

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE  
INSTITUTO METRÓPOLE DIGITAL - IMD  
CURSO DE BACHARELADO EM TECNOLOGIA DA  
INFORMAÇÃO

# **DOCUMENTAÇÃO DO INTERPRETADOR DE CCURE**

Dante Bezerra Pinto  
Henrique Lopes Fouquet  
João Pedro Silva  
Matheus Carvalho  
Paulo Vitor Lima Borges

Natal-RN  
2024

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE  
INSTITUTO METRÓPOLE DIGITAL - IMD  
CURSO DE BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

## **DOCUMENTAÇÃO DO INTERPRETADOR DE CCURE**

Dante Bezerra Pinto  
Henrique Lopes Fouquet  
João Pedro Silva  
Matheus Carvalho  
Paulo Vitor Lima Borges

Documentação apresentada à disciplina de Linguagens de Programação: Conceitos e Paradigmas, como requisito de parte da nota referente à unidade III

Natal-RN  
2024

# Sumário

<b>1</b>	<b>INTRODUÇÃO . . . . .</b>	<b>3</b>
<b>2</b>	<b>DESIGNS DE IMPLEMENTAÇÃO . . . . .</b>	<b>5</b>
<b>2.1</b>	<b>Transformação do código fonte em unidades léxicas . . . . .</b>	<b>5</b>
<b>2.2</b>	<b>Representação de símbolos, tabela de símbolos e funções asso-</b>	
	<b>ciadas . . . . .</b>	<b>5</b>
<b>2.3</b>	<b>Tratamento de estruturas condicionais e de repetição . . . . .</b>	<b>8</b>
2.3.1	Estrutura condicional (if-else) . . . . .	8
2.3.2	Estrutura de repetição (while) . . . . .	9
<b>2.4</b>	<b>Tratamento de subprogramas . . . . .</b>	<b>9</b>
2.4.1	Funções . . . . .	9
2.4.2	Procedimentos . . . . .	10
<b>2.5</b>	<b>Verificações realizadas . . . . .</b>	<b>11</b>
<b>3</b>	<b>INSTRUÇÕES DE USO DO INTERPRETADOR . . . . .</b>	<b>13</b>

# 1 Introdução

O presente relatório tem como objetivo apresentar uma visão geral do interpretador implementado para a linguagem *CCure* proposta, juntamente com sua documentação, designs de implementação e instruções de uso.

A linguagem *Ccure*, originalmente proposta pelo grupo, tem um foco voltado para a aplicação em sistemas críticos. Nesse contexto, a segurança se torna de preocupação primária, deixando a eficiência como preocupação secundária. Além disso, *Ccure* também tinha por objetivo oferecer tipos primitivos que pudessem auxiliar em aplicações científicas e facilitar o uso de certas operações para esses tipos. Devido a questões de tempo, nem todas as funcionalidades pensadas originalmente para a linguagem foram implementadas no interpretador. Nesse âmbito, apenas temos como tipo primitivo "novo" a matriz bidimensional, capaz de realizar operações de soma ou multiplicação por escalar, ou soma e multiplicação por matrizes.

De maneira geral, um código em *CCure* pode ser estruturado em 3 blocos de código:

- **Declaração de Tipos:** sessão opcional do código que, caso presente, armazena todas as declarações de tipos criados por usuário. Em *CCure*, nomes dados a tipos criados por usuário obrigatoriamente devem começar com letra maiúscula. Até o momento, apenas Registros são suportados como tipos criados por usuário.
- **Declaração de Subprogramas:** sessão opcional do código que, caso presente, armazena todas as declarações de subprogramas presentes no programa. Em *Ccure*, podem existir dois tipos de subprogramas declarados aqui:
  - Funções: subprogramas que podem receber parâmetros (apenas por valor), executam uma série de instruções e retornam um valor. Aqui, parâmetros só podem ser passados por valor a fim de evitar efeitos colaterais durante o cálculo do valor de uma expressão.
  - Procedimentos: subprogramas que podem receber parâmetros por valor ou referência, e realizam uma série de comandos.
- **Programa Principal:** sessão obrigatória que armazena a sequência de instruções principal do programa. Essas instruções podem conter:
  - Declaração de variável. Em *Ccure*, variáveis obrigatoriamente devem ser inicializadas ao serem declaradas.
  - Instrução de atribuição.
  - Instrução de chamada de procedimento.
  - Instrução de leitura (stup) e saída (puts) de dados.

