# Interpretador para linguagem Meia-Lua

**DIM0437**
**LINGUAGEM DE PROGRAMAÇÃO: CONCEITOS E PARADIGMAS**

Erick Willy Martins Machado
Francisco de Paiva
Gabriela Cavalcante da Silva
Giovanni Franco de Paula Rosario

# Sumário

# 1. Introdução

Com a propagação da computação nas mais diversas áreas, diferentes abordagens são propostas para diferentes problemas. Tendo como consequência o desenvolvimento de diferentes Linguagens de Programação com diferentes objetivos e propósitos para atender requisitos de domínios específicos.

A partir disso a linguagem meia-lua foi idealizada, inspirada na linguagem lua, seu objetivo principal é fornecer flexibilidade e aperfeiçoamento para o domínio de Inteligência Artificial, com ênfase em Jogos Digitais. O domínio de IA é caracterizado pelo forte uso de computação simbólica ao invés de computação numérica.

O objetivo do projeto a seguir é desenvolver uma ferramenta capaz de gerar de um interpretador para a linguagem proposta a partir de descrições léxicas, sintáticas e semânticas dadas para a linguagem meia-lua.

O projeto pode ser encontrado no repositório github.com/I-am-Gabi/meia-lua.

## 2. Visão Geral do Projeto

---

De modo geral, um interpretador de programa trabalha geralmente a partir do fluxo mostrado na figura a seguir, onde duas entradas são recebidas pelo interpretador, a primeira sendo o código fonte escrito na linguagem a ser interpretada, e uma entrada I  o qual tradicionalmente seria  entregue a versão para execução do programa. Após as duas entradas passarem pelo interpretador, é gerada uma saída do programa escrito como se o programa fosse executado sobre a entrada própria da linguagem L apontada na imagem.

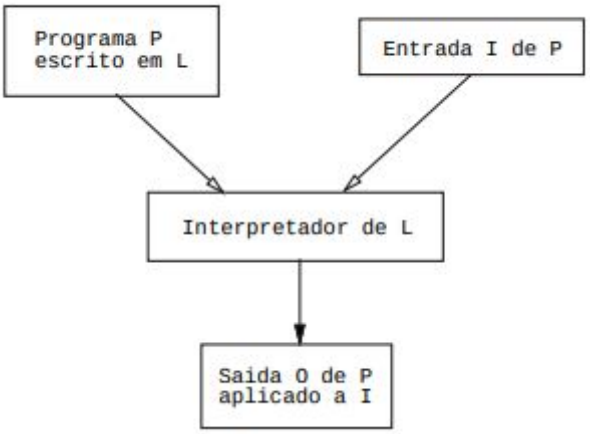



Figura 1 - Fluxo do funcionamento de um interpretador de linguagem de programação

No caso do presente projeto, o fluxo do trabalho do interpretador terá um caminho um pouco diferente, a primeira diferença é que o interpretador não receberá o código totalmente puro escrito na linguagem meia-lua, e sim uma descrição de estrutura sintática e semântica acerca do comportamento de cada componente da linguagem que compõem o código escrito. A partir disso, faz a necessidade de um tradutor para codificar os programas escritos em meia-lua e convertê-lo (traduzir) numa estrutura adequada para a leitura e interpretação de regras que foram definidas no interpretador, escrito na linguagem de programação funcional Haskell.

Simplificando, pode-se separar em três etapas a implementação e execução do interpretador e tradutor para receber um código escrito na sintaxe meia-lua.

1. Primeiro, foi implementado um interpretador para a linguagem meia-lua, escrito em Haskell, e a geração de um tradutor, também escrito em Haskell, capaz de transpor o código escrito na linguagem do projeto para a estrutura adequada para o interpretador.
2. Após a implementação, o fluxo de execução é dado por uma entrada de um programa escrito em meia-lua que é traduzir para as estruturas pré-definidas para o interpretador escritos em haskell, após isso o resultado dessa transposição para haskell é enviado como entrada para o interpretador.
3. Por último, a transposição gerada no passo dois é inserida no interpretador, resultando na execução do programa sobre sua entrada de código própria.

