Universidade Federal de Pernambuco – UFPE
Centro de Informática – CIn

Deville
Descrição e Implementação

Arthur Elihimas
Carlos Maciel
Itagiba Leão
João Paulo Fernandes
Severino Barros
Tiago Lemos

Recife, Junho de 2008

Deville
Descrição e Implementação

Arthur Elihimas
Carlos Maciel
Itagiba Leão
João Paulo Fernandes
Severino Barros
Tiago Lemos

{afce, cavmj, iccl, jpfb, sjbj, tlam}

Resumo

Este trabalho é parte das atividades da disciplina Paradigmas de Linguagens Computacionais – PLC (IF686), cadeira do ciclo básico dos cursos de Ciência e Engenharia da Computação do Centro de Informática (CIn) – Universidade Federal de Pernambuco (UFPE). Este documento descreve a especificação da linguagem *Deville* em termos de sua sintaxe e semântica, apresentadas através de uma abordagem informal. Esse trabalho é acompanhado pela construção de um interpretador da linguagem implementado em Haskell. A linguagem *Deville* ganhou este nome como forma de prestarmos uma pequena homenagem ao intrépido dançarino Damon "El Professor" Deville, cuja complexidade dos passos faz o impossível parecer trivial.

Índice

1 - Introdução

Neste documento está descrita a linguagem que foi desenvolvida durante o projeto da disciplina: *Deville*. Nele são apresentados aspectos relativos à sua sintaxe e semântica e alguns exemplos que mostram o passo-a-passo para a construção das estruturas que comporão os programas criados. Também serão mostrados exemplos completos de programas implementados em *Deville* e a forma como um interpretador para a linguagem foi projetado.

Este documento tem como objetivo servir como orientação para que qualquer programador possa, de posse do mesmo, construir um interpretador ou compilador para Deville. Para tanto algumas explicações técnicas sobre a linguagem também serão aqui abordadas.

2 - Sobre *Deville*

Deville é uma linguagem curta e simples que pertence ao paradigma imperativo, tem influência das linguagens C e Pascal. Possui apenas duas abstrações: variáveis e procedimentos parametrizados ou recursivos, tais procedimentos não permitem nenhum retorno. As variáveis podem ser usadas como ponteiros. Variáveis e procedimentos precisam ser declarados antes que seu uso seja solicitado. As palavras-chave não são reservadas. A linguagem oferece suporte aos tipos: inteiros, booleanos e strings (seqüência de caracteres). A linguagem distingue entre letras maiúsculas e minúsculas.

Cada uma das construções de *Deville* será delimitada por parênteses, sendo a maior delas o programa. Tal programa será formado por um ou mais comandos também delimitados por parênteses. Nas próximas seções a linguagem é especificada informalmente sob os aspectos de sua semântica e sintaxe.

2.1 – Comandos

Os programas escritos em *Deville* serão formados por um conjunto de comandos encadeados. Tais estruturas formam a base dos programas e deverão ser declaradas explicitamente através das palavras reservadas que estão no quadro abaixo.

```
COMANDODECLARACAO          COMANDOATRIBUICAO          COMANDOWHILE
COMANDOIFTHENELSE          COMANDOFOR                 COMANDOSKIP
COMANDOREPEATUNTIL         COMANDOIO                  COMANDOLISTACOMANDOS
COMANDOCHAMADAPROCEDIMENTO
```

A formação completa dos comandos de *Deville* ficará mais bem detalhada nas subseções seguintes. O que se pode adiantar é que cada um será formado por um ou mais elementos, como nos exemplos do próximo quadro. Esses exemplos demonstram a formação de um comando de declaração de variável (*COMANDODECLARACAO*) e um comando de repetição (*COMANDOFOR*).

```
(COMANDODECLARACAO (DECVARIAVEL (VARIAVEL "var1" (VALOR (VALInteiro 10)))))
(COMANDOFOR (FOR (ATRIBUICAO "i" (VALORINT (VALInteiro 1))) 30
(COMANDOSKIP))
```

2.2 – Tipos

Deville é fortemente tipada, possuindo como tipos primitivos string de caracteres, inteiro e booleano: o tipo String descreve um elemento que possui zero ou mais caracteres agrupados, o inteiro representa números inteiros e boolean poderá assumir apenas valor verdadeiro ou falso.

