
5

Programação CL

Neste capítulo são apresentados alguns exemplos complementares de programação em CL, tais como: introdução a programação estruturada e funcional; desenvolvimento de pacotes e o estabelecimento das regras para definição de comentários de código de programas.

5.1 PROGRAMAÇÃO ESTRUTURADA

A partir da exposição das diversas funcionalidades para a linguagem CL passa-se a ter em mãos as ferramentas necessárias para escrever programas mais complexos na forma de **scripts**.

Para demonstrar o uso da técnica de programação estruturada em CL considere um programa que a partir de um menu indique a possibilidade de escolha de uma entre quatro operações aritméticas que seja operada a partir do fornecimento de dois valores numéricos. O programa deve possuir a indicação de cinco opções de operação, sendo quatro para as operações aritméticas de soma, subtração, divisão e multiplicação e uma para a saída do programa. Se fornecido opção de menu diferente das opções esperadas o programa deve apresentar mensagem de erro. Caso seja indicado divisor zero para a operação de divisão a resposta deverá ser a mensagem “Erro”.

Programas escritos em estilo estruturado seguem em CL o padrão **bottom-up** de codificação, onde primeiro codifica-se as sub-rotinas de nível mais baixo do programa e em seguida se codifica as sub-rotinas de nível mais alto. Note que primeiramente é codificada as rotinas das operações aritméticas e no final é codificado o trecho principal do programa.

O código seguinte deve ser escrito em um editor de texto simples e gravado com o nome **calc1.lsp** ou **calc1.lisp**, onde **LSP** ou **LISP (Lisp Source Program)** indicam a identificação da extensão de arquivos de programas codificados em CL. A extensão **LSP** é uma forma mais antiga de representação, atualmente utiliza-se mais a forma **LISP**.

Note que todos os detalhes usados neste programa foram apresentados ao longo das páginas anteriores. Atente para um fato novo em que a definição de uma função pode conter internamente um conjunto de funções. Grave o programa proposto em uma pasta chamada “**FonteLISP**” a partir da unidade **C:**.

```
(defun calculadora ()  
  
(defun ADICAO ()  
  (let ((A 0) (B 0) (R 0))  
    (format t "~%Rotina de Adicao~%")  
    (format t "-----~%~%")  
    (princ "Entre valor <A> .: ") (finish-output)  
    (setf A (read))  
    (princ "Entre valor <B> .: ") (finish-output)  
    (setf B (read))  
    (setf R (+ A B))  
    (format t "Resultado .....: ~,2f~%" R)))  
  
(defun SUBTRACAO ()  
  (let ((A 0) (B 0) (R 0))  
    (format t "~%Rotina de Subtracao~%")  
    (format t "-----~%~%")  
    (princ "Entre valor <A> .: ") (finish-output)  
    (setf A (read))  
    (princ "Entre valor <B> .: ") (finish-output)  
    (setf B (read))  
    (setf R (- A B))  
    (format t "Resultado .....: ~,2f~%" R)))
```

```
(defun MULTIPLICACAO ()
  (let ((A 0) (B 0) (R 0))
    (format t "~%Rotina de Multiplicacao~%")
    (format t "-----~%~%")
    (princ "Entre valor <A> .: ") (finish-output)
    (setf A (read))
    (princ "Entre valor <B> .: ") (finish-output)
    (setf B (read))
    (setf R (* A B))
    (format t "Resultado .....: ~,2f~%" R)))

(defun DIVISAO ()
  (let ((A 0) (B 0) (R 0))
    (format t "~%Rotina de Divisao~%")
    (format t "-----~%~%")
    (princ "Entre valor <A> .: ") (finish-output)
    (setf A (read))
    (princ "Entre valor <B> .: ") (finish-output)
    (setf B (read))
    (if (= B 0)
        (format t "Resultado .....: Erro~%~%")
        (progn
          (setf R (/ A B))
          (format t "Resultado .....: ~,2f~%" R)))))

(loop
  (let ((OPCAO 0))
    (format t "~%Programa Calculadora~%")
    (format t "-----~%~%")
    (format t "[1] - Adicao~%")
    (format t "[2] - Subtracao~%")
    (format t "[3] - Multiplicacao~%")
    (format t "[4] - Divisao~%")
    (format t "[5] - Fim de programa~%~%")
```

```
(format t "Entre uma opcao: ") (finish-output)
(setf OPCAO (read))
(when (/= OPCAO 5)
  (progn
    (cond
      ((= OPCAO 1) (ADICAO))
      ((= OPCAO 2) (SUBTRACAO))
      ((= OPCAO 3) (MULTIPLICACAO))
      ((= OPCAO 4) (DIVISAO))
      (t (format t "Opcao invalida - tente novamente~%")))))
  (when (= OPCAO 5)
    (return))))
```

