****

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**](#_9dqwwen31lfc) **4**

[**2. Definição da DSL (Domain Specific Language)**](#_e2mzaag5zh4v) **5**

[2.1 EBNF (Extended Backus-Naur Form) da linguagem DMH](#_n9qx5c5npo1h) 5

[2.2 Diagramas de sintaxe da linguagem DMH](#_chlyrvfmd6ic) 8

[2.3 Exemplos de execução do código da linguagem DMH](#_pnhs6od5qln4) 13

[2.3.1 Exemplo 1: manipulação de variáveis](#_63psnjp6f8y2) 13

[2.3.2 Exemplo 2: estrutura de seleção (if..else)](#_jm0p8jixtri5) 14

[2.3.3 Exemplo 3: estrutura de repetição (while)](#_tjyqsaxs4lk3) 15

[2.3.4 Exemplo 4: manipulação de funções](#_pk99hnlauakc) 17

[2.3.5 Exemplo 5: cálculo de IMC (Índice de Massa Corporal)](#_vlgqscnsaxln) 17

[2.3.6 Exemplo 6: cálculo de fatorial](#_bbp715wt59rp) 19

[2.3.7 Exemplo 7: verificação de números primos](#_7hk2xhq3qfam) 20

[**3. Definição da AST (Abstract Syntax Tree)**](#_sir5v3iyl9t4) **23**

[**4. Conclusão**](#_6kwqjy3h0ir1) **25**

[**5. Referências Bibliográficas**](#_a7flrqbnsqf4) **26**

# 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**](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*.

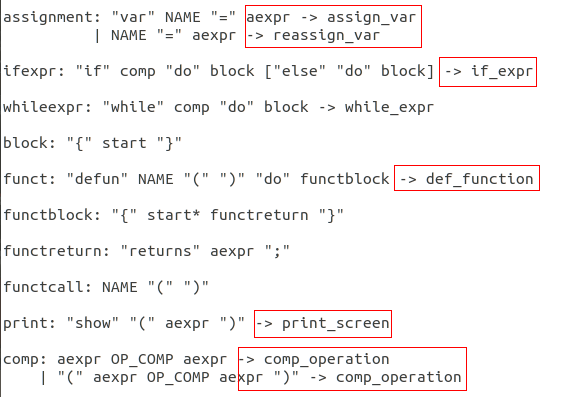


Figura 1 - Renomeação de regrasna biblioteca *Lark* com a utilização do *aliases*.

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.

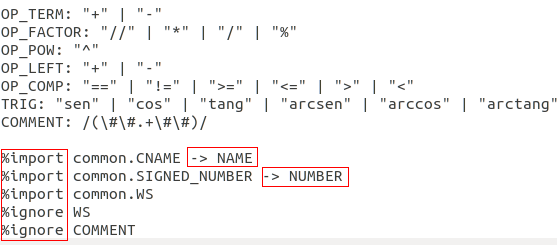


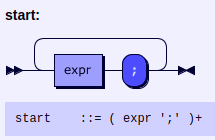
Figura 2 - Renomeação de símbolos terminais na biblioteca *Lark* com a utilização do *aliases*.

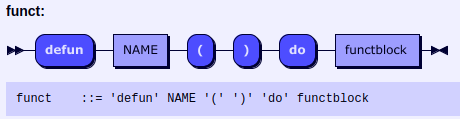
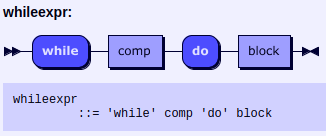
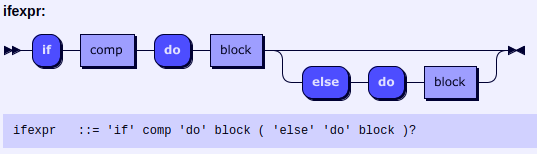
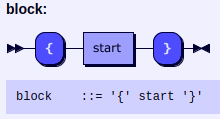
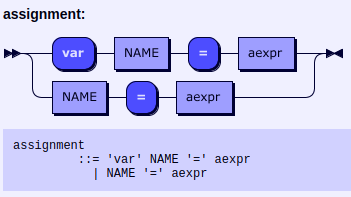
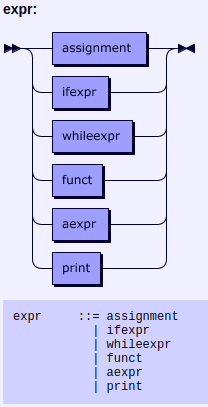
Note que além da renomeação das símbolos terminais é também possível as palavras reservadas *import*, para importar recursos disponíveis da biblioteca, como por exemplo na figura, onde são importados símbolos terminais correspondentes para nomeação de variáveis, funções, classes e afins (*commom.CNAME*) 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.