- Estrutura condicional (if-else)
- Estrutura de repetição (while), juntamente com instrução de saída de bloco (break)

Um exemplo de código em *CCure* se encontra abaixo:

```
typeDeclarations
  register Rational_t
    int numerador = 0;
    int denominador = 1;
  endRegister
endTypeDeclarations
subprograms
  fun build_rational(int a, int b) -> Rational_t
    Rational_t ret = Default(Rational_t);
    ret->numerador = a;
    ret->denominador = b;
    return ret;
  endFun
endSubprograms
program
  Rational_t a = build_rational(1, 2);
  puts("a = ");
  puts(a->numerador);
  puts("/");
  puts(a->denominador);
  puts("\n");
end
```

## 2 Designs de Implementação

### 2.1 Transformação do código fonte em unidades léxicas

A transformação do código fonte em unidades léxicas foi realizada através da ferramenta Alex. Com a ajuda dessa ferramenta, padrões reconhecidos como expressões regulares na sintaxe de Regex são mapeados para Tokens, tipos criados para representar cada unidade léxica lida no arquivo de entrada do interpretador. O resultado da aplicação do Alex é uma lista de tais Tokens, a qual será mais tarde processada pelo analisador sintático, verificando sua estrutura. Vale lembrar que este passo não se preocupa que o arquivo possua alguma estrutura específica, apenas reconhece os tokens os mapeia para tipos do Haskell.

Abaixo, estão alguns exemplos de mapeamentos de expressões regulares e seus respectivos tokens para os quais foram mapeados.

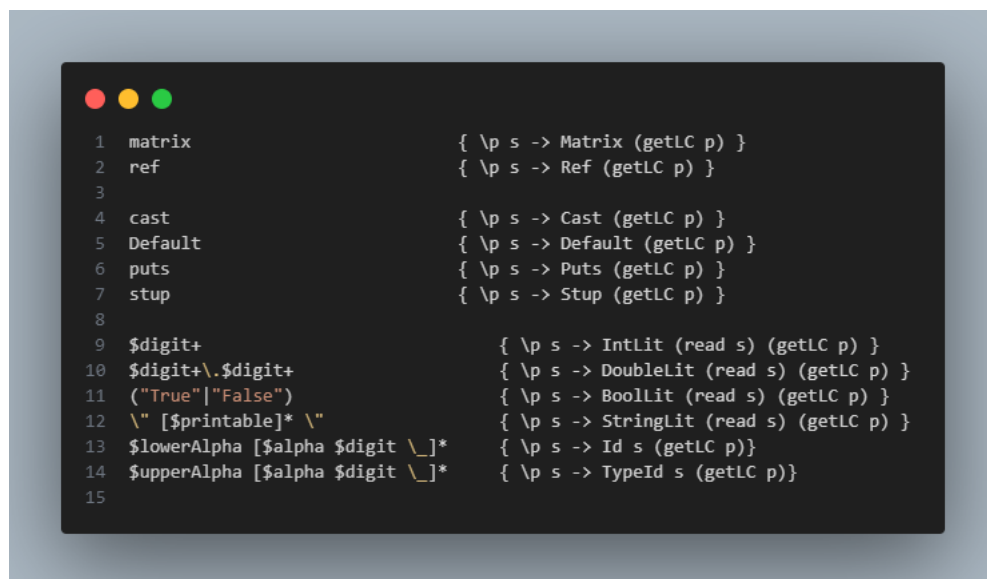


Figura 1 – Lexer

### 2.2 Representação de símbolos, tabela de símbolos e funções associadas

A representação dos símbolos lidos pela etapa léxica é feita criando um novo tipo em Haskell. Tal tipo foi chamado de *Token* e possui um representando para cada símbolo esperado pela linguagem. Durante o processamento sintático, as unidades léxicas são referidas como os elementos desse tipo.

A depender do tipo de unidade léxica, sua carga útil pode mudar. Alguns tokens possuem apenas uma tupla composta de dois inteiros, representando a linha e a coluna onde foram lidos

pelo Lexer. Alguns exemplos desses são as representações de *break*, *while*, *return*, *if*, *endIf*, *etc.* Mas tokens que podem assumir diferentes formas, como por exemplo literais inteiros, armazenam também em si o valor daquele literal. Por exemplo, o token *IntLit*, além de sua posição, armazena um Inteiro de Haskell. É possível ver alguns exemplos abaixo.

