Prelude AdvPL Documentation

Release latest

April 27, 2015

CAPÍTULO 1

Sobre o Prelude AdvPL

O Prelude AdvPL é uma biblioteca para AdvPL, escrita pela `NG Informática`, altamente compatível com Harbour, e que visa a implementação de conceitos funcionais na linguagem e melhoria na qualidade de código. A biblioteca implementa uma sintaxe expressiva e permite uma maior abstração do domínio da aplicação, evitando *boilerplate* e código desnecessário, além de ter um cuidado especial com a performance.

Através do pré-processador são implementadas features extras em cima da linguagem, mas sem quebrar qualquer compatibilidade com código legado. O Prelude AdvPL implementa três tipos de funções: **Prelude Functions**, **Cast Functions** e **Validate Functions**. **Prelude Functions** são comumente utilizadas para manipulação de dados, como listas, numéricos, strings ou blocos de código, mas de uma forma muito mais expressiva. **Cast Functions** servem para trabalhar em cima do sistema dinâmico de tipos da linguagem. Elas permitem a redefinição explícita de tipos de valores em casos específicos. **Validate Functions** servem para validar a coerência e consistência de dados. Algumas de suas validações aceitáveis são `CPF`, `CNPJ`, `Name`, `Even`, `Odd`, entre outros.

Abstrações Sintáticas:

1.1 Intervalos Numéricos

O Prelude AdvPL implementa uma sintaxe própria para intervalos numéricos, evitando que sejam criadas instruções desnecessárias para tal, como loops.

1.1.1 Definição Formal

```
range ::= @{ <expr> [, <expr> ] .. <expr> } ;
```

1.1.2 Exemplos

Temos duas opções para intervalos numéricos: `@{ x .. y }`, sendo `x` o valor inicial e `y` o valor final; e `@{ x, y .. z }`, sendo `x` o valor inicial, `z` o valor final, seguindo num intervalo de `y - x`. Exemplos:

```
#include "prelude.ch"

Function TestInterval()
   // aRange recebe { 1, 2, 3, 4, 5 }
   Local aRange := @{ 1 .. 5 }
   // aStepRange recebe { 1, 3, 5, 7, 9, 11, 13, 15 }
   Local aStepRange := @{ 1, 3 .. 15 }
   Return 0
```

1.2 Sintaxe Padrão

Abstrações sintáticas são implementadas para tornar o código mais compreensível. Chamadas abstraídas podem receber até **3** parâmetros e ser aninhadas dentro de outras chamadas.

1.2.1 Definição Formal

```
prelude-call ::= @<ident> { <expr> [, <expr> [, <expr> ] ] }
```

1.2.2 Exemplos

Vamos imaginar que queremos realizar a seguinte operação:

- **Criar uma lista** com os elementos de 1 a 50; `@{}`
- Obter os **20 primeiros elementos** desta; `@Take`
- **Mapear** e dobrar o valor de cada elemento obtido; `@Map`
- Mostrar ao usuário **cada** elemento em um `Alert`. `@Each`

Devemos fazer:

```
#include "prelude.ch"

Function Prelude()
   @Each { Fun (X) -> Alert( X ), ;
      @Map { Fun (Y) -> Y * 2, ;
         @Take { 20, @{ 1 .. 50 } } } }
   Return 0
```

1.3 Blocos de Primeira Classe

O Prelude AdvPL também abstrai blocos de primeira classe em forma de funções de primeira classe, utilizando uma sintaxe similar a `OCaml` ou `Livescript`. Blocos são, por padrão, definidos em AdvPL como `{ |Param| Expression }`. A única função dessa abstração é aplicar um *syntactic sugar* para `Fun ( Param ) -> Expression`.

Nota: `Lambda ( Param ) -> Expression` também é válido.

1.3.1 Definição Formal

```
fun ::= Fun ( [ <ident> [, <ident> [, <ident> ] ] ] ) -> <expr>
lambda ::= Lambda ( [ <ident> [, <ident> [, <ident> ] ] ] ): <expr>
```

1.3.2 Exemplos

Vamos realizar alguns testes com operações simples:

```
#include "prelude.ch"

Function TestBlocks()
   Local bAdd      := Fun ( X, Y ) -> X + Y
   Local bToString := Lambda ( X ): Str( X )

   Alert( Eval( bToString, Eval( bAdd, 10, 20 ) ) ) // => "30"
   Return 0
```

1.4 Sintaxe para Casting

Implementamos uma sintaxe especial para facilitar a transição de tipos de dados. Por padrão, os tipos suportados atualmente são `Str`, `Int` e `Number`.

