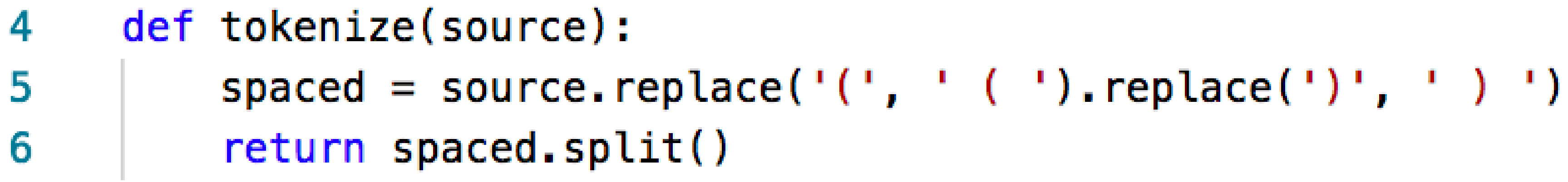
- **lê uma expressão (read)**
- **avalia a expressão (evaluate)**
- **exibe o resultado (print)**



```
$ ./repl.py
To exit, type .q
> (* 6 (+ 3 4))
42
> (define ! (n)
...  (if (< n 2)
...    1
...    (* n (! (- n 1)))
... ))
!
> (! 5)
120
> (! 30)
265252859812191058636308480000000
> 
```

