```
digit = "0" | "1" | "2" | "3" | "4 | "5" | "6" | "7" | "8" | "9";

comma = ",";

dot = ".";

semicolon = ";";

asign_operator = "=";

backslash = "\";

new_line = backsklash, "n";

comment = "#", [text];

library_name = identifier;

object_name = identifier;


letter = "a" | ... | "z" | "A" | ... | "Z";

quote = "'";

whitespace = " ";

identifier = letter, {letter | digit};

variable_name = identifier;

constant = digit, { digit };

text = quote, { letter | number | whitespace | character }, quote;


type = "bool" | "int" | "float" | "string" | "list";

bool_values = "false" | "true";

int_value = constant

float_value = constant, [ ".", constant ];

string_value = text;

variable_value = bool_value | int_value | float_value| string_value |
array;

array= "[", [or_expression, {comma , or_expression}] , "]";

array_name = variable_name;


sum_operator = "+" | "-";

multiply_operator = "*" | "/";

relational_operator = "<" | ">" | "<=" | ">=" | "==" | "!=";

and_operator = "and";

or_operator = "or";
```

```
negation_operator = "!";


parameters = [ variable_name, {comma, variable_name} ];

arguments = [ expression, {comma, expression} ] | lambda_expression;

function_name = identifier;

function_definition = "def", function_name, "(", parameters , ")" ,
statements;

break_statement = "break", semicolon ;

function_call = chained_expression, typical_function_call, semicolon;

chained_expression = identfier, ["(", arguments, ")"],
      {dot, (variable_name | typical_function_call)};

typical_function_call = function_name, '(', arguments, ')'

variable_assignment = object_expression, assign_operator,
assign_expression, semicolon;

object_expression = chained_access, variable_name;

assign_expression = or_expression;


include_statement = "from", library_name, "import", object_name,
      {coma, object_name}, semicolon;

lambda_expression = "$", variable_name, "=>", statements;

expression = or_expression;

or_expression = and_expression, {"or", and_expression};

and_expresion = relation_expresion, {"and", relation_condition};

relation_condition = arth_expression, [relational_operator,
arth_expression];

arth_expression = term, {sum_operator, term};

term = factor, { multiply_operator, factor };

factor = {negation_operator}, variable_value | object_expression |
function_call |"(", arth_expression, ")";

if = "if", "(", expression, ")", statements, ["else", statements];

while = "while", "(", expression, ")", statements;

return = "return", "(", expression, ")", semicolon;

statement = variable_assignment

            | function_call

            | if

            | while

            | break_statement
```

```
            | return;
statements = "{", {statement}, "}";
program = { include_statement | function_definition };
```