# Prelude AdvPL

NG Informática – 2015
Marcelo Camargo <marcelo.camargo@ngi.com.br>

# Sobre o Prelude AdvPL

O Prelude AdvPL é uma biblioteca para AdvPL altamente compatível com Harbour 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.

# Sobre o Prelude AdvPL

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

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

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

Validate Functions servem para validar a coerência e consistência de dados. Algumas de suas validações aceitáveis são CPF, CNPJ, Name, Even, Odd, entre outros.

# Intervalos Numéricos

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

# Intervalos Numéricos

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.

# Intervalos Numéricos (Exemplos)

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

# 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.

# Sintaxe Padrão (Exemplo)

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

# Sintaxe Padrão (Exemplo)

```
1 #include "prelude.ch"
2
3 Function Prelude()
4     @Each { Fun (X) -> Alert( X ), ;
5         @Map { Fun (Y) -> Y * 2, ;
6             @Take { 20, @{ 1 .. 50 } } } } }
7     Return 0
```

# 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 de Primeira Classe

Blocos são, por padrão, definidos em AdvPL como { |Param| Expression }. A única função dessa abstração é aplicar um *syntactic sugar* para:

Fun ( Param ) -> Expression

# Blocos de Primeira Classe (Exemplos)

## Vamos realizar alguns testes com operações simples:

```
1 #include "prelude.ch"
2
3 Function TestBlocks()
4    Local bAdd      := Fun ( X, Y ) -> X + Y
5    Local bToString := Fun ( X ) -> Str( X )
6
7    Alert( Eval( bToString, Eval( bAdd, 10, 20 ) ) ) // => "30"
8    Return 0
```

# 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.

# Sintaxe para Casting (Exemplos)

```
1 #include "prelude.ch"
2
3 Function TestCasting()
4    Local nStrToNumber := @Cast<Int> "18" ; // => 18
5        , cNumberToStr := @Cast<Str> 19   ; // => "19"
6        , nFloatToInt  := @Cast<Num> 15.4   // => 15
7    Return 0
```

# 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.

# Sintaxe Validate (Exemplos)

```
1 #include "prelude.ch"
2
3 Function ValidateTests()
4    @Validate<Email> "marcelocamargo@linuxmail.org" // => .T.
5    @Validate<CNPJ>  "62645927000136"               // => .T.
6    @Validate<CPF>   "45896587466"                  // => .F.
7    @Validate<Name>  "Marcelo Camargo"              // => .T.
8    Return 0
```

# Sintaxe Do .. In

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

# Sintaxe Do .. In (Exemplos)

```
1 #include "prelude.ch"
2
3 Function TestDo()
4     Do Alert >>= Str In 68
5     Return
```

# Aplicação de Bloco

Para facilitar a abstração, há aplicação de blocos (ou funções anônimas) para funções do Prelude AdvPL que recebam exatamente 2 argumentos.

# Aplicação de Bloco

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.

# Aplicação de Bloco (Exemplos)

```
1 #include "prelude.ch"
2
3 Function BlockApp()
4    Local aList  := @{ 1 .. 10 } ;
5         , bPrint := Fun ( X ) -> Alert( X )
6
7    @Each aList ::= bPrint
8    Return
```

# 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.

# Sintaxe de Retenção (Exemplos)

```
1 #include "prelude.ch"
2
3 Function TestOf()
4    Local aList := @{ 1 .. 50 }
5
6    @Take 5 Of aList // => { 1, 2, 3, 4, 5 }
7    Return
```

# 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.

# Sintaxe On (Exemplo)

```
1 #include "prelude.ch"
2
3 Function TestOn()
4    Local aRotina := { }
5
6    On aRotina aAdd { STR001, STR002, STR003, STR004 }
7
8    Return
```

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

# Sintaxe Just (Exemplo)

```
1 #include "prelude.ch"
2
3 Function TestJust( /* all optional */ nX, nY, nZ )
4    Just M->X Receives nX
5    Just M->Y Receives nY
6    Just M->Z Receives nZ
7    Return
```

# Documentação disponível em:

http://prelude.readthedocs.org/