DOUGLAS BOLIS LIMA

HARÃ HEIQUE DOS SANTOS

MARCOS ANTONIO CARNEIRO DE PAULA

Relatório Trabalho Final - Implementação de Linguagem de Domínio Específico (DSL)

Serra

2019

DOUGLAS BOLIS LIMA
HARÃ HEIQUE DOS SANTOS
MARCOS ANTONIO CARNEIRO DE PAULA

Relatório Trabalho Final - Implementação de Linguagem de Domínio Específico (DSL)

Trabalho apresentado na disciplina de Linguagens Formais e Autômatos no curso Bacharelado em Sistemas de Informação, do Instituto Federal do Espírito Santo - Campus Serra, orientado pelo docente Jefferson O. Andrade.

SUMÁRIO

1. Introdução

Implementação de uma *Linguagem de Domínio Específico (DSL)*, chamada ***DMH***, utilizando a linguagem de programação ***Python*** e a ferramenta voltada para o parse de qualquer gramática livre de contexto chamada ***Lark***.

A linguagem ***DMH***, nome proveniente das iniciais dos nomes dos autores (**D**ouglas, **H**arã e **M**arcos, é voltada exclusivamente no desenvolvimento de aplicações para cálculos matemáticos aritméticos e trigonométricos, onde são utilizados mecanismos de construção sintática tais como: *seleção* e *repetição*, além de mecanismos de *nomeação* (manipulação de variáveis) e *abstração* (funções), muito comumente utilizados nas linguagens de programação.

A ideia surgiu baseado numa extensão da implementação do parser descendente recursivo da linguagem livre de contexto chamada de ***MEL*** (*Micro Expression Language*), cuja foi desenvolvida ao longo período durante a disciplina de Linguagens Formais e Autômatos.

Para informações detalhadas sobre a linguagem desenvolvida, assim como seu código fonte acesse o link do seguinte repositório no github: **https://github.com/cardepaula/trabalho-final-lfa-DMH**.

2. Definição da DSL (*Domain Specific Language*)

A definição da *DSL* são divididas em duas partes, onde na seção 2.1 será apresentada a gramática da linguagem *DMH* no formato ***EBNF (Extended Backus-Naur Form)***. Já na seção 2.2 podem ser vistos os ***diagramas de sintaxe*** da linguagem, sendo uma outra maneira de representar uma linguagem livre de contexto. Por fim na seção 2.3 são mostrados exemplos de utilização da linguagem com o uso de arquivos de entrada do tipo *.dmh*, referente a linguagem, e seus respectivos resultados de saída.

2.1 *EBNF (Extended Backus-Naur Form)* da linguagem *DMH*

As regras de produção da gramática da DSL no formato ***EBNF*** é definida da seguinte maneira:

```
start = (expr ";")+

expr = assignment
     | ifexpr
     | whileexpr
     | funct
     | aexpr
     | print

assignment = "var" NAME "=" aexpr
           | NAME "=" aexpr

ifexpr = "if" comp "do" block ["else" "do" block]

whileexpr = "while" comp "do" block

block = "{" start "}"

funct = "defun" NAME "(" ")" "do" functblock

functblock = "{" start* functreturn "}"

functreturn = "returns" aexpr ";"

functcall = NAME "(" ")"

print = "show" "(" aexpr ")"

comp = aexpr OP_COMP aexpr
     | "(" aexpr OP_COMP aexpr ")"

aexpr = term
```

```
      | aexpr OP_TERM term

term = factor
     | term OP_FACTOR factor

factor = trig
       | factor OP_POW trig

trig = base
     | TRIG base

base = leftoperation
     | number
     | getvar
     | functcall
     | TRIG base
     | "(" aexpr ")"

leftoperation = OP_LEFT base

number = NUMBER

getvar = NAME

OP_TERM = "+" | "-"
OP_FACTOR = "//" | "*" | "/" | "%"
OP_POW = "^"
OP_LEFT = "+" | "-"
OP_COMP = "==" | "!=" | ">=" | "<=" | ">" | "<"
TRIG = "sen" | "cos" | "tang" | "arcsen" | "arccos" | "arctang"
COMMENT = /(\#\#.+\#\#)/
NAME = /[_a-zA-Z][_a-zA-Z0-9]*/
NUMBER = /-?\d+(\.\d+)?([eE][+-]?\d+)?/
```

Algumas observações importantes devem ser consideradas quanto a forma de escrita ***EBNF*** no uso da biblioteca ***Lark***. Uma delas é ela aceita a gramática de entrada no formato *EBNF*, mas também possibilita a utilização das chamadas *aliases*, que são pequenas setas (->) utilizadas na renomeação de regras (símbolo não-terminal) ou símbolos terminais, onde facilita bastante em determinadas situações como é mostrado na imagem abaixo retirada do código fonte da linguagem *DMH*.