Deville não permite conversão explicita ou implícita de tipos de elementos do programa, ou seja, o valor de uma variável que seja do tipo inteiro não poderá ser atribuído a uma variável de outro tipo. Isso é verificado também com os outros tipos definidos na linguagem

2.3 – Declarações

As declarações em *Deville* servem para indicar a criação de variáveis ponteiros e procedimentos. Alguns exemplos de declarações são indicados no quadro seguinte. Não será aprofundada aqui a discussão sobre as declarações, pois as próximas subseções elucidarão seu uso.

```
(COMANDODECLARACAO (DECVARIAVEL (PONTEIRO "ponteiro” String)))
(COMANDODECLARACAO (DECVARIAVEL (VARIAVEL "var"(VALOR(VALBooleano False))))
```

2.4 – Variáveis

Variáveis em *Deville* podem assumir qualquer valor de tipo (seção 2.2), sendo alguns exemplos de declaração descritos no código abaixo. Nos exemplos a palavra reservada DECVARIAVEL indica que será declarada um elemento variável, o texto entre aspas indica o seu identificador e a palavra VARIAVEL é componente da declaração para indicar explicitamente que o elemento declarado é uma variável e para diferenciar da declaração de ponteiros (vide seção 2.5). Em *Deville* após a declaração de uma variável é preciso indicar um valor que a mesma irá assumir. Nos exemplos isso é feito através da das palavras VALOR, VALInteiro, VALBooleano, VALString, que indica que um valor será declarado e seu tipo, respectivamente.

```
(DECVARIAVEL (VARIAVEL "var1" (VALOR (VALInteiro 10))))
(DECVARIAVEL (VARIAVEL "var2" (VALOR (VALBooleano True))))
(DECVARIAVEL (VARIAVEL "var3" (VALOR (VALString “declaracao string”))))
```

Todas as variáveis declaradas possuem escopo global, ou seja, será possível acessar qualquer elemento previamente declarado dentro de um trecho de código.

2.5 – Ponteiros

Deville permite a declaração de ponteiros da forma descrita no seguinte trecho de código. Sua declaração é semelhante à declaração de variáveis (seção 2.4) sendo a não necessidade de indicar um valor explicitamente a diferença essencial entre a declaração desta e daquela.

```
(DECVARIAVEL (PONTEIRO "varPonteiro" Int))
```

2.6 – Operadores

Deville possui, devido a pouca complexade de linguagem, um conjunto reduzido de operadores que poderão ser usados, com certas restrições, com os tipos definidos (2.2). Tais operadores poderão ter aridade um ou dois. A tabela a seguir possui uma descrição mais detalhada dos elementos citados nesta seção.

Operador	Aridade	Tipos	Exemplo
MENOS	1	Int	***MENOS*** Expressao
NOT	1	Boolean	***NOT*** Expressao
LENGTH	1	String	***LENGTH*** Expressao
EXPSOMA	2	Int	***EXPSOMA*** Expressao Expressao
EXPSUBTRACAO	2	Int	***EXPSUBTRACAO*** Expressao Expressao
EXPAND	2	Boolean	***EXPAND*** Expressao Expressao
EXPOR	2	Boolean	***EXPOR*** Expressao Expressao
EXPIGUAL	2	Int,Boolean,String	***EXPIGUAL*** Expressao Expressao
EXPCONCATENACAO	2	String	***EXPCONCATENACAO*** Expressao Expressao

Tabela de Operadores de Deville

Nesta tabela os exemplos de uso retratam uma abstração da forma real de uso desses operadores. Nela podemos notar que um dado operador sempre estará ligado a uma ou duas expressões, dependendo de sua aridade, e o mesmo sempre será escrito antes das

expressão(ões) a(os) qual(ais) está associado. Em seção posterior será demonstrada a forma de construção de expressões, com isso ficará mais bem elucidada a forma de usar os operadores aqui descritos.