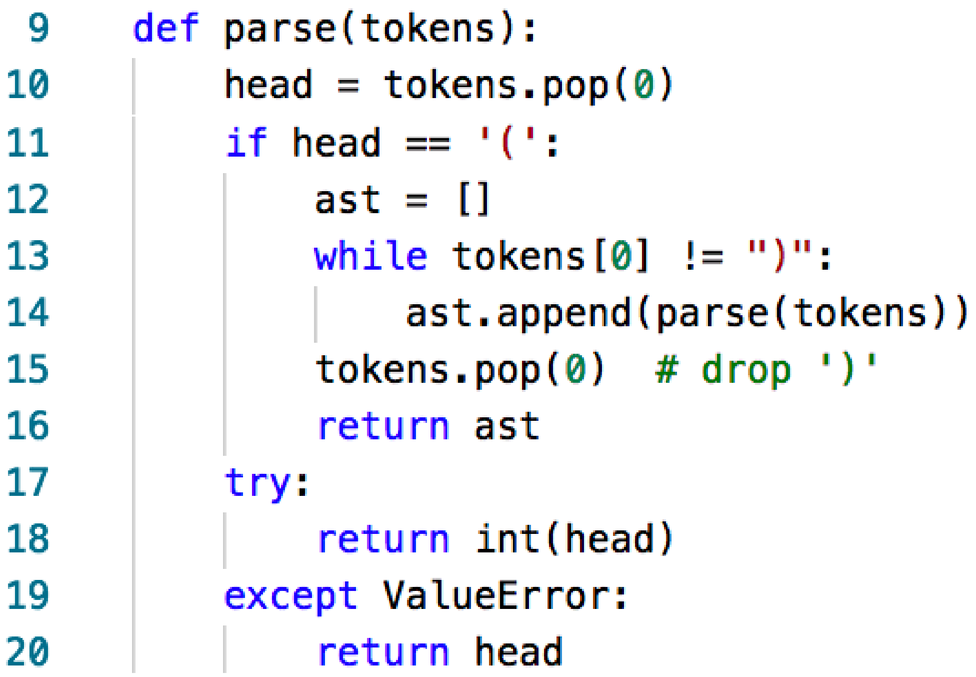
implementação da calculadora aritmética

tokenize



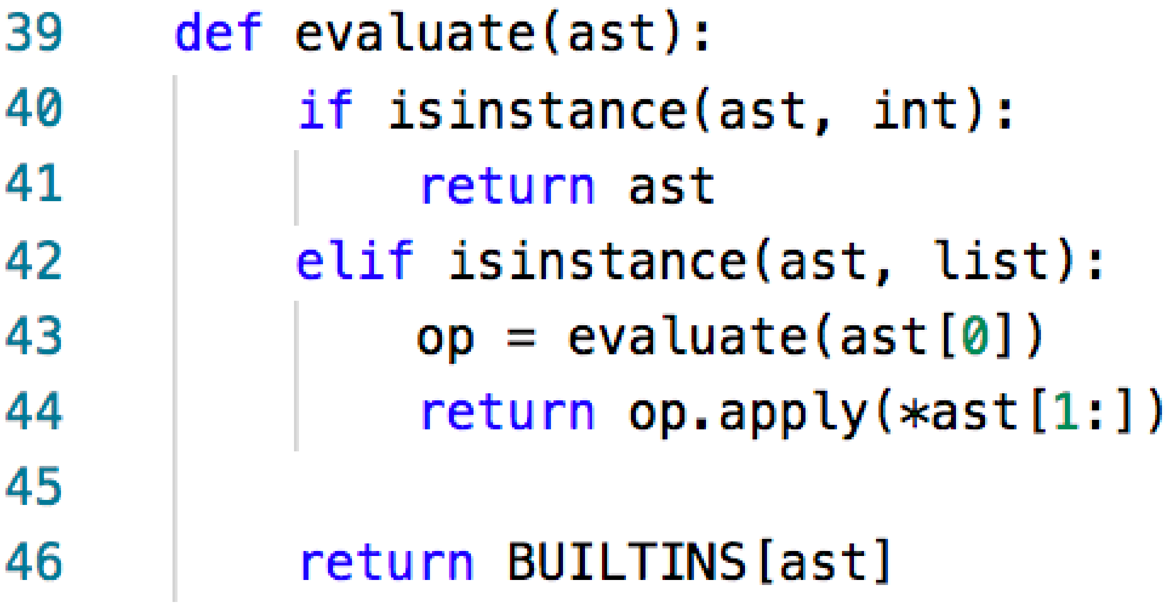
```python
def tokenize(source):
    spaced = source.replace('(', ' ( ').replace(')', ' ) ')
    return spaced.split()
```

parse: só caminhos felizes



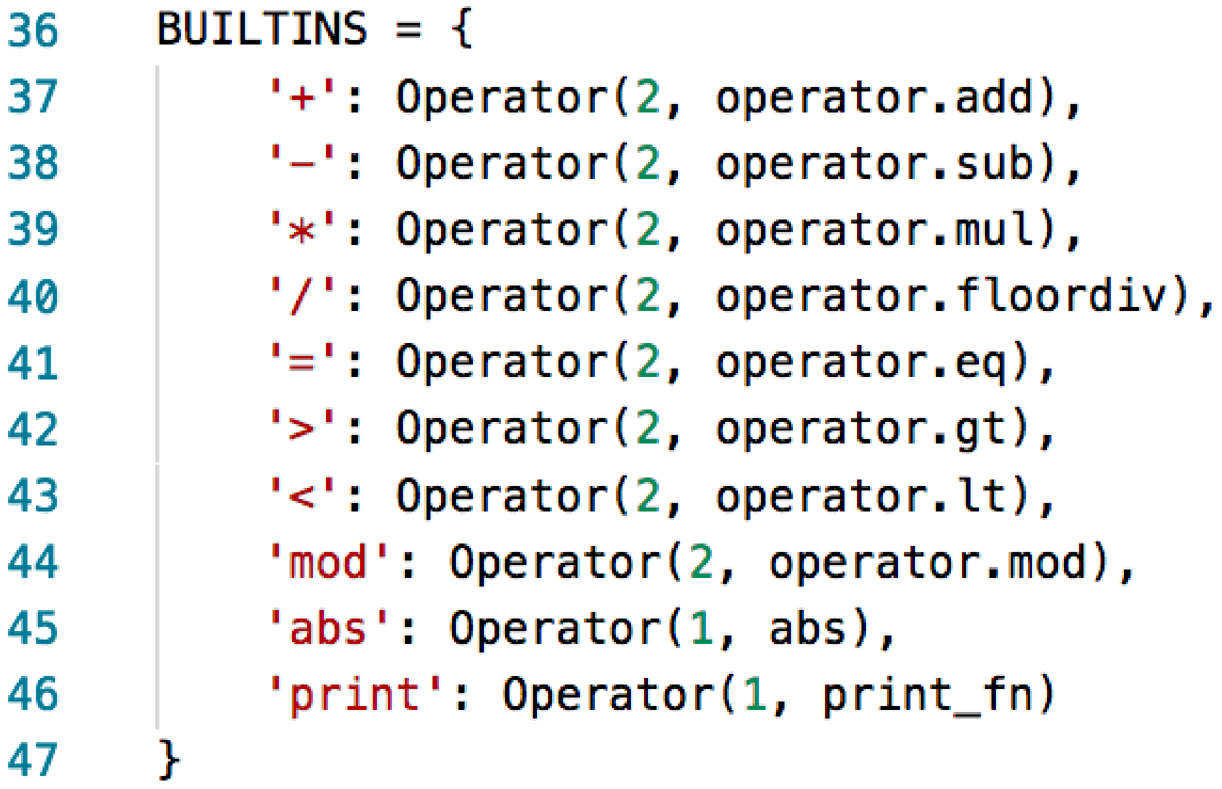
```python
def parse(tokens):
    head = tokens.pop(0)
    if head == '(':
        ast = []
        while tokens[0] != ")":
            ast.append(parse(tokens))
        tokens.pop(0)  # drop ')'
        return ast
    try:
        return int(head)
    except ValueError:
        return head
```