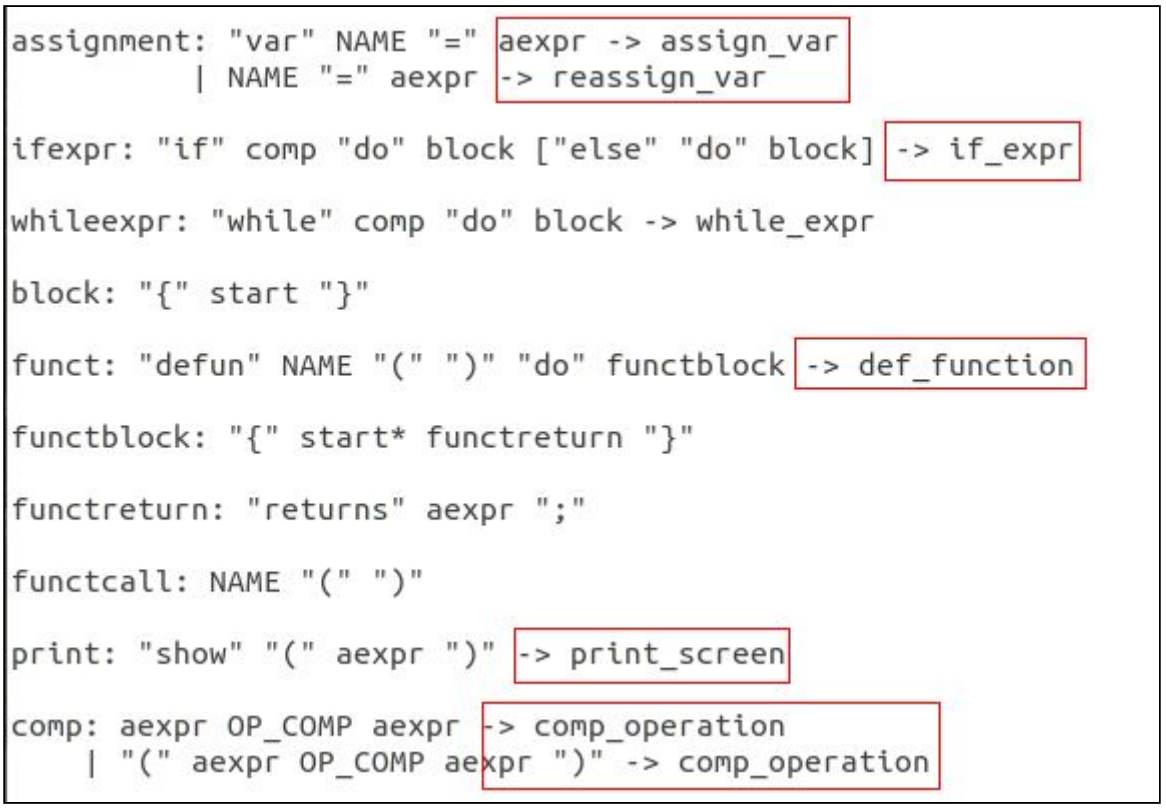


Figura 1 - Renomeação de regras na biblioteca *Lark* com a utilização do *aliases*.

Note na imagem que as partes marcadas em vermelhas correspondem ao uso do técnica *aliases* que renomeia toda a regra assim como sua produção para o nome que foi definido após a seta (->), ou seja, no seu lado direito. Logo a regra *assignment* será renomeada como *assign_var* ou *reassign_var* quando for criado a sua ***AST (Abstract Syntax Tree)***, o qual tem a vantagem de auxiliar no momento de codificação, pois o programador saberá exatamente quando está sendo declarado uma variável (*assign_var*) ou quando esta está sendo alterada por novos valores (*reassign_var*). Na imagem abaixo é possível notar que também pode ser utilizadas em símbolos terminais.

```
OP_TERM: "+" | "-"
OP_FACTOR: "//" | "*" | "/" | "%"
OP_POW: "^"
OP_LEFT: "+" | "-"
OP_COMP: "==" | "!=" | ">=" | "<=" | ">" | "<"
TRIG: "sen" | "cos" | "tang" | "arcsen" | "arccos" | "arctang"
COMMENT: /(\#\#.+\#\#)/

%import common.CNAME -> NAME
%import common.SIGNED_NUMBER -> NUMBER
%import common.WS
%ignore WS
%ignore COMMENT
```

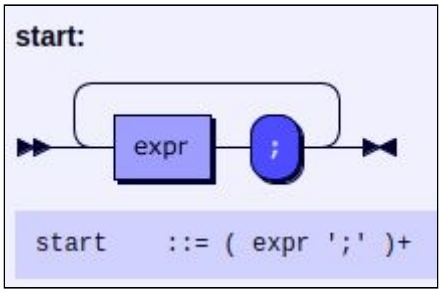
Figura 2 - Renomeação de símbolos terminais na biblioteca *Lark* com a utilização do *aliases*.

Note que além da renomeação das símbolos terminais é também possível as palavras reservadas *import*, para importar recursos disponíveis da biblioteca, como por exemplo na figura, onde são importados símbolos terminais correspondentes para nomeação de variáveis, funções, classes e afins (*commom.CNAME*) assim como para números em ponto flutuante (*commom.SIGNED_NUMBER*), e *ignore*, que como o nome sugere, serve para ignorar *tokens* que não são importantes para a gramática, mas que podem conter nas expressões de entrada que serão geradas as *AST's* correspondentes.

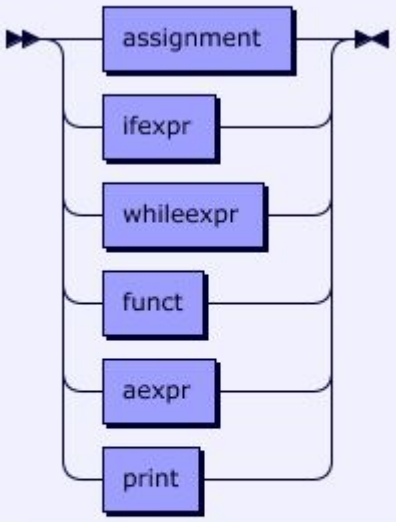
2.2 Diagramas de sintaxe da linguagem *DMH*

O ***diagrama de sintaxe*** representa uma maneira diferente de também representar a linguagem livre de contexto, representando o *EBNF* em um formato gráfico. Abaixo estão o conjunto de imagens que representa a linguagem dessa forma.