2.7 – Expressões

O entendimento da construção de expressões em *Deville* é fundamental para o funcionamento da linguagem. Tal declaração se verifica pelo fato de que as expressões estão presentes na maioria dos comandos, inclusive na própria expressão, que poderá ser recursivamente construída.

Temos dois tipos básicos de expressões: a expressão unária (EXPUNARIA) e a expressão binária (EXPBINARIA). Essas poderão ser combinadas para formar expressões mais complexas. Uma expressão poderá assumir um valor diretamente declarado (inteiro, string de caracteres ou booleano), o valor de um endereço, o conteúdo de um endereço, ou o conteúdo de uma variável. Além desses valores, uma expressão poderá ser formada por outra expressão, podendo esta ser unária ou binária, pois isso irá possibilitar o encadeamento dessas estruturas.

Os seguintes exemplos de códigos mostram as possíveis formas de delaração de expressões.

```
Declaração de expressão binária:
(EXPBINARIA ( EXPSOMA (VALOR (VALInTeiro 10)) (VALOR (VALInTeiro 10))))
(EXPBINARIA ( EXPCONCATENACAO (VALString “Damon ”) (VALSring “Deville”)))

Declaração de expressão unária anihada a uma binária:
(EXPBINARIA ( EXPAND ( EXPUNARIA ( NOT (VALOR (VALBooleano True)))) (
EXPUNARIA (VALOR (VALBooleano True)))))
```

2.8 – Estruturas de Controle

Nesta subseção são apresentadas as estruturas de controle presentes em *Deville*. Tal apresentação será acompanhada de um exemplo de uso, pois isso elucidará qualquer dúvida referente à sintaxe a ser empregada.

2.8.1 – If Then Else

Um escopo if só pode existir se tiver um else associado. O parâmetro dele é uma expressão booleana e else não possui parâmetro. A execução do bloco if só se realizará caso a condição booleana for verdadeira, e, caso contrário, será executado o bloco else.

```
(IFTHENELSE (EXPBINARIA (EXPOR (ID "a")  (ID "b")))
(COMANDOIO (WRITE (EXPBINARIA (EXPSUBTRACAO (VALOR
(VALInteiro 1)) (VALOR (VALInteiro 5))))))
(COMANDOIO (READ "b")))
```

Possui como parâmetros a atribuição inicial da variável de controle e a condição de controle. No seu corpo, os comandos os quais serão executados enquanto a condição de controle for satisfeita. A variável de controle deve ser declarada fora do escopo e obrigatoriamente deverá ser inteira. A condição de controle será uma expressão booleana que envolva a referida variável. No exemplo abaixo é assumido que a variável i já foi declarada. Tal declaração de variável ocorre da forma especificada na seção 2.2.

```
(FOR (ATRIBUICAO "i" (VALORINT (VALInteiro 1))) 30 (COMANDOSKIP) )
```

2.8.3 – While

Possui uma expressão booleana como parâmetro e um corpo com os comandos. A variável de iteração deverá ser inicializada fora do escopo e ser inteira. Os comandos serão executados enquanto a o parâmetro gerar uma condição verdadeira. É facultativa a inserção da variável na iteração, gerando, caso constatado a não alteração da mesma, um loop infinito.

```
(WHILE (EXPMENOR (ID "i") (VALOR (VALInteiro 100)))
    ((ATRIBUICAO "i"(EXPBINARIA (EXPSOMA (ID "i") (VALInteiro 1)))))
```

2.8.4 – Repeat Until

Esse comando é muito semelhante ao comando While, sendo a diferença fundamental entre eles o fato de que Repeat Until executa enquanto a expressão avaliada na estrutura é falsa. Isso é exatamente o oposto do que ocorre na avaliação da expressão do comando While.

```
(REPEATUNTIL
 ((ATRIBUICAO "i"(EXPBINARIA (EXPSOMA (ID "i") (VALInteiro 1))))
 (EXPMENOR (ID "i") (VALOR (VALInteiro 100))))
```

