# DCC024 Linguagens de Programação
# 2022.1

## Projeto 2

**Data de entrega:** 8 de julho de 2022

O Projeto 2 deve ser feito com a mesma dupla do Projeto 1. Ambos os estudantes receberão a mesma nota.

*Qualquer indício de fraude será comunicado às instâncias competentes da UFMG. Note que ambos os estudantes são responsáveis pela submissão, independentemente de como o trabalho é dividido entre eles.*

Descomprima o arquivo `project2.zip` e use a pasta `project2` extraída como a base para seu trabalho. A pasta contém arquivos que serão necessários para o projeto. Escreva suas soluções seguindo as instruções abaixo. A solução do projeto 1 deve ser incorporada à solução do projeto 2. **Podem ser feitas melhorias no parser que foi feito no projeto 1.** Ao fim comprima `project2` para um arquivo `project2.zip` e o submeta. *Tome cuidado para submeter o arquivo zip com a sua solução, não o original!*

Note que apenas um dos estudantes da dupla deve realizar a submissão.

**Atenção:** Seus arquivos *devem poder ser executados* sem erros de sintaxe ou tipagem. Perdas severas de pontos podem ser aplicadas se o código contiver tais erros.

# 1 Detalhes sobre entrega das repostas

- Para solução do projeto 2 que deverá ser entregue no dia 08/07/2022 até 23:59. Deverá conter uma implementação correta e completa dos módulos descritos na **Seção 3** .
- É parte do seu projeto criar casos de teste para o verificador de tipos e o interpretador. Alguns exemplos de testes estão disponíveis no arquivo em anexo.
- Ambas as submissões devem ser feitas **uma vez por dupla** através de arquivo `project2.zip` contendo a sua solução. **O mesmo aluno da dupla que fez a submissão do projeto 1 deve fazer a submissão do projeto 2.**

# 2 Sintaxe concreta

## 2.1 Valores

Termos em SML de tipo `plcValue` são usados para representar valores PLC. O interpretador PLC é essencialmente um conversor de termos `expr` para termos `plcValue`. Alguns exemplos de conversões:

| | Expressão |
|---|---|
| 1. | `ConI 15` |
| 2. | `ConB true` |
| 3. | `List []` |
| 4. | `List [ConI 6; ConB false]` |
| 5. | `Item (1, List [ConI 6; ConB false])` |
| 6. | `ESeq (SeqT BoolT)` |
| 7. | `Prim2 (";", Prim1 ("print", ConI 27), ConB true)` |
| 8. | `Prim1 ("print", ConI 27)` |
| 9. | `Prim2 ("::", ConI 3, Prim2 ("::", ConI 4, Prim2 ("::", ConI 5, ESeq (SeqT IntT))))` |
| 10. | `Anon (IntT, "x", Prim1(-", Var "x"))` |
| 11. | `Let ("x", ConI 9, Prim2 ("+", Var "x", ConI 1))` |
| 12. | `Let ("f", Anon (Int, "x", Var "x"), Call (Var "f", ConI 1))` |

| | Valor |
|---|---|
| 1. | `IntV 15` |
| 2. | `BoolV true` |
| 3. | `ListV []` |
| 4. | `ListV [IntV 6; BoolV false]` |
| 5. | `IntV 6` |
| 6. | `SeqV []` |
| 7. | `BoolV true` |
| 8. | `ListV []` |
| 9. | `SeqV [IntV 3; IntV 4; IntV 5]` |
| 10. | `Clos (, "x", Prim1(-", Var "x")), []` (para um ambiente vazio) |
| 11. | `IntV 10` |
| 12. | `IntV 1` |

Expressões de funções anônimas da forma `Anon (`$t$`, `$x$`, `$e$`)` devem ser avaliadas para o valor `Clos (, `$x$`, `$e$`, `$env$`)`, em que $env$ é o ambiente atual.

Com expressões da forma `Prim1("print", `$e$`)`, o interpretador deve primeiro avaliar $e$ para algum valor $v$, converter $v$ para sua representação como uma string na sintaxe concreta, e então imprimir essa string para a saída padrão (`stdout`) seguida de um caractere de quebra de linha. Para a conversão para string, utilize a função `val2string : plcVal -> string` do módulo `Absyn`.