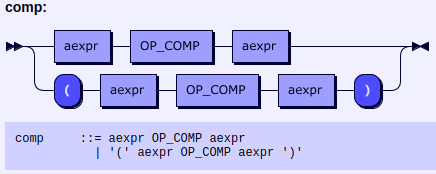
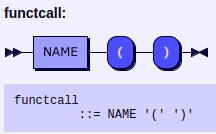
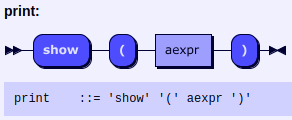
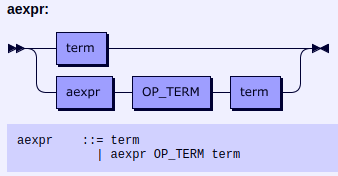
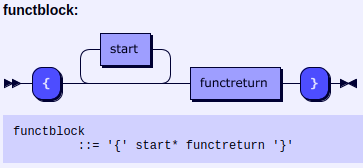
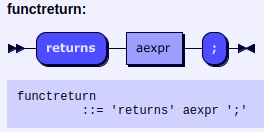
## 2.2 Diagramas de sintaxe da linguagem *DMH*

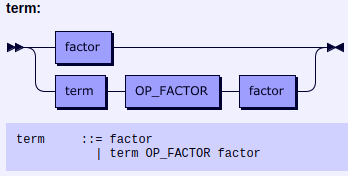
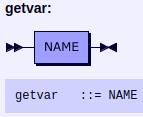
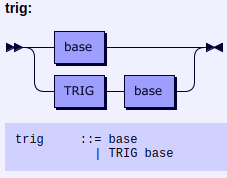
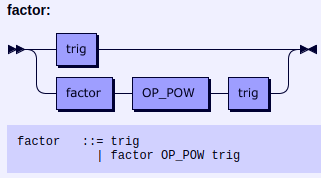
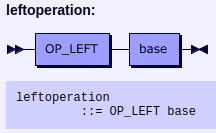
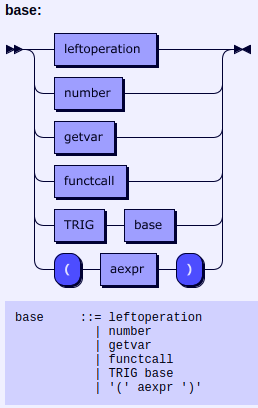
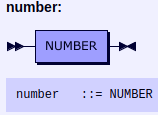
O ***diagrama de sintaxe*** representa uma maneira diferente de também representar a linguagem livre de contexto, representando o *EBNF* em um formato gráfico. Abaixo estão o conjunto de imagens que representa a linguagem dessa forma.

