UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
Linguagem de Programação: Conceitos e Paradigmas

Nome: Esther Maria da Silveira Wanderley
Nome: Isaque Barbosa Martins
Nome: Thuanny Carvalho Rolim de Albuquerque
Nome: Vinícius Oliveira da Silva
Nome: Gabriel Henrique Campos Medeiros

Relatório sobre a Linguagem de Programação produzida

1. Introdução

Neste projeto, implementamos um interpretador em Haskell para a linguagem **LEGOLAN**. O objetivo deste trabalho é explorar e aplicar conceitos de linguagens de programação e paradigmas de implementação de interpretadores. A linguagem criada pelo nosso grupo possui características que focam em estruturas de dados, procurando facilitar suas representações para o programador.
O público-alvo da **LEGOLAN** são pesquisadores da área de estruturas de dados, visando facilitar a implementação de algoritmos e operações com determinadas estruturas de dados.
O principal diferencial da **LEGOLAN** é que ela busca trazer as principais estruturas dados (matriz, lista, árvore) como tipos primitivos da linguagem, ou seja, o usuário não precisa se preocupar com a implementação dessas estruturas.

2. Design da Implementação

2.1 Transformação do Código-Fonte em Unidades Léxicas

A transformação do código-fonte em unidades léxicas (tokenização) é a primeira etapa do processo de interpretação. Utilizamos a biblioteca **Alex** para a criação do analisador léxico, que identifica e categoriza os diferentes componentes do código-fonte, como palavras-chave, identificadores, operadores, etc.

- **Estrutura Geral:**

```
tokens :-                    ---  Inicia a definicao das regras de
    tokenizacao.
  $white+                        -- Ignora espacos em branco (um ou mais).
  "#".* ; -- Ignora comentarios que comecam com # e vao ate o final da
      linha.
  ...
  $alpha [$alpha $digit \_ \']*              { \p s -> Id s (getLC p) }
```

- **Padrões e Ações:**

```
    ":"                                          { \p s -> Colon (getLC p) }

--- ":": Padrao que corresponde ao caractere :
---{ \p s -> Colon (getLC p) }: Acao que cria um token Colon com a posicao
    (linha e coluna) do caractere. getLC p e uma funcao que obtem a
    posicao do token.
```

- **Exemplos de tokens:**

  i. Símbolos e operadores:

```
  -- cria um token "SemiColon"
  ";"                   { \p s -> SemiColon (getLC p) }

   -- cria um token "Comma"
 ","  { \p s -> Comma (getLC p) }

  -- cria um token "LeftParenthesis"
 "("  { \p s -> LeftParenthesis (getLC p) }

  -- cria um token "RightParenthesis"
 ")"  { \p s -> RightParenthesis (getLC p) }

-- cria um token "LeftCurlyBrackets"
 "{"  { \p s -> LeftCurlyBrackets (getLC p) }

 -- cria um token "RightCurlyBrackets"
"}"   { \p s -> RightCurlyBrackets (getLC p) }

 -- cria um token " To"
"->"  { \p s -> To (getLC p) }

 -- cria um token " Assign"
"="   { \p s -> Assign (getLC p) }
```

  ii. Operadores Matemáticos:

```

        -- cria um token "Plus"
 "+"                            { \p s -> Plus (getLC p) }

        -- cria um token "Minus"
 "-"                            { \p s -> Minus (getLC p) }

        -- cria um token "Times"
 "*"                            { \p s -> Times (getLC p) }

        -- cria um token "Divider"
 "/"                            { \p s -> Divider (getLC p)

        -- cria um token "IntegerDivider"
"//"                            { \p s -> IntegerDivider (getLC p) }

        -- cria um token "Exponent"
"**"                           { \p s -> Exponent (getLC p) }
```

iii. Operadores Lógicos:

```
1
2          -- cria um token "And"
3     "&&"                          { \p s -> And (getLC p) }
4
5          -- cria um token "Or"
6     "||"                          { \p s -> Or (getLC p) }
7
8          -- cria um token "Xor"
9     "^"                           { \p s -> Xor (getLC p) }
10
11         -- cria um token "Not"
12    "!"                           { \p s -> Not (getLC p) }
13
14         -- cria um token "Less"
15    "<"                           { \p s -> Less (getLC p) }
16
17         -- cria um token "LessEqual"
18    "<="                          { \p s -> LessEqual (getLC p) }
19
20        -- cria um token "Greater"
21    ">"                           { \p s -> Greater (getLC p) }
22
23        -- cria um token "GreaterEqual"
24    ">="                          { \p s -> GreaterEqual (getLC p) }
25
26        -- cria um token "Equal"
27    "=="                          { \p s -> Equal (getLC p) }
28
29        -- cria um token "Different"
30    "!="                          { \p s -> Different (getLC p) }
```

iv. Tipos:

```
1
2     --Cada um cria um token Type com a string correspondente e a
          posicao.
3
4      "int"                            { \p s -> Type s (getLC p) }
5     "float"                           { \p s -> Type s (getLC p) }
6     "char"                            { \p s -> Type s (getLC p) }
7     "string"                          { \p s -> Type s (getLC p) }
8     "bool"                            { \p s -> Type s (getLC p) }
9     "list"                            { \p s -> Type s (getLC p) }
10    "stack"                           { \p s -> Type s (getLC p) }
11    "queue"                           { \p s -> Type s (getLC p) }
12    "matrix"                          { \p s -> Type s (getLC p) }
13    "graph"                           { \p s -> Type s (getLC p) }
14    "tree"                            { \p s -> Type s (getLC p) }
15    "n-array"                         { \p s -> Type s (getLC p) }
```

v. Estruturas de Controle:

```
1
2   --Cada um cria um token correspondente (If, EndIf, Elif, Else) com a
        posicao
3
4     "if"                          { \p s -> If (getLC p) }
5     "end_if"                      { \p s -> EndIf (getLC p) }
6     "elif"                        { \p s -> Elif (getLC p) }
7     "else"                        { \p s -> Else (getLC p) }
8     "for"                         { \p s -> For (getLC p) }
9     "end_for"                     { \p s -> EndFor (getLC p) }
10    "while"                       { \p s -> While (getLC p) }
11    "end_while"                   { \p s -> EndWhile (getLC p) }
12    "return"                      { \p s -> Return (getLC p) }
13    "declaration"                 { \p s -> Declaration (getLC p) }
14    "end_declaration"             { \p s -> EndDeclaration (getLC p) }
15    "func"                        { \p s -> Func (getLC p) }
16    "end_func"                    { \p s -> EndFunc (getLC p) }
17    "main"                        { \p s -> Main (getLC p) }
18    "end_main"                    { \p s -> EndMain (getLC p) }
```