A partir do **script** de programa gravado é possível fazer seu uso de duas maneiras diferentes: uma executando o programa de dentro do ambiente e outra executando o programa no sistema operacional a partir do ambiente e mantendo-se no sistema.

Para executar o programa calc1.lisp no **prompt** do sistema operacional e manter-se ao final da execução no próprio sistema sem que ocorra na memória o carregamento do ambiente CL siga as instruções seguintes.

Com o uso do programa CLISP execute a instrução.

```
C:\> clisp calc1.lisp
```

Com o uso do programa SBCL execute a instrução.

```
C:\> sbcl --script calc1.lisp
```

Para executar o programa **calc1.lisp** de dentro dos ambientes CLISP e SBCL siga as seguintes instruções.

Com o programa CLISP carregado execute a instrução seguinte que efetua o carregamento do programa **calc1.lisp** na memória.

```
[n]> (load "C:\\FonteLISP\\calc1.lisp")
;; Loading file C:\\FonteLISP\\calc1.lisp ...
;; Loaded file C:\\FonteLISP\\calc1.lisp
```

T

Com o programa SBCL carregado execute a instrução seguinte que efetua o carregamento do programa **calc1.lisp** na memória.

```
* (load "C:\\FonteLISP\\calc1.lisp")
T
```

Após o carregamento na memória do programa **calc1.lisp** para usá-lo basta apenas executar a instrução.

```
>>> (calculadora)
```

A partir deste momento escolha a opção desejada no menu e informe os valores solicitados. Veja cada uma das opções. Teste a entrada de valor zero para o divisor. Tente efetuar a entrada de opções diferentes das opções indicadas no menu. Veja todas as possibilidades do programa.

5.2 DEFINIÇÃO DE COMENTÁRIOS

Além das definições de instruções, funções e programas escritos em CL é adequado conhecer as regras de definição das linhas de comentários dos códigos de um programa, que podem ser estabelecidas a partir de três símbolos diferentes com: comentários alados (#| |#), ponto e vírgula (;) e aspas inglesas ("").

Os comentários alados são iniciados com (#|) e finalizados com (|#) servindo para a definição de blocos de comentários que ocupem mais de uma linha aceitando-se o uso de blocos aninhados simulando sub níveis. Por vezes essa forma de comentário pode ser usada para indicar a eliminação temporária de trechos de códigos na fase de testes de um programa por permitir abranger grupos de linhas.

```
#| Sub-rotina destinada a geração de valores aleatórios definidos
   em uma faixa numérica delimitada entre um valor inicial repre-
   sentado pelo argumento INICIO e um valor final representado
   pelo argumento FIM.
#
   Os valores dos argumentos INICIO e FIM poderão ser defini-
   dos a partir do uso números inteiros e/ou reais.
|
|#
```

```
(defun sorteio (INICIO FIM)
  (+ INICIO
     (random (1+ (- FIM INICIO)))
     (make-random-state t))))
```

Quanto ao uso do símbolo ponto e vírgula (;) este pode ser definido a partir de quatro variações descritas a seguir.