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

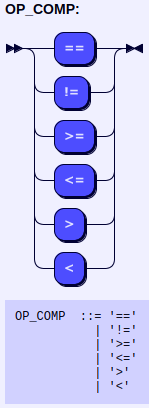
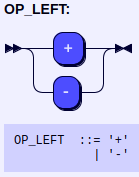
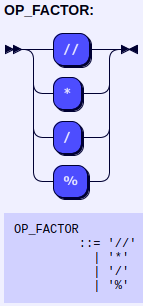
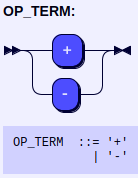
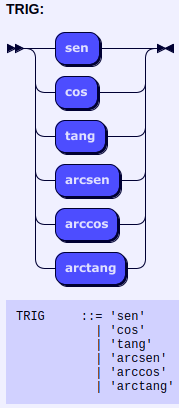
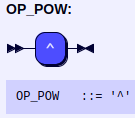


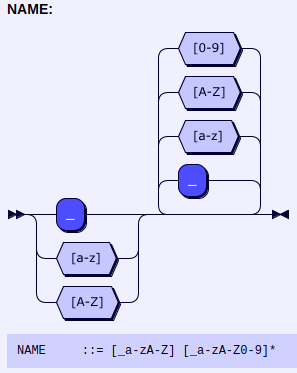
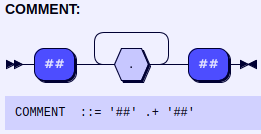
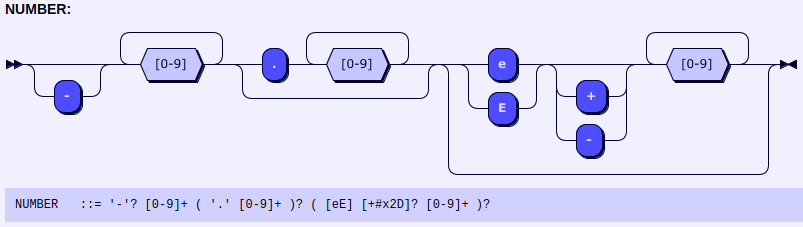






Conjunto de imagens que representam os símbolos terminais da linguagem *DMH*:





Caso deseje visualizar mais informações sobre o ***diagrama de sintaxe*** da linguagem implementada basta acessar o arquivo ***syntax\_diagram.html***, onde nele contém todas as regras de produção e símbolos terminais definidos na gramática de maneira mais detalhada. O arquivo se encontra no diretório *./documentation/images/diagram\_syntax/****syntax\_diagram.html***.

## 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.

# 

Figura 3 - Código fonte do arquivo *teste1\_variaveis.dmh*.

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

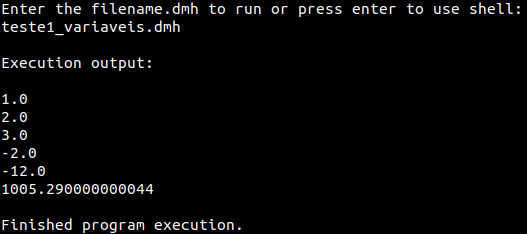


Figura 4 - Resultado de saída do arquivo *teste1\_variaveis.dmh*.

A seguir está o link da imagem da *AST* geradacorrespondente 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.

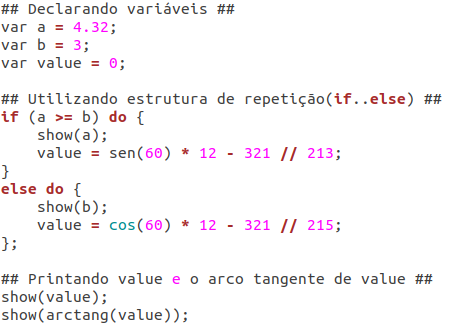


Figura 5 - Código fonte do arquivo *teste2\_estrutura\_selecao.dmh*.

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

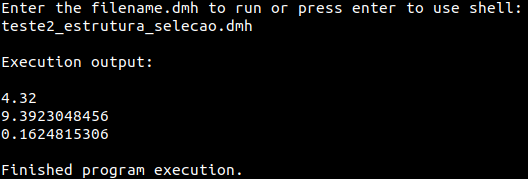


Figura 6 - Resultado de saída do arquivo *teste2\_estrutura\_selecao.dmh*.

A seguir está o link da imagem da *AST* geradacorrespondente 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.