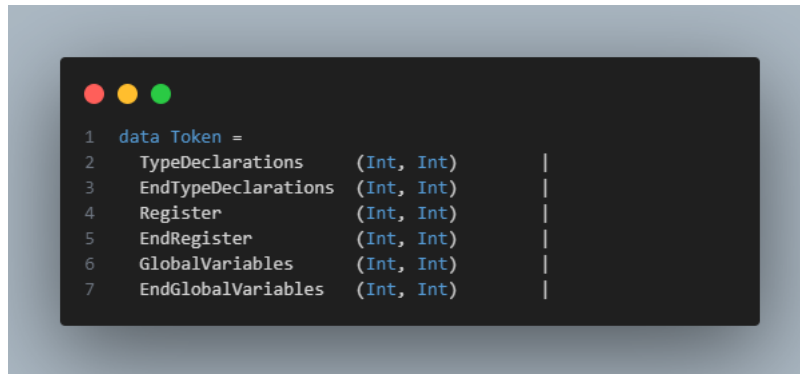


Figura 2 – Representação de Símbolos

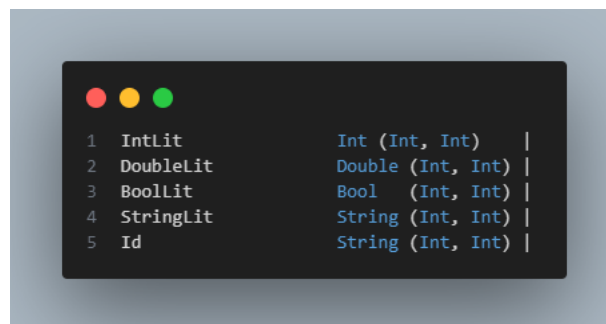


Figura 3 – Representação de Símbolos

O estado da linguagem *CCure* é composto por 8 componentes, descritos com mais detalhes a seguir:

- **Tabela de Símbolos:** armazena as variáveis que já foram declaradas até o momento atual de execução, bem como seus valores e atributos. Cada elemento da tabela de símbolos possui os seguintes campos:
  - Nome: representa o identificador da variável.
  - Escopo: representa o escopo no qual aquela variável foi declarada.
  - Pilha de profundidade dinâmica e valores: cada elemento dessa pilha representa a profundidade dinâmica na qual uma variável foi declarada, juntamente com seu valor associado.
- **Pilha de Escopo:** armazena uma pilha de todos os escopos já visitados até o momento corrente de execução. Quando se adentra em um determinado escopo, existem duas possibilidades:

- Caso seja um escopo de bloco (como em um while ou if), é adicionado na pilha o escopo atual concatenado com o escopo armazenado anteriormente no topo da pilha.
- Caso seja um escopo de um subprograma, o nome daquele subprograma é adicionado no topo da pilha (sem concatenar com o anterior).

Analogamente, ao sair de um escopo, o topo da pilha de escopos é removida.

- **Pilha de Loop:** armazena uma pilha do estado de execução de todos os loops correntes. O estado de um determinado loop pode ser das seguintes formas:
  - OK: Simboliza que o loop está em execução normal.
  - BREAK: Simboliza que a execução durante o loop foi desativada por uma instrução de saída de bloco (break).
- **Profundidade Dinâmica:** armazena a profundidade dinâmica atual. Quando um subprograma é chamado, essa profundidade é incrementada em 1. Analogamente, quando se sai do escopo de um subprograma, ela é decrementada em 1.
- **Tipos criados pelo Usuário:** armazena uma lista dos tipos declarados pelo usuário. Atualmente, apenas damos suporte para declaração de Registros pelo usuário. Cada elemento desse campo possui os seguintes atributos:
  - Nome: simboliza o nome dado àquele tipo declarado pelo usuário
  - Lista de nome e valor: armazena uma lista de todos os atributos daquele tipo, contendo, para cada elemento, o nome e valor padrão do atributo.
- **Procedimentos:** armazena uma lista contendo informações sobre todos os procedimentos declarados pelo usuário. Um dado elemento dessa lista possui os seguintes campos:
  - Nome: identificador daquele procedimento.
  - Corpo: armazena uma lista de símbolos referentes à sequência de instruções presentes no corpo do procedimento.
  - Lista de parâmetros: armazena informações sobre cada parâmetro recebido pelo procedimento, como seu identificador, tipo e se é passado por valor ou por referência.
- **Funções:** armazena uma lista contendo informações sobre todas as funções declaradas pelo usuário. Um dado elemento dessa lista possui todos os campos de um elemento da lista de Procedimentos, mas inclui também o seguinte campo:
  - Tipo: armazena o tipo retornado pela função.
- **Flag de execução:** indica se a execução do programa está ativa ou não. Caso esteja ativa, a alteração do estado é permitida.