A indicação de comentários com o uso de um ponto e vírgula (;) são usados para descrever as ações de certa linha de código. Quando usados vários comentários em esses devem ser alinhados a partir da mesma posição usada para o comentário da coluna anterior.

```
(defun sorteio (INICIO FIM)      ; Função "sorteio" com INICIO e FIM
  (+ INICIO                  ; soma o valor do INICIO ao sorteio
     (random (1+ (- FIM INICIO)) ; do valor entre 1 e FIM - INICIO a
     (make-random-state t)))    ; partir do modo randômico variável
```

A indicação de comentários com dois pontos e vírgulas (;;) são usados para descrever o propósito das linhas de código ou o estado operacional de certo programa naquele ponto, são usados na definição de comentários comuns. Este tipo de comentário fica alinhado com o mesmo nível de recuo do código quando assim for usado.

```
;; Função "sorteio" que efetua a apresentação de valor randômico entre
;; um valor inicial (INICIO) e valor final (FIM) informados.
(defun sorteio (INICIO FIM)
  ;; Função sorteio com simulação de geração automática de semente
  (+ INICIO
     (random (1+ (- FIM INICIO)))
     (make-random-state t))))
```

A indicação de comentários com três pontos e vírgulas (;;;) são usados para descrever uma seção de programa. Esses comentários devem ser iniciados na margem esquerda do código.

;;; Função para geração de valores aleatórios

```
(defun sorteio (INICIO FIM)
  (+ INICIO
      (random (1+ (- FIM INICIO)))
      (make-random-state t))))
```

A indicação de comentários com quatro pontos e vírgulas (;;;;) são usadas para descrever observações em nível de arquivo dando normalmente explicações gerais sobre o conteúdo do arquivo contendo um conjunto de funcionalidades. Esses comentários devem ser iniciados na margem esquerda do código.

**;;;; Instruções para teste de execução recursiva de cálculo de
;;;; factorial que indicam os resultados de 0 a 15.**

;;; Função recursiva para cálculo de factorial

```
(defun fat (N)
  ;; Cálculo de factorial ocorrerá apenas com valores numéricos
  ;; inteiros.
  (if (<= N 0) ; Se N menor ou igual a zero
      1           ; retorna 1.
      (* N (fat (- N 1)))))
```

;;; Parte principal que apresenta as fatoriais de 0 a 15.

```
(loop for I from 0 to 15
      do (format t "~2d! = ~13d~%" I (fat I)))
```

A indicação de comentários entre aspas inglesas ("") são usadas para descrever uma forma de documentação incorporada ao código de uma função que pode ser visualizado com a função documentation. Isto é válido desde que esse comentário seja a primeira expressão na forma de cadeia associada a partir da definição de uma função podendo ser uma descrição com qualquer quantidade de linhas. A tabulação no uso desse tipo de comentário ocorre no mesmo nível de definição da primeira instrução em relação a definição da função.

```
(defun fat (N)
  "Efetua o cálculo de N fatorial"
  (if (<= N 0) ; Se N menor ou igual a zero
      1           ; retorna 1.
      (* N (fat (- N 1)))))
```

A partir da definição de comentários para documentação é possível obtê-los a partir do uso da instrução.

```
>>> (documentation 'fat 'function)
"Efetua cálculo de fatorial"
```

A função documentation pode ser usada para obter a descrição de certo recurso da linguagem CL como, por exemplo, ver a descrição de alguma função interna. No entanto, se a descrição não existir será retornado o valor NIL.

Os detalhes aqui expostos seguem orientações gerais existentes e recomendadas na documentação disponibilizada para a linguagem CL. Existem variações praticadas que não estão sendo aqui consideradas. Manteve-se o senso comum usado por grande parte dos programadores CL.