2.8.5 – Skip

O comando Skip não executará operação alguma, sendo por isso o mais simples comando da linguagem. Esse comando é semelhante a operações de linguagem de máquina que não executam alterações no estado da máquina.

```
(COMANDOSKIP)
```

2.9 – Procedimentos

Em Devill procedimentos poderão ser criados e posteriormente chamados dentro do programa. Um procedimento possui uma lista de parâmetros com zero ou mais elementos e não retornam valor algum. Esta estrutura poderá ter variáveis declaradas internamente que estarão no escopo global. No quadro abaixo está um exemplo de declaração de procedimento e o uso através de sua chamada. Nele admitimos que as variáveis x e y foram declaradas anteriormente da forma descrita na seção 2.4.

```
Declaração de procedimento:
(COMANDODECLARACAO (DECLARACAO (DECPROCEDIMENTO(PROCEDIMENTO “proc”
(PARAMVARIAVEL “X” Boolean))
(DECVARIAVEL (VARIAVEL "i" (VALOR (VALInteiro 10))))
(COMANDOFOR (FOR (ATRIBUICAO “i” (VALORINT (VALInteiro 1))) 30
(COMANDOSKIP)))))

Chamada de procedimento:
(COMANDOCHAMADAPROCEDIMENTO (CALLPROCEDIMENTO “proc” (EXPRESSAO ID “y”) ))
```

Na declaração de um procedimento poderemos ter zero ou mais parâmetros que serão passados através de uma lista. No exemplo abaixo está descrita a forma como construir uma lista de parâmetros. As palavras PARAMVARIAVEL e PARAMPONTEIRO indicam que o parâmetro será uma variável e ponteiro, respectivamente.

```
((PARAMVARIAVEL “X” Boolean),(PARAMVARIAVEL “Y” Int),(PARAMPONTEIRO “z”
Int))
```

Para a chamada de procedimentos os parâmetros que serão passados deverão estar na forma de expressões (seção 2.7).

2.10 - Comando de Entrada e Saída

Existem dois comandos com os quais é possível realizar entrada e saída de dados no programa: READ lê uma entrada do teclado e armazena na variável indicada pelo identificador, WRITE que avalia o valor da expressão especificada e imprime seu resultado na tela. Na seqüência é demonstrados exemplos de uso dos comandos citados nesta seção.

```
(COMANDOIO (WRITE (VALOR( VALInteiro 0))))
(COMANDOIO (READ “x”))
```

2.11 – Comando de Atribuição

Uma atribuição de valor a uma é mostrada nos exemplos abaixo. Neste exemplo temos a atribuição a uma variável anteriormente declarada .

```
(COMANDOATRIBUICAO (ATRIBUICAO "variavel" ( VALInteiro 20)))
```

2.12 - Programa

Como foi dito no inicio desse documento, um programa será construído de vários comandos encadeados. No exemplo abaixo é mostrado como fazer a declaração de um programa. A palavra Comando que aparece entre parênteses poderá ser qualquer comando especificado na seção 2.1.

```
(PROGRAMA (Comando),(Comando),(Comando)...)
```

3. Implementação do Projeto

Nesta seção será feita uma apresentação a respeito dos elementos que formam a linguagem *Deville*.