De maneira similar a *generics* em C#, os tipos são aplicados dentro de *tags*.

1.4.1 Definição Formal

```
casting ::= @Cast\< <type> \> <expr>
```

1.4.2 Exemplos

```
#include "prelude.ch"

Function TestCasting()
   Local nStrToNumber := @Cast<Int> "18" ; // => 18
       , cNumberToStr := @Cast<Str> 19   ; // => "19"
       , nFloatToInt  := @Cast<Num> 15.4   // => 15
   Return 0
```

1.5 Sintaxe Validate

Validate atua de maneira muito similar a **Cast**, recebendo, como parâmetro entre as tags, o "**tipo**" a ser validado. As opções atualmente disponíveis são `CPF`, `CNPJ`, `Name`, `Even`, `Odd`, `Positive`, `Negative`, `CEP` e `Email`.

1.5.1 Definição Formal

```
validate ::= @Validate\< <type> \> <expr>
```

1.5.2 Exemplos

```
#include "prelude.ch"

Function ValidateTests()
   @Validate<Email> "marcelocamargo@linuxmail.org" // => .T.
   @Validate<CNPJ>  "62645927000136"               // => .T.
   @Validate<CPF>   "45896587466"                  // => .F.
```

```
   @Validate<Name>  "Marcelo Camargo"              // => .T.
   Return 0
```

1.6 Sintaxe Do ... In

Recebe dois identificadores de funções, aplica *function-composition*, isto é, une as duas funções e aplica à expressão recebida.

1.6.1 Definição Formal

1.6.2 Exemplos

```
#include "prelude.ch"

Function TestDo()
   Do Alert >>= Str In 68
   Return
```

1.7 Aplicação de Bloco

Para facilitar a abstração, há aplicação de blocos (ou funções anônimas) para funções do Prelude AdvPL que recebam exatamente **2** argumentos. Por padrão, o argumento antes do operador é o valor ao qual o bloco será aplicado e o elemento posterior ao operador, o operando direito, será o bloco que será aplicado. É possível usar essa abstação para outras coisas, que não a aplicação de blocos, contudo, não é aconselhável.

1.7.1 Definição Formal

```
block-app ::= @<ident> <expr> ::= <expr>
```

1.7.2 Exemplos

```
#include "prelude.ch"

Function BlockApp()
   Local aList  := @{ 1 .. 10 } ;
       , bPrint := Fun ( X ) -> Alert( X )

   @Each aList ::= bPrint
   Return
```

1.8 Sintaxe de Retenção

Exatamente o oposto à aplicação de bloco. Recebe dois operandos entre `Of`. O primeiro operando será o valor que será retido e o segundo será o elemento que terá seu valor retido. Pode ser usada para outras coisas além de retenção, no entando, não é aconselhável por puras questões de legibilidade.

1.8.1 Definição Formal

```
of-op ::= @<ident> <expr> Of <expr>
```

1.8.2 Exemplos

```
#include "prelude.ch"

Function TestOf()
   Local aList := @{ 1 .. 50 }

   @Take 5 Of aList // => { 1, 2, 3, 4, 5 }
   Return
```

1.9 Sintaxe On

Aplicação de função. O operando à esquerda corresponde ao primeiro parâmetro, o operando central corresponde à função e o operando à direita corresponde ao segundo parâmetro.

1.9.1 Definição Formal

```
on ::= On <ident> <ident> <expr>
```

1.9.2 Exemplos

```
#include "prelude.ch"

Function TestOn()
   Local aRotina := { }

   On aRotina aAdd { STR001, STR002, STR003, STR004 }

   Return
```

1.10 Sintaxe Just

Se o último operando à direita for um valor diferente de `Nil`, o operando à esquerda recebe-o.

1.10.1 Definição Formal

```
just ::= Just <ident> -> <ident> Receives <expr>
```

1.10.2 Exemplos

```
#include "prelude.ch"

Function TestJust( /* all optional */ nX, nY, nZ )
   Just M->X Receives nX
   Just M->Y Receives nY
   Just M->Z Receives nZ
   Return
```

1.11 Keywords e Operadores

Algumas palavras-chave e operadores foram adicionados como *aliases* para as já existentes, entre eles:

1.11.1 Keywords

Keyword	Correspondente	Token
Let	Local	T_LOCAL
Extern	Private	T_PRIVATE
True	.T.	T_TRUE
False	.F.	T_FALSE

1.11.2 Operadores

Operador	Correspondente	Token
<-	:=	T_ASSIGNMENT
Is	==	T_STRICTEQUALS
Like	=	T_EQUALS
Or	.Or.	T_OR
And	.And.	T_AND

1.11.3 Operador Ternário

Foi implementado **If** como expressão, retornando valores, em sua equivalência a `IIf`, onde podemos fazer, por exemplo: `Let cName <- If 1 < 2 Then "Foo" Else "Bar"`.