5.3 PACOTES (PACKAGE)

O sistema da linguagem CL mantém internamente em seu ambiente operacional uma tabela contendo os símbolos disponíveis no sistema e também aqueles definidos na etapa da programação, como: variáveis especiais (globais), funções, macros, entre outros. Quando um novo símbolo é definido este é automaticamente adicionado a esta tabela (conhecida como pacote do sistema).

Os pacotes são um recurso existente na linguagem CL que permite dividir o estado de memória em áreas de trabalho denominadas nomes de espaços (**namespace**). Cada pacote permite definir conjuntos de símbolos formados por variáveis globais, funções, macros ou outro elemento que ficam separados do restante do sistema.

O motivo para se usar pacotes (há quem os chame de módulos) é permitir a definição controlada de uma área de trabalho isolada onde os programadores podem criar seus símbolos sem se preocupar se determinado símbolo criado por ele entra em conflito com outro símbolo usado no sistema ou mesmo escrito por outro programador. Os pacotes reduzem eventuais conflitos entre códigos produzidos inde-

pendentemente por programadores, podendo também ser úteis na definição de arquivos de programas como bibliotecas.

O ambiente CL possui um conjunto de pacotes (que podem ser diferentes entre as diversas distribuições CL existentes), dos quais se destacam três pacotes padrão mais comuns e existentes em qualquer distribuição, sendo:

- **common-lisp** - pacote padrão que possui os símbolos relacionados a todas as funções e variáveis globais definidas pelo padrão da linguagem e disponibilizados para o uso geral da linguagem;
- **keyword** - pacote usado para a identificação de todos os nomes internos da linguagem iniciados por dois pontos;
- **common-lisp-user** – pacote, também referenciado como **cl-user** que utiliza além do pacote padrão *common-lisp*, todos os demais pacotes relacionados as operações de edição e depuração do ambiente de trabalho.

Para visualizar o pacote ativo em certo momento basta solicitar que seja apresentado o conteúdo definido na variável especial ***PACKAGE*** a partir da instrução.

```
>>> *package*
```

No programa CLISP é apresentada a resposta.

```
#<PACKAGE COMMON-LISP-USER>
```

No programa SBCL é apresentada a resposta.

```
#<PACKAGE "COMMON-LISP-USER">
```

O programa SBCL mostra o nome do pacote COMMON-LISP-USER entre aspas inglesas diferentemente do programa CLISP que o indica sem aspas. Além do uso da apresentação do conteúdo da variável ***PACKAGE*** é possível obter essa informação por meio das instruções.

```
>>> common-lisp:*package*
>>> cl:*package*
```

Para ver todos os pacotes existentes no ambiente execute a instrução.

```
>>> (list-all-packages)
```

A lista de pacotes apresentada é diferente entre os ambientes de programação CL existentes, pois isso depende da distribuição do ambiente CL. Considerando as

distribuições CLISP e SBCL tem-se como padrão os três pacotes antes indicados, como: `common-lisp`, `keyword` e `common-lisp-user`.

A criação de pacotes é realizada com o uso da função `defpackage` a partir da sintaxe.

```
(defpackage <:nome>
  (:use <:pacote1> [<:pacote2> ... [<pacoteN>]])
  (:export <:símbolo1> [<:símbolo2> ... [<símboloN>]]))
```

Onde `:nome` representa a definição do nome do pacote que deve ser definido sem o uso de espaços em branco, `:use` se refere ao vínculo e uso de pacotes necessários ao desenvolvimento do pacote criado e `:export` se refere ao uso opcional de símbolos que são definidos externamente para o pacote criado.