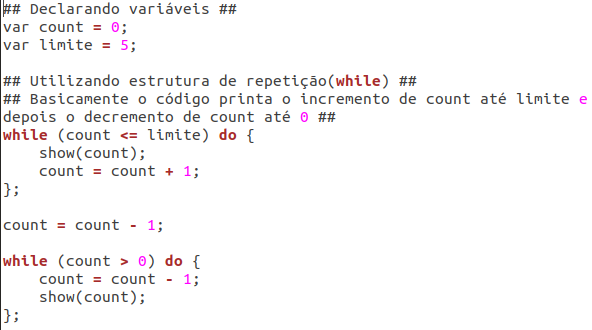


Figura 7 - Código fonte do arquivo *teste3\_estrutura\_repeticao.dmh*.

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

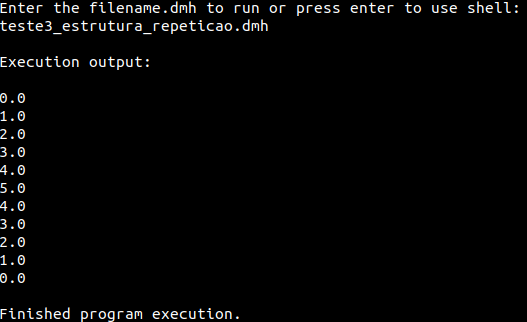


Figura 8 - Resultado de saída do arquivo *teste3\_estrutura\_repeticao.dmh*.

A seguir está o link da imagem da *AST* geradacorrespondente 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.

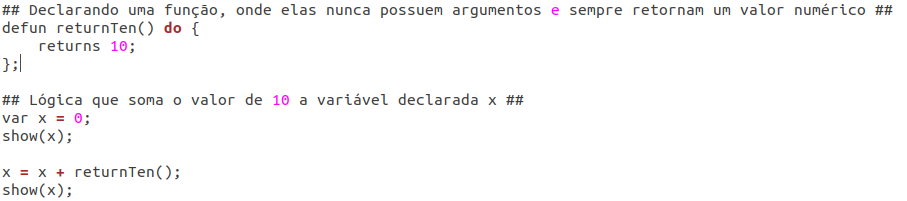


Figura 9 - Código fonte do arquivo *teste4\_funcao.dmh*.

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

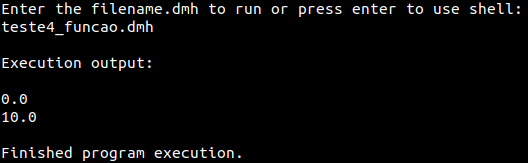


Figura 10 - Resultado de saída do arquivo *teste4\_funcao.dmh*.

A seguir está o link da imagem da *AST* geradacorrespondente 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.

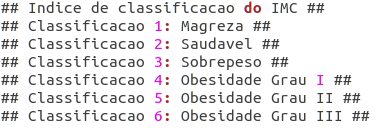


Figura 11 - Classificações correspondentes do IMC.

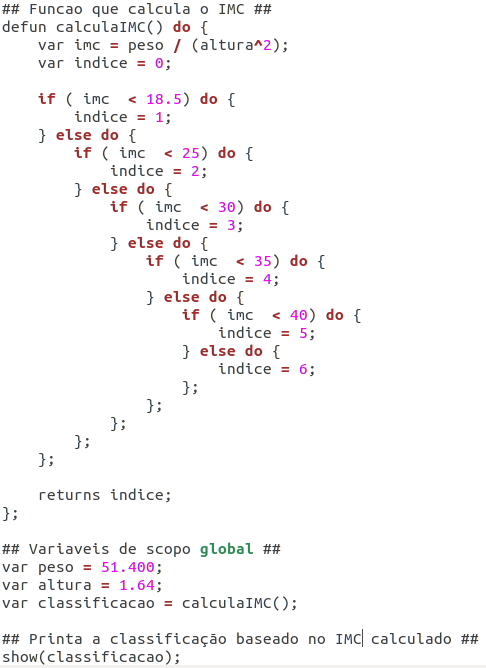


Figura 12 - Código fonte do arquivo *teste5\_calculo\_imc.dmh*.