vi. Palavras-Chaves:

```
1
2         -- Cria tokens das palavras-chaves
3
4    "typedef"                        { \p s -> Typedef (getLC p)}
5     "struct"                        { \p s -> Struct (getLC p)}
6     "scan"                          { \p s -> Scan s (getLC p) }
7     "print"                         { \p s -> Print s (getLC p) }
```

vii. Constantes e Literais:

```
1
2       -- Cria um token BoolValue com o valor True or False
3
4   "true"                        { \p s -> BoolValue True(getLC p) }
5  "false"                        { \p s -> BoolValue False(getLC p) }
6
7   -- $digit+ \. $digit+: Corresponde a numeros de ponto flutuante (como
        123.456) e cria um token FloatValue.
8
9  -- $digit+: Corresponde a inteiros (como 123) e cria um token IntValue.
10
11
12 -- \" [$white $alpha $digit ! \_ \']* \": Corresponde a strings (como
       "hello") e cria um token StringValue.
13
14 --\' $printable \': Corresponde a caracteres (como 'a') e cria um token
       CharValue.
15
16
17    $digit+ \. $digit+          { \p s -> FloatValue (read s) (getLC p) }
18    $digit+                     { \p s -> IntValue (read s) (getLC p) }
19   \" [$white $alpha $digit ! \_ \']* \"        { \p s -> StringValue
        (cleanString s) (getLC p) }
20   \' $printable \'             { \p s -> CharValue (cleanChar s) (getLC p) }
```

- **Identificadores**

```
-- $alpha [$alpha $digit \_ \']*: Corresponde a identificadores (como
    variableName). Comeca com uma letra ($alpha) e pode ser seguido por
    letras, digitos, sublinhados ou aspas simples.


-- { \p s -> Id s (getLC p) }: Cria um token Id com a string correspondente
   e a posicao.

    $alpha [$alpha $digit \_ \']*              { \p s -> Id s (getLC p) }
```

Este conjunto de regras define como transformar o texto de entrada em uma sequência de tokens, que serão usados pelo analisador sintático para construir a árvore sintática do programa. Cada regra consiste em um padrão de expressão regular e uma ação correspondente para criar o token apropriado com a informação de posição.

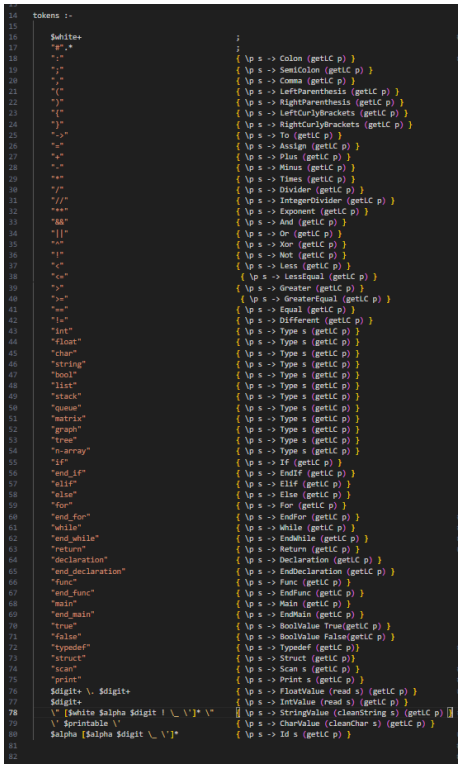


Figura 1: tokens da **LEGOLAN**

- **Definição de TypeValue** :

```
--- data TypeValue define um tipo dado algebrico, que pode representar
    diferentes tipos de valores com informacoes de posicao (linha e coluna)
    no codigo fonte


data TypeValue =
    IntType Int (Int, Int) |
    FloatType Float (Int, Int) |
    StringType String (Int, Int) |
    CharType Char (Int, Int) |
    BoolType Bool (Int, Int) |
    ListType (Int, [TypeValue]) (Int, Int) |
    StructType [(String, TypeValue)] (Int, Int)
    deriving (Eq)
```

Estes dados TypeValue são utilizados para armazenar os valores de cada tipo específico de nossa linguagem, carregando também informações adicionais, como o tamanho de uma

lista.

- **Instância da Classe show para TypeValue**

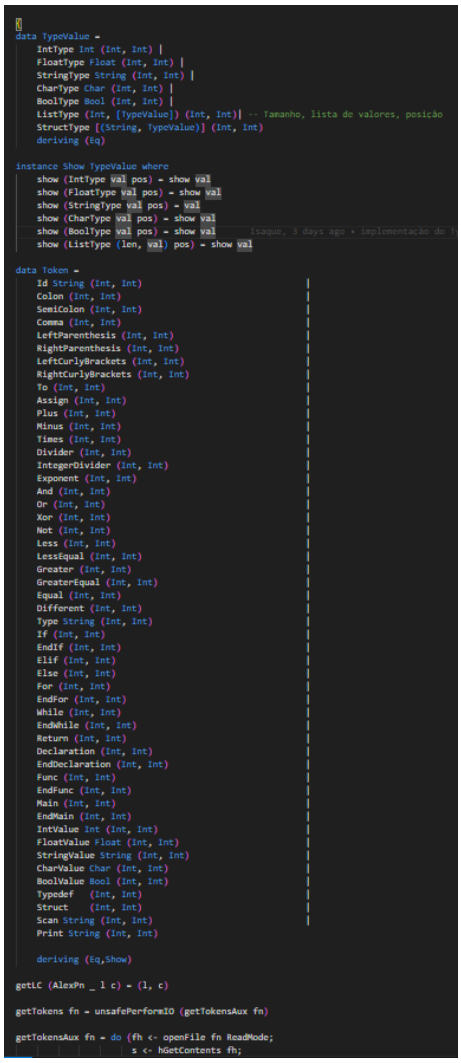
```haskell
--- "instance Show TypeValue" define como 'TypeValue' deve ser convertido
    em uma string para exibicao

--- a funcao show implementa a conversao para cada construtor de
    'TypeValue', exibindo apenas o valor, sem a posicao
instance Show TypeValue where
    show (IntType val pos) = show val
    show (FloatType val pos) = show val
    show (StringType val pos) = val
    show (CharType val pos) = show val
    show (BoolType val pos) = show val
    show (ListType (len, val) pos) = show val
```