Execute as instruções seguintes para criar três pacotes no programa CLISP.

```
[n]> (defpackage :pacote1 (:use :common-lisp))
#<PACKAGE PACOTE1>
```

```
[n]> (defpackage :pacote2 (:use :common-lisp))
#<PACKAGE PACOTE2>
```

```
[n]> (defpackage :pacote3 (:use :common-lisp))
#<PACKAGE PACOTE3>
```

Execute as instruções seguintes para criar três pacotes no programa SBCL.

```
* (defpackage :PACOTE1 (:use :common-lisp))
#<PACKAGE "PACOTE1">
```

```
* (defpackage :PACOTE2 (:use :common-lisp))
#<PACKAGE "PACOTE2">
```

```
* (defpackage :PACOTE3 (:use :common-lisp))
#<PACKAGE "PACOTE3">
```

Na sequência se executada a função `list-all-packages` será apresentado na lista de pacotes do ambiente a existência dos três pacotes criados.

Quando um pacote é criado este pode ter seu nome modificado a qualquer momento com o uso da função `rename-package` operada a partir da sintaxe.

```
(rename-package '<nome_antigo> '<nome_novo>)
```

Onde **nome_antigo** representa o nome do pacote existente em memória e a indicação **nome_novo** se refere ao novo nome do pacote existente.

Observe a seguir a instrução de mudança do nome do pacote PACOTE1 para o nome PACOTE4 no programa CLISP.

```
[n]> (rename-package 'PACOTE1 'PACOTE4)
#<PACKAGE PACOTE4>
```

Observe a seguir a instrução de mudança do nome do pacote PACOTE1 para o nome PACOTE4 no programa SBCL.

```
* (rename-package 'PACOTE1 'PACOTE4)
#<PACKAGE "PACOTE4">
```

Um pacote criado pode ser removido com o uso da função `delete-package`, a partir da sintaxe.

```
(delete-package <'nome>)
```

Onde **nome** representa o nome do pacote a ser removido.

Observe a instrução para remoção dos pacotes anteriormente definidos.

```
>>> (delete-package 'PACOTE2)
```

T

```
>>> (delete-package 'PACOTE3)
```

T

```
>>> (delete-package 'PACOTE4)
```

T

A partir da apresentação de uma visão básica sobre o gerenciamento de pacotes será definido um novo pacote chamado TESTE a partir da instrução.

```
>>> (defpackage :TESTE (:use :common-lisp))
```

Para se fazer uso efetivo de um pacote é necessário usar a função `in-package` que possui a sintaxe.

```
(in-package <nome>)
```

Onde `nome` representa o nome do pacote a ser colocado em uso na memória.

No programa CLISP execute a instrução a seguir e veja a mudança do *prompt*.

```
[n]> (in-package TESTE)
#<PACKAGE TESTE>
TESTE[n]>
```

No programa SBCL execute a instrução seguinte.

```
* (in-package TESTE)
#<PACKAGE "TESTE">
*
```

No programa CLISP o pacote em uso é indicado ao lado esquerdo do *prompt*, mas no programa SBCL o *prompt* padrão é mantido. Uma forma segura de verificar em qual pacote o programa SBCL está a fazer uso é utilizar uma das instruções.

```
* *package*
* common-lisp:*package*
* cl:*package*
```

A partir da definição e ativação do nome de espaço TESTE será neste local definida a função `mensagem` que indicará a apresentação do texto "Acao executada na area TESTE" quando executada no nome de espaço TESTE. Assim sendo, crie a seguinte função.

```
>>> (defun mensagem () "Mensagem em TESTE")
MENSAGEM
```

Na sequência execute a chamada da função `mensagem` e observe o resultado apresentado.

```
>>> (mensagem)
"Mensagem em TESTE"
```

Para sair do nome de espaço TESTE e retornar ao nome de espaço padrão execute a instrução.

```
>>> (in-package COMMON-LISP-USER)
```

Neste momento se executada a função `mensagem` ocorrerá um erro por esta função não estar ativa na área de memória COMMON-LISP-USER. Desta forma, para executar a função `mensagem` da área TESTE execute a instrução.

```
>>> (teste::mensagem)
```

```
"Mensagem em TESTE"
```

Veja que para acionar a ação de uma função de um nome de espaço não ativo é necessário indicar o nome do pacote e o nome da função separada por um operador de escopo indicado pelos símbolos duplos de dois pontos (::).