Para a implementação do interpretador, diversas funções envolvendo o estado foram utilizadas. Algumas delas são listadas a seguir:

- *isInSymTable*: Checa se uma determinada variável se encontra na tabela de símbolos
- *symtable\_get*: Retorna o valor associado à uma determinada variável presente na tabela de símbolos.
- *symtable\_update*: Atualiza o valor de uma dada variável na tabela de símbolos
- *insertUserType*: Insere um novo tipo declarado pelo usuário na lista de Tipos declarados pelo Usuário
- *isInUserTypes*: Verifica se um determinado tipo criado pelo usuário já foi declarado.
- *getUserType*: Retorna o registro associado a um determinado tipo declarado pelo usuário.
- *isRegAttr*: Verifica se um atributo pertence a um determinado registro.
- *getRegAttr*: Retorna o valor associado a um atributo de um determinado registro.

## 2.3 Tratamento de estruturas condicionais e de repetição

### 2.3.1 Estrutura condicional (if-else)

A sintaxe sintática de uma estrutura condicional em *CCure* é dada da seguinte forma (onde a utilização de "else" é opcional):

```
if (<expr >
    <stmts1 >
[ else ]
    <stmts2 >
endIf
```

Durante a execução de uma estrutura condicional, a expressão encontrada em *<expr>* é avaliada. Caso seja avaliada para um valor não booleano, um erro de tipagem é detectado, parando o programa.

Caso seja avaliada como True, *<stmts1>* serão parseados com a execução ativa. Após isso, a execução é desligada até a chegada do "endIf", onde ela é novamente ligada.

Caso seja avaliada como False, a execução é desligada, e *<stmts1>* serão parseados apenas sintaticamente. Caso se encontre um "else", a execução é novamente ligada, de forma a parsear *<stmts2>*. Caso não, a execução é ligada ao se encontrar "endIf".

### 2.3.2 Estrutura de repetição (while)

A sintaxe sintática de uma estrutura de repetição *while* em *CCure* é dada da seguinte forma

```
while(<expr>)  
  <stmts>  
endWhile
```

A expressão *<expr>* é avaliada de maneira semelhante à do tópico 2.3.1. Caso verdadeira, os comandos no blocos serão parseados com a execução ativa. Após isso, o *<expr>* é avaliado novamente e, caso verdadeira, o ciclo se repete. O programa só irá sair do bloco quando a expressão for falsa, ou quando uma instrução de "break" ou "return" seja encontrada em *<stmts>*.

## 2.4 Tratamento de subprogramas

### 2.4.1 Funções

Em *CCure*, uma função deve ser previamente declarada na seção *subprograms* do código. Um exemplo de estrutura sintática de uma declaração de função se encontra abaixo:

```
subprograms  
  fun <funId> (<funArgs>) -> <returnType>  
    <stmts>  
  endFun  
endSubprograms
```

Aqui, os seguintes símbolos representam o seguinte:

- *<funID>*: identificador da função declarada. Caso já exista alguma outra função declarada anteriormente com esse identificador, a execução será interrompida com um erro.
- *<funArgs>*: lista de zero ou mais argumentos da função declarada. Cada argumento é composto por seu tipo, precedendo seu identificador. Múltiplos argumentos devem ser separados por vírgula.
- *<returnType>*: tipo de retorno esperado para a função. Caso esse tipo seja inválido (como um tipo declarado pelo usuário que não foi declarado, por exemplo), a execução será interrompida com um erro.

Dado que uma função foi declarada corretamente, chamadas para aquela função podem ser realizadas em expressões. Uma chamada de função pode ser sintaticamente estruturada da seguinte forma:

```
<funCall> = <funId> (<funCallArgs>)
```

Aqui, `<funId>` representa o identificador da função, e `<funCallArgs>` a lista de argumentos a ser passada. Cada um desses argumentos são expressões, e devem ser separados por vírgulas. A quantidade de parâmetros reais, assim como o tipo de cada um deles, deve ser compatível com os parâmetros formais da função. Caso isso não ocorra, a execução será interrompida com uma mensagem de erro.