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

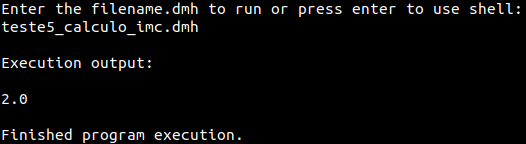


Figura 13 - Resultado de saída do arquivo *teste5\_calculo\_imc.dmh*.

A seguir está o link da imagem da *AST* geradacorrespondente 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.

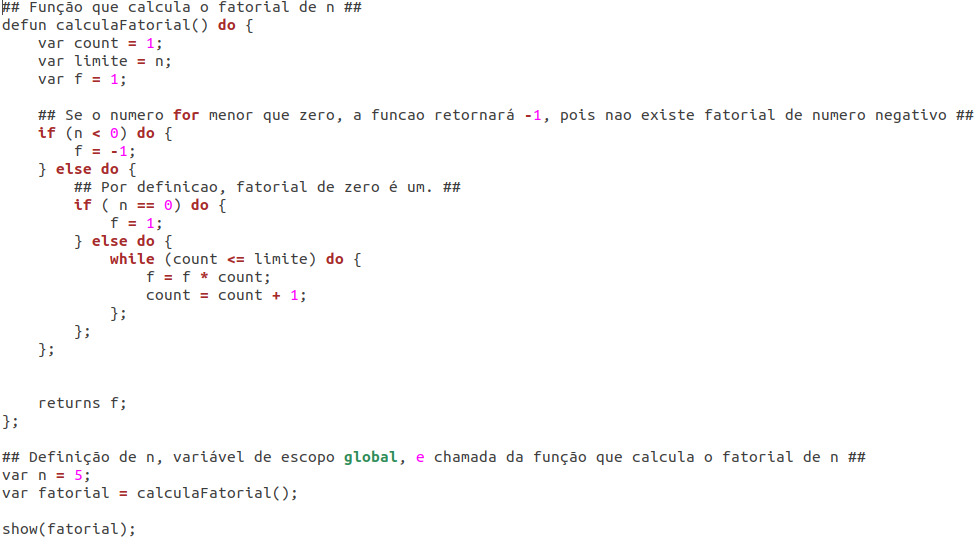


Figura 14 - Código fonte do arquivo *teste6\_calculo\_fatorial.dmh*.

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

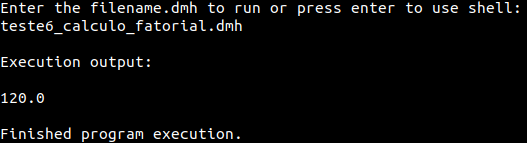


Figura 15 - Resultado de saída do arquivo *teste6\_calculo\_fatorial.dmh*.

A seguir está o link da imagem da *AST* geradacorrespondente 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.

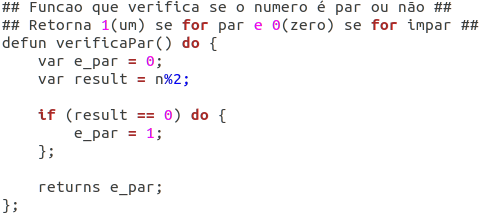


Figura 16 - Código do arquivo *teste7\_verifica\_primos.dmh (função verificaPar())*.

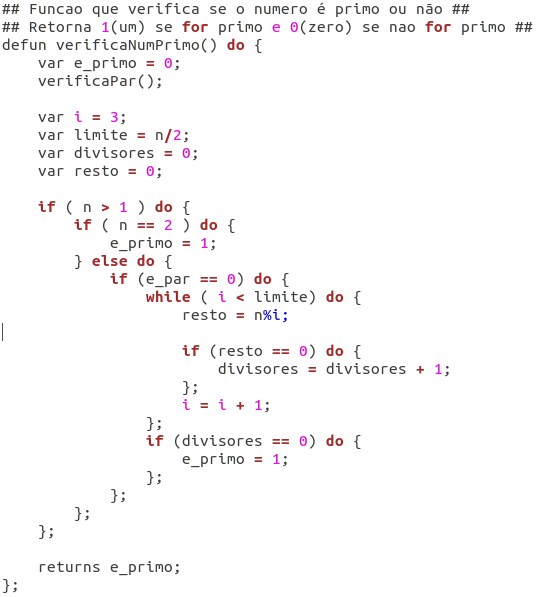


Figura 17 - Código do arquivo *teste7\_verifica\_primos.dmh (função verificaNumPrimo())*.