- **Definição de 'Token'**
  data Token: Define um tipo de dado algébrico chamado Token, representando diferentes tipos de tokens que o lexer pode reconhecer, cada um com sua posição no código fonte.

```
data Token =
    Id String (Int, Int)                                |
    Colon (Int, Int)                                    |
    SemiColon (Int, Int)                                |
    Comma (Int, Int)                                    |
    LeftParenthesis (Int, Int)                          |
    RightParenthesis (Int, Int)                         |
    LeftCurlyBrackets (Int, Int)                        |
    RightCurlyBrackets (Int, Int)                       |
    To (Int, Int)                                       |
    Assign (Int, Int)                                   |
    Plus (Int, Int)                                     |
    Minus (Int, Int)                                    |
    Times (Int, Int)                                    |
    Divider (Int, Int)                                  |
    IntegerDivider (Int, Int)                           |
    Exponent (Int, Int)                                 |
    And (Int, Int)                                      |
    Or (Int, Int)                                       |
    Xor (Int, Int)                                      |
    Not (Int, Int)                                      |
    Less (Int, Int)                                     |
    LessEqual (Int, Int)                                |
    Greater (Int, Int)                                  |
    GreaterEqual (Int, Int)                             |
    Equal (Int, Int)                                    |
    Different (Int, Int)                                |
    Type String (Int, Int)                              |
    If (Int, Int)                                       |
    EndIf (Int, Int)                                    |
    Elif (Int, Int)                                     |
    Else (Int, Int)                                     |
    For (Int, Int)                                      |
    EndFor (Int, Int)                                   |
    While (Int, Int)                                    |
    EndWhile (Int, Int)                                 |
    Return (Int, Int)                                   |
    Declaration (Int, Int)                              |
    EndDeclaration (Int, Int)                           |
    Func (Int, Int)                                     |
    EndFunc (Int, Int)                                  |
    Main (Int, Int)                                     |
    EndMain (Int, Int)                                  |
    IntValue Int (Int, Int)                             |
    FloatValue Float (Int, Int)                         |
    StringValue String (Int, Int)                       |
    CharValue Char (Int, Int)                           |
    BoolValue Bool (Int, Int)                           |
    Typedef   (Int, Int)                                |
    Struct    (Int, Int)                                |
    Scan String (Int, Int)                              |
    Print String (Int, Int)

    deriving (Eq, Show) --Permite a comparacao de igualdade e a conversao
        para string dos valores de Token.
```



```
{
data TypeValue =
    IntType Int (Int, Int) |
    FloatType Float (Int, Int) |
    StringType String (Int, Int) |
    CharType Char (Int, Int) |
    BoolType Bool (Int, Int) |
    ListType (Int, [TypeValue]) (Int, Int)| -- Tamanho, lista de valores, posição
    StructType [(String, TypeValue)] (Int, Int)
    deriving (Eq)

instance Show TypeValue where
    show (IntType val pos) = show val
    show (FloatType val pos) = show val
    show (StringType val pos) = val
    show (CharType val pos) = show val
    show (BoolType val pos) = show val      Isaque, 3 days ago • implementação do T
    show (ListType (len, val) pos) = show val

data Token =
    Id String (Int, Int)                    |
    Colon (Int, Int)                        |
    SemiColon (Int, Int)                    |
    Comma (Int, Int)                        |
    LeftParenthesis (Int, Int)              |
    RightParenthesis (Int, Int)             |
    LeftCurlyBrackets (Int, Int)            |
    RightCurlyBrackets (Int, Int)           |
    To (Int, Int)                           |
    Assign (Int, Int)                       |
    Plus (Int, Int)                         |
    Minus (Int, Int)                        |
    Times (Int, Int)                        |
    Divider (Int, Int)                      |
    IntegerDivider (Int, Int)               |
    Exponent (Int, Int)                     |
    And (Int, Int)                          |
    Or (Int, Int)                           |
    Xor (Int, Int)                          |
    Not (Int, Int)                          |
    Less (Int, Int)                         |
    LessEqual (Int, Int)                    |
    Greater (Int, Int)                      |
    GreaterEqual (Int, Int)                 |
    Equal (Int, Int)                        |
    Different (Int, Int)                    |
    Type String (Int, Int)                  |
    If (Int, Int)                           |
    EndIf (Int, Int)                        |
    Elif (Int, Int)                         |
    Else (Int, Int)                         |
    For (Int, Int)                          |
    EndFor (Int, Int)                       |
    While (Int, Int)                        |
    EndWhile (Int, Int)                     |
    Return (Int, Int)                       |
    Declaration (Int, Int)                  |
    EndDeclaration (Int, Int)               |
    Func (Int, Int)                         |
    EndFunc (Int, Int)                      |
    Main (Int, Int)                         |
    EndMain (Int, Int)                      |
    IntValue Int (Int, Int)                 |
    FloatValue Float (Int, Int)             |
    StringValue String (Int, Int)           |
    CharValue Char (Int, Int)               |
    BoolValue Bool (Int, Int)               |
    Typedef   (Int, Int)                    |
    Struct    (Int, Int)                    |
    Scan String (Int, Int)                  |
    Print String (Int, Int)

    deriving (Eq,Show)

getLC (AlexPn _ l c) = (l, c)

getTokens fn = unsafePerformIO (getTokensAux fn)

getTokensAux fn = do {fh <- openFile fn ReadMode;
                      s <- hGetContents fh;
```

Figura 2: TypeValue e Token da **LEGOLAN**

2.2 Memória da Linguagem

A estrutura de memória *MemoryState* gerencia variáveis, funções, escopos e estruturas dentro de um programa. Ela permite a inserção, atualização e remoção de variáveis em tabelas de símbolos globais e locais, além de manter flags para controle de execução e análise de estruturas.

- Definição do Tipo *MemoryState*

```
type MemoryState = (
                    Bool, -- Flag
                    [(Token, TypeValue)], -- Symtable
                    [(Token, [(Token, TypeValue)], [Token])], -- Funcs
                    [[(Token, TypeValue)]], -- Structs
                    CallStack, -- Callstack
                    Bool, -- StructFlag
                    Bool -- FuncFlag
                    )
type CallStack = [(Token, [(Token, TypeValue)], [Token])]
```

  - **Bool (Flag)** : Uma flag generalizada que indica se o bloco de código atual deve ser analisado semanticamente.