evaluate: só caminhos felizes



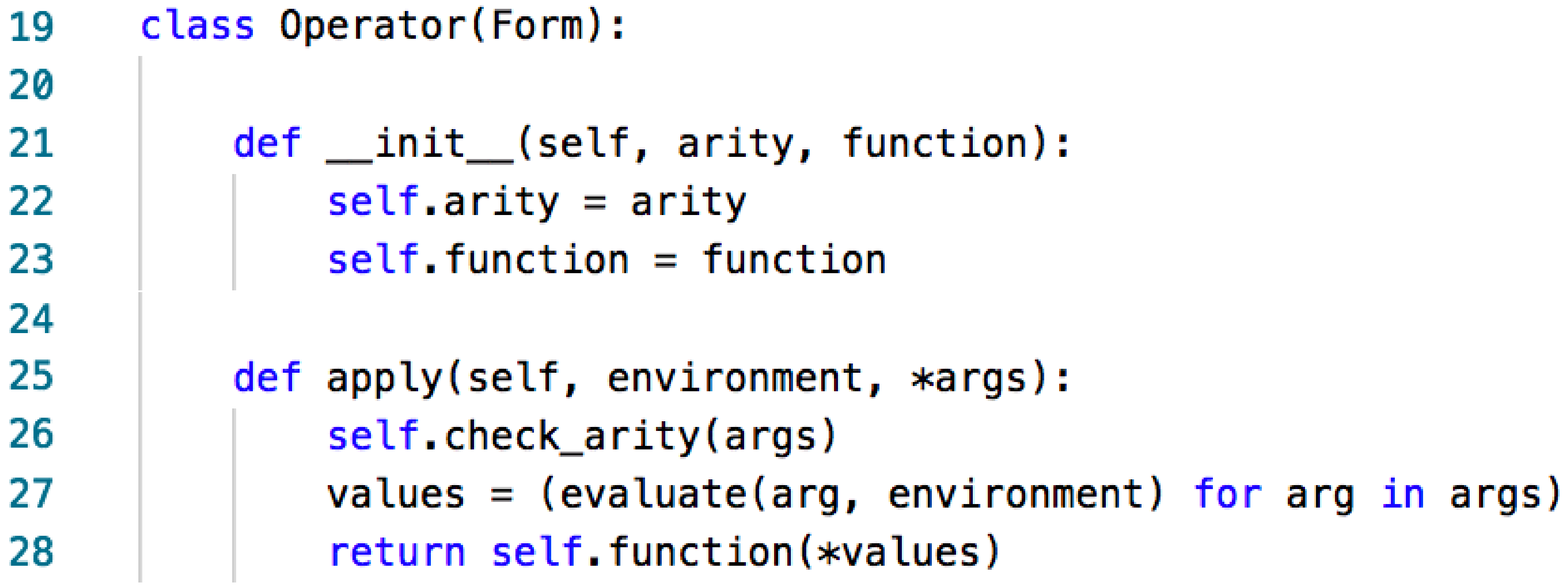
```python
def evaluate(ast):
    if isinstance(ast, int):
        return ast
    elif isinstance(ast, list):
        op = evaluate(ast[0])
        return op.apply(*ast[1:])

    return BUILTINS[ast]
```

BUILTINS



```
BUILTINS = {
        '+': Operator(2, operator.add),
        '-': Operator(2, operator.sub),
        '*': Operator(2, operator.mul),
        '/': Operator(2, operator.floordiv),
        '=': Operator(2, operator.eq),
        '>': Operator(2, operator.gt),
        '<': Operator(2, operator.lt),
        'mod': Operator(2, operator.mod),
        'abs': Operator(1, abs),
        'print': Operator(1, print_fn)
}
```