Figura 18 - Chamada de *verificaNumPrimo().*

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

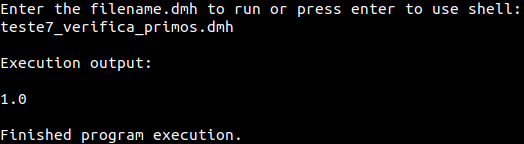


Figura 19 - Resultado de saída do arquivo *teste7\_verifica\_primos.dmh*.

A seguir está o link da imagem da *AST* geradacorrespondente 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.

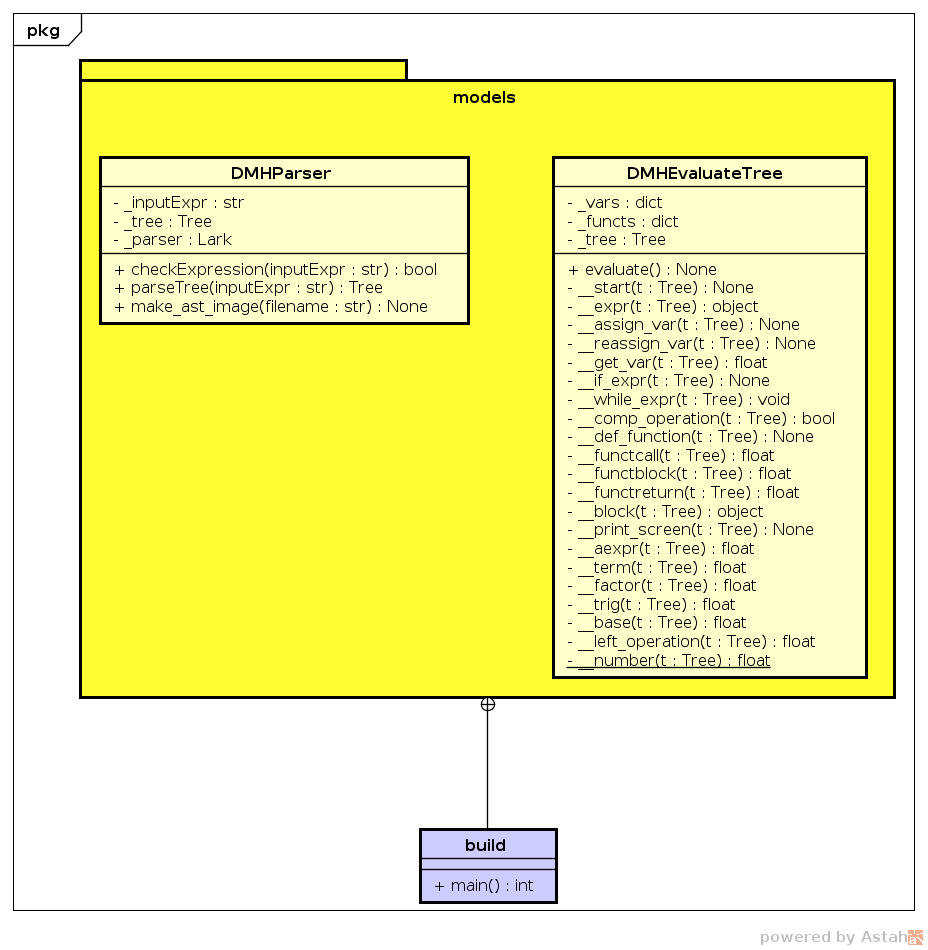


Figura 20 - Diagrama de classes para a representação da *AST*.

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>