A partir do que foi exposto é possível criar de forma simples um pacote matemático que contenha algumas constantes não existentes no ambiente CL e algumas funções auxiliares. Para tanto, escreva em um editor de texto simples o código seguinte gravando-o na pasta **FonteLISP** com o nome **math.lisp**.

```
;;;; Pacote .....: MATH  
;;;; Autor .....: Augusto Manzano  
;;;; Finalidade ...: Apoio a operacoes matematicas
```

```
;;; Definicao do pacote  
(defpackage :MATH  
  (:use :common-lisp))
```

```
;;; Pacote colocado em uso  
(in-package MATH)
```

```
;;; Definicao de constantes de apoio  
(defconstant M_E 2.718281828459045235360287471352662497757)  
(defconstant M_LOG2E 1.4426950408889634073599)  
(defconstant M_LOG10E 0.434294481903251827651128)  
(defconstant M_LN2 0.693147180559945309417232121458176568075)  
(defconstant M_LN10 2.302585092994045684017991)
```

```
(defconstant M_PI2 1.570796326794896619231321691639751442098)
(defconstant M_PI4 0.785398163397448309615660845819875721049)
(defconstant M_SQRT2 1.41421356237309504880168872420969807)
(defconstant M_SQRT1_2 0.707106781186547524400844)

;;; Função para obtencao de quociente inteiro
(defun div (A N)
  (multiple-value-bind (Q) (floor A N) Q))

;;; Função recursiva para cálculo de fatorial
(defun fact (N &optional (BASE 1))
  (if (< N 0)
      (format t "Error~%")
      (if (and (>= N 0) (< N 2))
          BASE
          (fact (- N 1) (* BASE N)))))

;;; Função ambiguidade
;;; Dada uma lista de valores a função escolherá apenas um dos
;;; valores, sem que se saiba de antemão qual valor será escolhido
(defun amb (&rest VALORES)
  ; VALORES representa uma lista de itens separados por espaço
  ; em branco
  (nth (random (length VALORES)) VALORES))

;;; Função para geração de faixa de valores
(defun range (INICIO FIM)
  (loop for VALOR from INICIO to FIM
        collect VALOR))

(in-package COMMON-LISP-USER)
```

Assim que o arquivo **math.lisp** estiver escrito e gravado saia do ambiente CL. Faça novo acesso e execute a instrução.

```
>>> (load "math.lisp")
```

E em seguida, para testar o pacote carregado, execute as instruções.

```
>>> (math::m_pi2
```

```
1.5707964
```

```
>>> (math::fact 5)
```

```
120
```

```
>>> (math::range 1 5)
```

```
(1 2 3 4 5)
```

```
>>> (math::amb 5 3 1)
```

```
5
```

A definição da função `amb` (ambiguidade) do pacote `math` tem por finalidade devolver aleatoriamente um dos elementos indicados na lista de valores definida. Devido ao uso de `&rest` é possível especificar uma quantidade indefinida de elementos para a lista de valores da função.

Um uso prático para a função `amb` pode ser aplicado a necessidade de saber quais valores em certa equação gera determinada resposta. Por exemplo, quando que os valores das variáveis **A** e **B** geram o resultado **5** a partir de $5 = A + B$. Note a instrução a seguir.

```
>>> (let ((a (math::amb 1 2 3 4))
          (b (math::amb 1 2 3 4)))
      (if (= 5 (+ a b)) (list a b)))
```

Veja que serão apresentados os valores para as variáveis **A** e **B** quando a soma desses valores for igual a **5**. Caso não seja o resultado **5** o retorno será **NIL**. Para testar a instrução anterior execute-a algumas vezes em seu ambiente até que os valores que geram o resultado **5** seja apresentado.