Com a invocação da função, o corpo da função será parseado e executado. Isso implica em atualizar a Stack de Escopo e Profundidade Dinâmica do estado. Caso se chegue no final do corpo da função, a execução será interrompida com uma mensagem de erro (já que um "return" não foi encontrado). Além disso, caso um "return" seja encontrado com tipo incompatível ao tipo de retorno da função, outro erro será exibido. Ao voltar da execução da função, a Stack de Escopo e Profundidade Dinâmica do estado são novamente atualizados.

## 2.4.2 Procedimentos

Procedimentos possuem um comportamento semelhante ao de funções, porém com algumas diferenças:

- Na declaração dos argumentos do procedimento, pode-se especificar que estes serão passados por valor ou por referência. Caso seja por valor, a construção do argumento equivale à da função. Caso seja por referência, o argumento deve ser precedido por "ref".
- Procedimentos não possuem retorno.
- Procedimentos não são tratados como expressões, mas sim como instruções.

Um exemplo de estrutura sintática de uma declaração de procedimento em *CCure* se encontra abaixo:

```
subprograms
  proc <procId> (<procArgs>)
    <stmts>
  endProc
endSubprograms
```

Dado que um procedimento foi declarado corretamente, chamadas para aquele procedimento podem ser realizados. A sintaxe de uma instrução que corresponde a uma chamada de procedimento se encontra abaixo:

```
<procId> (<procCallArgs>) ;
```

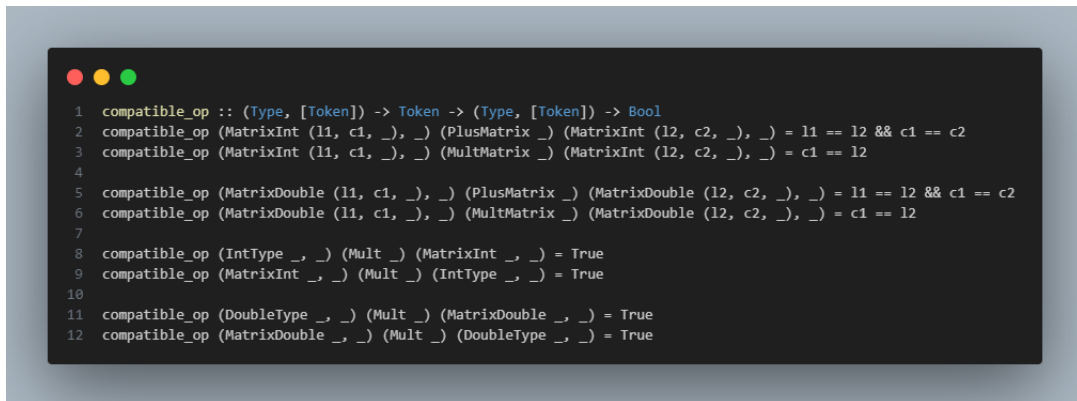
Analogamente ao caso das funções, os parâmetros reais aqui devem corresponder em número e em tipo aos formais. Além disso, aqueles que devem ser passados por referência devem ser precedidos por "ref". Vale ressaltar que não é permitido passar valores literais por referência (caso ocorra, uma mensagem de erro será exibida).

## 2.5 Verificações realizadas

Em alinhamento com a proposta da linguagem (segurança), é necessário que se façam várias verificações em prol desse objetivo. Tais verificações são inúmeras e de vários âmbitos diferentes. Abaixo alguma delas serão detalhadas.

- **Verificações de operações:**

Para que seja possível realizar qualquer operação em *CCure*, é necessário que seus operandos sejam compatíveis. Por exemplo, só é possível realizar uma soma entre duas matrizes caso suas dimensões sejam idênticas, além de que estas devem ter mesmo tipo em suas entradas, assim, foi necessário criar uma função de verificação para tal caso:



```

1 compatible_op :: (Type, [Token]) -> Token -> (Type, [Token]) -> Bool
2 compatible_op (MatrixInt (l1, c1, _), _) (PlusMatrix _) (MatrixInt (l2, c2, _), _) = l1 == l2 && c1 == c2
3 compatible_op (MatrixInt (l1, c1, _), _) (MultMatrix _) (MatrixInt (l2, c2, _), _) = c1 == l2
4
5 compatible_op (MatrixDouble (l1, c1, _), _) (PlusMatrix _) (MatrixDouble (l2, c2, _), _) = l1 == l2 && c1 == c2
6 compatible_op (MatrixDouble (l1, c1, _), _) (MultMatrix _) (MatrixDouble (l2, c2, _), _) = c1 == l2
7
8 compatible_op (IntType _, _) (Mult _) (MatrixInt _, _) = True
9 compatible_op (MatrixInt _, _) (Mult _) (IntType _, _) = True
10
11 compatible_op (DoubleType _, _) (Mult _) (MatrixDouble _, _) = True
12 compatible_op (MatrixDouble _, _) (Mult _) (DoubleType _, _) = True