3.1. Elementos DATA e TYPE

Elementos DATA: sua utilização foi necessária para definição dos elementos gramáticais da linguagem *Deville*; Elementos TYPE: foram utilizados para definir os tipos usados pela linguagem. No quadro abaixo está descrita a gramática utilizada na implementação deste projeto.

```
type Id = String

type ListaParametros = [Parametro]

data Programa = PROGRAMA Comando

data Comando = COMANDODECLARACAO ComandoDeclaracao
             | COMANDOATRIBUICAO Atribuicao
             | COMANDOWHILE While
             | COMANDOFOR For
             | COMANDOIFTHENELSE IfThenElse
             | COMANDOREPEATUNTIL RepeatUntil
             | COMANDOIO Io
             | COMANDOSKIP
             | COMANDOCHAMADAPROCEDIMENTO ChamadaProcedimento
             | COMANDOLISTACOMANDOS Comando Comando

data Atribuicao = ATRIBUICAO Id Expressao

data Expressao = VALOR Valor
               | EXPUNARIA ExpUnaria
               | EXPBINARIA ExpBinaria
               | ID Id
               | ENDERECO Id
               | CONTEUDO Id
```

```
data Valor = VALInteiro Int

           | VALBooleano Bool

           | VALPonteiro Id

           | VALString String deriving (Eq, Ord, Show)

data ExpUnaria = MENOS Expressao

               | NOT Expressao

               | LENGTH Expressao

data ExpBinaria = EXPSOMA Expressao Expressao

                | EXPSUBTRACAO Expressao Expressao

                | EXPAND Expressao Expressao

                | EXPOR Expressao Expressao

                | EXPIGUAL Expressao Expressao

                | EXPCONCATENACAO Expressao Expressao

data ComandoDeclaracao = DECLARACAO Declaracao Comando

data Declaracao = DECVARIAVEL DeclaracaoVariavel
                | DECPROCEDIMENTO DeclaracaoProcedimento
                | LISTADECLARACOES Declaracao Declaração

data DeclaracaoVariavel = VARIAVEL Id Expressao
                        | PONTEIRO Id Tipo

data DeclaracaoProcedimento = PROCEDIMENTO Id ListaParametros Comando

data Parametro = PARAMVARIAVEL Id Tipo
               | PARAMPONTEIRO Id Tipo deriving (Eq)

data Tipo = String
          | Int
          | Boolean deriving (Eq)

data While = WHILE Expressao Comando

data For = FOR Atribuicao Int Comando

data IfThenElse = IFTHENELSE Expressao Comando Comando

data RepeatUntil = REPEATUNTIL Comando Expressão

data Io = WRITE Expressao
        | READ Id

data ChamadaProcedimento = CALLPROCEDIMENTO Id ListaExpressao

data ListaExpressao = EXPRESSAO Expressao
                    | LISTAEXPRESSAO Expressao ListaExpressao
```

3.2 Funções de validação

Foram implementadas com o objetivo de avaliar os elementos usados nos comandos da linguagem *Deville.* A avaliação verifica se variáveis foram declaradas, se os tipos correspondem aos tipos esperados, etc. Assim a linguagem consegue oferecer algum suporte a conflitos de tipos e variáveis.

3.3 Memória e Binding

A fim de objetivar a facilidade de criação da memória foi usado o conceito de pilhas, simuladas por duas listas definidas como tipo: uma para variáveis e outra para procedimentos. Entretanto, o binding é compartilhado tanto por variáveis como procedimentos.

3.3.1. Pilha da Memória de Variáveis

É uma pilha constituída de várias células contendo o endereço e o valor (conteúdo) da variável especificada.

3.3.2. Pilha da Memória de Procedimentos

É uma pilha com várias células, cada uma contendo o endereço, a lista de parâmetros e o comando a ser executado pelo procedimento em a ser utilizado.

3.3.3. Pilha do Binding

É uma pilha constituída de várias células, contendo cada uma um identificador, um endereço e um tipo. O identificador pode ser de uma variável, de um ponteiro ou de um procedimento. Dessa forma foi possível criar o compartilhamento de espaço por variáveis e ponteiros. Para diferenciá-los é preciso apenas que se avalia o identificador.

3.4. Avaliação de Expressões

Como a maioria das expressões possui operadores predefinidos a forma de avaliação é convencional e ocorre a partir das expressões que estão mais a esquerda. No entanto o operador de igualdade, por trabalhar com tipos diferentes precisa avaliar seus operadores quanto a compatibilidade antes de iniciar sua execução.

4. Estudo de Caso

Nessa seção apresentamos alguns dos exemplos escritos na linguagem *Deville.* Para uma maior compreensão dos exemplos, inicialmente iremos apresentar um programa semelhante escrito em pseudo-código que simule uma linguagem de mais alto nível. Em seguida o código em *Deville* será apresentado.

