Cython

Relatório I: Análise Léxica e apresentação da linguagem

Matheus S. Pinheiro Bittencourt e Thales A. Zirbel Hübner

20 de Março de 2018

Universidade Federal de Santa Catarina

1. Linguagem

A linguagem sendo criada, de nome cython, usa em grande parte a sintaxe de python com algumas características de c++, como tipagem e a possibilidade de passar referências para funções.

É incluído na linguagem operações booleanas e aritméticas, definição e chamada de funções, asserções, estruturas if, while e for e matrizes. Os tipos primitivos definidos criados são int, float, char e void.

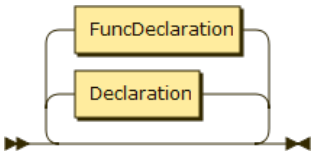
2. Especificação Formal

A seguir está a gramática geradora da linguagem e os grafos EBNF.

```
Program     ::=   (Declaration | FuncDeclaration)*

Declaration ::= IDENTIFIER ':' Type ('=' Expression)? '\n'
FuncDeclaration ::= 'def' IDENTIFIER '(' ArgsList? ')' '->' Type Block

Block ::=   'begin' InnerBlock 'end'
InnerBlock  ::= '\n' ((Declaration | Statement | Expression) '\n')*

Statement   ::=
      IfStmt
      | ForStmt
      | WhileStmt
      | ReturnStmt

Expression  ::=
      Expression '+' Expression
      | Expression '-' Expression
      | Expression '*' Expression
      | Expression '/' Expression
      | Expression '**' Expression
      | Expression 'and' Expression
      | Expression 'or' Expression
      | 'not' Expression
      | '-' Expression
      | Expression '>' Expression
      | Expression '<' Expression
      | Expression '>=' Expression
      | Expression '<=' Expression
      | Expression '==' Expression
      | Expression '!=' Expression
```

```
      | '(' Expression ')'
      | Assignment
      | AtomExpr

AtomExpr ::=
      Name
      | FuncCall
      | NUMBER
      | CHAR
      | STRING
      | BOOL

Assignment ::= Name '=' Expression

FuncCall    ::=   IDENTIFIER '(' (Expression (',' Expression)* )? ')'

IfStmt      ::=   'if' Expression 'do' InnerBlock ('elif' Expression
InnerBlock)* ('else' Expression InnerBlock)? 'end'

ForStmt     ::=   'for' (Expression | Declaration) ';' Expression ';'
Expression Block

WhileStmt   ::=   'while'     Expression Block

ReturnStmt  ::= 'return' Expression

ArgsList    ::= IDENTIFIER '&'? ':' Type (',' IDENTIFIER '&'? ':' Type)*

Type  ::=
      Type '[' NUMBER? ']'
      | 'int'
      | 'float'
      | 'char'
      | 'void'

Name  ::=
      Name '[' Expression ']'
      | IDENTIFIER

IDENTIFIER  ::=   [A-z_][0-9A-z_]*
NUMBER      ::=   [0-9]+('.'[0-9]+)?
CHAR  ::= '\'' [^\']? '\''
STRING      ::= '\"' [^\"]* '\"'
BOOL  ::= 'True' | 'False'
```

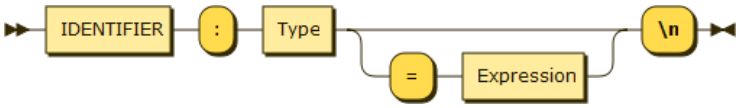
Program:




```
Program  ::= ( Declaration | FuncDeclaration )*
```

no references

Declaration:

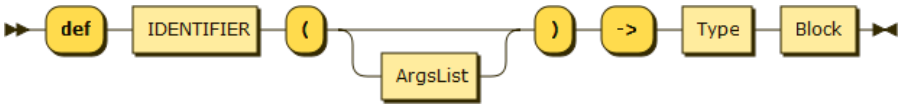



```
Declaration
        ::= IDENTIFIER ':' Type ( '=' Expression )? '\n'
```

referenced by:

- ForStmt
- InnerBlock
- Program

FuncDeclaration:




```
FuncDeclaration
        ::= 'def' IDENTIFIER '(' ArgsList? ')' '->' Type Block
```

referenced by:

- Program

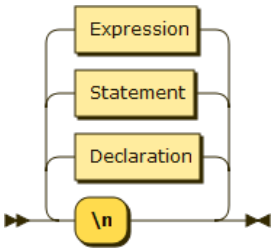
Block:


```
Block    ::= 'begin' InnerBlock 'end'
```

referenced by:

- ForStmt
- FuncDeclaration
- WhileStmt

InnerBlock:

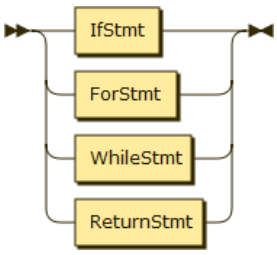



```
InnerBlock
        ::= '\n' ( ( Declaration | Statement | Expression ) '\n' )*
```

referenced by:

- Block
- IfStmt

Statement:

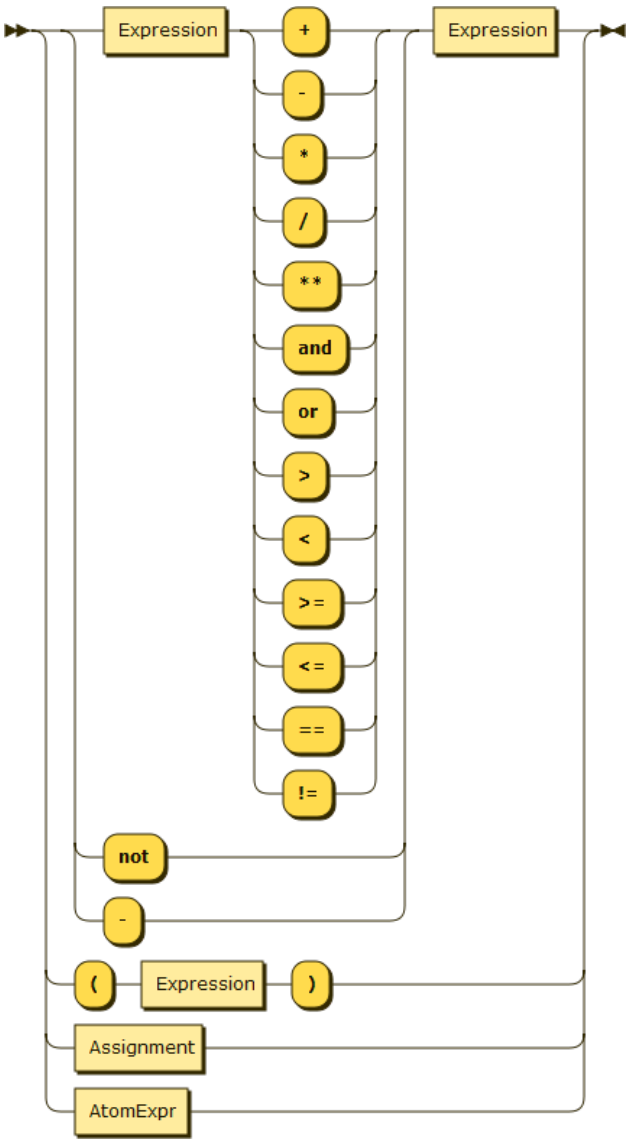



```
Statement
         ::= IfStmt
           | ForStmt
           | WhileStmt
           | ReturnStmt
```

referenced by:

- InnerBlock

Expression:



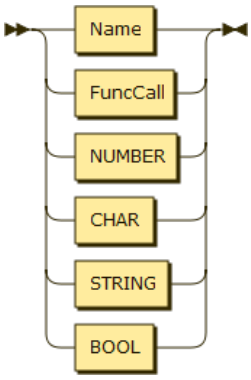

```
Expression
         ::= ( Expression ( '+' | '-' | '*' | '/' | '**' | 'and' | 'or' | '>' | '<' | '>=' | '<=' | '==' | '!=' ) | 'not' | '-' ) Expression
           | '(' Expression ')'
           | Assignment
           | AtomExpr
```

referenced by:

- Assignment
- Declaration
- Expression
- ForStmt
- FuncCall

- IfStmt
- InnerBlock
- Name
- ReturnStmt
- WhileStmt

AtomExpr:

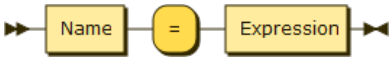



```
AtomExpr ::= Name
           | FuncCall
           | NUMBER
           | CHAR
           | STRING
           | BOOL
```

referenced by:

- Expression

Assignment:

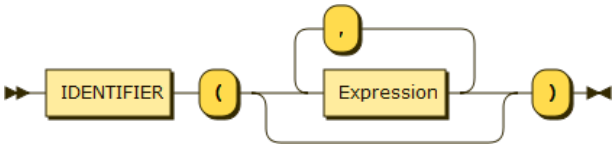



```
Assignment
        ::= Name '=' Expression
```

referenced by:

- Expression

FuncCall:

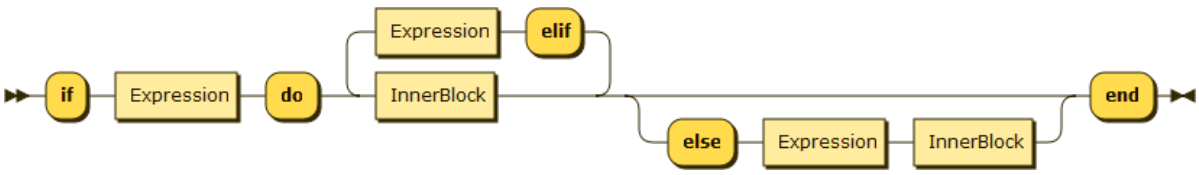



```
FuncCall ::= IDENTIFIER '(' ( Expression ( ',' Expression )* )? ')'
```

referenced by:

- AtomExpr

IfStmt:

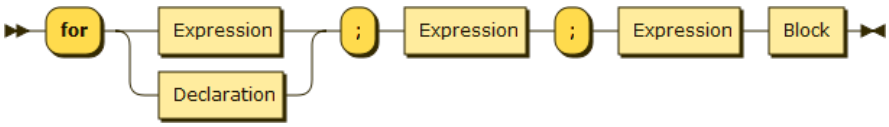



```
IfStmt   ::= 'if' Expression 'do' InnerBlock ( 'elif' Expression InnerBlock )* ( 'else' Expression InnerBlock )? 'end'
```

referenced by:

- Statement

ForStmt:




```
ForStmt  ::= 'for' ( Expression | Declaration ) ';' Expression ';' Expression Block
```

referenced by:

- Statement

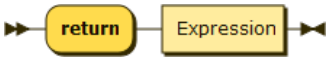
WhileStmt:


```
WhileStmt
         ::= 'while' Expression Block
```

referenced by:

- Statement

ReturnStmt:

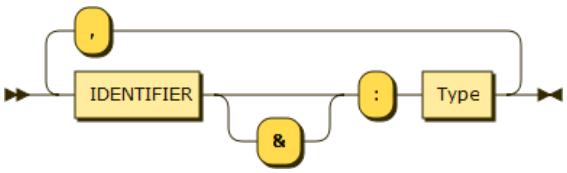



```
ReturnStmt
         ::= 'return' Expression
```

referenced by:

- Statement

ArgsList:

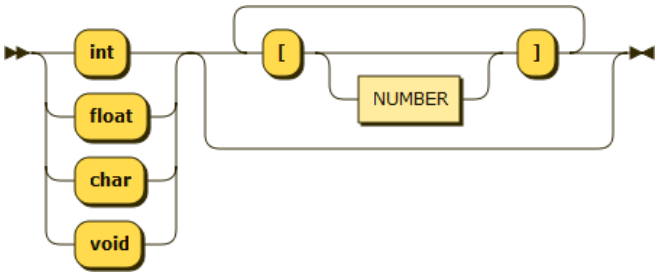



```
ArgsList ::= IDENTIFIER '&'? ':' Type ( ',' IDENTIFIER '&'? ':' Type )*
```

referenced by:

- FuncDeclaration

Type:

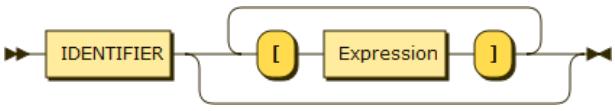



```
Type     ::= ( 'int' | 'float' | 'char' | 'void' ) ( '[' NUMBER? ']' )*
```

referenced by:

- ArgsList
- Declaration
- FuncDeclaration

Name:

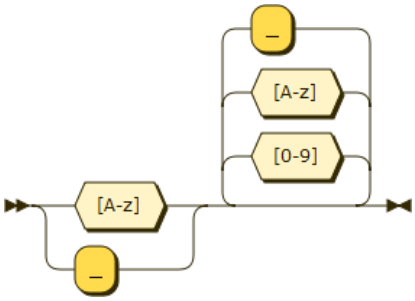



```
Name     ::= IDENTIFIER ( '[' Expression ']' )*
```

referenced by:

- Assignment
- AtomExpr

IDENTIFIER:

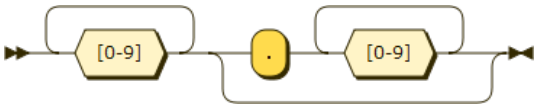



```
IDENTIFIER
         ::= [A-z_] [0-9A-z_]*
```

referenced by:

- ArgsList
- Declaration
- FuncCall
- FuncDeclaration
- Name

NUMBER:

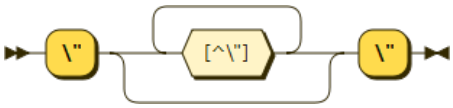



```
NUMBER   ::= [0-9]+ ( '.' [0-9]+ )?
```

referenced by:

- AtomExpr
- Type

CHAR:

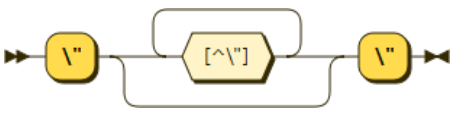



```
CHAR     ::= '\"' [^\"]* '\"'
```

referenced by:

- AtomExpr

STRING:

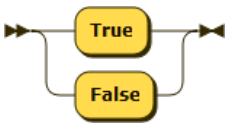



```
STRING   ::= '\"' [^\"]* '\"'
```

referenced by:

- AtomExpr

BOOL:




```
BOOL      ::= 'True'
            | 'False'
```

referenced by:

- AtomExpr

---

3. Analisador Léxico

Utilizando FLEX foi realizado um analisador correspondente a gramática da linguagem escolhida e o código está abaixo.

```
%{
#include <iostream>
#include <lexical_error.h>
%}

%option yylineno noyywrap noinput nounput nodefault

%%

[ \t] /* ignore whitespace */
\n+   std::cout << "NL" << std::endl;  /* ignores multiple new lines */

#[^\n]*\n   /* single line comment */
\"\"\"[^\"]*\"\"\"\n?   /* multi-line comment */

def   std::cout << "DEF" << std::endl;
if    std::cout << "IF" << std::endl;
elif  std::cout << "ELIF" << std::endl;
else  std::cout << "ELSE" << std::endl;
for   std::cout << "FOR" << std::endl;
while std::cout << "WHILE" << std::endl;
do    std::cout << "DO" << std::endl;
begin std::cout << "BEGIN" << std::endl;
end   std::cout << "END" << std::endl;
return      std::cout << "RETURN" << std::endl;
True|False  std::cout << "<BOOL, " << yytext << ">" << std::endl;

int   std::cout << "INT" << std::endl;
float std::cout << "FLOAT" << std::endl;
char  std::cout << "CHAR" << std::endl;
void  std::cout << "VOID" << std::endl;

":"   std::cout << "COLON" << std::endl;
";"   std::cout << "SEMICOLON" << std::endl;
","   std::cout << "COMMA" << std::endl;
"->"  std::cout << "ARROW" << std::endl;
"&"   std::cout << "AMPERSEND" << std::endl;
```

```
"+"   std::cout << "PLUS" << std::endl;
"-"   std::cout << "MINUS" << std::endl;
"*"   std::cout << "TIMES" << std::endl;
"/"   std::cout << "DIV" << std::endl;
"**"  std::cout << "EXP" << std::endl;

and   std::cout << "AND" << std::endl;
or    std::cout << "OR" << std::endl;
not   std::cout << "NOT" << std::endl;

"<"   std::cout << "LT" << std::endl;
">"   std::cout << "GT" << std::endl;
"<="  std::cout << "LE" << std::endl;
">="  std::cout << "GE" << std::endl;
"=="  std::cout << "EQ" << std::endl;
"!="  std::cout << "NE" << std::endl;

"("   std::cout << "LPAREN" << std::endl;
")"   std::cout << "RPAREN" << std::endl;
"["   std::cout << "LBRACKET" << std::endl;
"]"   std::cout << "RBRACKET" << std::endl;

"="   std::cout << "ASSIGN" << std::endl;

[a-zA-Z_][a-zA-Z_0-9]* {
      std::cout << "<IDENTIFIER, " << yytext << ">" << std::endl;
}
[0-9]+(\.[0-9]+)? std::cout << "<NUMBER_L, " << yytext << ">" <<
std::endl;
\'[^\']?\'  std::cout << "<CHAR_L, " << yytext << ">" << std::endl;
\"[^\"]*\"  std::cout << "<STRING_L, " << yytext << ">" << std::endl;
<<EOF>>     yyterminate();
.     throw lexical_error(yylineno, yytext);

%%

int main(int argc, char** argv) {
      if (argc > 1)
            yyin = std::fopen(argv[1], "r");

      try {
            yylex();
      } catch (const lexical_error& e) {
            std::cout << "[Line " << e.line() << "] " << e.what() <<
std::endl;
```

```
	}
	yypop_buffer_state();  /* clean scanner memory */

	if (yyin != stdin)
		std::fclose(yyin);
}
```