1.12 Gerenciador de Pacotes

Foi implementando um *package manager*, responsável, no momento, apenas por identificação e versionamento de um determinado programa, baseando-se na sintaxe `Package <string>(Version: <number>) Where`, o que definirá, estaticamente, `Package` e `_NVERSAO`, para uso interno.

Tipos de Funções:

1.13 Cast Functions

Cast Functions são funções padrão de conversão explícita de tipos.

@Cast<Int>
Retorna um valor numérico inteiro a partir de um valor numérico com parte inteira e decimal informado como parâmetro, desconsiderando todos os dígitos à direta do ponto decimal.

```
@Cast<Int> 87.2 // => 87
```

@Cast<Num>
Converte uma sequência de caracteres que contêm dígitos em um valor numérico.

```
@Cast<Num> "9587" // => 9587
```

@Cast<Str>
A partir de um numérico, esta função retorna uma string formatada, inserindo espaços (" ") à esquerda e/ou o símbolo decimal (".") em suas casas, de acordo com a informação do parâmetro.

```
@Cast<Str> 987 // "987"
```

1.14 Prelude Functions

Prelude Functions são as mais importantes funções do Prelude AdvPL. Elas são basicamente responsáveis por toda manipulação de dados que ocorre. Estão definidas no núcleo da biblioteca e possuem uma sintaxe própria para uso.

Bool @All { Block, Array }
Retorna verdadeiro se todos os itens na lista são verdadeiros quando aplicados ao teste.

Bool @AndList { Array }
Retorna falso se qualquer item na lista é falso. Caso contrário, retorna verdadeiro.

Bool @Any { Block, Array }
Retorna verdadeiro se quaisquer dos itens da lista são verdadeiros. Caso contrário, retorna falso.

Array @Compact { Array }
Retorna uma lista contendo apenas os valores verdadeiros da lista recebida.

Array @Concat { Array }
Concatena uma lista de listas juntas, a um nível **1** de imersão.

Array @Each { Block, Array }
Aplica um bloco para cada elemento na lista e retorna a lista original. Retorna a própria lista, comumente utilizada para efeitos colaterais.

Array @EachIndex { Block, Array }
Similar, quase o mesmo comportamento que a função `@Each`, mas aplica-se com dois parâmetros, o elemento em questão e o índice deste. Isso é útil quando realmente precisa-se saber a posição em que o elemento se encontra.

Number @ElemIndex { Mixed, Array }
Retorna o índice da primeira ocorrência do elemento em questão. Retorna nulo caso nenhum elemento como o passado seja encontrado.

Array @ElemIndices { Mixed, Array }
Retorna uma array contendo todos os índices que correspondem ao elemento em questão. Retorna um array vazio caso nenhum elemento seja encontrado.

Bool @Even { Number }
Retorna se o número dado é par.

Array @Explode { String }
Transforma uma string em um array de caracteres.

`Array @Filter { Block, Array }`
Retorna uma nova lista composta de todos os itens que passam no teste do bloco dado.

`Mixed @Find { Block, Array }`
Retorna o primeiro item que passa no teste do bloco dado. Retorna nula caso todos os elementos falhem na aplicação do teste.

`Number @FindIndex { Block, Array }`
Retorna o índice do primeiro elemento a passar no predicado. Retorna nulo caso nenhum dos elementos passe no teste em questão.

`Array @FindIndices { Block, Array }`
Retorna um array contendo os índices dos elementos a passarem no predicado. Caso nenhum elemento passe no teste, retorna um array vazio.

`Number @GCD { Number, Number }`
Retorna um máximo denominador comum de dois números.

`Mixed @Head { Array }`
Retorna o primeiro item de um array.

`Mixed @Id { Mixed }`
Retorna o próprio elemento. Útil como placeholder.

`Array @Initial { Array }`
Retorna todos os itens de uma lista, exceto o último.

`Number @LCM { Number, Number }`
Retorna o mínimo múltiplo comum de dois números.

`Array @Map { Block, Array }`
Aplica um bloco para cada item em uma lista e produz uma nova lista com os resultados. O tamanho da lista retornada é igual ao da lista dada.

`Array @MapIndex { Block, Array }`
Quase o mesmo que `@Map`, contudo, aplica-se com dois parâmetros, ambos o item e o índice. É útil quando realmente precisa-se conhecer a posição onde o elemento se encontra. Retorna a lista modificada.