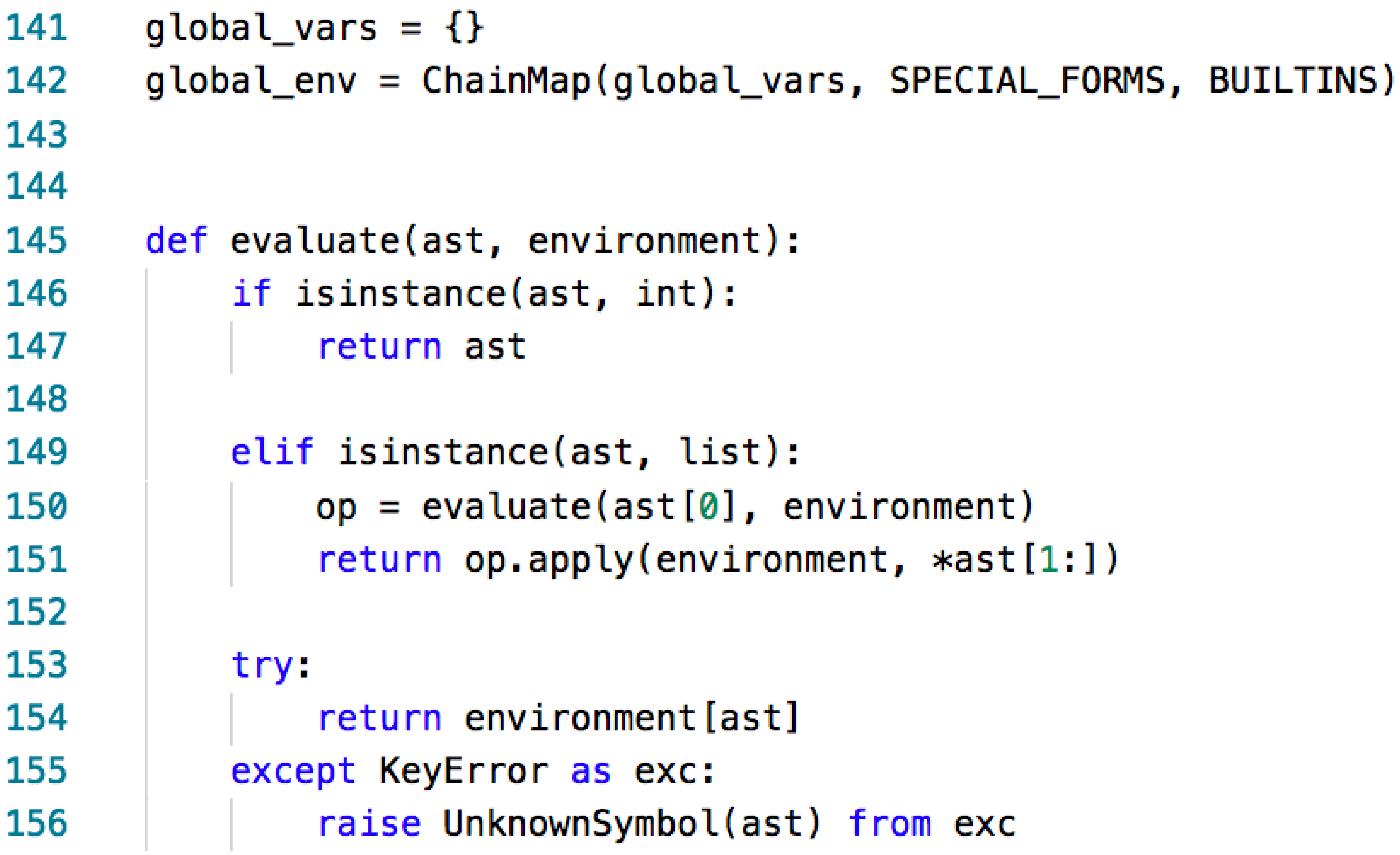
Operator



```
class Operator(Form):

    def __init__(self, arity, function):
        self.arity = arity
        self.function = function

    def apply(self, environment, *args):
        self.check_arity(args)
        values = (evaluate(arg, environment) for arg in args)
        return self.function(*values)
```