4.1 Programa I

Programa que executa um laço *for* e declara duas variáveis: uma externa e uma interna ao laço (escopos globais e locais, respectivamente) . O programa também soma as duas variáveis a cada execução do *for*.

4.1.1 Código em Alto Nível

```
var a = 0
      a = 0
      for (a = 0; a < 20; a = a + 1){
            var b = 0
           b = b + a
      }
```

4.1.2 Código em *Deville*

```
execPrograma [] (PROGRAMA (COMANDODECLARACAO (DECLARACAO (DECVARIAVEL
(VARIAVEL "a" (VALOR (VALInteiro 0)))) (COMANDOFOR (FOR (ATRIBUICAO "a"
(VALOR (VALInteiro 0))) 20
(COMANDOLISTACOMANDOS (COMANDOATRIBUICAO (ATRIBUICAO "a" (EXPBINARIA
(EXPSOMA (ID "a") (VALOR (VALInteiro 1))))))
(COMANDOLISTACOMANDOS (COMANDODECLARACAO(DECLARACAO (DECVARIAVEL (VARIAVEL
"b" (VALOR (VALInteiro 0))))(COMANDOSKIP)))
(COMANDOATRIBUICAO (ATRIBUICAO "b" (EXPBINARIA (EXPSOMA (ID "a") (ID
"b")))))))))))
```

4.1.3 Saída do Console

```
"Memoria: Variaveis: [ (22, 21) (21, 20) (20, 19) (19, 18) (18, 17) (17,
16) (16, 15) (15, 14) (14, 13) (13, 12) (12, 11) (11, 10) (10, 9) (9, 8)
(8, 7) (7, 6) (6, 5) (5, 4) (4, 3) (3, 2) (2, 1) (1, 21)  ]

Procedimentos: [  ]

Vinculacao:  | \"b\" => 22 | \"b\" => 21 | \"b\" => 20 | \"b\" => 19 |
\"b\" => 18 | \"b\" => 17 | \"b\" => 16 | \"b\" => 15 | \"b\" => 14 | \"b\"
=> 13 | \"b\" => 12 | \"b\" => 11 | \"b\" => 10 | \"b\" => 9 | \"b\" => 8 |
\"b\" => 7 | \"b\" => 6 | \"b\" => 5 | \"b\" => 4 | \"b\" => 3 | \"b\" => 2
| \"a\" => 1 | "
```

Legenda : as variáveis são apresentadas na saída com o endereço que a identifica mais o seu valor ao final da execução como na seguinte generalização => (endereço, valor).

Obs.: podemos notar que o excessivo número de variáveis se deve a declaração que ocorre sempre que o laço é executado.

4.2 Programa II

Programa que exemplifica o uso do comando *while* em *Deville*

4.2.1 Código em Alto Nível

```
var a = 1;
var b = 2;
while (a != 100) {
      a = a + 1;
}
```

4.2.2 Código em *Deville*

```
execPrograma [] (PROGRAMA (COMANDOLISTACOMANDOS (COMANDODECLARACAO (DECLARACAO
(DECVARIAVEL (VARIAVEL "a" (VALOR (VALInteiro 1)))) (COMANDOSKIP)))

 (COMANDOLISTACOMANDOS (COMANDODECLARACAO (DECLARACAO (DECVARIAVEL (VARIAVEL "b" (VALOR
(VALInteiro 2)))) (COMANDOSKIP)))

(COMANDOWHILE (WHILE (EXPUNARIA (NOT (EXPBINARIA (EXPIGUAL (ID "a") (VALOR (VALInteiro 100))))))
(COMANDOATRIBUICAO (ATRIBUICAO "a" (EXPBINARIA (EXPSOMA (ID "a") (VALOR (VALInteiro 1)))))))))))
```

4.2.3 Saída do Console

```
"Memoria: Variaveis: [ (2, 2) (1, 100)]

 Procedimentos: [ ]

Vinculacao:  | \"b\" => 2 | \"a\" => 1 | "
```

