DCC024 Linguagens de Programação
2020.2

Projeto da disciplina

Datas de entrega: 12 de Março (parcial), 19 de Março (final)

O projeto deve ser feito em dupla. 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 `project.zip` e use a pasta `project` extraída como a base para seu trabalho. A pasta contém arquivos que serão necessários para o projeto. Escreva suas soluções seguindo as instruções abaixo. Ao fim comprima `project` para um arquivo `projectsol.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. Veja a Seção 7 para uma descrição detalhada de como a entrega deve ser feita.

1 A linguagem PLC

Neste projeto você irá desenvolver em SML um interpretador, e ferramentas relacionadas, para a linguagem PLC, que pode ser vista como uma extensão da linguagem "micro-ML" de expressões utilizada durante a primeira parte da disciplina. A linguagem incorpora vários conceitos de programação vistos durante a disciplina. Ela é uma linguagem puramente funcional, estática e estritamente tipada, com escopo estático, e de ordem superior. Entre as funcionalidades presentes em PLC que não abordamos diretamente durante as aulas está um tipo para sequências e funções primitivas para sua manipulação. O tipo de sequências em PLC é similar ao tipo de listas em SML, com valores consistindo de uma série ordenada e imutável de elementos do mesmo tipo. A linguagem possui também funções anônimas, um casamento de padrões simplificado, e um comando de impressão.

As principais limitações de PLC com respeito a linguagens funcionais "reais", impostas em nome da simplicidade, são a ausência de comandos para ler entrada do console ou de arquivos; funções são monomórficas e podem ser recursivas mas não mutuamente recursivas; parâmetros formais de funções devem ser explicitamente tipados; funções recursivas devem declarar seus tipos de retorno; e casamento de padrões é restrito a comparação de valores, i.e. corresponde a açúcar sintático para uma série de comandos `if-then-else`. Por último, não estão presentes tipos básicos comuns como caracteres ou strings, ou tipos estruturados como tipos de dados algébricos e estruturas.

```
fun inc (Int x) = x + 1;
fun add (Int x, Int y) = x + y;
fun cadd (Int x) = fn (Int y) => x + y end;
var y = add(3, inc(4));
var x = cadd(3)(7-y);
var z = x * 3;
fun rec fac (Int n) : Int =
  match n with
  | 0 -> 1
  | 1 -> 1
  | _ -> n * fac(n - 1)
  end
;
print x; print y;
x :: y :: z :: fac(z) :: ([Int] [])
```

Figura 1: Um programa PLC.

A Figura 1 mostra um exemplo de um programa PLC. O programa define uma função não-recursiva de primeira ordem `inc` de tipo `Int -> Int`; uma função não-recursiva de primeira ordem `add` de tipo `(Int,Int) -> Int`; uma função não-recursiva de ordem superior `cadd`; variáveis `x`, `y` e `z`; e uma função recursiva de primeira ordem `fac`. O escopo de cada uma dessas funções e variáveis inclui as declarações e expressões que as seguem. No exemplo, as expressões após a declaração de `fac` também são separadas por ponto e vírgula. Quando usado com expressões, ponto e vírgula é um operador binário associativo à direita tal que e_1 `;` e_2 é avaliado para o valor de e_2, quaisquer que sejam as expressões e_1 e e_2. No programa do exemplo, a primeira expressão imprime no console o valor de `x` e `y` e então produz a lista consistindo dos valores de `x`, `y`, `z`, `fac(z)` e `y`. A função `cadd` toma um inteiro `x` e produz a função anônima `fn (Int y) => x + y end`, a qual toma um inteiro `y` e produz o valor de `x + y`. A função `fac` implementa o fatorial da entrada, tomando um valor do tipo `Int` e produzindo outro.