Dúvidas sobre a interpretação de outras expressões bem tipadas não cobertas acima devem ser esclarecidas pelo tópico de discussões no Moodle.

## 3 Implementação

Nesta segunda parte a sua implementação de PLC deve ser dividida nos arquivos descritos abaixo, cada um representando um módulo do arcabouço de tratamento de programas PLC. É preciso seguir essa modularização para seu benefício e do da avaliação de seu código.

- `PlcChecker`
  Este módulo é responsável pela checagem de tipos. Ele será provido no arquivo `PlcChecker.sml`. Ele deve prover uma função `teval : expr -> plcType env -> plcType` que, data uma expressão $e$ em sintaxe abstrata e um ambiente de tipos para as variáveis livres em $e$ (pode não

haver nenhuma), produz o tipo de $e$ naquele ambiente se $e$ é bem-tipada e falha (produzindo uma das exceções já presentes no arquivo) caso contrário. A implementação de `teval` deve seguir as regras de tipagem especificadas no Apêndice A.

- `PlcInterp`
  Este módulo é responsável pela interpretação de programas PLC. Ele será provido no arquivo `PlcInterp.sml`. Ele deve prover uma função `eval : expr -> plcValue env -> plcValue` que, dada uma expressão $e$ *bem-tipadas* e um ambiente de valores para as variáveis livres de $e$ (pode não haver nenhuma), produz o valor de $e$ naquele ambiente.

  Erros de interpretação devem gerar as respectivas exceções já presentes em `PlcInterp.sml`.

  Perceba que se espera que `eval` se perca (nunca produzindo um valor) se $e$ denota uma computação infinita; por exemplo, se $e$ vem de um programa como `fun rec f(Int x):Int = f(x - 1); f(0)`.

- `Plc`
  Este módulo, no arquivo `Plc.sml`, define uma função `run : expr -> string` que toma uma expressão $e$ em sintaxe abstrata, faz sua checagem de tipos com `teval`, a avalia com `eval`, e produz a string contendo o valor e o tipo de $e$ em sintaxe concreta. Exceções geradas por `teval` ou `eval` devem tratadas em `run`, produzindo mensagens de erro significativas, condizentes com a exceção disparada.

  Usando a função `run` justo com `fromString` ou `fromFile` é possível testar sua implementação do verificador de tipos e do interpretador.

# 4 Exceptions

1. exception `EmptySeq`
   A sequência de entrada não contém nenhum elemento
2. exception `UnknownType`
   É usada nas situações onde nenhuma das específicas se encaixa.
3. exception `NotEqTypes`
   Se os tipos usados numa comparação são diferentes.
4. exception `WrongRetType`
   O tipo de retorno da função não condiz com o corpo da mesma.
5. exception `DiffBrTypes`
   Os tipos da expressões dos possíveis caminhos de um If divergem
6. exception `IfCondNotBool`
   A condição do if não é booleana
7. exception `NoMatchResults`
   Não há resultados para a expressão match
8. exception `MatchResTypeDiff`
   O tipo de algum dos casos em match difere dos demais

9. exception `MatchCondTypesDiff`
   O tipo das opções de match difere do tipo da expressão passada para Match

10. exception `CallTypeMisM`
    Você está passando pra uma chamada de função um tipo diferente do qual ela suporta

11. exception `NotFunc`
    Você está tentando chamar algo que não é uma função.

12. exception `ListOutOfRange`
    Tentativa de acessar um elemento fora dos limites da lista

13. exception `OpNonList`
    Tentativa de acessar um elemento em uma expressão que não é uma lista.

**Perguntas e Respostas**

1. Fiquei na dúvida entre o uso do `NoMatchResults`(7) dentro de um Match. Entendi que o `MatchResType`(8) é usado entre a comparação dos tipos dos retornos do Match apenas. O `MatchCondTypes`(9) entendi que seria o que você mesmo disse acima. Porém, o `NoMatchResults`(7) seria a comparação entre quais elementos? "Não haver resultado para a expressão Match"seria o caso de eu ter um Match sem opções ou o caso semelhante de "O tipo das opções de match difere do tipo da expressão passada para Match"(9)?