Obs.: a variável de endereço 1 (variável "a") inicialmente declarada com valor igual a 1, apresenta seu valor final devido as sucessivas iterações

4.3 Programa III

Programa que exemplifica o uso do comando *if then else* em *Deville.*

4.3.1 Código em Alto Nível

```
var a = True;
var b = False;
if (a or b) then
   write (1 - 5);
else
   read (b);
var c = 10;
```

4.3.2 Código em *Deville* (para expressão avaliada como verdadeira)

```
execPrograma [VALBooleano True] (PROGRAMA (COMANDOLISTACOMANDOS
(COMANDODECLARACAO (DECLARACAO (DECVARIAVEL (VARIAVEL "a" (VALOR
(VALBooleano True)))) (COMANDOSKIP)))
(COMANDOLISTACOMANDOS (COMANDODECLARACAO (DECLARACAO (DECVARIAVEL (VARIAVEL
"b" (VALOR (VALBooleano False))))(COMANDOSKIP)))
 (COMANDOLISTACOMANDOS (COMANDOIFTHENELSE (IFTHENELSE (EXPBINARIA (EXPOR
(ID "a") (ID "b")))
(COMANDOIO (WRITE (EXPBINARIA (EXPSUBTRACAO (VALOR (VALInteiro 1)) (VALOR
(VALInteiro 5)))))) (COMANDOIO (READ "b"))))
(COMANDODECLARACAO (DECLARACAO (DECVARIAVEL (VARIAVEL "c" (VALOR
(VALInteiro 10)))) (COMANDOSKIP)))))))
```

4.3.3 Saída do Console(para expressão avaliada como verdadeira)

```
"Memoria: Variaveis: [ (3, 10) (2, False) (1, True)  ]

 Procedimentos: [ ]

 Vinculacao:  | \"c\" => 3 | \"b\" => 2 | \"a\" => 1 | "
```

4.3.2 Código em *Deville* (para expressão avaliada como falsa)

```
execPrograma [VALBooleano True] (PROGRAMA (COMANDOLISTACOMANDOS
(COMANDODECLARACAO (DECLARACAO (DECVARIAVEL (VARIAVEL "a" (VALOR
(VALBooleano True)))) (COMANDOSKIP)))
 (COMANDOLISTACOMANDOS (COMANDODECLARACAO (DECLARACAO (DECVARIAVEL
(VARIAVEL "b" (VALOR (VALBooleano False))))(COMANDOSKIP)))
 (COMANDOLISTACOMANDOS (COMANDOIFTHENELSE (IFTHENELSE (EXPBINARIA (EXPOR
(ID "b") (ID "b")))
 (COMANDOIO (WRITE (EXPBINARIA (EXPSUBTRACAO (VALOR (VALInteiro 1)) (VALOR
(VALInteiro 5)))))) (COMANDOIO (READ "b"))))
 (COMANDODECLARACAO (DECLARACAO (DECVARIAVEL (VARIAVEL "c" (VALOR
(VALInteiro 10)))) (COMANDOSKIP))) ))))
```

4.3.3 Saída do Console(para expressão avaliada como falsa)

```
"Memoria: Variaveis: [ (3, 10) (2, True) (1, True)  ] Procedimentos: [  ] Vinculacao:  | \"c\" => 3 | \"b\" => 2 |
\"a\" => 1 | "
```

Obs.: como a expressão do comando IFTHENELSE foi avaliada como falsa, o *else* foi executado e a variável "b" de endereço 2 inicialmente declarada com valor False, teve seu valor modificado para True.

4.4 Programa IV

Programa que faz o uso de ponteiros e do comando de concatenação.