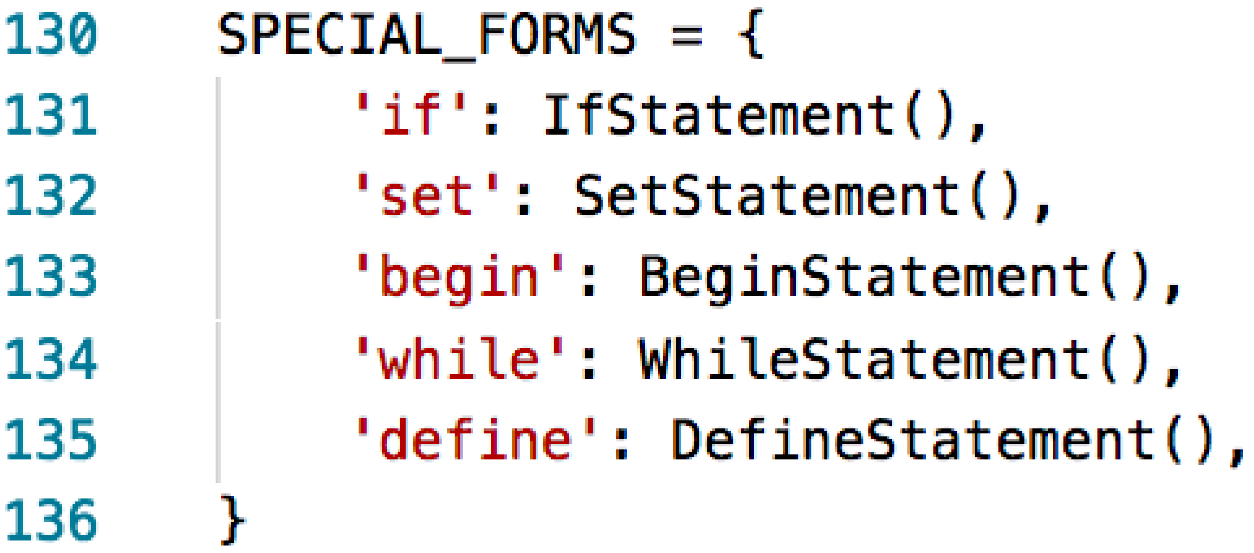
implementação de SubPascal

evaluate



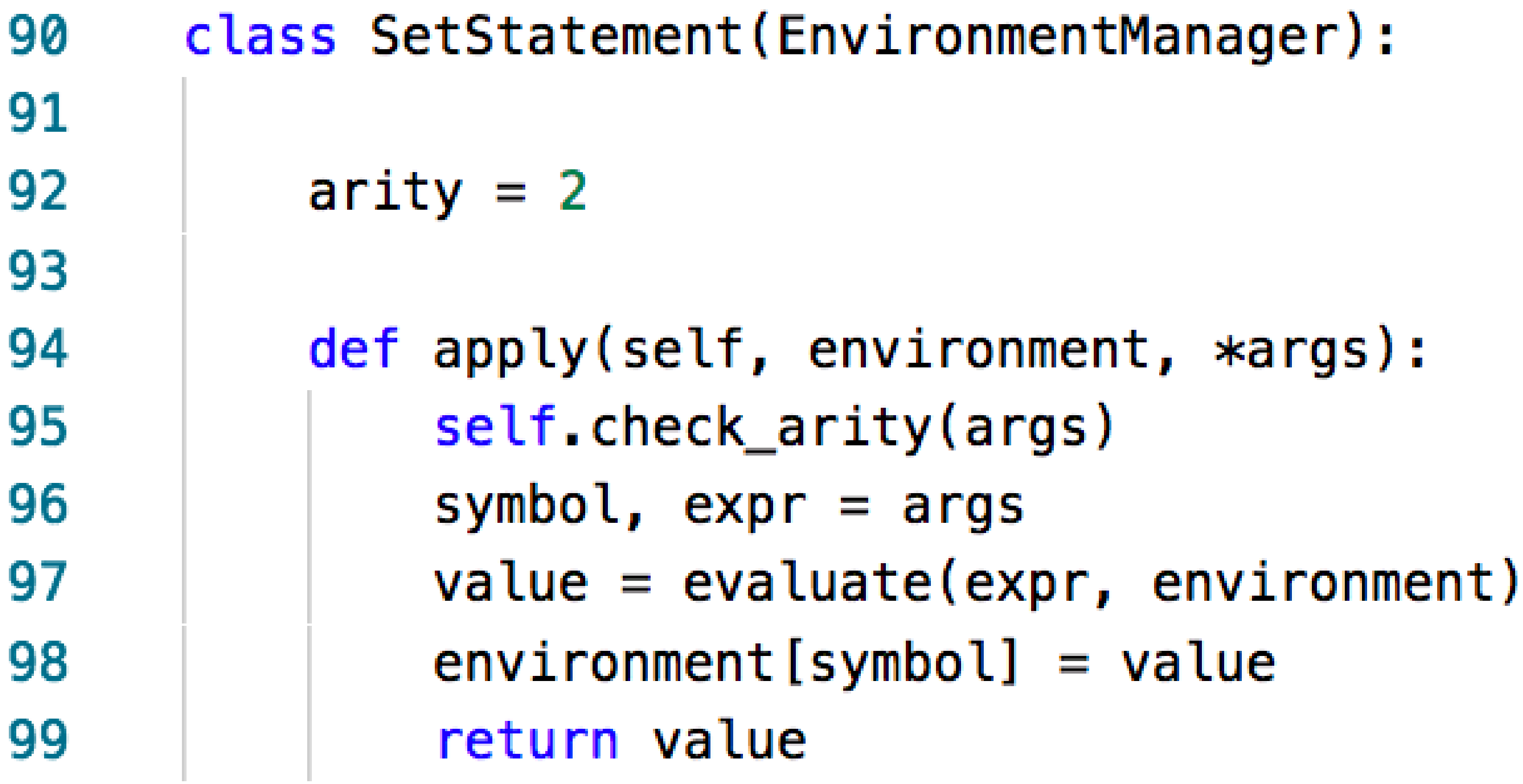
```python
global_vars = {}
global_env = ChainMap(global_vars, SPECIAL_FORMS, BUILTINS)


def evaluate(ast, environment):
    if isinstance(ast, int):
        return ast

    elif isinstance(ast, list):
        op = evaluate(ast[0], environment)
        return op.apply(environment, *ast[1:])

    try:
        return environment[ast]
    except KeyError as exc:
        raise UnknownSymbol(ast) from exc
```