Declarações de funções devem incluir o tipo de retorno *apenas* se a função for recursiva. Declarações recursivas também precisam do qualificador `rec` após a palavra chave `fun`. Como PLC possui funções anônimas, declarações de funções não-recursivas são de fato açúcar sintático. Ou seja, um programa como

$$\texttt{fun } f\texttt{(}t\ x\texttt{) = } e\ \texttt{;}\ e_1$$

é tratado como o programa

$$\texttt{var } f\ \texttt{= fn (}t\ x\texttt{) => } e\ \texttt{end ;}\ e_1$$

em que f se torna a variável de ordem superior com tipo t `->` t_e (em que t_e é o tipo de e) cujo valor é a função anônima `fn (`$t\ x$`) =>` e `end`. Assim, as únicas declarações de funções primitivas são as de funções recursivas.

Uma restrição no uso de `;` é que declarações (de variáveis ou funções) não podem seguir expressões, a não ser que elas incluam um bloco delimitado por chaves. Por exemplo:

```
var E = ([Int] []);
fun reverse ([Int] s) = {
  fun rec rev ([Int] s1, [Int] s2): [Int] =
    match s1 with
    | E -> s2
    | _ -> {
             var h = hd(s1);
             var t = tl(s1);
             rev(t, h::s2)
           }
    end
  ;
  rev(s, E)
};
reverse (1::2::3::E)
```

Figura 2: Um programa PLC com funções definidas localmente.

```
1 - 3; var x = 4; 2 * x
```

não é permitido, enquanto

```
1 - 3; {var x = 4; 2 * x}
```

é. Resumindo, o último argumento de `;` deve ser uma expressão, não uma declaração.

Sequências de declarações e expressões envoltas por chaves são tratadas como expressões atômicas, isto é, elas podem ser usadas em qualquer ponto em que expressões podem ser usadas. Isto permite por exemplo declarar variáveis locais e funções dentro de outras funções, como no programa da Figura 3.

2 Tipos, anotações de tipos e tipagem estática

Sequências em PLC são essencialmente o mesmo que listas em SML, com `[]` denotando a sequência vazia e `::` denotando o construtor de sequências não-vazias. Note, no entanto, que a sequência vazia deve ser explicitamente tipada quando quer que seja usada, como visto nos programas das Figuras 1– 3. Isto é para que a checagem de tipos seja simplificada significativamente, similarmente com a convenção de que parâmetros formais de funções sejam explicitamente tipados e de que funções recursivas devam declarar seu tipo de retorno. O último caso simplifica a checagem de tipos do corpo da função, que inclui ocorrências do nome da função (em chamadas recursivas).

Como a linguagem é de ordem superior, é possível definir e utilizar os combinadores que vimos anteriormente, apenas com a restrição de que eles não podem ser polimórficos.[1] Exemplos de tais funções são dados na Figura 2. A função `map` é como aquela que estamos acostumados, exceto que

[1] Esta é uma limitação séria, já que agora é necessário por exemplo definir uma função `map` para cada possível instanciação do tipo paramétrico `('a -> 'b) -> 'a list -> 'a list` que `map` teria e.g. em SML. Assim, para usar `map` para sequências de inteiros seria necessário definir a função com o tipo `(Int -> Int) -> [Int] -> [Int]`, bem como definir outras funções `map` para sequências com outros tipos. Esta restrição no entanto facilita a checagem de tipos.

```
fun twice (Int -> Int f) = fn (Int x) => f(f(x)) end ;
fun rec map (Int -> Int f) : ([Int] -> [Int]) =
  fn ([Int] s) =>
    if ise(s) then s else f(hd(s)) :: map(f)(tl(s))
  end ;
fun square (Int x) = x * x ;
fun inc (Int x) = x + 1 ;
var E = ([Int] []) ;
var s1 = map (fn (Int x) => 2*x end) (10::20::30::E) ;
var s2 = map (twice(inc)) (s1) ;
(s1, s2)
```

Figura 3: Um programa PLC com combinadores.

restrita a sequências de inteiros como entrada e saída, bem como com uma declaração um pouco mais verbosa do que em SML.

2.1 Tipos e operadores

A linguagem possui os tipos, e operações sobre eles, a seguir. Seu interpretador deve prover suporte a todos a eles.