4.4.1 Código em Alto Nível

```
var a = "SPORT SEMPRE CAMPEAO";
ptr b = ^String
b = &a;
var c = *b ++ " O SPORT QUE EMOCIONA"
```

4.1.2 Código em *Deville*

```
execPrograma [] (PROGRAMA (COMANDOLISTACOMANDOS (COMANDODECLARACAO
(DECLARACAO (DECVARIAVEL (VARIAVEL "a" (VALOR (VALString "SPORT SEMPRE
CAMPEAO"))))(COMANDOSKIP)))
(COMANDOLISTACOMANDOS (COMANDODECLARACAO (DECLARACAO (DECVARIAVEL (PONTEIRO
"b" String))(COMANDOSKIP)))
(COMANDOLISTACOMANDOS (COMANDOATRIBUICAO (ATRIBUICAO "b" (ID "a")))
(COMANDODECLARACAO (DECLARACAO (DECVARIAVEL (VARIAVEL "c" (EXPBINARIA
(EXPCONCATENACAO (ID "b") (VALOR (VALString " O SPORT QUE EMOCIONA"))))))
(COMANDOSKIP)))))))
```

4.1.3 Saída do Console

```
Memoria: Variaveis: [ (3, SPORT SEMPRE  CAMPEAO O SPORT QUE EMOCIONA) (2,
SPORT SEMPRE  CAMPEAO) (1, SPORT SEMPRE  CAMPEAO)  ]
Procedimentos: [  ]
 Vinculacao:  | \"c\" => 3 | \"b\" => 2 | \"a\" => 1 | "
```

Obs.: a variável "c" de endereço 3 apresenta o resultado da concatenação na saída.

4.5 Programa V

Programa que faz o uso de um comando *repeat until*.

4.5.1 Código em Alto Nível

```
var a = 1;
var b = 2;
repeat{
      a = a + 1;
}until (a != 10)
```

4.5.2 Código em *Deville*

```
execPrograma [] (PROGRAMA (COMANDOLISTACOMANDOS (COMANDODECLARACAO
(DECLARACAO (DECVARIAVEL(VARIAVEL "a" (VALOR (VALInteiro 1))))
(COMANDOSKIP)))
(COMANDOLISTACOMANDOS (COMANDODECLARACAO (DECLARACAO (DECVARIAVEL (VARIAVEL
"b" (VALOR (VALInteiro 2)))) (COMANDOSKIP)))
(COMANDOREPEATUNTIL (REPEATUNTIL (COMANDOLISTACOMANDOS (COMANDOSKIP)
(COMANDOATRIBUICAO (ATRIBUICAO "a" (EXPBINARIA (EXPSOMA (ID "a") (VALOR
(VALInteiro 1)))))))
(EXPBINARIA (EXPIGUAL (ID "a") (VALOR (VALInteiro 10)))))))))
```

Obs.: a variável "a" de endereço 1 inicialmente declarada com valor igual a 1 apresenta o resultado da sucessão de iterações aplicadas sobre a mesma nas execuções do laço *repeat until*.

4.5.3 Saída do Console

```
"Memoria: Variaveis: [ (2, 2) (1, 10)  ]
 Procedimentos: [  ]
 Vinculacao:  | \"b\" => 2 | \"a\" => 1 | "
```

Obs.: a variável "a" de endereço 1 inicialmente declarada com valor igual a 1 apresenta o resultado da sucessão de iterações aplicadas sobre a mesma nas execuções do laço *repeat until*.

5. Conclusão

Neste documento apresentamos a linguagem *Deville* e os detalhes de sua implementação e de sua definição sintática, além disso apresentamos programas escritos em *Deville* com a função de exibir alguns exemplos simples de comandos existentes nas linguagens de programação tradicionais.

A escolha da linguagem Haskell para implementação do interpretador da linguagem *Deville* trouxe as vantagens da recursividade simples e da manipulação de listas, que simularam as pilhas da memória. Uma alternativa bem mais eficiente em termos de simplicidade de implementação do que se fossem usados elementos semelhantes como *arrays* ou *vetores* presentes em outras linguagens.

Além disso para projetos futuros da linguagem *Deville* é preciso que se observe com mais cuidado determinadas escolhas, como a pilha de bindings. No momento é um espaço compartilhado entre variáveis e procedimentos. Mas pode ser mais interessante criar uma pilha para cada, a fim de evitar problemas com tempo de busca.