Conjunto de imagens que representam as regras da linguagem *DMH*:

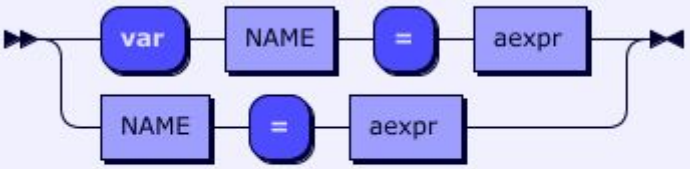


expr:



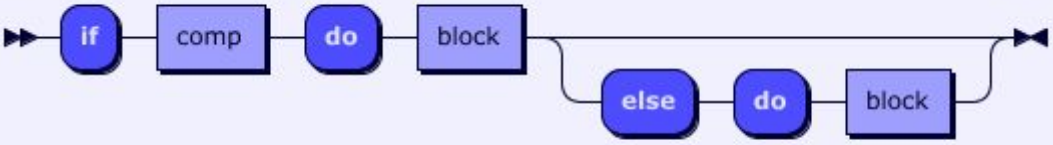

```
expr      ::= assignment
            | ifexpr
            | whileexpr
            | funct
            | aexpr
            | print
```

assignment:




```
assignment
          ::= 'var' NAME '=' aexpr
            | NAME '=' aexpr
```

ifexpr:

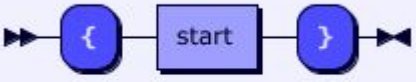



```
ifexpr   ::= 'if' comp 'do' block ( 'else' 'do' block )?
```

whileexpr:

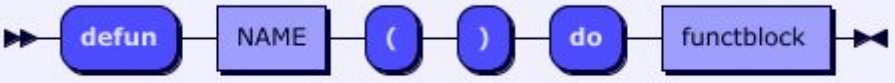

```
whileexpr
          ::= 'while' comp 'do' block
```

block:



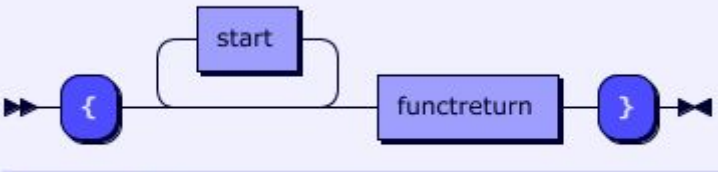

```
block    ::= '{' start '}'
```

funct:




```
funct    ::= 'defun' NAME '(' ')' 'do' functblock
```

functblock:




```
functblock
        ::= '{' start* functreturn '}'
```

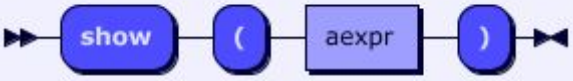
functreturn:


```
functreturn
        ::= 'returns' aexpr ';'
```

functcall:

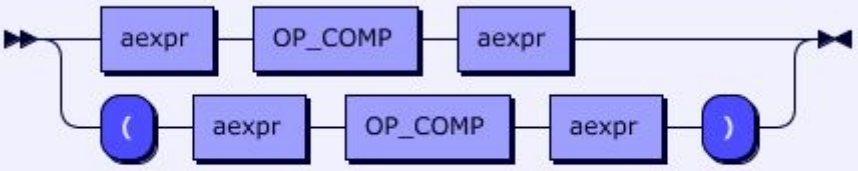

```
functcall
        ::= NAME '(' ')'
```

print:



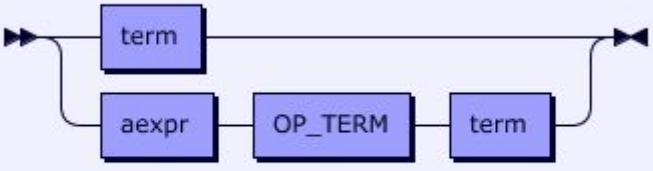

```
print    ::= 'show' '(' aexpr ')'
```

comp:



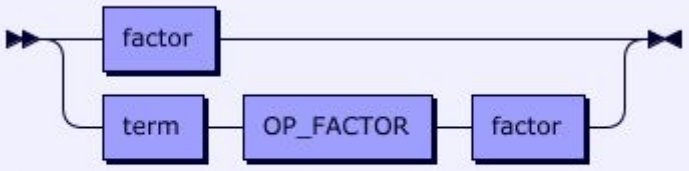

```
comp     ::= aexpr OP_COMP aexpr
           | '(' aexpr OP_COMP aexpr ')'
```

aexpr:



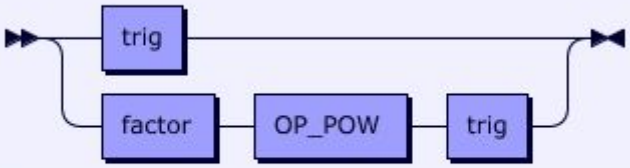

```
aexpr    ::= term
           | aexpr OP_TERM term
```

term:



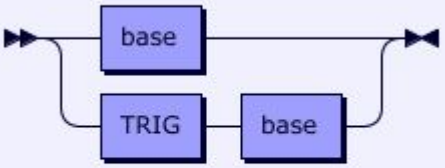

```
term     ::= factor
           | term OP_FACTOR factor
```

factor:



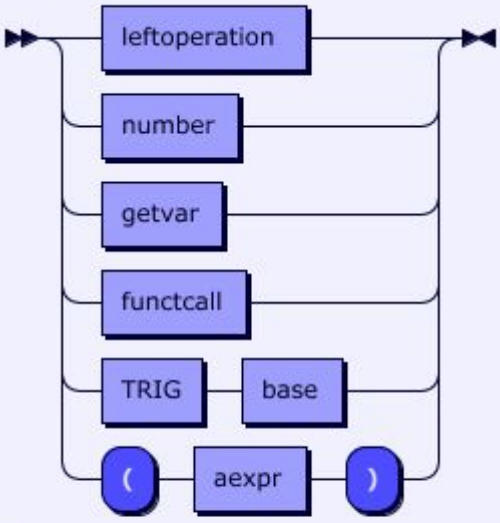

```
factor   ::= trig
           | factor OP_POW trig
```

trig:



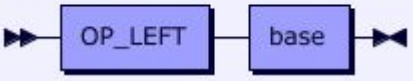

```
trig     ::= base
           | TRIG base
```

base:



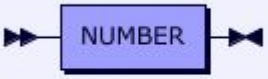

```
base     ::= leftoperation
           | number
           | getvar
           | functcall
           | TRIG base
           | '(' aexpr ')'
```

leftoperation:



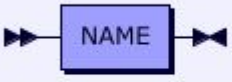

```
leftoperation
         ::= OP_LEFT base
```

number:



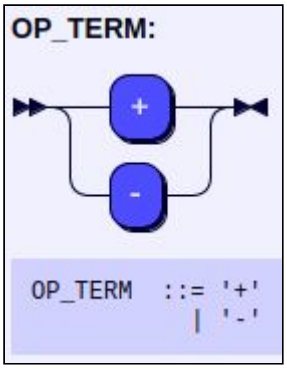

```
number   ::= NUMBER
```

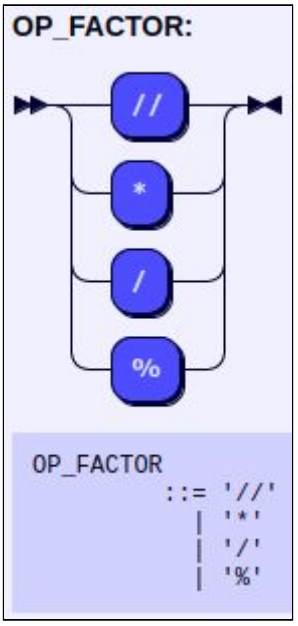
getvar:

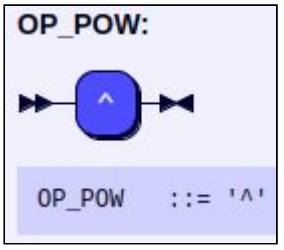


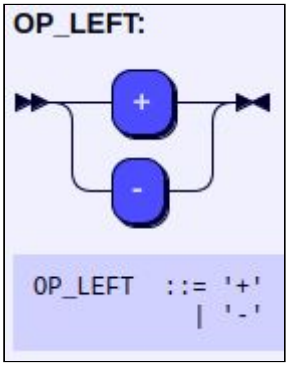

```
getvar   ::= NAME
```

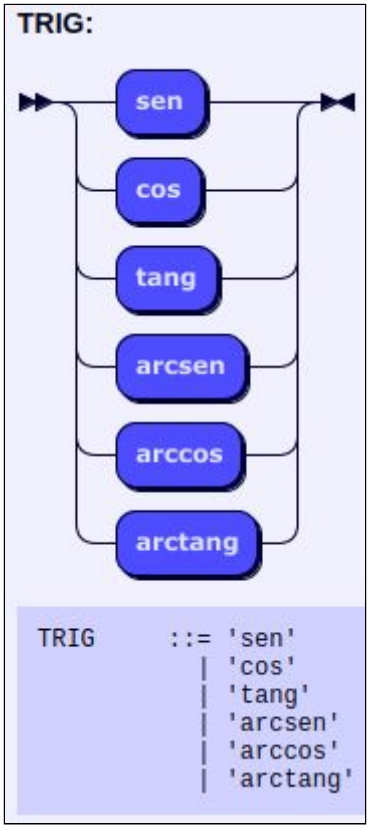
Conjunto de imagens que representam os símbolos terminais da linguagem *DMH*:

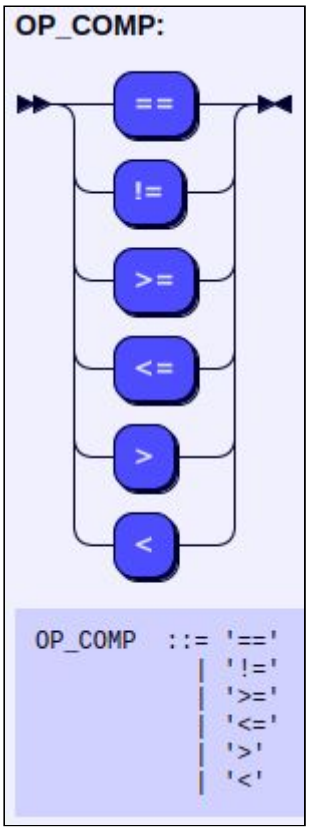


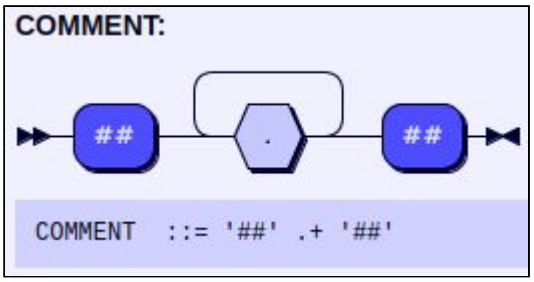


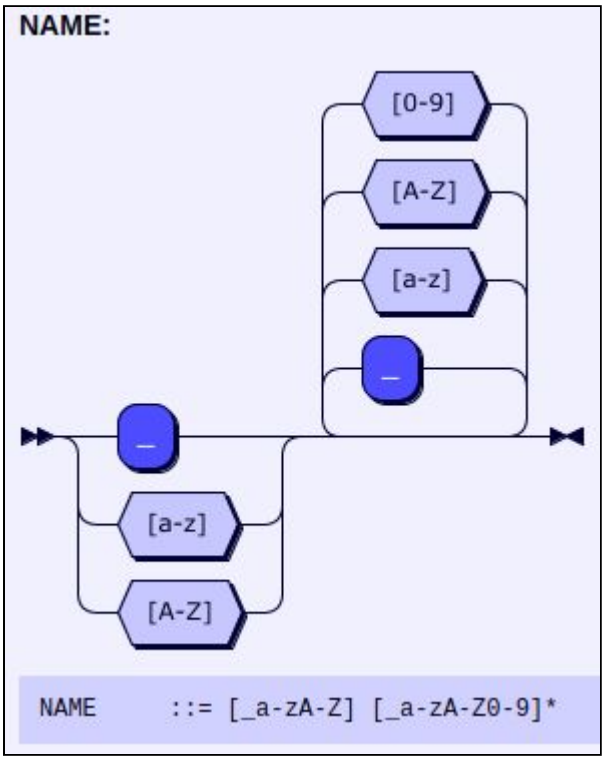


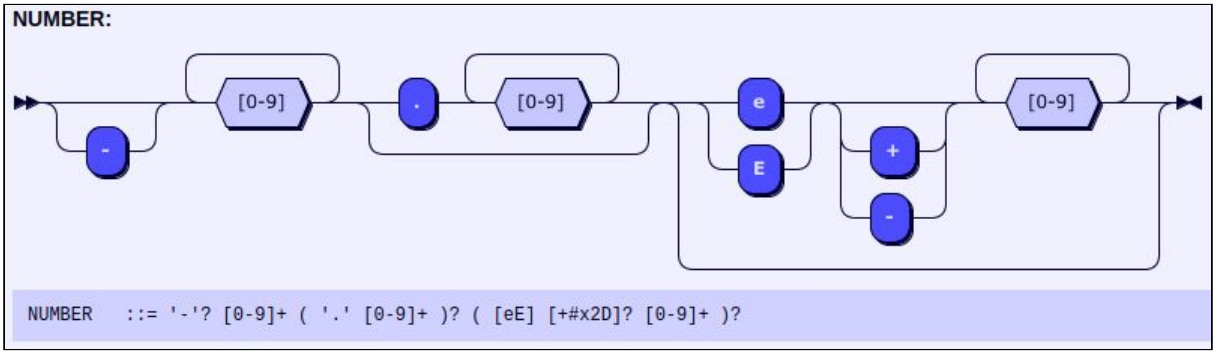












2.3 Exemplos de execução do código da linguagem DMH

Abaixo estão alguns exemplos da utilização da linguagem de forma a ilustrar como são realizadas as construções sintáticas e a possibilidade de seu uso. Uma observação importante que todos exemplos possuem links para sua correspondente *AST (Abstract Syntax Tree)*, onde não foram colocadas as imagens no relatório devido às dimensões e a ruim visibilidade. Logo basta clicar nos links e checar as árvores.

2.3.1 Exemplo 1: manipulação de variáveis

A imagem abaixo mostra o código fonte do *teste1_variaveis.dmh* com simples código, onde basicamente são realizadas declarações, alterações de valores e

recuperação de dados das variáveis, onde estas são sempre de escopo *global*, além de “printar” na tela com os valores de cada uma das variáveis utilizando a palavra reservada *show*. Importante perceber que na linguagem tudo que estiver entre “## . ## “ é ignorado, pois são comentários comumente utilizados em linguagens de programação.

```
## Declarando variáveis que são sempre de escopo global ##
var a = 1;
var b = 2;
var c = 3;

## Printando as variáveis definidas ##
show(a); show(b); show(c);

## Alterando o valor das variáveis definidas com cálculo de expressões matemáticas ##
a = +++2 - 4.0 / ----1.; ## a = -2 ##
b = -((2+2)*2)-((2-0)+2); ## b = -12 ##
c = (-2.3)^2 + 2.2E1 * 2e-12 + 1e+3; ## c = 1005.29 ##

## Printando as modificações realizadas nas variáveis ##
show(a); show(b); show(c);
```

Figura 3 - Código fonte do arquivo *teste1_variaveis.dmh*.

A imagem abaixo mostra saída produzida após a execução do arquivo.

```
Enter the filename.dmh to run or press enter to use shell:
teste1_variaveis.dmh

Execution output:

1.0
2.0
3.0
-2.0
-12.0
1005.290000000044

Finished program execution.
```

Figura 4 - Resultado de saída do arquivo *teste1_variaveis.dmh*.

A seguir está o link da imagem da *AST* gerada correspondente da execução deste código:
https://github.com/cardepaula/trabalho-final-lfa-DMH/blob/master/ast_outfiles/teste1_variaveis.dmh.png?raw=true

2.3.2 Exemplo 2: estrutura de seleção (*if..else*)

A imagem abaixo mostra o código fonte do arquivo *teste2_estrutura_selecao.dmh*, o qual basicamente mostra a utilização da estrutura de seleção (*if..else*) implementada na linguagem. Caso o condicional do *if* seja verdadeiro é executado todo seu bloco de instruções, caso contrário será executado o que está no *else*, isso caso exista o *else*, pois ele é opcional.

```
## Declarando variáveis ##
var a = 4.32;
var b = 3;
var value = 0;

## Utilizando estrutura de repetição(if..else) ##
if (a >= b) do {
    show(a);
    value = sen(60) * 12 - 321 // 213;
}
else do {
    show(b);
    value = cos(60) * 12 - 321 // 215;
};

## Printando value e o arco tangente de value ##
show(value);
show(arctang(value));
```

Figura 5 - Código fonte do arquivo *teste2_estrutura_selecao.dmh*.

A imagem abaixo mostra saída produzida após a execução do arquivo.

```
Enter the filename.dmh to run or press enter to use shell:
teste2_estrutura_selecao.dmh

Execution output:

4.32
9.3923048456
0.1624815306

Finished program execution.
```

Figura 6 - Resultado de saída do arquivo *teste2_estrutura_selecao.dmh*.

A seguir está o link da imagem da *AST* gerada correspondente da execução deste código:

https://github.com/cardepaula/trabalho-final-lfa-DMH/blob/master/ast_outfiles/teste2_estrutura_selecao.dmh.png?raw=true

2.3.3 Exemplo 3: estrutura de repetição (while)

A imagem abaixo mostra o código fonte do arquivo *teste3_estrutura_repeticao.dmh*, o qual basicamente mostra a utilização da estrutura de repetição (while) implementada na linguagem. Enquanto a condição da repetição for verdadeira são executados todas instruções de seu bloco, caso contrário não são mais executados.

```
## Declarando variáveis ##
var count = 0;
var limite = 5;

## Utilizando estrutura de repetição(while) ##
## Basicamente o código printa o incremento de count até limite e
depois o decremento de count até 0 ##
while (count <= limite) do {
    show(count);
    count = count + 1;
};

count = count - 1;

while (count > 0) do {
    count = count - 1;
    show(count);
};
```

Figura 7 - Código fonte do arquivo *teste3_estrutura_repeticao.dmh*.