  - **[(Token, TypeValue)] (Symtable)**: Esta lista de tuplas é referente a tabela de símbolos do programa principal, sendo uma lista de pares de tokens e valores de tipo. Cada par representa uma variável e seu valor.

  - **[(Token, [(Token, Token)], [Token])] (Funcs)**: Esta memória é utilizada para armazenar a sintaxe da implementação de subprogramas do código, onde o primeiro elemento é um Token Id referente ao nome do subprograma, temos uma lista de variáveis locais onde, inicialmente, é armazenado valores padrões somente para os parâmetros, e uma lista de Tokens referente aos *statements* locais da função.

  - **[[(Token, TypeValue)]] (Structs)**: Uma pilha de registros, que armazena os registros criados pelo usuário, onde o Token é o nome do registro, e o TypeValue é o StructType possuindo as variáveis do registro.

  - **[[(Token, [(Token, Token)], [Token])]] (CallStack)**: Uma pilha de funções instanciadas. Esta memória é utilizada assim que uma função é chamada em um programa principal, ou local. É resgatado a sintaxe armazenada em *funcs* e instanciada com os valores reais sobre os parâmetros, além de definir o input do parsec para os stmts da função instanciada.

  - **Bool (StructFlag):** Uma flag que indica se estamos realizando uma definição ou assinatura de estrutura.
  - **Bool (FuncFlag):** Uma flag que indica se estamos realizando o parser de uma função. Com esta flag, somos capazes de declarar e atribuir variáveis locais, ao invés de fazê-lo globalmente.

- **Flags de Execução**

```
-- Function to set the flag to True
setFlagTrue :: MemoryState -> MemoryState
setFlagTrue (flag, vars, funcs, structs, callstack, structflag) = (True,
    vars, funcs, structs, callstack, structflag)

-- Function to set the flag to False
setFlagFalse :: MemoryState -> MemoryState
setFlagFalse (flag, vars, funcs, structs, callstack, structflag) = (False,
    vars, funcs, structs, callstack, structflag)

isFlagTrue :: MemoryState -> Bool
isFlagTrue (flag, _, _, _, _, _) = flag
```

  - **setFlagTrue e setFlagFalse**: Funções que alteram a *flag* de execução para True ou False, respectivamente.
  - **isFlagTrue**: Função que verifica se a *flag* de execução está definido como True.

- **Flags de Estrutura**

```haskell
-- Function to set the flag to True
setStructFlagTrue :: MemoryState -> MemoryState
setStructFlagTrue (flag, vars, funcs, structs, callstack, structflag) =
    (flag, vars, funcs, structs, callstack, True)

-- Function to set the flag to False
setStructFlagFalse :: MemoryState -> MemoryState
setStructFlagFalse (flag, vars, funcs, structs, callstack, structflag) =
    (flag, vars, funcs, structs, callstack, False)

isStructFlagTrue :: MemoryState -> Bool
isStructFlagTrue (_, _, _, _, _, structflag) = structflag
```

  - **setStructFlagTrue e setStructFlagFalse**: Funções que alteram a *structFlag* para True ou False, respectivamente.

  - **isStructFlagTrue**: Função que verifica se a *structFlag* está definido como True.

- **Flags de funções**

```haskell
-- Function to set the flag to True
setFuncFlagTrue :: MemoryState -> MemoryState
setFuncFlagTrue (flag, vars, funcs, structs, callstack, structflag,
    funcFlag) = (flag, vars, funcs, structs, callstack, structflag, True)

-- Function to set the flag to False
setFuncFlagFalse :: MemoryState -> MemoryState
setFuncFlagFalse (flag, vars, funcs, structs, callStack, structFlag,
    funcFlag) =
  let newFuncFlag = (not (null callStack) && funcFlag)
  in (flag, vars, funcs, structs, callStack, structFlag, newFuncFlag)


isFuncFlagTrue :: MemoryState -> Bool
isFuncFlagTrue (_, _, _, _, _, _, funcFlag) = funcFlag
```

  - **setFuncFlagTrue e setFuncFlagFalse**: Funções que alteram a *funcFlag* para True ou False, respectivamente.

  - **isFuncFlagTrue**: Função que verifica se a *funcFlag* está definida como True.

2.3 Representação de Símbolos, Tabela de Símbolos e Funções Associadas

A tabela de símbolos é uma estrutura de dados crucial que armazena informações sobre variáveis no programa principal. Implementamos uma tabela de símbolos utilizando uma tupla **(Token,TypeValue)**, que permite inserções e consultas eficientes.

- **symtableGet**

```haskell
symtableGet :: (Token) -> MemoryState -> Maybe TypeValue
symtableGet _ (_, [], _, _, _, _) = fail "variable not found"
symtableGet (Id idStr1 pos1) (_, (Id idStr2 pos2, value2) : listTail, _, _,
    _, _) =
  if idStr1 == idStr2
    then Just value2
    else symtableGet (Id idStr1 pos1) (False, listTail, [], [], [], False)
symtableGet _ _ = Nothing
```

  - **symtableGet**: Recebe um Token ID e o MemoryState. Verifica se a variável existe na tabela de símbolos e, se existir, retorna seu valor. Caso contrário, lança um erro para o usuário.

- **symtableInsert**

```haskell
symtableInsert :: (Token, TypeValue) -> MemoryState -> MemoryState
symtableInsert newSymbol@(newId, _) state@(flag, symtable, funcs, structs,
    callstack, structflag, funcFlag) =
  case symtableGet newId state of
    Nothing -> (flag, symtable ++ [newSymbol], funcs, structs, callstack,
        structflag, funcFlag)
    Just _  -> error $ "Variable " ++ show newId ++ " already declared."
```

– **symtableInsert**: Insere um novo par (Token ID, TypeValue) na tabela de símbolos. Verifica se a variável já não existe nela e, se existir, lança um erro.

- **symtableUpdate**

```
symtableUpdate :: (Token, TypeValue) -> MemoryState -> MemoryState
symtableUpdate _ (_, [], _, _, _, _, _) = error "variable not found"
symtableUpdate (Id idStr1 pos1, value1) (flag, (Id idStr2 pos2, value2) :
    listTail, funcs, structs, callstack, structflag, funcFlag)
  | idStr1 == idStr2 = (flag, (Id idStr1 pos2, value1) : listTail, funcs,
      structs, callstack, structflag, funcFlag)
  | otherwise =
      let (flag', updatedSymtable, funcs', structs', callstack',
          structflag', funcFlag') = symtableUpdate (Id idStr1 pos1, value1)
          (flag, listTail, funcs, structs, callstack, structflag, funcFlag)
       in (flag', (Id idStr2 pos2, value2) : updatedSymtable, funcs',
           structs', callstack', structflag', funcFlag')
```

– **symtableUpdate**: Recebe uma tupla (Token ID, TypeValue) e atualiza o valor na tabela de símbolos, se o Token ID já estiver na tabela.