```

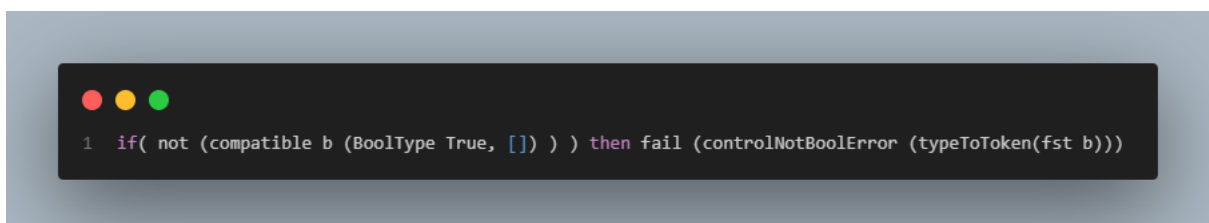
Figura 4 – Verificação de soma de matrizes

Além dessa, também estão ilustradas verificações usuais para multiplicação por escalar e multiplicação de matrizes.

Também foram feitas verificações usuais para operações booleanas e aritméticas.

- **Verificações de controle de fluxo**

Também foram feitas verificações nas estruturas de repetição e condicional. É necessário verificar que a expressão que vem logo após do uso de uma destas é booleana. Por exemplo, após identificar um *while statement*, o parser procura por uma expressão e a avalia, caso esta avaliação não produza um booleano, um erro é lançado.



```

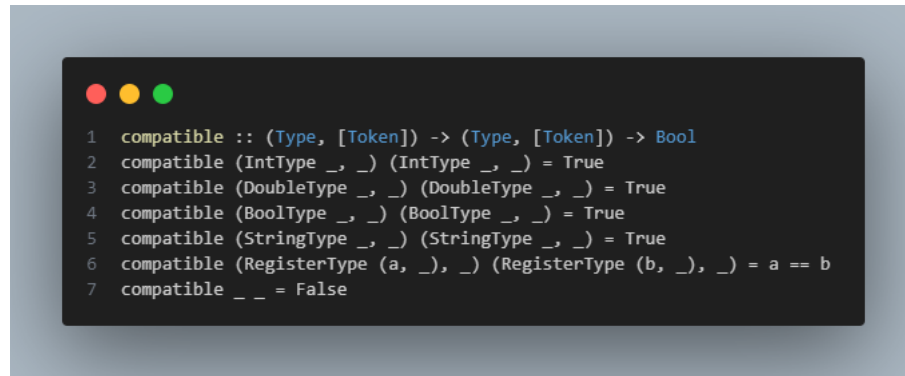
1 if( not (compatible b (BoolType True, []) ) ) then fail (controlNotBoolError (typeToToken(fst b)))

```

Figura 5 – Verificação em While

- **Verificações de compatibilidade na atribuição**

Assim como em operações, é necessário que sejam verificados os tipos dos valores que se deseja atribuir a uma variável. Por exemplo, só é possível atribuir literais inteiros para variáveis do tipo primitivo *int*. Funções auxiliares para tais verificações foram criadas e são ilustradas abaixo.



```
1 compatible :: (Type, [Token]) -> (Type, [Token]) -> Bool
2 compatible (IntType _, _) (IntType _, _) = True
3 compatible (DoubleType _, _) (DoubleType _, _) = True
4 compatible (BoolType _, _) (BoolType _, _) = True
5 compatible (StringType _, _) (StringType _, _) = True
6 compatible (RegisterType (a, _), _) (RegisterType (b, _), _) = a == b
7 compatible _ _ = False
```

Figura 6 – Funções de compatibilidade para atribuição

### 3 Instruções de uso do interpretador

Após baixar os arquivos disponíveis no repositório (<https://github.com/DanteAugusto/LPCP>), em seu terminal adentre a pasta *ccure* e execute o seguinte comando:

```
./ccure.sh <caminho_para_programa.ccr>
```

Além disso, é requisitado que Haskell e Alex estejam instalados na máquina. Para instruções mais detalhadas, consulte o arquivo README.md no repositório referenciado.