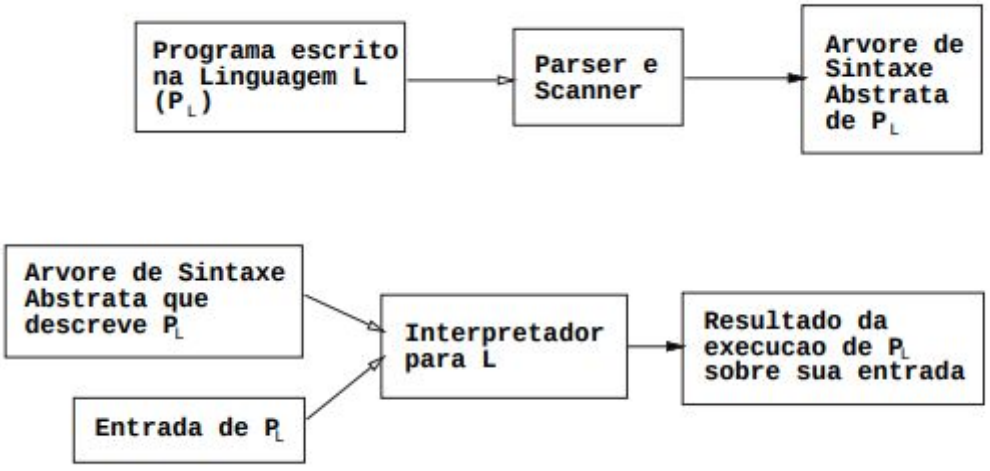



Figura 2 - Fluxo do funcionamento do interpretador e respectivos analisadores

O esquema acima deixa mais claro o funcionamento e compreensão das ferramentas desenvolvidas.

## 3. Transformação do código-fonte em unidades léxicas

---

Unidades léxicas são conjuntos de caracteres que representam o resultado de uma interpretação de tokens a partir de regras fornecidas para esses respectivos tokens. No interpretador, foi utilizado a ferramenta para a linguagem Haskell chamada Alex, para auxiliar essa atividade a ser realizada. Onde a partir do código escrito na sintaxe de meia-lua que é recebido pelo interpretador, a ferramenta, analisa toda entrada e os transforma em tokens a partir das regras previamente definidas,

Para descrever os aspectos lexicais da linguagem, foi utilizado a BNF estendida, na qual {a} significa 0 ou mais a's e [a] significa um a opcional. Não-terminais são mostrados como non-terminal, palavras-chave são mostradas como kword e outros símbolos terminais são mostrados como `=´. Após seguidas avaliações e discussões, estes aspectos foram aprimorados até que o desenvolvimento do interpretador em si da linguagem tivesse início, através da linguagem de programação funcional Haskell, através dos analisadores léxicos e sintáticos.

Abaixo, é possível visualizar uma tabela com os tokens e suas respectivas regras de interpretação para o Alex. A seguir dessa tabela, pode-se observar em outra tabela os tipos que os tokens podem assumir.

| **Token** | **Regra Lexical** |
|---|---|
| $digit = 0-9 | digits |
| $alpha = [a-zA-Z] | alphabetic characters |
| $white+ | ; |
| "--".*. | ; |
| int | { \p s -> TypeInt p } |
| float | { \p s -> TypeFloat p } |
| string | { \p s -> TypeString p } |
| bool | { \p s -> TypeBoolean p } |
| $digit+\.$digit+ | { \p s -> FloatLit p (read s) } |
| $digit+ | { \p s -> IntLit p (read s) } |
| \: | { \p s -> Colon p } |
| \; | { \p s -> SemiColon p } |
| \. | { \p s -> EndPoint p } |
| \=\= | { \p s -> SymEq p } |
| \!\= | { \p s -> SymNotEq p } |
| \<\= | { \p s -> SymLessThanEq p } |