- **symtableRemove**

```
symtableRemove :: (Token, TypeValue) -> MemoryState -> MemoryState
symtableRemove _ (_, [], _, _, _, _, _) = error "variable not found"
symtableRemove (id1, v1) (flag, (id2, v2) : listTail, funcs, structs,
    callstack, structflag, funcFlag)
  | id1 == id2 = (flag, listTail, funcs, structs, callstack, structflag,
      funcFlag)
  | otherwise =
      let (flag', updatedSymtable, funcs', structs', callstack',
          structflag', funcFlag') = symtableRemove (id1, v1) (flag,
          listTail, funcs, structs, callstack, structflag, funcFlag)
       in (flag', (id2, v2) : updatedSymtable, funcs', structs',
           callstack', structflag', funcFlag')
```

– **symtableRemove**: Recebe uma tupla (Token ID, Token Value) e remove -a da tabela de símbolos, se o Token ID já existir na tabela.

2.4 **Tratamento de Estruturas Condicionais e de Repetição**
As estruturas condicionais (if, else) e de repetição (while, for) são tratadas por meio de funções específicas que analisam e executam os blocos de código associados.

- Estruturas Condicionais:
  As estruturas condicionais são essenciais para controlar o fluxo de execução do programa. No nosso interpretador, as instruções condicionais são processadas de forma a verificar a validade das expressões associadas e, com base nisso, decidir qual bloco de código deve ser executado.

  i. **'if'** :
  A instrução if é tratada da seguinte maneira:

  – **Tokenização:** O token if é identificado e a expressão condicional é avaliada.

  – **Análise da Condição:** A condição dentro da instrução if é avaliada.

  – **Execução do Bloco de Código:** Se a condição for verdadeira, o bloco de código associado ao if é executado.

  – **Estruturas Aninhadas:** Estruturas if podem ser aninhadas, e o interpretador trata esses aninhamentos recursivamente.

  – **Manipulação da Tabela de Símbolos:** Dependendo do estado da análise (semântica ou sintática), e do escopo sendo tratado (global ou local), a tabela de símbolos necessária é atualizada ou restaurada após a execução do bloco if.

```
ifStmt :: ParsecT [Token] MemoryState IO [Token]
ifStmt = do
  ifLiteral <- ifToken
  expression <- ifParenthesisExpression
  colonLiteral <- colonToken
  state <- getState
  if isFlagTrue state
    then do
      if isFuncFlagTrue state
        then do
          let nLocalVar = getLocalSymtableLength state
          let result = evaluateCondition expression
          if result
            then do
              stmtsBlock <- stmts
              skip' <- manyTill anyToken (lookAhead endifToken)
              endIfLiteral <- endifToken
              semiCol1 <- semiColonToken
              updateState (removeLocalSymtableUntilLenght nLocalVar)
              return ([ifLiteral] ++ [expression] ++ [colonLiteral] ++
                  stmtsBlock)
            else do
              skip' <- manyTill anyToken (lookAhead elifToken <|>
                  lookAhead elseToken <|> lookAhead endifToken)
              elifStmt' <- elifStmt
              if null elifStmt'
                then do
                  skip' <- manyTill anyToken (lookAhead elseToken <|>
                      lookAhead endifToken)
                  elseStmt' <- elseStmt
                  endIfLiteral <- endifToken
                  semiCol <- semiColonToken
                  updateState (removeLocalSymtableUntilLenght nLocalVar)
                  return ([ifLiteral] ++ [expression] ++ [colonLiteral]
                      ++ elseStmt')
                else do
                  skip' <- manyTill anyToken (lookAhead endifToken <|>
                      lookAhead endifToken)
                  endIfLiteral <- endifToken
                  semiCol <- semiColonToken
                  updateState (removeLocalSymtableUntilLenght nLocalVar)
                  return ([ifLiteral] ++ [expression] ++ [colonLiteral]
                      ++ elifStmt')
        else do
          let nVar = getSymtableLenght state
          let result = evaluateCondition expression
          if result
            then do
              stmtsBlock <- stmts
              skip' <- manyTill anyToken (lookAhead endifToken)
              endIfLiteral <- endifToken
              semiCol1 <- semiColonToken
              updateState (symtableRemoveUntilLenght nVar)
              return ([ifLiteral] ++ [expression] ++ [colonLiteral] ++
                  stmtsBlock)
            else do
              skip' <- manyTill anyToken (lookAhead elifToken <|>
                  lookAhead elseToken <|> lookAhead endifToken)
              elifStmt' <- elifStmt
              if null elifStmt'
                then do
                  skip' <- manyTill anyToken (lookAhead elseToken <|>
                      lookAhead endifToken)
                  elseStmt' <- elseStmt
                  endIfLiteral <- endifToken
                  semiCol <- semiColonToken
                  updateState (symtableRemoveUntilLenght nVar)
                  return ([ifLiteral] ++ [expression] ++ [colonLiteral]
                      ++ elseStmt')
                else do
                  skip' <- manyTill anyToken (lookAhead endifToken <|>
                      lookAhead endifToken)
                  endIfLiteral <- endifToken
                  semiCol <- semiColonToken
```

```
                    updateState (symtableRemoveUntilLenght nVar)
                    return ([ifLiteral] ++ [expression] ++ [colonLiteral]
                        ++ elifStmt')
    else
      return []
```

ii. **'elif'** :
A instrução elif fornece uma forma de adicionar condições adicionais após um if. A análise e execução seguem estas etapas:

- **Tokenização:** O token elif é identificado e a expressão condicional é avaliada.
- **Análise da Condição:** A condição dentro da instrução elif é avaliada.
- **Execução do Bloco de Código:** Se a condição for verdadeira, o bloco de código associado ao elif é executado.
- **Verificação de Condições Adicionais:** Se a condição do elif for falsa, outras condições elif subsequentes são avaliadas até encontrar uma verdadeira ou chegar ao else.

```
elifStmt :: ParsecT [Token] MemoryState IO [Token]
elifStmt =
  ( do
      elifLiteral <- elifToken
      expression <- ifParenthesisExpression
      colonLiteral <- colonToken
      let result = evaluateCondition expression
      if result
        then do
          stmtsBlock <- stmts
          return ([elifLiteral] ++ [expression] ++ [colonLiteral] ++
              stmtsBlock)
        else do
          skip' <- manyTill anyToken (lookAhead elifToken)
          elifStmt' <- elifStmt
          return elifStmt'
  )
    <|> return []
```

iii. **'else'** :
A instrução else fornece um bloco de código alternativo que é executado se nenhuma das condições if ou elif forem verdadeiras.

