# Sucuri

**Uma linguagem baseada em Python**

João Paulo T. I. Z., Ranieri S. A., William K. A.

26 de Setembro de 2017

## A linguagem

A linguagem é planejada tendo como base algumas ideias de Python, Javascript e Haskell. Para geração do analisador léxico, foi utilizada as ferramentas FLEX (para especificação do léxico) e BISON (para gerar o código-fonte do analisador).

Exemplo de código válido na linguagem Sucuri:

```
# Geometry example module.

# Simple Point class
export class Point
    let x = 0
    let y = 0

    let new(self, x, y)
        self.x = x
        self.y = y

    let __sub__(self, b)
        return Vector(b.x - self.x, b.y - self.y)


# Alias example
export let Vector = Point

# Distance from a to b
export let distance(a, b)
    return b - a

# Simple Rectangle class
export class Rectangle
    let top_left = Point(0, 0)
    let bottom_right = Point(0, 0)

    let new(self, top_left, bottom_right)
        self.top_left = top_left
        self.bottom_right = bottom_right

    let width(self)
        return (self.bottom_right - self.top_left).x

    let height(self)
        return (self.bottom_right - self.top_left).y
```

## Especificação Sintática

```
code
    ::= program
    | imports program

atom
    ::= IDENTIFIER
    | INTEGER_LITERAL
    | FLOAT_LITERAL
    | STRING_LITERAL
    | '(' expr ')'
    | '[' expr ']'

/* expressions */
atom_expr
    ::= atom
    | atom trailer

trailer
    ::= '(' ')'
    | '(' arglist ')'
    | '[' expr ']'
    | '.' IDENTIFIER

exponential_expr
    ::= atom_expr
    | exponential_expr POW atom_expr

unary_expr
    ::= NOT unary_expr
    | SUB unary_expr
    | exponential_expr

multiplicative_expr
    ::= unary_expr
    | multiplicative_expr MUL unary_expr
    | multiplicative_expr DIV unary_expr

additive_expr
    ::= multiplicative_expr
    | additive_expr ADD multiplicative_expr
    | additive_expr SUB multiplicative_expr

relational_expr
    ::= additive_expr
    | relational_expr LT additive_expr
    | relational_expr LE additive_expr
    | relational_expr GT additive_expr
    | relational_expr GE additive_expr

equality_expr
```

```
    ::= relational_expr
    | equality_expr EQ relational_expr
    | equality_expr NE relational_expr

logical_expr
    ::= equality_expr
    | logical_expr AND equality_expr
    | logical_expr OR equality_expr
    | logical_expr XOR equality_expr

expr
    ::= logical_expr

exprlist
    ::= expr
    | exprlist ',' expr

/* module system */
imports
    ::= import_stmt NEWLINE
    | imports import_stmt NEWLINE

import_stmt
    ::= IMPORT dotted_as_names
    | IMPORT dotted_as_names ','
    | FROM dotted_name IMPORT import_as_names
    | FROM dotted_name IMPORT import_as_names ','

/* import a.b.c */
dotted_as_names
    ::= dotted_as_name
    | dotted_as_names ',' dotted_as_name

dotted_as_name
    ::= dotted_name
    | dotted_name AS IDENTIFIER

/* from a.b import c */
import_as_names
    ::= import_as_name
    | import_as_names ',' import_as_name

import_as_name
    ::= IDENTIFIER
    | IDENTIFIER AS IDENTIFIER

dotted_name
    ::= IDENTIFIER
    | dotted_name '.' IDENTIFIER

program
    ::= stmt
```

```
    | definition
    | stmt program
    | definition program

definition
    ::= function_definition
    | class_definition
    | EXPORT function_definition
    | EXPORT class_definition


function_definition
    ::= LET IDENTIFIER '(' ')' scope
    | LET IDENTIFIER '(' function_params_list ')' scope

function_params_list
    ::= identifier_list
    | identifier_list ',' variadic_param
    | variadic_param

identifier_list
    ::= IDENTIFIER
    | IDENTIFIER EQ atom
    | identifier_list ',' IDENTIFIER
    | identifier_list ',' IDENTIFIER EQ atom

variadic_param
    ::= ELLIPSIS IDENTIFIER

scope
    ::= NEWLINE INDENT inner_scope DEDENT

inner_scope
    ::= stmt NEWLINE
    | inner_scope stmt NEWLINE


class_definition
    ::= CLASS IDENTIFIER class_scope

class_scope
    ::= NEWLINE INDENT inner_class_scope DEDENT

inner_class_scope
    ::= function_definition
    | inner_class_scope function_definition

stmt
    ::= assignment_expr
    | function_call
    | compound_stmt
    | THROW expr
```

```
    | RETURN exprlist

assignment_expr
    ::= LET IDENTIFIER EQ atom
    | IDENTIFIER EQ atom
    | IDENTIFIER '[' atom ']' EQ atom

function_call
    ::= IDENTIFIER '(' ')'
    | IDENTIFIER '(' arglist ')'

arglist
    ::= atom
    | arglist ',' atom

/* flow control */
compound_stmt
    ::= if_stmt
    | while_stmt
    | for_stmt

if_stmt
    ::= IF expr scope
    | IF expr scope ELSE scope

while_stmt
    ::= WHILE expr scope

for_stmt
    ::= FOR exprlist IN expr scope
```

## Arquivos

Os arquivos FLEX e BISON são respectivamente `sucuri.l` e `sucuri.y`. Exemplos de programas válidos se encontram na pasta `examples/`. O código fonte do analisador é `sucuri.yy.c`. Os logs de saída aplicados no exemplo `examples/geometry.scr` estão no arquivo `geometry-parse.ylog`.