A imagem abaixo mostra saída produzida após a execução do arquivo.

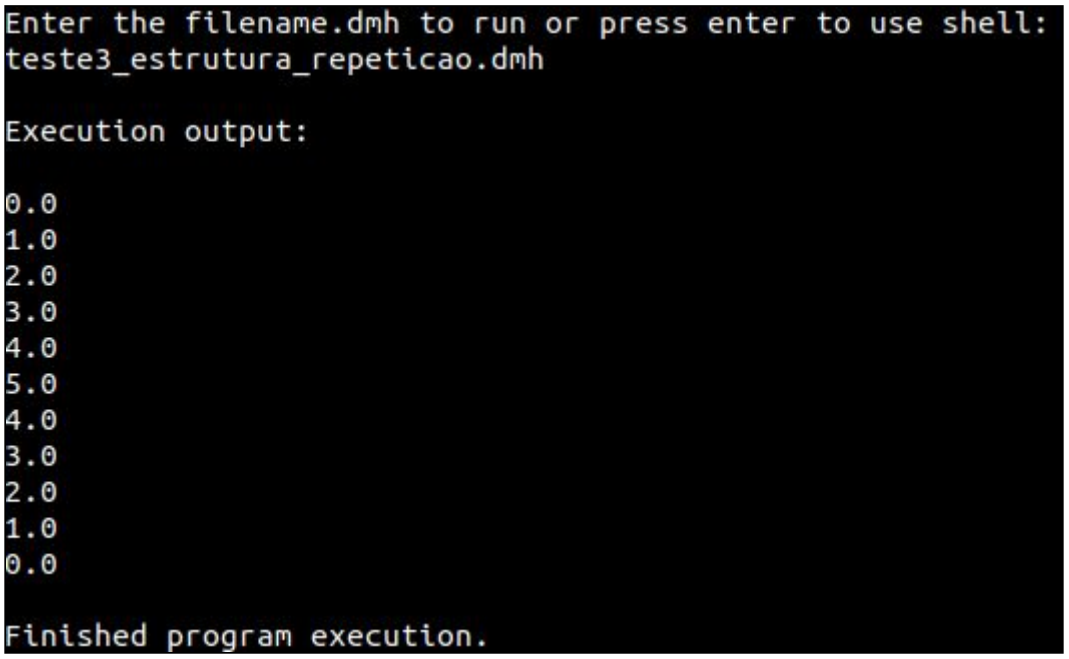


Figura 8 - Resultado de saída do arquivo *teste3_estrutura_repeticao.dmh*.

A seguir está o link da imagem da *AST* gerada correspondente da execução deste código:
https://github.com/cardepaula/trabalho-final-lfa-DMH/blob/master/ast_outfiles/teste3_estrutura_repeticao.dmh.png?raw=true

2.3.4 Exemplo 4: manipulação de funções

A imagem abaixo mostra o código fonte do arquivo *teste4_funcao.dmh*, o qual basicamente mostra a utilização de uma função declarada e logo depois chamada. Importante ressaltar que as funções na linguagem *DMH* não possuem argumentos e sempre retorna um valor numérico em ponto flutuante.

```
## Declarando uma função, onde elas nunca possuem argumentos e sempre retornam um valor numérico ##
defun returnTen() do {
    returns 10;
};

## Lógica que soma o valor de 10 a variável declarada x ##
var x = 0;
show(x);

x = x + returnTen();
show(x);
```

Figura 9 - Código fonte do arquivo *teste4_funcao.dmh*.

A imagem abaixo mostra saída produzida após a execução do arquivo.

```
Enter the filename.dmh to run or press enter to use shell:
teste4_funcao.dmh

Execution output:

0.0
10.0

Finished program execution.
```

Figura 10 - Resultado de saída do arquivo *teste4_funcao.dmh*.

A seguir está o link da imagem da *AST* gerada correspondente da execução deste código:
https://github.com/cardepaula/trabalho-final-lfa-DMH/blob/master/ast_outfiles/teste4_funcao.dmh.png?raw=true

2.3.5 Exemplo 5: cálculo de IMC (Índice de Massa Corporal)

Abaixo está a imagem com código fonte do arquivo *teste5_calculo_imc.dmh*, que realiza o cálculo de IMC dado uma altura e peso. Note que neste exemplo já mostra uma aplicação mais próximo da realidade do dia a dia.

```
## Indice de classificacao do IMC ##
## Classificacao 1: Magreza ##
## Classificacao 2: Saudavel ##
## Classificacao 3: Sobrepeso ##
## Classificacao 4: Obesidade Grau I ##
## Classificacao 5: Obesidade Grau II ##
## Classificacao 6: Obesidade Grau III ##
```

Figura 11 - Classificações correspondentes do IMC.

```
## Funcao que calcula o IMC ##
defun calculaIMC() do {
    var imc = peso / (altura^2);
    var indice = 0;

    if ( imc  < 18.5) do {
        indice = 1;
    } else do {
        if ( imc  < 25) do {
            indice = 2;
        } else do {
            if ( imc  < 30) do {
                indice = 3;
            } else do {
                if ( imc  < 35) do {
                    indice = 4;
                } else do {
                    if ( imc  < 40) do {
                        indice = 5;
                    } else do {
                        indice = 6;
                    };
                };
            };
        };
    };

    returns indice;
};

## Variaveis de scopo global ##
var peso = 51.400;
var altura = 1.64;
var classificacao = calculaIMC();

## Printa a classificação baseado no IMC calculado ##
show(classificacao);
```

Figura 12 - Código fonte do arquivo *teste5_calculo_imc.dmh*.

A imagem abaixo mostra saída produzida após a execução do arquivo.

```
Enter the filename.dmh to run or press enter to use shell:
teste5_calculo_imc.dmh

Execution output:

2.0

Finished program execution.
```

Figura 13 - Resultado de saída do arquivo *teste5_calculo_imc.dmh*.