SPECIAL_FORMS



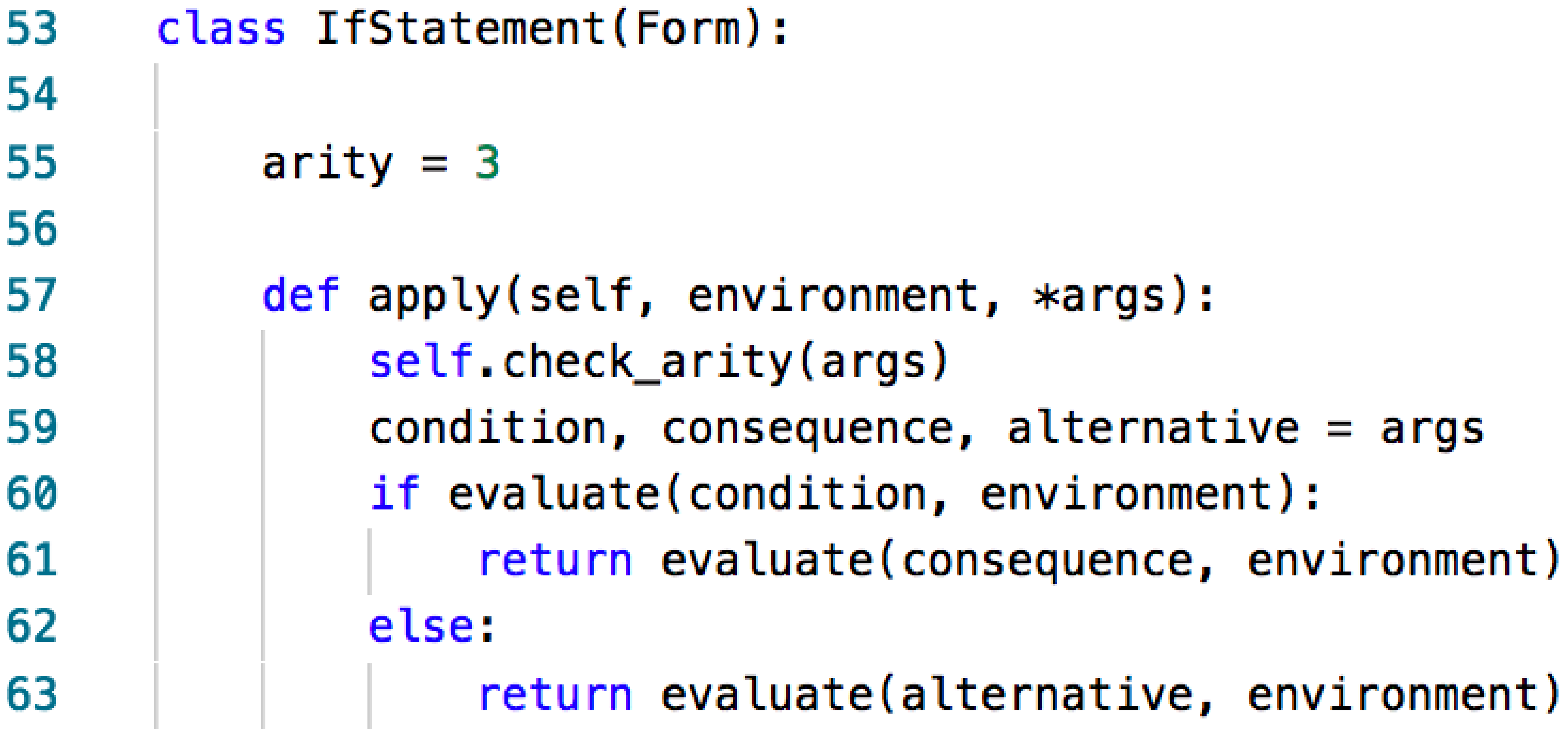
```
SPECIAL_FORMS = {
    'if': IfStatement(),
    'set': SetStatement(),
    'begin': BeginStatement(),
    'while': WhileStatement(),
    'define': DefineStatement(),
}
```