Tipo *Nil* O tipo `Nil`, similar ao tipo `unit` em SML, contém um único valor. Operadores pré-definidos para lidar com valores `Nil` são: `() : Nil`, o único valor deste tipo, e `print : `τ` -> Nil`, para qualquer tipo τ. A última função deve sempre produzir `()` mas possui o efeito colateral de imprimir no console (*standard output*) uma representação textual de seu valor de entrada.

Tipo *Boolean* O tipo `Bool` é o tipo Booleano usual. Além das constantes `true` e `false`, ele possui operadores pré-definidos `&& : (Bool, Bool) -> Bool` para conjunção Booleana e `! : Bool -> Bool` para negação Booleana. Dois outros operadores são`=` e `!=`, ambos de tipo `(`τ`, `τ`) -> Bool` para quaisquer tipos *iguais* τ (veja abaixo), respectivamente para comparações de igualdade e desigualdade.

Tipo *Integer* O tipo `Int` é o tipo inteiro usual cujas constantes são todos os numerais. Ele possui as operações binárias infixas usuais `+`, `-`, `*`, `/`, `<`, e `<=`, com o significado esperado. As primeiras quatro possuem tipo`(Int, Int) -> Int`. As últimas duas possuem tipo `(Int, Int) -> Bool`. O operador `-` é também unário, com tipo `Int -> Int`.

Tipo *List* Para quaisquer tipos PLC $\tau_1, \ldots, \tau_n$ com $n > 1$, é possível construir listas de tipos `(`τ_1`, ..., `τ_n`)`. O construtor de listas é o operador "mixfix" com multiaridade `(_, ..., _)`. Para todo $n > 0$, $i \in \{1, \ldots, n\}$ e tipos $\tau_1, \ldots, \tau_n$, há também um seletor de elementos pós-fixo `[`i`] : (`τ_1`, ..., `τ_n`) -> `τ_i que produz o i-ésimo elemento da lista de entrada.

Tipos *Function* Funções que tomam como entrada um tipo τ_1 e produzem uma saída do tipo τ_2 possuem tipo τ_1 `->` τ_2. O operator *arrow* `->` é associativo à direita.

Tipos *Sequence* Para qualquer tipo PLC τ é possível construir sequências de tipo `[`τ`]`. Perceba que isto significa que é possível construir sequências de sequências, sequências de listas, e

assim vai. Os operadores pré-definidos, e polimórficos, que lidam com valores sequências são listados abaixo:

- `[]` : `[`τ`]`, para qualquer tipo τ. A sequência vazia de elementos de tipo τ.
- `::` : `(`τ`, [`τ`]) -> [`τ`]`, para qualquer tipo τ. O operador infixo, associativo à direita, para construção de sequências.
- `ise` : `[`τ`] -> Bool`, para qualquer tipo τ. Produz `true` se a sequência de entrada é vazia e `false` caso contrário.
- `hd` : `[`τ`] ->` τ, para qualquer tipo τ. Produz a cabeça da sequência de entrada se a entrada não é vazia, e produz uma exceção caso contrário.
- `tl` : `[`τ`] -> [`τ`]`, para qualquer tipo τ. Produz a calda da sequência de entrada se a entrada não é vazia, e produz uma exceção caso contrário.

Tipos de igualdade Estes são os tipos sem ocorrências de `->` neles. Eles são definidos indutivamente tais que: (i) `Bool`, `Int`, e `Nil` são tipos de igualdade; (ii) se τ é um tipo de igualdade, então `[`τ`]` também é; (iii) se $\tau_1, \ldots, \tau_n$, com $n > 1$, são tipos de igualdade, então `(`τ_1`, ..., `τ_n`)` também é; (iv) nada mais é um tipo de igualdade. Lembre que `=` e `!=` se aplicam apenas a valores de um tipo de igualdade.

Outro operador infixo pré-definido é `;` que tem tipo `(`τ_1`, `τ_2`) ->` τ_2 para quaisquer tipos τ_1 e τ_2. Ele funciona avaliando, em ordem, seus argumentos e produzindo o valor de seu segundo argumento. Este operador é mais útil quando o primeiro argumento contém aplicações da função `print`.