- **Tokenização:** O token else é identificado.
- **Execução do Bloco de Código:** O bloco de código associado ao else é executado incondicionalmente, já que todas as condições anteriores falharam.

```
elseStmt :: ParsecT [Token] MemoryState IO [Token]
elseStmt =
  ( do
      elseLiteral <- elseToken
      colonLiteral <- colonToken
      stmtsBlock <- stmts
      return ([elseLiteral] ++ [colonLiteral] ++ stmtsBlock)
  )
    <|> return []
```

- **Abordagem Implementada** :
Utilizamos uma abordagem baseada em análise de tokens com a biblioteca Parsec em Haskell. Esta abordagem permite:

  * **Análise Léxica e Sintática:** Separação clara entre a análise de tokens e a avaliação das condições.

* **Manutenção da Tabela de Símbolos:** A tabela de símbolos é manipulada para garantir que as variáveis locais e globais sejam tratadas corretamente durante a análise e execução.

* **Tratamento de Blocos de Código:** Blocos de código são lidos e executados com base nos resultados das avaliações das condições.

Esta implementação garante que as estruturas condicionais sejam tratadas de forma eficiente e correta, mantendo o fluxo de controle do programa conforme esperado.

- **Estruturas de Repetição**:
  As estruturas de repetição, como while e for, são essenciais para executar blocos de código repetidamente com base em uma condição. Abaixo, descrevemos como essas estruturas são tratadas no seu interpretador usando a biblioteca Parsec em Haskell.

  - **'while'**:
    A instrução while é tratada da seguinte maneira:

    * **Tokenização:** O token while é identificado e a expressão condicional é armazenada sintaticamente.

    * **Análise da Condição:** Em um loop, a condição dentro da instrução while é avaliada.
    * **Execução do Bloco de Código:** Quando a condição for verdadeira, o bloco de código associado ao while é definido como input do parser e executado. Ao finalizar o bloco de código, definimos a expressão como input novamente e a reavaliamos, fazendo isto recursivamente até a condição se tornar falsa.

    * **Manipulação da Tabela de Símbolos:** Variáveis locais e globais são manipuladas corretamente durante a execução do loop.

```
whileStmt :: ParsecT [Token] MemoryState IO [Token]
whileStmt = do
  whileLiteral <- whileToken
  expressionTokens <- manyTill anyToken (lookAhead colonToken)
  colonLiteral <- colonToken
  stmtsBlock <- nestedWhileTokens 0
  endWhileLiteral <- endWhileToken
  semiCol <- semiColonToken

  memoryState <- getState
  input <- getInput

  let loop = do
        memoryState <- getState
        if isFlagTrue memoryState
          then do
            if isFuncFlagTrue memoryState
              then do
                let nLocalVar = getLocalSymtableLength memoryState
                setInput expressionTokens
                expressionValue <- ifParenthesisExpression
                let condition = evaluateCondition expressionValue
                if condition
                  then do
                    setInput stmtsBlock
                    _ <- many stmts
                    updateState (removeLocalSymtableUntilLenght
                        nLocalVar)
                    loop
                  else setInput input
              else do
                let nVar = getSymtableLenght memoryState
                setInput expressionTokens
                expressionValue <- ifParenthesisExpression
                let condition = evaluateCondition expressionValue
                if condition
                  then do
                    setInput stmtsBlock
                    _ <- many stmts
                    updateState (symtableRemoveUntilLenght nVar)
                    loop
                  else setInput input
          else setInput input

  loop

  return ([whileLiteral] ++ expressionTokens ++ [colonLiteral] ++
      stmtsBlock ++ [endWhileLiteral] ++ [semiCol])

nestedWhileTokens :: Int -> ParsecT [Token] MemoryState IO [Token]
nestedWhileTokens nestDepth = do
  tokens <- manyTill anyToken (lookAhead endWhileToken <|> lookAhead
      whileToken)
  next <- lookAhead anyToken
  case next of
    While _ -> do
      whileToken' <- whileToken
      nestedFor <- nestedWhileTokens (nestDepth + 1)
      tokens' <- manyTill anyToken (lookAhead endWhileToken)
      return (tokens ++ [whileToken'] ++ nestedFor ++ tokens')
    _ -> do
      if nestDepth == 0
        then return tokens
        else do
          endWhile' <- endWhileToken
          return (tokens ++ [endWhile'])
```

– **'for'**:
A instrução for é tratada da seguinte maneira:

* **Tokenização:** O token for é identificado a primeira atribuição é análisada semânticamente, a expressão de iteração é armazenada sintáticamente assim como a atribuição iterativa.

* **Análise da Condição**: Em um loop, a condição dentro da instrução for é avaliada.
* **Execução do Bloco de Código:** Se a condição for verdadeira, o bloco de código associado ao for é definido como input e executado. Após seu término, a atribuição iterativa é analisada semanticamente e em seguida o bloco de expressão.