SetStatement



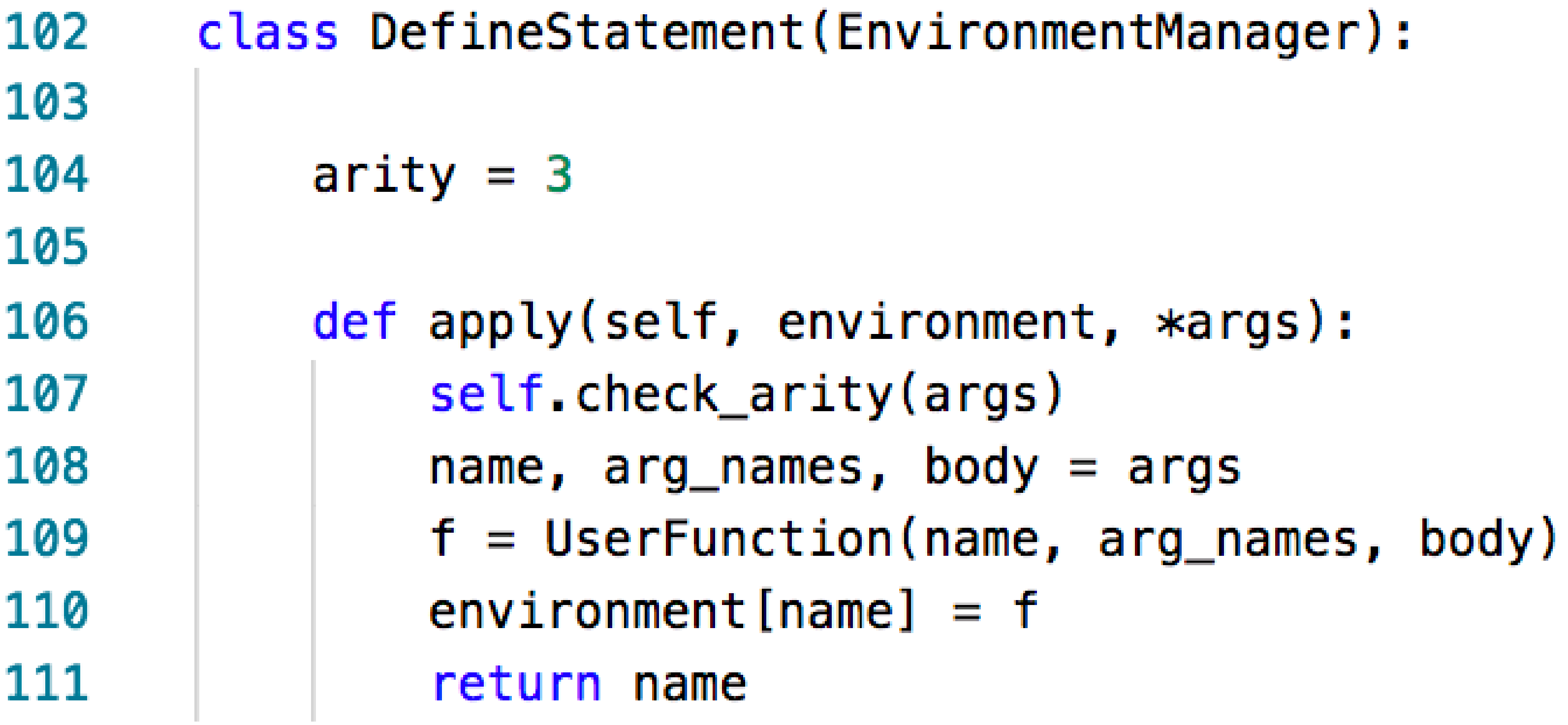
```
class SetStatement(EnvironmentManager):

    arity = 2

    def apply(self, environment, *args):
        self.check_arity(args)
        symbol, expr = args
        value = evaluate(expr, environment)
        environment[symbol] = value
        return value
```