3 Sintaxe concreta

A sintaxe concreta de PLC é descrita pela gramática abaixo, em que símbolos não-terminais são escritos entre chaves angulares e o símbolo inicial é `<prog>`.

3.1 Regras de produção

```
<prog> ::= <expr> | <decl> ; <prog>

<decl> ::=
    var <name> = <expr>
  | fun <name> <args> = <expr>
  | fun rec <name> <args> : <type> = <expr>

<expr> ::=
    <atomic expr>                                  atomic expression
  | <app expr>                                     function application
  | if <expr> then <expr> else <expr>              conditional expression
  | match <expr> with <matchexpr>                  match expression
  | ! <expr>                                       unary operator application
  | - <expr>
  | hd <expr>
  | tl <expr>
  | ise <expr>
```

```
  | print <expr>
  | <expr> && <expr>                                binary operator application
  | <expr> + <expr>
  | <expr> - <expr>
  | <expr> * <expr>
  | <expr> / <expr>
  | <expr> = <expr>
  | <expr> != <expr>
  | <expr> < <expr>
  | <expr> <= <expr>
  | <expr> :: <expr>
  | <expr> ; <expr>
  | <expr> [ <nat> ]

<atomic expr> ::=
    <const>                                         constant literal
  | <name>                                          function, variable or parameter name
  | { <prog> }                                      local scope block
  | ( <expr> )                                      parenthesized expression
  | ( <comps> )                                     list
  | fn <args> => <expr> end                         anonymous function

<app expr> ::=                                      function application
    <atomic expr> <atomic expr>
  | <app expr> <atomic expr>

<const> ::=
    true | false
  | <nat>                                           numerals
  | ( )                                             nil value
  | ( <type> [ ] )                                  type-annotated empty sequence

<comps> ::=                                         list components
    <expr> , <expr>
  | <expr> , <comps>

<matchexpr> ::=                                     match cases
    end
  | '|' <condexpr> -> <expr> <matchexpr>

<condexpr> ::=                                      values to be matched against
    <expr>
  | '_'

<args> ::=                                          function arguments
    ( )
  | ( <params> )

<params> ::=
    <typed var>
  | <typed var> , <params>
```

```
<typed var> ::= <type> <name>                    typed variable

<type> ::=
     <atomic type>
   | ( <types> )                                 list type
   | [ <type> ]                                  sequence type
   | <type> -> <type>                            function type

<atomic type> ::=
     Nil                                         Nil type
   | Bool                                        Boolean type
   | Int                                         integer type
   | ( <type> )

<types> ::=
     <type> , <type>
   | <type> , <types>
```

3.2 Regras léxicas

O não-terminal `<name>` é um token definido pela expressão regular

```
['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '_' '0'-'9']*
```

excluindo os seguintes nomes, que são palavras-chave:

```
  Bool else end false fn fun hd if Int ise
match Nil print rec then tl true var with __
```

O não-terminal `<nat>` é um token definido pela expressão regular `[0-9]+`.

3.3 Precedência de operadores

Os vários operadores e palavras-chave possuem a seguinte precedência, da menor para a maior, com operadores na mesma linha tendo a mesma precedência.

`; ->`	(associativo à direita)
`if`	(não-associativo)
`else`	(associativo à esquerda)
`&&`	(associativo à esquerda)
`= !=`	(associativo à esquerda)
`< <=`	(associativo à esquerda)
`::`	(associativo à direita)
`+ -`	(associativo à esquerda)
`* /`	(associativo à esquerda)
`not hd tl ise print` f	(não-associativo)
`[`	(associativo à esquerda)