| | |
|---|---|
| \>\= | { \p s -> SymGreaterThanEq p } |
| \! | { \p s -> SymNot p } |
| \&\& | { \p s -> SymAnd p } |
| \|\| | { \p s -> SymOr p } |
| true | { \p s -> SymTrue p } |
| false | { \p s -> SymFalse p } |
| \< | { \p s -> SymLessThan p } |
| \> | { \p s -> SymGreaterThan p } |
| \+ | { \p s -> SymOpPlus p } |
| \- | { \p s -> SymOpMinus p } |
| * | { \p s -> SymOpMult p } |
| \/ | { \p s -> SymOpDiv p } |
| \^ | { \p s -> SymOpExp p } |
| \% | { \p s -> SymOpMod p } |
| \+\+ | { \p s -> SymOpPlusPlus p } |
| \-\- | { \p s -> SymOpMinusMinus p } |
| \+\= | { \p s -> SymOpPlusAssign p } |
| \-\= | { \p s -> SymOpMinusAssign p } |
| *\= | { \p s -> SymOpMultAssign p } |
| \/\= | { \p s -> SymOpDivAssign p } |
| \= | { \p s -> Attrib p } |
| \( | { \p s -> OpenParenth p } |
| \) | { \p s -> CloseParenth p } |
| \[ | { \p s -> OpenBracket p } |
| \] | { \p s -> CloseBracket p } |
| \{ | { \p s -> OpenScope p } |
| \} | { \p s -> CloseScope p } |

| | |
|---|---|
| \=\> | { \p s -> SymPtrOp p } |
| \$ | { \p s -> SymAdressOp p } |
| print | { \p s -> Print p } |
| scan | { \p s -> Scan p } |
| do | { \p s -> Do p } |
| for | { \p s -> For p } |
| while | { \p s -> While p } |
| if | { \p s -> If p } |
| then | { \p s -> Then p } |
| else | { \p s -> Else p } |
| procedure | { \p s -> Procedure p } |
| function | { \p s -> Function p } |
| return | { \p s -> Return p } |
| break | { \p s -> Break p } |
| continue | { \p s -> Continue p } |
| table | { \p s -> Table p } |
| $alpha [$alpha $digit _ \']* | { \p s -> Id p s } |
| $alpha [$alpha $digit _ \']* | { \p s -> Id p s } |
| \" [^\"]* \" | { \p s -> StrLit p (firstLast s) } |

Tabela 1 - Tokens associados a sua respectiva regra léxica

| **Tipos de dados de tokens possíveis.** |
|---|
| TypeInt AlexPosn |
| TypeFloat AlexPosn |
| TypeString AlexPosn |
| TypeBoolean AlexPosn |

| Attrib AlexPosn |
| --- |
| OpenParenth AlexPosn |
| CloseParenth AlexPos |
| OpenBracket AlexPosn |
| CloseBracket AlexPosn |
| OpenScope AlexPosn |
| CloseScope AlexPosn |
| Colon AlexPosn |
| SemiColon AlexPosn |
| Comma AlexPosn |
| EndPoint AlexPosn |
| SymPtrOp AlexPosn |
| SymAdressOp AlexPosn |
| Print AlexPosn |
| Scan AlexPosn |
| Do AlexPosn |
| If AlexPosn |
| Then AlexPosn |
| Else AlexPosn |
| Elif AlexPosn |
| For AlexPosn |
| While AlexPosn |
| Procedure AlexPosn |
| Function AlexPosn |
| Return AlexPosn |
| Break AlexPosn |
| Continue AlexPosn |

| Table AlexPosn |
| --- |
| FloatLit AlexPosn Float |
| IntLit AlexPosn Int |
| StrLit AlexPosn String |
| SymOpPlus AlexPosn |
| SymOpMinus AlexPosn |
| SymOpMult AlexPosn |
| SymOpDiv AlexPosn |
| SymOpExp AlexPosn |
| SymOpMod AlexPosn |
| SymOpPlusPlus AlexPosn |
| SymOpMinusMinus AlexPosn |
| SymOpPlusAssign AlexPosn |
| SymOpMinusAssign AlexPosn |
| SymOpMultAssign AlexPosn |
| SymOpDivAssign AlexPosn |
| SymEq AlexPosn |
| SymNotEq AlexPosn |
| SymLessThanEq AlexPosn |
| SymGreaterThanEq AlexPosn |
| SymNot AlexPosn |
| SymAnd AlexPosn |
| SymOr AlexPosn |
| SymTrue AlexPosn |
| SymFalse AlexPosn |
| SymLessThan AlexPosn |
| SymGreaterThan AlexPosn |

| Id AlexPosn String |
|---|

Tabela 2 - Tipos possíveis que os tokens podem assumir para interpretação da ferramenta Alex