4. Exemplos

Foram realizados 4 exemplos curtos para exemplificar o funcionamento do parser. Para realizar a execução dos exemplos, é necessário realizar o comando ***make*** no diretório raiz do código e então executar a seguinte linha de código, ainda no diretório raiz:

```
	./cython exemples/nome_do_arquivo_de_exemplo.cy
```

4.1 Fatorial recursivo

Foi criado um código para teste do analisador, definindo uma função de nome factorial. O nome do arquivo de exemplo é “factorial.cy”.

```
def factorial(n: int) -> int begin
	if n == 1 do
		return 1
	else
		return factorial(n-1) * factorial(n-2)
	end
end
```

O log de saída do arquivo foi o seguinte:

DEF
<IDENTIFIER, factorial>
LPAREN
<IDENTIFIER, n>
COLON
INT
RPAREN
ARROW
INT
BEGIN
NL
IF
<IDENTIFIER, n>
EQ
<NUMBER_L, 1>

DO
NL
RETURN
<NUMBER_L, 1>
NL
ELSE
NL
RETURN
<IDENTIFIER, factorial>
LPAREN
<IDENTIFIER, n>
MINUS
<NUMBER_L, 1>
RPAREN
TIMES
<IDENTIFIER, factorial>
LPAREN
<IDENTIFIER, n>
MINUS
<NUMBER_L, 2>
RPAREN
NL
END
NL
END
NL

4.2 Máximo de um vetor

Um pequeno código para verificar o uso de matrizes. O nome do arquivo de teste é “find_max.cy”.

```
def find_max(v: int[], size: int) -> int begin
      max = v[0]
      for i: int = 1; i < size; i = i + 1 begin
            if v[i] > max do
                  max = v[i]
            end
      end
      return max
end
```

Segue o log de saída ao executar o parser.

DEF
<IDENTIFIER, find_max>
LPAREN

<IDENTIFIER, v>
COLON
INT
LBRACKET
RBRACKET
COMMA
<IDENTIFIER, size>
COLON
INT
RPAREN
ARROW
INT
BEGIN
NL
<IDENTIFIER, max>
ASSIGN
<IDENTIFIER, v>
LBRACKET
<NUMBER_L, 0>
RBRACKET
NL
FOR
<IDENTIFIER, i>
COLON
INT
ASSIGN
<NUMBER_L, 1>
SEMICOLON
<IDENTIFIER, i>
LT
<IDENTIFIER, size>
SEMICOLON
<IDENTIFIER, i>
ASSIGN
<IDENTIFIER, i>
PLUS
<NUMBER_L, 1>
BEGIN
NL
IF
<IDENTIFIER, v>
LBRACKET
<IDENTIFIER, i>
RBRACKET
GT
<IDENTIFIER, max>

DO
NL
<IDENTIFIER, max>
ASSIGN
<IDENTIFIER, v>
LBRACKET
<IDENTIFIER, i>
RBRACKET
NL
END
NL
END
NL
RETURN
<IDENTIFIER, max>
NL
END
NL

4.3 Erro léxico

Foi criado um exemplo para causar um erro léxico ao ser executado. O nome do arquivo é “lex_error1.cy”.

```
c: char = '
```

Log gerado:
<IDENTIFIER, c>
COLON
CHAR
ASSIGN
[Line 1] lexical error, unknown symbol '

4.4 Referência

Uma exemplificação de passagem de referência por uma função. O arquivo tem nome “reference.cy”.

```
def f(a&: int) -> void:
      a = 1

a = 0
f(a)
print(a)  # should print '1'
```

Log resultante:

DEF
<IDENTIFIER, f>
LPAREN
<IDENTIFIER, a>
AMPERSEND
COLON
INT
RPAREN
ARROW
VOID
COLON
NL
<IDENTIFIER, a>
ASSIGN
<NUMBER_L, 1>
NL
<IDENTIFIER, a>
ASSIGN
<NUMBER_L, 0>
NL
<IDENTIFIER, f>
LPAREN
<IDENTIFIER, a>
RPAREN
NL
<IDENTIFIER, print>
LPAREN
<IDENTIFIER, a>
RPAREN