em que f é um nome de uma função definida pelo usuário.

```
type plcType =
    | IntT                                  //  Int
    | BoolT                                 //  Bool
    | FunT  of plcType * plcType            //  type -> type
    | ListT of plcType list                 //  Nil and (type, ..., type)
    | SeqT  of plcType                      // [type]

type expr =
    | ConI   of int                                     // integer constants
    | ConB   of bool                                    // Boolean constants
    | ESeq   of plcType                                 // typed empty sequence constant
    | Var    of string                                  // variables
    | Let    of string * expr * expr                    // expressions with variable declaration
    | Letrec of string * plcType * string               // expressions with recursive function decl.
                * plcType * expr * expr
    | Prim1  of string * expr                           // unary operators
    | Prim2  of string * expr * expr                    // binary operators
    | If     of expr * expr * expr                      // if construct
    | Match  of expr * (expr option * expr) list        // match construct
    | Call   of expr * expr                             // function application
    | List   of expr list                               // Nil Constant / list construction
    | Item   of int * expr                              // List selector application
    | Anon   of plcType * string * expr                 // anonymous function

type plcVal =
    | BoolV of bool                                     // Booleans
    | IntV  of int                                      // integers
    | ListV of plcVal list                              // lists
    | SeqV  of plcVal list                              // sequences
    | Clos  of string * string * expr * plcVal env      // closures
```

Figura 4: Sintaxe abstrata para programas PLC.

4 Sintaxe abstrata

Por uniformidade, e para facilitar o projeto, a sintaxe abstrata (AST) de PLC, para tipos, expressões e valores, deverá seguir os tipos de dados algébricos na Figura 4. Você deve usar esta sintaxe abstrata em sua implementação. A árvore de sintaxe abstrata também está disponível no módulo `Absyn`.

4.1 Tipos

Termos em SML de tipo `plcType` são usados para codificar tipos PLC. Alguns exemplos de códigos PLC e sua respectiva sintaxe abstrata:

Sintaxe concreta	Sintaxe abstrata
`Int`	`IntT`
`Nil`	`ListT []`
`Int -> Int`	`FunT (IntT, IntT)`
`Int -> Int -> Bool`	`FunT (IntT, FunT (IntT, BooT))`
`(Int -> Int) -> Bool`	`FunT (FunT (IntT, IntT), BooT)`
`(Int, Int, Bool)`	`ListT [IntT; IntT; BooT]`
`(Int, Int) -> Bool`	`FunT (ListT [IntT; IntT], BooT)`
`[Int]`	`SeqT IntT`
`[(Bool,Int)]`	`SeqT (List [BooT; IntT])`

Perceba que o construtor `ListT` de `plcType` é usado para representar tanto o tipo `Nil`, com `ListT []`, como tipo tipos de listas, com `ListT [`τ_1`; ...; `τ_n`]`, para $n > 1$.

4.2 Expressões

Termos em SML de tipo `exp` são usados para codificar programas e expressões PLC. Alguns exemplos de códigos PLC e sua respectiva sintaxe abstrata:

Sintaxe concreta	Sintaxe abstrata
`15`	`ConI 15`
`true`	`ConB true`
`()`	`List []`
`(6, false)`	`List [ConI 6; ConB false]`
`(6, false)[1]`	`Item (1, List [ConI 6; ConB false])`
`([Bool] [])`	`ESeq (SeqT BoolT)`
`print x; true`	`Prim2 (";", Prim1 ("print", Var "x"), ConB true)`
`3::7::t`	`Prim2 ("::", ConI 3, Prim2 ("::", ConI 7, Var "t"))`
`fn (Int x) => -x end`	`Anon (IntT, "x", Prim1(-", Var "x"))`
`var x = 9; x + 1`	`Let ("x", ConI 9, Prim2 ("+", Var "x", ConI 1))`
`fun f(Int x) = x; f(1)`	`Let ("f", Anon (IntT, "x", Var "x"), Call ("f", ConI 1))`

`match x with`
`| 0 -> 1`
`| _ -> -1`
`end`
	`Match (Var "x",`
	`[(Some (ConI 0), ConI 1);`
	`(None, Prim1 (-",ConI 1))])`

`fun rec f(Int n) =`
`if n <= 0 then 0`
`else n + f(n-1) ;`
`f(5)`
	`Letrec ("f", IntT, "n", IntT,`
	`If (Prim2 («=", Var "n", ConI 0), ConI 0,`
	`Prim2 ("+", Var "n", Call (Var "f", ...))),`
	`Call (Var "f", ConI 5))`

O construtor `List`, que recebe uma lista de expressões como argumentos, é usado para representar expressões de listas. Ele também é usado para representar a expressão `Nil`, isto é, `()`, como `List []`. Perceba que constante para sequência vazia, `ESeq`, carrega com ela o tipo da sequência, o que é necessário para checagem de tipos. Perceba também que `[i]`, representado pelo construtor `Item`, é tratado como um operador binário por conveniência; no entanto, seu segunda argumento, `i`, deve ser um numeral.