   `NoMatchResults` é usado quando não há expressões para se fazer o casamento, i.e., a lista de expressões de `MatchExpr` é vazia.

2. Além disso, o `MatchResType`(8) seria usado tanto na verificação de tipos das opções quanto dos tipos dos retornos da opções?

   Não sei se entendi muito bem o que você quis dizer com "tipos das opções"e "tipos de retorno das opções". Mas, de toda forma, `MatchResTypeDiff` só é usado na verificação dos tipos das expressões que compõem a lista de Match.

3. A exceção `ListOutOfRange` é para o caso de uma expressão Item estar tentando acessar um índice inválido na lista. (Regra 25)

   Sobre `EmptySeq`, ela está relacionada com a regra 7. Ela trata uma expressão ESeq que deve retornar o tipo t apenas se t for um tipo sequência, isto é, ESeq (SeqT t). Caso contrário, uma sequência construída com tipo que não é SeqT, você deve disparar `EmptySeq`.

# 5 Fluxo Geral do Trabalho

Pra deixar mais claro como o fluxo do trabalho deve funcionar, abaixo temos um exemplo de execução do interpretador de Plc, desde o programa em sintaxe concreta até sua avaliação.

Vamos usar o seguinte exemplo:

```
fun rec f1(Int x):Int = x + 1; f1(12)
```

Ao fazer parsing desse programa, obtemos a expressão em sintaxe abstrata:

```
Letrec("f1",IntT,"x",IntT,Prim2 ("+",Var "x",ConI 1),Call (Var "f1",ConI 12))
```

Essa expressão agora deve ser passada para a função run do arquivo `Plc.sml`. Essa função irá chamar `teval` e `eval` para fazer checagem de tipos e avaliação da expressão em si.
A saída esperada é:

```
"13 :  Int":  string
```

Se quiserem testar, carreguem todos os módulos do interpretador de Plc no interpretador de SML, e vocês podem fazer:

```
-val e = fromString "fun rec f1(Int x):Int = x + 1; f1(12)";

-run e;

 val it = "13 :  Int":  string
```

Note que nesse caso, o ambiente passado para a função está vazio, mas as funções `teval` e `eval` podem receber algum ambiente não vazio.

**Importante:** O formato da saída deve ser esse: valor `":"` tipo.

**Importante:** Vocês também devem tratar a exceção *SymbolNotFound* de `Environ.sml`

# A Regras de tipagem de PLC

A seguir, $x$ denota nomes de variáveis ou funções; $n$ denota numerais; $e$, $e_1$, $e_2$ denotam expressões PLC; $s$, $t$, $t_i$ denotam tipos PLC; $\rho$ denota um ambiente de tipos, isto é, um mapa parcial de nomes de variáveis ou funções para tipos; $\rho[x \mapsto t]$ denota o ambiente que mapeia $x$ para $\tau$ e é de outra forma idêntico a $\rho$; $type(e,\ \rho) = t$ abrevia a sentença: "o tipo da expressão $e$ no ambiente $\rho$ é $\tau$."

As regras abaixo definem o sistema de tipos para PLC. Uma expressão $e$ é bem-tipada e tem tipo $\tau$ em ambiente de tipagem $\rho$ se e somente se pode-se concluir $type(e,\ \rho) = \tau$ de acordo com essas regras.

1. $type(x,\ \rho) = \rho(x)$
2. $type(n,\ \rho) =$ `Int`
3. $type($`true`$,\ \rho) =$ `Bool`
4. $type($`false`$,\ \rho) =$ `Bool`

5. $type(\texttt{()},\ \rho) = \texttt{Nil}$

6. $type(\texttt{(}e_1\texttt{, ..., }e_n\texttt{)},\ \rho) = \texttt{(}t_1\texttt{, ..., }t_n\texttt{)}$ se $n > 1$ e $type(e_i,\ \rho) = t_i$ para todo $i = 1, \ldots, n$

7. $type(\texttt{(}t\ \texttt{[])},\ \rho) = t$ se $t$ é um tipo sequência.

