Sucuri

Uma linguagem baseada em Python

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

22 de Agosto 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 Léxica

Inicialmente são definidas algumas regex de apoio:

```
D [0-9]                          % Reconhece dígitos

L [a-zA-Z_!@$?]                  % Reconhece qualquer símbolo
                                 % possível em um identificador

NO_SQUOTE_STRING_LITERAL [^']*   % Qualquer _string literal_
                                 % que não possua aspas simples

NO_DQUOTE_STRING_LITERAL [^"]*   % Qualquer _string literal_
                                 % que não possua aspas duplas
```

Além de duas funções, `count()`, que realiza contagem de colunas para gerar uma melhor mensagem de erro (caso ocorra), e `indent_level()`, que informa o nível de identação atual.

Identificadores

Identificadores são compostos por qualquer sequência de `L` ou `D` não separados por espaços, podendo conter "." (não no início, no final nem sucedidos por dígitos).

Literais

São assumidos como literais de inteiros qualquer construção de somente dígitos:

```
# Inteiros válidos:

1
10
0
0000  % Tratado como 0
300
-10   % É reconhecido o "10" como literal inteiro e o "-" como operador unário
      % operado sobre o "10"
```

Assim, sua *regex* se torna `{D}+` (1 ou mais dígitos consecutivos).

São assumidos como ponto-flutuante todo literal composto por números e que tenha um "." no início, meio ou fim do literal:

```
1   % Inteiro
1.  % Float
.1  % Float
-1. % "1." reconhecido como literal float, unário "-" operado em "1."

.   % Erro léxico
```

Assim, sua *regex* é separada em duas:

- `"."{D}+` — Reconhece *floats* iniciados em ".";
- `{D}+"."{D}*` — Reconhece *floats* com "."no meio ou final;

String literals são compostos de qualquer sequência de caracteres que:

- Estão entre aspas simples (') e não possuem outra aspa simples no meio (reconhecido pela *regex* `"'"{NO_SQUOTE_STRING_LITERAL}"'"`).
- Estão entre aspas dupla (") e não possuem outra aspa dupla no meio (reconhecido pela *regex* `"\""{NO_DQUOTE_STRING_LITERAL}"\""`).

Operadores:

```
% Unários
not
-

% Comparativos
!=
=
<
<=
>
>=


% Matemáticos
+
-
*
**
/

% Lógicos
and
or
xor

% Outros
(
)
```

Palavras reservadas:

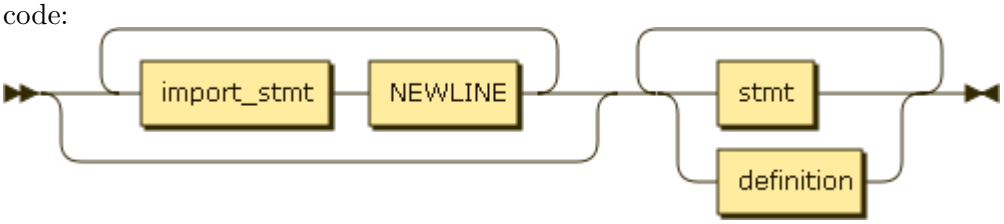
```
% Estruturas de controle
class  % Define um novo tipo
if
else
for
while

% Retornos
return
throw

as     % Serve para alias
export % Define o elemento como público
from   % Para importação parcial de um módulo
```

```
import % Para importar um módulo
in     % Para iteração (for i in set)
let    % Definição
```

Há também a definição de elipse (`...`) para parâmetros variádicos.

Grafo de sintaxe e especificação EBNF

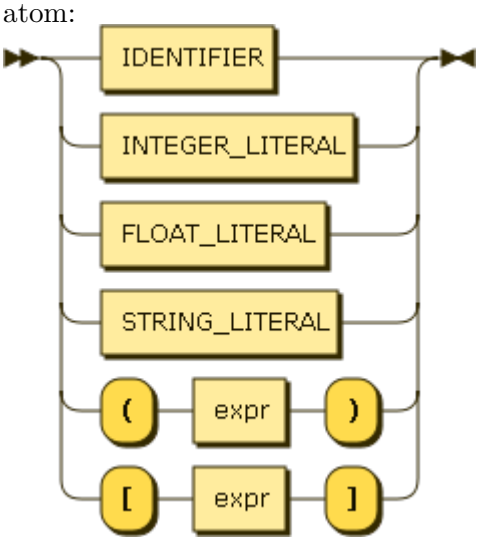
code:




```
code     ::= ( import_stmt NEWLINE )* ( stmt | definition )+
```

no references

atom:

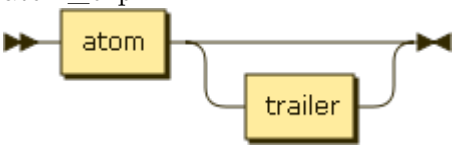



```
atom     ::= IDENTIFIER
           | INTEGER_LITERAL
           | FLOAT_LITERAL
           | STRING_LITERAL
           | '(' expr ')'
           | '[' expr ']'
```

referenced by:

- arglist
- assignment_expr
- atom_expr
- function_params_list

atom_expr:

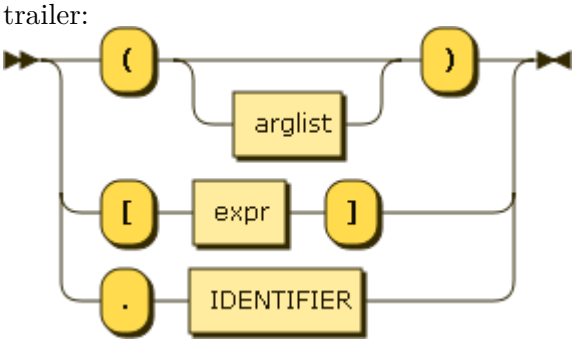



```
atom_expr
     ::= atom trailer?
```

referenced by:

- unary_expr

trailer:

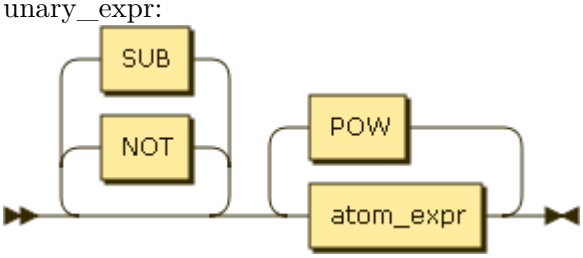



```
trailer  ::= '(' arglist? ')'
           | '[' expr ']'
           | '.' IDENTIFIER
```

referenced by:

- atom_expr

unary_expr:

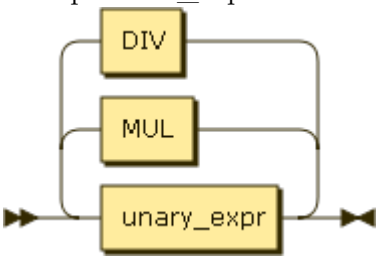



```
unary_expr
     ::= ( NOT | SUB )* atom_expr ( POW atom_expr )*
```

referenced by:

- multiplicative_expr

multiplicative_expr:

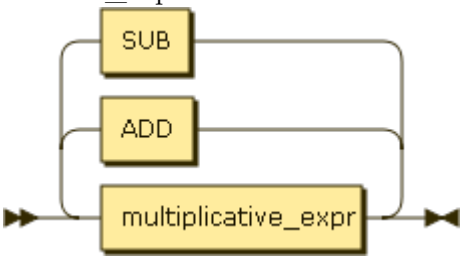



```
multiplicative_expr
     ::= unary_expr ( ( MUL | DIV ) unary_expr )*
```

referenced by:

- additive_expr

additive_expr:

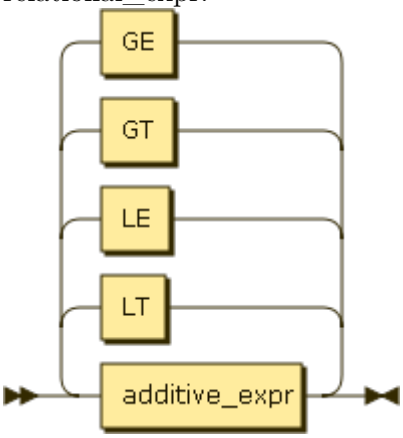



```
additive_expr
     ::= multiplicative_expr ( ( ADD | SUB ) multiplicative_expr )*
```

referenced by:

- relational_expr

relational_expr:

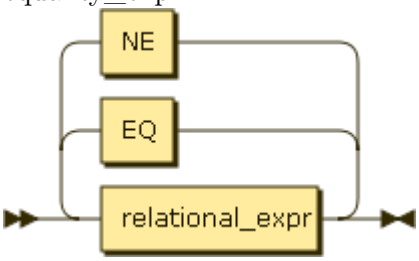



```
relational_expr
     ::= additive_expr ( ( LT | LE | GT | GE ) additive_expr )*
```

referenced by:

- equality_expr

equality_expr:

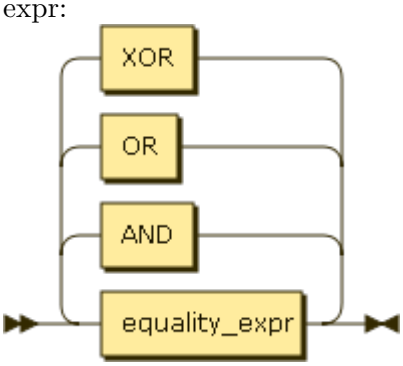



```
equality_expr
     ::= relational_expr ( ( EQ | NE ) relational_expr )*
```

referenced by:

- expr

expr:

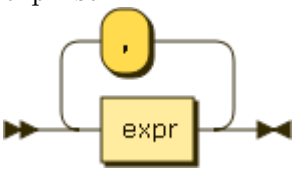



```
expr     ::= equality_expr ( ( AND | OR | XOR ) equality_expr )*
```

referenced by:

- atom
- exprlist
- for_stmt
- if_stmt
- stmt
- trailer
- while_stmt

exprlist:

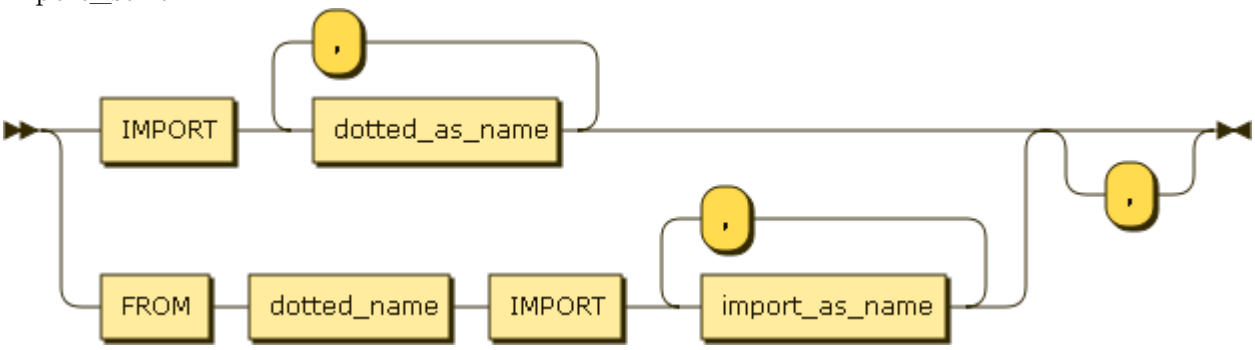



```
exprlist ::= expr ( ',' expr )*
```

referenced by:

- for_stmt
- stmt

import_stmt:

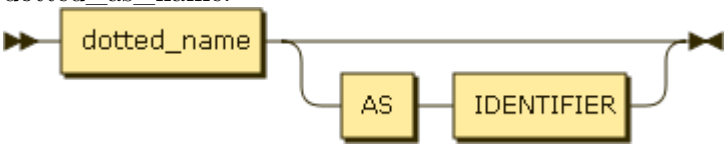



```
import_stmt
     ::= ( IMPORT dotted_as_name ( ',' dotted_as_name )* | FROM dotted_name IMPORT in
```

referenced by:

- code

dotted_as_name:

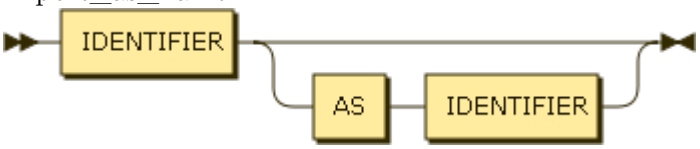



```
dotted_as_name
     ::= dotted_name ( AS IDENTIFIER )?
```

referenced by:

- import_stmt

import_as_name:

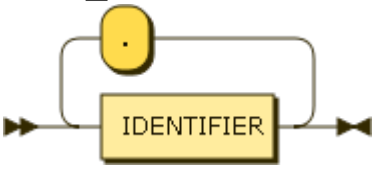



```
import_as_name
     ::= IDENTIFIER ( AS IDENTIFIER )?
```

referenced by:

- import_stmt

dotted_name:

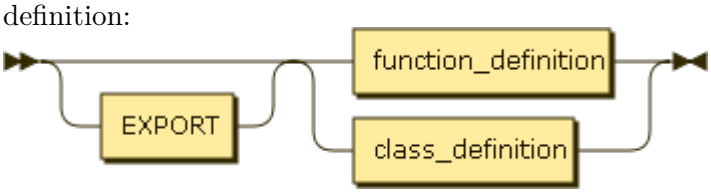



```
dotted_name
     ::= IDENTIFIER ( '.' IDENTIFIER )*
```

referenced by:

- dotted_as_name
- import_stmt

definition:

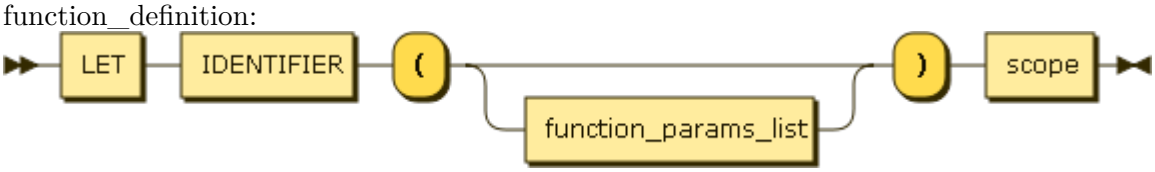



```
definition
     ::= EXPORT? ( function_definition | class_definition )
```

referenced by:

- code

function_definition:

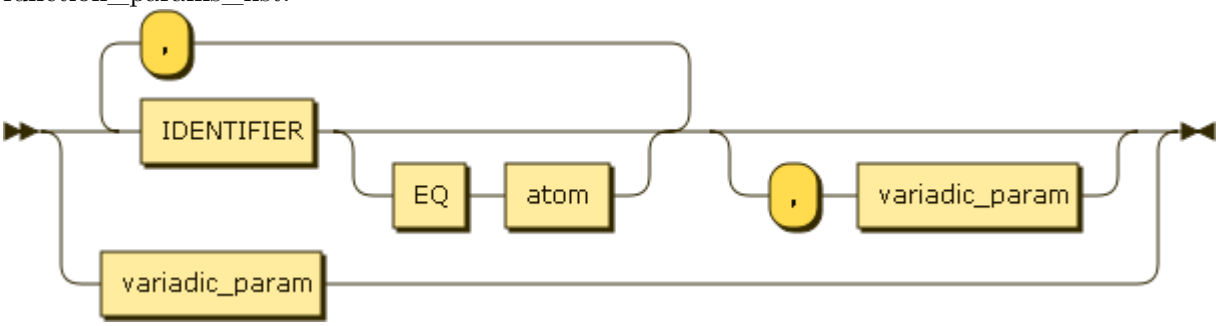



```
function_definition
     ::= LET IDENTIFIER '(' function_params_list? ')' scope
```

referenced by:

- class_scope
- definition

function_params_list:

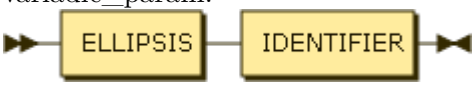



```
function_params_list
     ::= IDENTIFIER ( EQ atom )? ( ',' IDENTIFIER ( EQ atom )? )* ( ',' variadic_para
       | variadic_param
```

referenced by:

- function_definition

variadic_param:

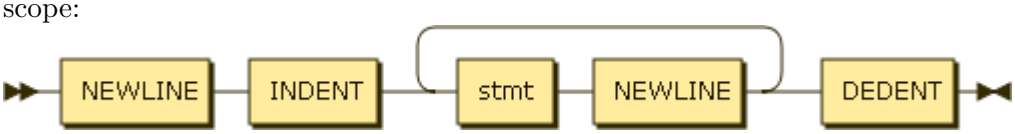



```
variadic_param
     ::= ELLIPSIS IDENTIFIER
```

referenced by:

- function_params_list

scope:




```
scope    ::= NEWLINE INDENT ( stmt NEWLINE )+ DEDENT
```

referenced by:

- for_stmt
- function_definition
- if_stmt
- while_stmt

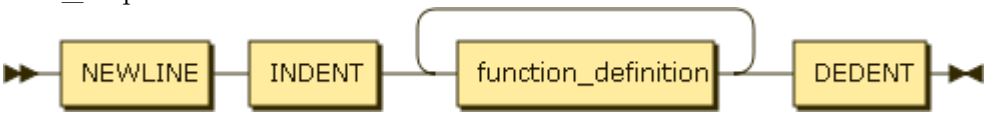
class_definition:


```
class_definition
     ::= CLASS IDENTIFIER class_scope
```

referenced by:

- definition

class_scope:

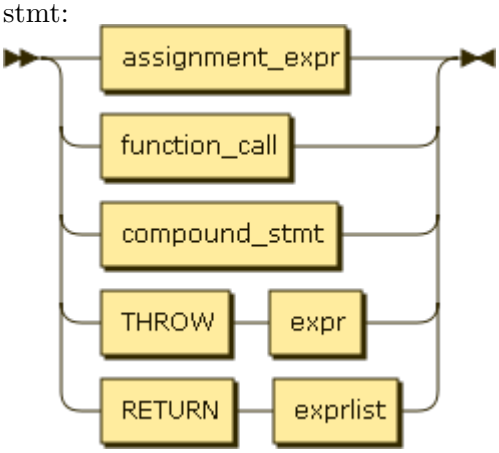



```
class_scope
     ::= NEWLINE INDENT function_definition+ DEDENT
```

referenced by:

- class_definition

stmt:

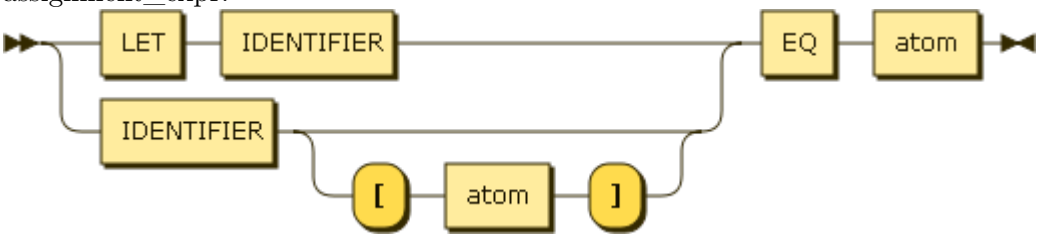



```
stmt      ::= assignment_expr
            | function_call
            | compound_stmt
            | THROW expr
            | RETURN exprlist
```

referenced by:

- code
- scope

assignment_expr:

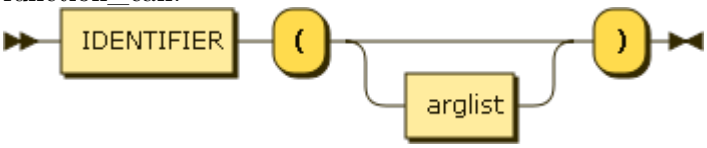



```
assignment_expr
     ::= ( LET IDENTIFIER | IDENTIFIER ( '[' atom ']' )? ) EQ atom
```

referenced by:

- stmt

function_call:

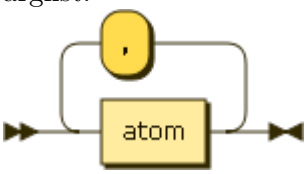



```
function_call
     ::= IDENTIFIER '(' arglist? ')'
```

referenced by:

- stmt

arglist:

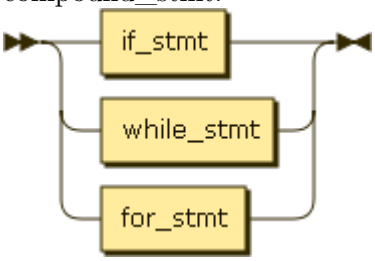



```
arglist  ::= atom ( ',' atom )*
```

referenced by:

- function_call
- trailer

compound_stmt:

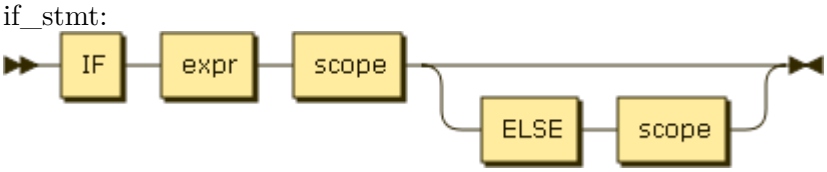



```
compound_stmt
     ::= if_stmt
       | while_stmt
       | for_stmt
```

referenced by:

- stmt

if_stmt:




```
if_stmt  ::= IF expr scope ( ELSE scope )?
```

referenced by:

- compound_stmt

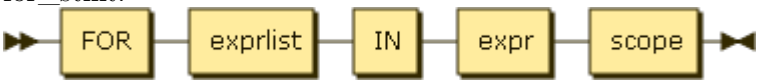
while_stmt:


```
while_stmt
     ::= WHILE expr scope
```

referenced by:

- compound_stmt

for_stmt:




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

referenced by:

- compound_stmt

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`.