* **Manipulação da Tabela de Símbolos:**Variáveis locais e globais são manipuladas corretamente durante a execução do loop.

```
forStmt :: ParsecT [Token] MemoryState IO [Token]
forStmt = do
  forLiteral <- forToken
  leftParenthesis <- leftParenthesisToken
  assign' <- assign
  semiCol' <- semiColonToken
  expressionTokens <- manyTill anyToken (lookAhead semiColonToken)
  semiCol'' <- semiColonToken
  updateAssign <- manyTill anyToken (lookAhead rightParenthesisToken)
  rightParenthesis <- rightParenthesisToken
  colon' <- colonToken
  stmtsBlock <- nestedForTokens 0
  endFor <- endForToken
  semiCol <- semiColonToken
  memoryState <- getState
  input <- getInput
  let loop = do
        memoryState <- getState
        if isFlagTrue memoryState
          then do
            if isFuncFlagTrue memoryState
              then do
                let nLocalVar = getLocalSymtableLength memoryState
                setInput expressionTokens
                expressionValue <- relatOrLogicExpression
                let condition = evaluateCondition expressionValue
                if condition
                  then do
                    setInput stmtsBlock
                    _ <- many stmts
                    setInput updateAssign
                    assign'' <- assign
                    updateState (removeLocalSymtableUntilLenght
                        nLocalVar)
                    loop
                  else setInput input
              else do
                let nVar = getSymtableLenght memoryState
                setInput expressionTokens
                expressionValue <- relatOrLogicExpression
                let condition = evaluateCondition expressionValue
                if condition
                  then do
                    setInput stmtsBlock
                    _ <- many stmts
                    setInput updateAssign
                    assign'' <- assign
                    updateState (symtableRemoveUntilLenght nVar)
                    loop
                  else setInput input
          else setInput input

  loop

  return ([forLiteral] ++ [colon'] ++ stmtsBlock ++ [endFor] ++
      [semiCol])

nestedForTokens :: Int -> ParsecT [Token] MemoryState IO [Token]
nestedForTokens nestDepth = do
  tokens <- manyTill anyToken (lookAhead endForToken <|> lookAhead
      forToken)
  next <- lookAhead anyToken
  case next of
    For _ -> do
      forToken' <- forToken
      nestedFor <- nestedForTokens (nestDepth + 1)
      tokens' <- manyTill anyToken (lookAhead endForToken)
      return (tokens ++ [forToken'] ++ nestedFor ++ tokens')
    _ -> do
      if nestDepth == 0
        then return tokens
        else do
          endFor' <- endForToken
          return (tokens ++ [endFor'])
```

- **Abordagem Implementada**
  Utilizamos uma abordagem baseada em análise de tokens com a biblioteca Parsec em Haskell. Esta abordagem permite:

  * Análise Léxica e Sintática: Separação clara entre a análise de tokens e a avaliação das condições.

  * Manutenção da Tabela de Símbolos: A tabela de símbolos é manipulada para garantir que as variáveis locais e globais sejam tratadas corretamente durante a análise e execução.

  * Tratamento de Blocos de Código: Blocos de código são lidos e executados com base nos resultados das avaliações das condições.

  Esta implementação garante que as estruturas de repetição sejam tratadas de forma eficiente e correta, mantendo o fluxo de controle do programa conforme esperado.

2.5 Tratamento de Subprogramas

Os subprogramas, incluindo funções e procedimentos, são blocos de código que podem ser chamados e executados a partir de diferentes pontos em um programa. Eles permitem a modularização do código, promovendo reutilização e facilitando a manutenção. Em linguagens de programação, os subprogramas são definidos com um nome e um conjunto de parâmetros e podem retornar um valor (no caso de funções).

- **Chamada de Função:**

  A chamada de função envolve os seguintes passos:

  - Tokenização e Análise: Identificação do identificador da função, parênteses e parâmetros.

  - Verificação de Parâmetros: Verificação se existe uma função com o identificador fornecido, que aceite o mesmo número e tipo de parâmetros.

  - Manipulação da Pilha de Chamadas: Atualização do estado para refletir a nova chamada de função, incluindo a atualização da pilha de chamadas.

  - Execução da Função: Avaliação das instruções da função.

  - Manipulação do Estado Pós-Chamada: Atualização do estado após a execução da função, incluindo a passagem de valores de retorno.

```
    funcStmt :: ParsecT [Token] MemoryState IO [Token]
funcStmt = do
  id <- idToken
  leftpar <- leftParenthesisToken
  parameters' <- parametersExprBlock
  rightPar <- rightParenthesisToken
  semiCol <- semiColonToken
  state <- getState
  input <- getInput
  let parametersValues = parametersValuesFromIDs parameters' state
  let funcMemoryInstance@(idFunc, funcMemory, funcStmts) =
      checkFunctionParameters id parametersValues state

  updateState (callStackPush funcMemoryInstance)
  state' <- getState

  updateState setFuncFlagTrue
  setInput funcStmts
  stmts' <- stmts
  setInput input

  newState <- getState
  let topStack = callStackGet newState

  updateState (const (callStackPop newState))
  removedState <- getState

  updateState (passResultValue parameters' topStack)
  updatedState <- getState

  updateState setFuncFlagFalse
  updateState setFlagTrue
  return stmts'
```

- Expressão de função:
  As expressões de função são tratadas de forma semelhante às chamadas de função, com a diferença de que elas retornam um valor específico.

```
    funcExpr :: ParsecT [Token] MemoryState IO Token
funcExpr = do
  id <- idToken
  leftpar <- leftParenthesisToken
  parameters' <- parametersExprBlock
  rightPar <- rightParenthesisToken
  state <- getState
  input <- getInput

  let parametersValues = parametersValuesFromIDs parameters' state
  let funcMemoryInstance@(idFunc, funcMemory, funcStmts) =
      checkFunctionParameters id parametersValues state

  updateState (callStackPush funcMemoryInstance)
  state' <- getState

  updateState setFuncFlagTrue
  setInput funcStmts
  stmts' <- stmts
  setInput input

  newState <- getState
  let topStack = callStackGet newState

  updateState (const (callStackPop newState))
  removedState <- getState

  updateState (passResultValue parameters' topStack)
  updatedState <- getState

  updateState setFuncFlagFalse
  updateState setFlagTrue

  return (stmts' !! (length stmts' - 2))
```

- Principais Componentes:

  – Verificação de Parâmetros:
  A função **checkFunctionParameters** é responsável por verificar se uma função com o identificador fornecido existe e se aceita os parâmetros fornecidos.

```
checkFunctionParameters :: String -> [Value] -> MemoryState ->
    (String, FunctionMemory, [Token])
checkFunctionParameters id parameters state = -- Implementacao da
    verificacao de parametros
```

  – Manipulação da Pilha de Chamadas:
  A manipulação da pilha de chamadas envolve operações como callStackPush, callStackPop, e callStackGet, que gerenciam a pilha de chamadas.

```
callStackPush :: FunctionMemoryInstance -> MemoryState ->
    MemoryState
callStackPush instance state = -- Implementacao da operacao de push na
    pilha

callStackPop :: MemoryState -> MemoryState
callStackPop state = -- Implementacao da operacao de pop na pilha

callStackGet :: MemoryState -> FunctionMemoryInstance
callStackGet state = -- Implementacao da operacao de get na pilha
```

  – Simulação de Passagem por referência :
  Para simular a passagem por referência no nosso interpretador, utilizamos a técnica de passagem por valor resultado. Isso significa que os valores dos parâmetros são copiados para dentro da função, e os resultados são copiados de volta para os parâmetros originais após a execução da função. Esse método permite que as funções possam modificar os valores dos parâmetros, simulando o comportamento da passagem por referência.