IfStatement



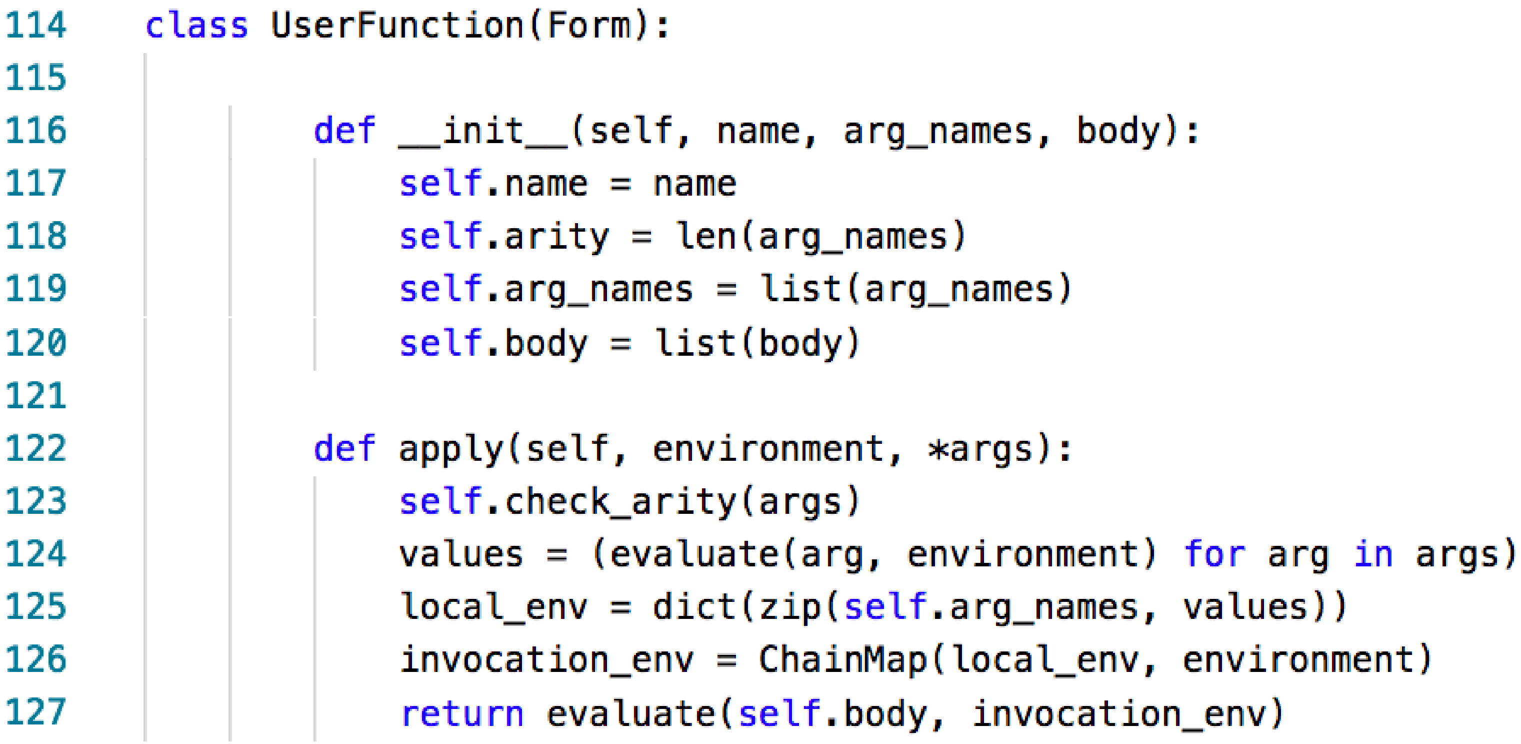
```
class IfStatement(Form):

    arity = 3

    def apply(self, environment, *args):
        self.check_arity(args)
        condition, consequence, alternative = args
        if evaluate(condition, environment):
            return evaluate(consequence, environment)
        else:
            return evaluate(alternative, environment)
```

DefineStatement



```python
class DefineStatement(EnvironmentManager):

    arity = 3

    def apply(self, environment, *args):
        self.check_arity(args)
        name, arg_names, body = args
        f = UserFunction(name, arg_names, body)
        environment[name] = f
        return name
```

UserFunction



```python
class UserFunction(Form):

    def __init__(self, name, arg_names, body):
        self.name = name
        self.arity = len(arg_names)
        self.arg_names = list(arg_names)
        self.body = list(body)

    def apply(self, environment, *args):
        self.check_arity(args)
        values = (evaluate(arg, environment) for arg in args)
        local_env = dict(zip(self.arg_names, values))
        invocation_env = ChainMap(local_env, environment)
        return evaluate(self.body, invocation_env)
```