## 4. Análise sintática do interpretador

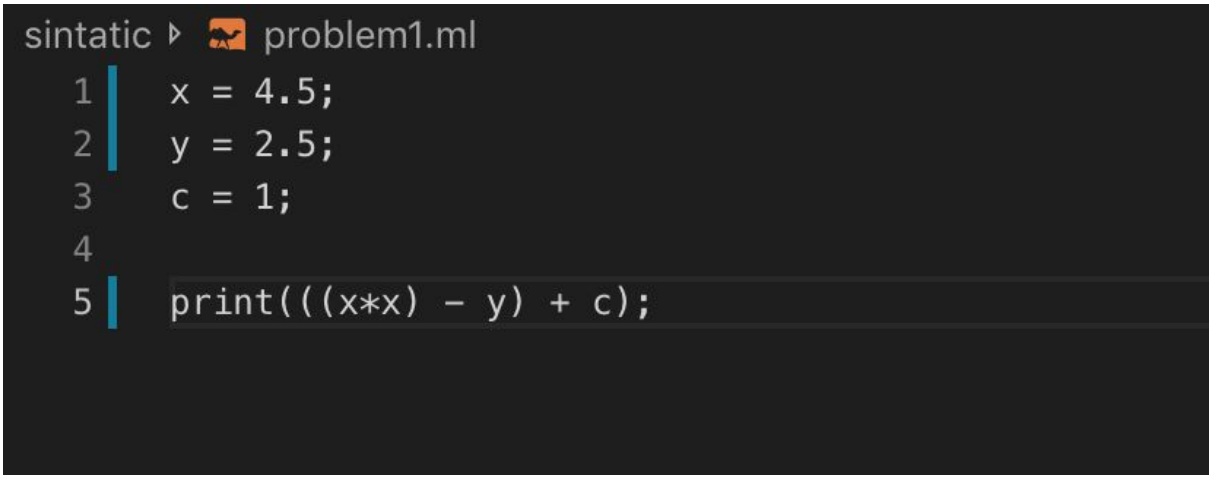
---

Em linguagem natural, o valor sintático é o papel que uma determinada palavra desempenha dentro de uma frase, por exemplo. No caso de linguagem de programação, temos que as palavras das frases são os tokens gerados pelo analisador léxico e o seu papel é justamente definido pelo analisador sintático, que no presente projeto foi escolhido a biblioteca Parsec, também para a linguagem Haskell.

Através da sintaxe da biblioteca, foram escritas diversas funções para interpretar adequadamente os tokens gerados pelo analisador léxico e dar algum significado para esses tokens. A seguir tem-se um trecho de código que representa a função de Parser do token de soma “**+**” da linguagem meia-lua.

```
symOpPlusToken :: ParsecT [Token] st IO (Token)
symOpPlusToken = tokenPrim show update_pos get_token where
 get_token (SymOpPlus pos) = Just (SymOpPlus pos)
 get_token _               = Nothing
```

Abaixo, pode-se ver um trecho de código que recebe as seguintes instruções escritas na sintaxe meia-lua e após o analisador léxico e sintático obtém-se a seguinte saída:



Programa 1 na linguagem meia-lua

```
[Id (AlexPn 0 1 1) "x",Attrib (AlexPn 2 1 3),FloatLit (AlexPn 4 1 5) 4.5,Se
micolon (AlexPn 7 1 8),Id (AlexPn 10 2 1) "y",Attrib (AlexPn 12 2 3),FloatL
it (AlexPn 14 2 5) 2.5,Semicolon (AlexPn 17 2 8),Id (AlexPn 20 3 1) "c",Att
rib (AlexPn 22 3 3),IntLit (AlexPn 24 3 5) 1,Semicolon (AlexPn 25 3 6),Prin
t (AlexPn 28 5 1),OpenParenth (AlexPn 33 5 6),OpenParenth (AlexPn 34 5 7),O
penParenth (AlexPn 35 5 8),Id (AlexPn 36 5 9) "x",SymOpMult (AlexPn 37 5 10
),Id (AlexPn 38 5 11) "x",CloseParenth (AlexPn 39 5 12),SymOpMinus (AlexPn
41 5 14),Id (AlexPn 43 5 16) "y",CloseParenth (AlexPn 44 5 17),SymOpPlus (A
lexPn 46 5 19),Id (AlexPn 48 5 21) "c",CloseParenth (AlexPn 49 5 22),Semico
lon (AlexPn 50 5 23)]
```

Resultado do analisador léxico

## 5. Análise semântica do interpretador

---

Ao obter todas interpretações devidas dos tokens gerados pelo analisador léxico, é preciso definir como essas interpretações irão interagir entre si, ou seja, como cada token irá interagir junto a outros através de expressões. Nesta etapa da implementação do interpretador, também foi utilizada a biblioteca Parsec, onde a expressão que foi recebida é transformada em uma árvore com a decomposição de todos tokens. Para a análise dessas sentenças existem níveis de prioridade execução do interpretador, por exemplo uma expressão *3+(4*5-(9+6))*, será convertida em uma árvore conforme mostrado na figura 3 e os níveis considerados para o interpretador podem ser vistos na tabela 3.

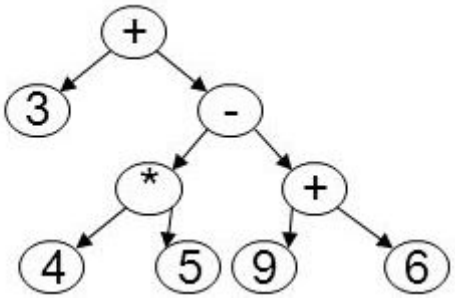



Figura 3 - Decomposição da expressão em *3+(4*5-(9+6))* em uma árvore.

| Token/Expressão | Nível |
|---|---|
| + e - | 1 |
| * e / | 2 |

| | |
|---|---|
| ^ | 3 |
| ( ) | 4 |

Em um exemplo mais prático, pode-se ter o seguinte programa escrito em meia-lua e o respectivo resultado do analisador semântico:

sintatic ▸ problem1.ml

```
1   x = 4.5;
2   y = 2.5;
3   c = 1;
4
5   print(((x*x) - y) + c);
```

Programa 1 na linguagem meia-lua

```
[Variable "x" FloatType (Float 4.5),Variable "y" FloatType (Float 2.5),Variable "c" IntType (Int 1)]
[Variable "x" FloatType (Float 4.5),Variable "y" FloatType (Float 2.5)]
[Variable "x" FloatType (Float 4.5)]
[]
18.75
```

Analisador Semântico

## 6. Instruções de uso do interpretador

Para utilizar o interpretador da linguagem meia-lua deve-se primeiramente ter o ambiente da linguagem Haskell que pode ser obtido em: https://www.haskell.org/downloads/

Em seguida deve-se efetuar o download e instalação da biblioteca de analisador léxico Alex, tais instruções podem ser vistas em: https://www.haskell.org/alex/

Por último na parte de preparação de ambiente deve-se instalar a biblioteca de análise sintática e semântica Parsec, o manual de instalação, tal como os arquivos necessários podem ser encontradas no seguinte link:

https://web.archive.org/web/20140528151730/http://legacy.cs.uu.nl/daan/parsec.html

Para execução do interpretador deve-se primeiro executar o script de execução *run.sh* na pasta '/lex', note que é necessário configurar o path correspondendo ao seu computador no arquivo *run.sh*. Isso irá executar a ferramenta Alex e construir o módulo de Tokens do projeto. Feito isso, deve-se executar o script de execução *run.sh* na pasta '/sintatic' para executar o interpretador, mais uma vez defina o path correspondendo ao seu ambiente de execução. O arquivo de código "problem1.ml" na linguagem meia-lua será interpretado e sua saída apresentada.