A seguir está o link da imagem da *AST* gerada correspondente da execução deste código:

https://github.com/cardepaula/trabalho-final-lfa-DMH/blob/master/ast_outfiles/teste5_calculo_imc.dmh.png?raw=true

2.3.6 Exemplo 6: cálculo de fatorial

Abaixo está a imagem com código fonte do arquivo *teste6_calculo_fatorial.dmh* que realiza o cálculo do fatorial dado um valor n. Neste exemplo também mostra também uma outra aplicação em que a linguagem seria suficientemente capaz de resolver.

```
## Função que calcula o fatorial de n ##
defun calculaFatorial() do {
    var count = 1;
    var limite = n;
    var f = 1;

    ## Se o numero for menor que zero, a funcao retornará -1, pois nao existe fatorial de numero negativo ##
    if (n < 0) do {
        f = -1;
    } else do {
        ## Por definicao, fatorial de zero é um. ##
        if ( n == 0) do {
            f = 1;
        } else do {
            while (count <= limite) do {
                f = f * count;
                count = count + 1;
            };
        };
    };


    returns f;
};

## Definição de n, variável de escopo global, e chamada da função que calcula o fatorial de n ##
var n = 5;
var fatorial = calculaFatorial();

show(fatorial);
```

Figura 14 - Código fonte do arquivo *teste6_calculo_fatorial.dmh*.

A imagem abaixo mostra saída produzida após a execução do arquivo.

```
Enter the filename.dmh to run or press enter to use shell:
teste6_calculo_fatorial.dmh

Execution output:

120.0

Finished program execution.
```

Figura 15 - Resultado de saída do arquivo *teste6_calculo_fatorial.dmh*.

A seguir está o link da imagem da *AST* gerada correspondente da execução deste código:
https://github.com/cardepaula/trabalho-final-lfa-DMH/blob/master/ast_outfiles/teste6_calculo_fatorial.dmh.png?raw=true

2.3.7 Exemplo 7: verificação de números primos

Abaixo está a imagem com código fonte do arquivo *teste7_verifica_primos.dmh*, que checa se um número é primo ou não. Note que este código de uma das maneiras mais otimizadas. Outro ponto importante é a chamada da função *verificaPar()* dentro da função *verificaNumPrimo()*. Aqui mostra um exemplo em que também pode ser aplicado em um contexto real do dia a dia.

```
## Funcao que verifica se o numero é par ou não ##
## Retorna 1(um) se for par e 0(zero) se for impar ##
defun verificaPar() do {
    var e_par = 0;
    var result = n%2;

    if (result == 0) do {
        e_par = 1;
    };

    returns e_par;
};
```

Figura 16 - Código do arquivo *teste7_verifica_primos.dmh (função verificaPar())*.

```
## Funcao que verifica se o numero é primo ou não ##
## Retorna 1(um) se for primo e 0(zero) se nao for primo ##
defun verificaNumPrimo() do {
    var e_primo = 0;
    verificaPar();

    var i = 3;
    var limite = n/2;
    var divisores = 0;
    var resto = 0;

    if ( n > 1 ) do {
        if ( n == 2 ) do {
            e_primo = 1;
        } else do {
            if (e_par == 0) do {
                while ( i < limite) do {
                    resto = n%i;

                    if (resto == 0) do {
                        divisores = divisores + 1;
                    };
                    i = i + 1;
                };
                if (divisores == 0) do {
                    e_primo = 1;
                };
            };
        };
    };

    returns e_primo;
};
```

Figura 17 - Código do arquivo *teste7_verifica_primos.dmh (função verificaNumPrimo())*.

```
var n = 977;
var primo = verificaNumPrimo();
show(primo);
```

Figura 18 - Chamada de *verificaNumPrimo()*.

A imagem abaixo mostra saída produzida após a execução do arquivo.

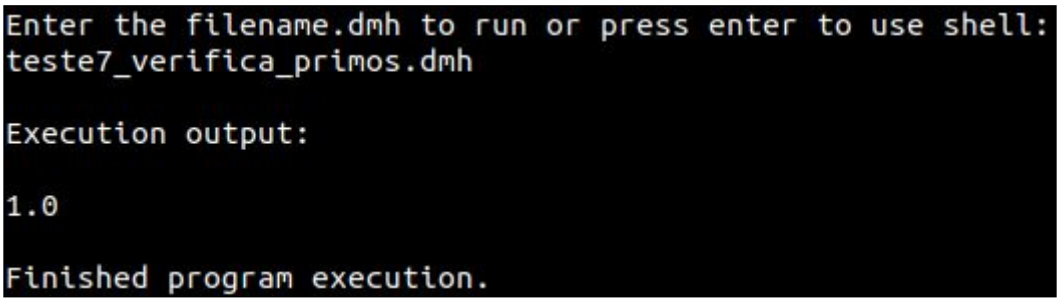


Figura 19 - Resultado de saída do arquivo *teste7_verifica_primos.dmh*.

A seguir está o link da imagem da *AST* gerada correspondente da execução deste código:
https://github.com/cardepaula/trabalho-final-lfa-DMH/blob/master/ast_outfiles/teste7_verifica_primos.dmh.png?raw=true

Existem outros 3 arquivos exemplos para testes da linguagem. Para executá-los basta, caso esteja executando a aplicação, seguir os mesmos passos que os anteriores, colocando o nome do *arquivo.dmh* desejado.

3. Definição da AST (*Abstract Syntax Tree*)

A estrutura de dados definida criada para representar a ***AST (Abstract Syntax Tree)*** é simples, onde são utilizados duas classes principais: *DMHParser* e *DMHEvaluateTree*. Abaixo está um diagrama de classes que representa essa estrutura em alto nível.