Funções anônimas da forma `fn (τ x) => e end` são representadas como `Anon (τ', x, e')`, em que τ' é a representação em sintaxe abstrata do tipo τ e e' é a representação em sintaxe abstrata para o corpo da função e.

Otherwise, the conversion to abstract syntax should be generally done as in Hw6. In particular, multi-argument functions should also be converted as in Hw6, using nested `Let` expressions.

4.3 Values

F# terms of type `plcValue` are used to encode PLC values. The PLC interpreter is essentially a converter from `expr` terms to `plcValue` terms. Here are examples of such conversions.

	Expression
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 ("f", ConI 1))`

	Value
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")), [])` (in case of an empty environment)
11.	`IntV 10`
12.	`IntV 1`

Anonymous function expressions of the form `Anon (t, x, e)` should evaluate to the value `Clos (, x, e, env)` where *env* is the current environment.

With expressions of the form `Prim1("print", e)`, the interpreter should first evaluate e to some value v, convert v to a string representation in concrete syntax, and then print that string to the standard output followed by a new line character. For the string conversion, you can use the helper function `val2string : plcVal -> string` already provided in module `Absyn`.

What other well-typed PLC expressions should evaluate to should be clear from Hw6. If you are not clear about specific cases, please ask the instructors.

5 Parsing

Um parser para PLC deve ser especificado preenchendo os arquivos `PlcParser.lex` e `PlcParser.yacc`. O parser deve utilizar a sintaxe abstrata definida na Seção 4. As funções auxiliares definidas `PlcParserAux.sml` devem ser também completadas e utilizadas.

A função `makeFun` deve ser usada na regra de produção de funções recursivas, tomando, em ordem, o nome da função, a lista de parâmetros, o tipo de retorno, o corpo da função, e a expressão em que a definição da função é utilizada. Se a lista de parâmetros contém apenas um argumento, `makeFun` produz a AST

$$\texttt{Letfun (}f\texttt{, }x_1\texttt{, }t_1\texttt{, }e_1\texttt{, }t\texttt{, }e_2\texttt{)}$$

Se a lista de parâmetros é vazia, ela produz a AST `Letfun (`f`, `x`, ListT [], `e_1`, `t`, `e_2`)` que corresponde em sintaxe concreta à declaração `fun `f`(Nil `x`): `t` = `e_1`; `e_2.

Se a lista de parâmetros possui dois elementos, `makeFun` produz a AST

$$\texttt{Letfun (}f\texttt{, }x\texttt{, }t\texttt{, }e_1'\texttt{, }t\texttt{, }e_2\texttt{)}$$

com t = `ListT [`t_1`; `t_2`]` e e_1' = `Let(`x_1`, Item(1, `x`), Let(`x_2`, Item(2, `x`), `e_1`))`, que corresponde em sintaxe concreta à declaração `fun `f`((`t_1`,`t_2`) `x`): `t` = {var `x_1` = `x`[1]; var `x_2` = `x`[2]; `e_1`}; `e_2.

O comportamento para $n > 2$ é similar mas com mais construtores `Let` aninhados.

Em todos os casos, x acima é a string `"$list"`. A escolha deste nome para a variável de entrada de f é arbitrária mas começa com `$` para garantir que o nome não coincide com qualquer variável no programa PLC.

Quando $n > 1$, `makeFun` usa as funções auxiliares `makeType` e `makeFunAux`, que produzem, respectivamente, t e e_1' acima. Estas duas funções tem implementações incorretas no esqueleto dado, que devem ser substituídas por implementações com o comportamento acima.