8. $type(\texttt{var } x \texttt{ = } e_1 \texttt{ ; } e_2,\ \rho) = t_2$ se
   $type(e_1,\ \rho) = t_1$ e $type(e_2,\ \rho[\texttt{x} \mapsto t_1]) = t_2$ para algum tipo $t_1$

9. $type(\texttt{fun rec } f\ \texttt{(}t\ x\texttt{) : } t_1 \texttt{ = } e_1 \texttt{ ; } e_2,\ \rho) = t_2$
   se $type(e_1,\ \rho[f \mapsto t \texttt{ -> } t_1][x \mapsto t]) = t_1$ e $type(e_2,\ \rho[f \mapsto t \texttt{ -> } t_1]) = t_2$

10. $type(\texttt{fn (}s\ x\texttt{) => } e \texttt{ end},\ \rho) = s \texttt{ -> } t$ se $type(e,\ \rho[x \mapsto s]) = t$

11. $type(e_2\texttt{(}e_1\texttt{)},\ \rho) = t_2$ se $type(e_2,\ \rho) = t_1 \texttt{ -> } t_2$ e $type(e_1,\ \rho) = t_1$ para algum tipo $t_1$

12. $type(\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2,\ \rho) = t$ se $type(e,\ \rho) = \texttt{Bool}$ e $type(e_1,\ \rho) = type(e_2,\ \rho) = t$

13. $type(\texttt{match } e \texttt{ with | } e_1 \texttt{ -> } r_1 \texttt{ | ...| } e_n \texttt{ -> } r_n,\ \rho) = t$ se

    (a) $type(e,\ \rho) = type(e_i,\ \rho)$, para cada $e_i$ diferente de '_', e

    (b) $type(r_1,\ \rho) = \ldots = type(r_n,\ \rho) = t$

14. $type(\texttt{!}e,\ \rho) = \texttt{Bool}$ se $type(e,\ \rho) = \texttt{Bool}$

15. $type(\texttt{-}e,\ \rho) = \texttt{Int}$ se $type(e,\ \rho) = \texttt{Int}$

16. $type(\texttt{hd(}e\texttt{)},\ \rho) = t$ se $type(e,\ \rho) = \texttt{[}t\texttt{]}$

17. $type(\texttt{tl(}e\texttt{)},\ \rho) = \texttt{[}t\texttt{]}$ se $type(e,\ \rho) = \texttt{[}t\texttt{]}$

18. $type(\texttt{ise(}e\texttt{)},\ \rho) = \texttt{Bool}$ se $type(e,\ \rho) = \texttt{[}t\texttt{]}$ para algum tipo $t$

19. $type(\texttt{print(}e\texttt{)},\ \rho) = \texttt{Nil}$ se $type(e,\ \rho) = t$ para algum tipo $t$

20. $type(e_1 \texttt{ \&\& } e_2,\ \rho) = \texttt{Bool}$ se $type(e_1,\ \rho) = type(e_2,\ \rho) = \texttt{Bool}$

21. $type(e_1 \texttt{ :: } e_2,\ \rho) = \texttt{[}t\texttt{]}$ se $type(e_1,\ \rho) = t$ e $type(e_2,\ \rho) = \texttt{[}t\texttt{]}$

22. $type(e_1\ op\ e_2,\ \rho) = \texttt{Int}$ se $op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$ e $type(e_1,\ \rho) = type(e_2,\ \rho) = \texttt{Int}$

23. $type(e_1\ op\ e_2,\ \rho) = \texttt{Bool}$ se $op \in \{\texttt{<}, \texttt{<=}\}$ e $type(e_1,\ \rho) = type(e_2,\ \rho) = \texttt{Int}$

24. $type(e_1\ op\ e_2,\ \rho) = \texttt{Bool}$ se $op \in \{\texttt{=}, \texttt{!=}\}$ e $type(e_1,\ \rho) = type(e_2,\ \rho) = t$ para algum **tipo de igualdade** $t$

25. $type(e\ \texttt{[}i\texttt{]},\ \rho) = t_i$ se $type(e,\ \rho) = \texttt{(}t_1\texttt{, ..., }t_n\texttt{)}$ para algum $n > 1$ e tipos $t_1, \ldots, t_n$, e $i \in \{1, \ldots, n\}$

26. $type(e_1 \texttt{ ; } e_2,\ \rho) = t_2$ se $type(e_1,\ \rho) = t_1$ para algum tipo $t$ e $type(e_2,\ \rho) = t_2$