A implementação de subprogramas no seu interpretador é crucial para permitir a modularização e reutilização de código. A abordagem baseada em Parsec em Haskell permite uma análise sintática e semântica eficiente das chamadas de função e expressões de função, garantindo que o fluxo de controle do programa seja mantido corretamente.Além disso, a simulação da passagem por referência através da passagem por valor resultado proporciona flexibilidade na manipulação de parâmetros dentro das funções.

2.6 Verificações Realizadas

Implementamos diversas verificações para assegurar a corretude do programa, como foram apresentados acima, funcções como:

- **symtableInsert**
- **symtableGet**
- **symtableUpdate**
- **symtableRemove**
- **callStackGet**
- **callStackPop**
- **callStackUpdateTop**

Dentre outras, são funções que fazem verificação de erros de tipos, e existências de variáveis.

3. Instruções de Uso do Interpretador

Para utilizar o interpretador, siga os passos abaixo:

- Compilação do Léxico e geração do Parser:

```
alex tokens.x
```

- Compilação do Interpretador:
  Compile o interpretador utilizando o GHC (Glasgow Haskell Compiler):

```
ghc parser.hs
```

- Execução do Interpretador:
  Execute o interpretador passando o arquivo de código-fonte da linguagem criada:

```
./parser
```

- Exemplos de Entrada:

```
./parser exemplos/exemplo_a_sua_escolha
```

  Aqui estão alguns exemplos de programas escritos na linguagem criada:

  – exemplo1_declarations.txt

```
   declaration:
    a: int;
    b: int;
    c: int;
    d: int;
end_declaration
```

  – exemplo2_uma_atribuicao.txt

```
  declaration:
    a: int;
    c: int;
end_declaration

main:
    #a: int;
    b: int;
    d: int;
    a = 10;
end_main
```

  – exemplo3_atribuicao_tipos_simples.txt

```
  declaration:
    a: int;
    b: string;
    c: char;
    d: float;
    e: bool;
end_declaration

main:
    a = 10;
    b = "oi";
    c = 'o';
    d = 10.102;
    e = false;
end_main
```

  – exemplo4_atribuicao_por_expressao.txt

```
 declaration:
    b: int;
end_declaration

main:
    b = 10 + 5;
end_main
```

– exemplo5_atribuicoes_por_expressoes.txt

```
 declaration:
    a: int;
    b: int;
    c: int;
    d: float;
    e: int;
    f: int;
    problem_1: float;
    g: bool;
    h: int;
    i: int;
    j: bool;
    k: bool;
    l: bool;
    m: bool;
    n: bool;
    o: bool;
    p: bool;
    q: bool;
    r: bool;
    s: bool;
    t: bool;
    u: bool;
    v: bool;
    w: bool;
    x: bool;
    y: bool;
    z: bool;
end_declaration

main:
    a = 10 + 5; # 15
    b = 10 - 5; # 5
    c = 10 * 2; # 20
    d = 10 / 2; # 5.0
    e = 10 // 2; # 5
    f = 10 ** 2; # 100
    problem_1 = 10.1 ** 2 - 5.5 + 5; # 95
    g = !false; # True
    h = (10 + 5); # 15
    i = 10; # 10
    j = 5 < 10 + 5; # True
    k = 5 + 5 < 10 + 5; # True
    l = 5 - 5 < 10 - 5; # True
    m = 5 * 5 > 5; # True
    n = 5/5 >= 1; # True
    o = 10 // 2 <= 5; # True
    p = 10 ** 2 != 99; # True
    r = (10 ** 2) == (10 ** 2); # True
    s = 10 > 5; # True
    t = (10 > 5) && (5 < 10); # True
    u = (10 != 5) || (5 > 10); # True
    v = (10 < 5) && (5 < 10); # False
    w = (10 < 5) && (11 < 6); # False
    x = (10 != 10) || (10 == 10); # True
    y = (10 != 10) || (9 != 9); # False
    #z
end_main
```

– exemplo5.1_problema_1.txt

```
 declaration:
    problem_1: float;
    a: float;
    b: int;
    c: float;
    d: int;
end_declaration

main:
    a = 10.1;
    b = 2;
    c = 5.5;
    d = 5;
    problem_1 = a ** b - c + d; # 101.51
    print(problem_1);
end_main
```

– exemplo6_if_stmts.txt

```
        declaration:
    a: int;
    b: int;
    c: int;
end_declaration

main:
    if(11 < 10):
        a = 5;
    elif(10 > 11):
        b = 10;
    elif(10 > 9):
        c = 15;
    else:
        a = 15;
        b = 15;
        c = 15;
    end_if;
end_main
```

– exemplo7_while_stmts.txt

```
     declaration:
    a: int;
    b: int;
    c: int;
end_declaration

main:
    a = 5;
    while(a < 10):
        a = a + 1;
    end_while;
    print(a);
end_main
```

– exemplo8_for_stmts.txt

```
     declaration:
    a: int;
    x: int;
end_declaration

main:
    for(x = 0; x < 10; x = x + 1):
        a = a + 5;
    end_for;

    print(a);
end_main
```

– exemplo9_scan.txt

```
        declaration:
    a: int;
end_declaration

main:
    a = scan("Oi ");
end_main
```

– exemplo10_problema_2.txt

```
      declaration:
    entry: int;
    count1: int;
    count2: int;
    count3: int;
    count4: int;
end_declaration

main:
    entry = scan("Informe um valor que deseja verificar");
    while(entry >= 0):
        if ((entry >= 0) && (entry <= 25)):
            count1 = count1 + 1;
        elif ((entry >= 26) && (entry <= 50)):
            count2 = count2 + 1;
        elif ((entry >= 51) && (entry <= 75)):
            count3 = count3 + 1;
        elif ((entry >= 76) && (entry <= 100)):
            count4 = count4 + 1;
        end_if;
        entry = scan("Informe um valor que deseja verificar");
    end_while;

    print(count1);
    print(count2);
    print(count3);
    print(count4);
end_main
```

– exemplo11_print.txt

```
    declaration:
    entry: int;
    numero: float;
    teste: string;
    cara : char;
    flasg : bool;
end_declaration

main:
    entry = 1;
    numero = 63.89;
    teste = "teste";
    cara = 'k';
    flasg = true;
    print(teste);
    print(entry);
    print(numero);
    print(cara);
    print(flasg);
end_main
```