Para gerar o lexer e o parser a partir dos arquivos `PlcParser.lex` e `PlcParser.yacc`, uma vez que estejam completos, utilize os programas ml-lex e ml-yacc. Por exemplo, com os comandos:

```
$ ml-lex PlcLexer.lex

$ ml-yacc PlcParser.yacc
```

Estes comandos geraram os arquivos necessários para se transformar uma string na sintaxe concreta para uma AST. Alguns links úteis para estudar sobre ml-lex e ml-yacc:

- Manual oficial ml-lex: `https://www.smlnj.org/doc/ML-Lex/manual.html`
- Manual oficial ml-yac: `http://www.smlnj.org/doc/ML-Yacc/`
- *User's guide* completo: `http://rogerprice.org/ug/ug.pdf`

6 Implementação

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.

- `Environ`
  Este módulo define o tipo de um ambiente genérico e uma função de `lookup` para ele. Ele já é provido completo através do arquivo `project/Environ.sml` em `project.zip`. Você precisará de instanciações deste tipo e usará `lookup` no verificador de tipos e no interpretador.

- `Absyn`
  Este módulo define a sintaxe abstrata de PLC. Ele já é provido completo no arquivo `project/Absyn.sml` em `project.zip`. Ele contém uma função auxiliar `val2string` que pode ser usada para implementar `print`.

- `PlcParserAux`
  Este módulo define funções auxiliares para parsing. As implementações no arquivo `project/PlcParserAux.sm` estão incompletas e devem ser completadas por você.

- `PlcParser`
  Este módulo contem o parser para a linguagem PLC, nos arquivos `PlcParser.yacc.sig` e `PlcParser.yacc.sml`, a serem gerados automaticamente através do processo descrito na Seção 5.

- `Lexer`
  Este módulo contem o lexer para a linguagem PLC, no arquivo `PlcLexer.lex.sml`, a ser gerado automaticamente através do processo descrito na Seção 5. O lexer pode prover suporte a comentários, que em PLC seguem o formato `(* ... *)`, mas isto não é obrigatório.

- `Parse`
  Este módulo, provido no arquivo `Parse.sml`, define a função `fromString`, que faz parsing de um programa PLC a partir de uma string, e a função `fromFile`, que faz parsing de um programa PLC em um arquivo de texto. Você pode usar essas funções para testar seu parser, seguindo o arquivo `testParser.sml`.

- `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.

7 Detalhes sobre entrega das repostas

- O projeto deverá ser entregue em duas partes:

  1. Uma solução parcial do projeto até 23:59 de 12/03/2021. Esta solução parcial deve conter lexer e parser corretos e completos para a linguagem PLC.
     Um conjunto de casos de testes é provido em `testParserCases.sml` para ajudar neste processo. Perceba que os casos de teste para o parser não necessariamente correspondem a programas corretamente tipados que que tenham uma avaliação que faça sentido.
     Não realizar esta entrega ou prover um parser não executável ou que falhe mais do 5 dos testes dados gerará perda significativa de pontos. Caso contrário a nota final será de acordo com a avaliação da solução final.
  2. A solução final deverá ser submetida até 23:59 de 19/03/2021. Esta solução deve conter uma implementação correta e completa de todos os módulos descritos acima.
     É parte do seu trabalho criar casos de teste para o verificador de tipos e o interpretador.
  3. Ambas as submissões devem ser feitas uma vez por dupla através de arquivo `project.zip` contendo os módulos descritos acima (de acordo com a solução ser a parcial ou a final).

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; ρ 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 τ e é de outra forma idêntico a ρ; $type(e,\ \rho) = t$ abrevia a sentença: "o tipo da expressão e no ambiente ρ é τ."

As regras abaixo definem o sistema de tipos para PLC. Uma expressão e é bem-tipada e tem tipo τ em ambiente de tipagem ρ 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. $type($`()`$,\ \rho) =$ `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 $type(e_2, \rho) = t_2$