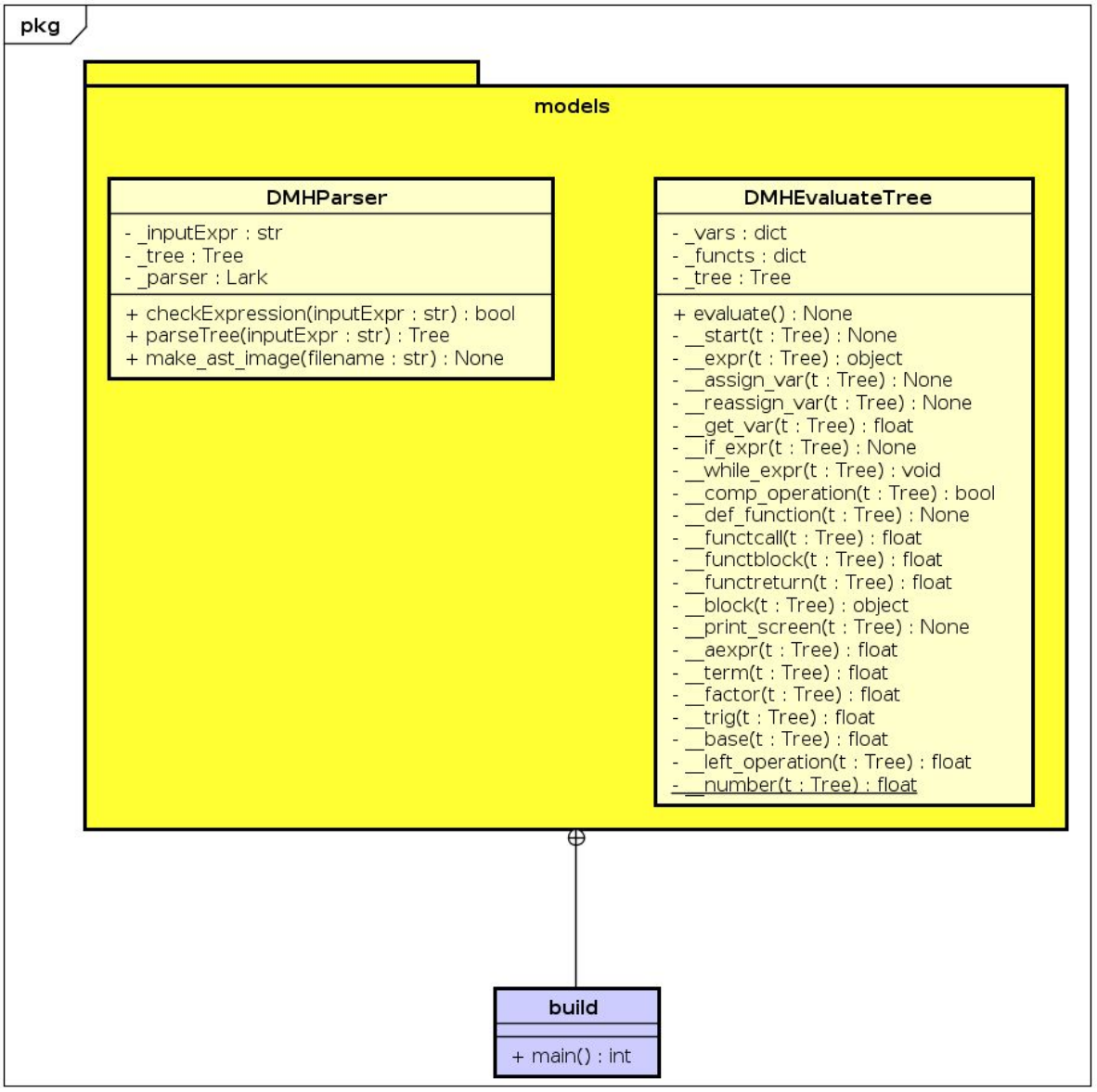


Figura 20 - Diagrama de classes para a representação da *AST*.

A seguir está o link da imagem do diagrama de classes, pois a imagem pode não estar tão nítida e de fácil visualização:
https://github.com/cardepaula/trabalho-final-lfa-DMH/blob/master/documentation/images/DMH%20class%20diagram.png?raw=true

Note que a classe *DMHEvaluateTree* possui conjunto de métodos privados que representam as *regras ou símbolos não terminais* da gramática. Já os símbolos *terminais* são utilizados dentro das regras representando as folhas da *AST* passada para a instância do objeto *DMHEvaluateTree*. O módulo *build.py* que é responsável por manipular os objetos dessas classes de forma que o usuário pode interagir tanto com entrada de um arquivo do formato .dmh quanto do console (CLI).

Abaixo uma tabela que descreve cada classe da utilizada na construção da linguagem.

Descrição das classes do diagrama de classes	
Classe	**Objetivo**
DMHParser	Classe responsável por realizar o parser tree (AST) da expressão/código passada como entrada seguindo as regras definidas pela gramática da linguagem DMH.
DMHEvaluateTree	Classe responsável por realizar o *evaluation* da árvore (AST) da expressão/código, executando assim o código.

Tabela 1 - Descrição das principais classes para representação da *AST*.

4. Conclusão

A implementação de uma DSL (Domain Specific Language) é importante, pois mostra os princípios de uma linguagem de programação, abrindo margem para uma visão mais crítica.

É perceptível notar que quando se está criando uma *DSL*, o intuito é que ela seja concisa e que descreve as informações de maneira clara, limpa, atacando um objetivo específico, ou seja, o foco da linguagem.

Entretanto para todos os prós, existem os seus contras e na criação de uma *DSL*, percebe-se na correta escrita da gramática, o qual caso não seja definida de forma correta causa muito retrabalho e a complexidade que ela pode tomar ao longo do desenvolvimento.

5. Referências Bibliográficas

Lark Parser

- GitHub: https://github.com/lark-parser/lark
- Documentação: https://lark-parser.readthedocs.io/en/latest/

Railroad Diagram Generator

- https://www.bottlecaps.de/rr/ui

Wikipedia

- https://en.wikipedia.org/wiki/Domain-specific_language
- https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form
- https://en.wikipedia.org/wiki/Syntax_diagram