`Mixed @Maximum { Array }`
Percorre uma lista de itens comparáveis e retorna o maior deles.

`Number @Mean { Array }`
Retorna a média dos valores de uma lista.

`Mixed @Minimum { Array }`
Percorre uma lista de itens comparáveis e retorna o menos deles.

`Number @Negate { Number }`
A negação de um dado número.

`Bool @Odd { Number }`
Retorna se o dado número é ímpar.

`Bool @OrList { Array }`
Retorna verdadeiro se qualquer item da lista é verdadeiro. Caso contrário, retorna falso.

`Array @Partition { Block, Array }`
Equivalente a `{ @Filter { f, xs }, @Reject { f, xs } }`, mas mais eficiente, utilizando apenas um loop.

`Number @Pi { }`
Retorna os 16 primeiros números de pi.

`Number @Product { Array }`
Obtém o produto de todos os itens na lista.

`Array @Range { Number, Number }`
Recebe dois valores inteiros e retorna um array dentro daquele intervalo.

`Number @Recipe { Number }`
Retorna 1 divido pelo valor dado.

`Array @Reject { Block, Array }`
Como `@Filter`, mas a nova lista é composta de todos os itens que **falham** no teste da função.

`Array @Reverse { Array }`
Reverte e retorna uma lista.

`Number @SigNum { Number }`
Toma um número e retorna -1, 0 ou 1, isto dependendo do operador unário em questão do número (-, 0, +).

`Array @Slice { Number, Number, Array }`
Corta uma lista e retorna o pedaço dela, recebendo os valores inicial e final e a lista a ser cortada.

`Array @Sort { Array }`
Organiza em ordem ascendente uma lista. Não modifica a entrada.

`Array @StepRange { Number, Number, Number }`
Recebe três inteiros e retorno um array indo do primeiro número até o segundo e seguindo um intervalo do terceiro menos o primeiro.

`Number @Sum { Array }`
Soma todos os itens de uma lista e os retorna.

`Array @Split { String, String }`
Recebe um delimitador e uma string. Retorna um array separando a string pelo dado delimitador.

`String @Stringify { Array }`
Transforma um array de caracteres em uma string.

`Array @Tail { Array }`
Retorna todos, menos o último item de uma lista.

`Array @Take { Number, Array }`
Obtém os primeiros `n` elementos de uma lista.

`Array @TakeWhile { Block, Array }`
Obtém os primeiros elementos da lista até que algum deles falhe no teste do dado bloco.

`Number @Tau { }`
Retorna o valor de **tau**, isto é, 2 * pi.

`Array @Zip { Array, Array }`
Une elementos de duas listas de mesmo tamanho. Útil para simular dicionários.

`Array @ZipWith { Block, Array, Array }`
Une elementos de duas listas de mesmo tamanho aplicando um bloco a elas. Retorna um único array contendo o resultado.

1.15 Validate Functions

Validate Functions são funções gerais de validação. Todas **devem** ter como retorno um valor *Booleano*.

@Validate<CEP>
Retorna a validade de um CEP no formato "99999-999".

```
@Validate<CEP> "88590-000" // .T.
```

@Validate<CNPJ>
Retorna a validade de um CNPJ (apenas números).

```
@Validate<CNPJ> "30226467000115" // .T.
```

@Validate<CPF>
Retorna a validade de um CPF (apenas números).

```
@Validate<CPF> "15985496883" // .F.
```

@Validate<Email>
Valida um e-mail segundo os padrões atuais.

```
@Validate<Email> "marcelocamargo@linuxmail.org" // .T.
```

@Validate<Even>
Valida a paridade de um número.

```
@Validate<Even> 15 // .F.
```

@Validate<Name>
Verifica se o nome é válido, composto apenas por caracteres latinos.

```
@Validate<Name> "Marcelo Haskell Camargo" // .T.
```

@Validate<Negative>
Validação de negatividade numeral.

```
@Validate<Negative> 15 // .F.
```

@Validate<Number>
Retorna se determinado valor é válido como numeral.

```
@Validate<Number> "15" // .T.
```

@Validate<Odd>
Retorna se um número é ímpar.

```
@Validate<Odd> 15 // .T.
```

@Validate<Positive>
Validação de positividade numeral.

```
@Validate<Positive> 15 // .T.
```

Informações?

Marcelo Camargo <marcelo.camargo@ngi.com.br> <marcelocamargo@linuxmail.org>

Última modificação por **Marcelo Camargo** às 15:02 do dia 08/04/2015.