A função definida como `range` gera uma lista de valores numéricos a partir da indicação de um valor menor e outro maior informado como seus argumentos. Para criar uma lista de valores usa-se a cláusula de construção `collect` que na ação de iteração controlada pelo laço e ao receber certo valor o coloca sequencialmente em

uma lista. Note que quando se usa `collect` não se faz uso da cláusula `do` para a macro `loop`.

A necessidade de indicar o nome do pacote e a funcionalidade a ser utilizada é que garante a possibilidade de se fazer uso de nomes de recursos repetidos em pacotes diferentes.

5.4 PROGRAMAÇÃO FUNCIONAL

O paradigma de programação imperativa teve seu início marcado com o lançamento da linguagem FORTRAN em 1954, dando origem inicialmente ao paradigma de programação semiestruturada, tornando-se mais tarde paradigma de programação estruturada com o lançamento da linguagem ALGOL de 1958, a qual influenciou posteriormente outras linguagens, destacando-se em 1965 a linguagem Simula, primeira linguagem a utilizar o paradigma de programação orientada a objetos.

De certa forma, em contradição ao estilo proposto pelo paradigma imperativo (semiestruturado, estruturado e orientado a objetos) surgiu em 1958 o paradigma declarativo funcional, tendo como seu primeiro exemplar a linguagem LISP como seu representante.

A programação semiestruturada, estruturada e orientada a objetos, por serem um ramo da programação imperativa descreve as operações computacionais como ações executadas por meio de instruções que mudam o estado dos dados armazenados em memória por intermédio de variáveis, ou seja, enfatiza ações com mudanças no estado do programa. Já a programação funcional por ser um paradigma declarativo trata suas operações a partir da avaliação de funções matemáticas, evitando o uso de estados mutáveis comum no paradigma imperativo.

Na programação funcional, de um ponto de vista geral, um dos tipos de dados mais usados e populares é a **lista**, que permite simular computacionalmente a aplicação de conjuntos do ponto de vista estudado pela **teoria de conjuntos**.

As operações relacionadas a manipulação de conjuntos em CL são principalmente realizadas com o uso dos tipos de dados `list`, `array` e `function`. Assim sendo, são indicados neste tópico apenas os detalhes relacionados a aplicação matemática sobre conjuntos e suas relações a partir do uso de funções.

A programação funcional tem como princípio operacional básico o uso de funções de ordem superior, também chamadas de funções de alta ordem que possuem

como característica operacional a capacidade de receber como argumento uma função ou retornar como resultado uma função.

Considere como exemplo o uso da função simples `soma-raizes` que tem por finalidade somar as raízes quadradas definidas entre dois valores inteiros delimitados pelos argumentos I (início) e F (fim) e obtidos a partir do uso da função `sqrt`.

```
>>> (defun soma-raizes (I F)
  (if (> I F)
    0
    (+ (sqrt I) (soma-raizes (1+ I) F))))
```

Veja que a função `soma-raizes` faz uso do recurso `(1+ I)` para incrementar o valor **1** junto a variável **I** a cada vez que a ação recursiva é processada na soma do resultado de uma raiz com o próximo valor.

Para verificar a ação da operação da função `soma-raizes` execute a instrução seguinte e observe a apresentação do resultado da soma das raízes de **1** a **5**.

```
>>> (soma-raizes 1 5)
8.382332
```

As funções `soma-raizes` e `sqrt` não estão sendo usadas como funções de primeira ordem, pois são funções simples, sendo necessário extrair a raiz quadrada de cada valor para em seguida produzir o somatório de cada um dos valores calculados. E se quiser fazer o somatório dos valores pares entre dois limites? Seria necessário criar outra função de somatório para atender a esta necessidade. Agora considere usar apenas uma função para calcular o somatório, por exemplo, das raízes quadradas, dos quadrados ou qualquer outra ação entre os limites estabelecidos.

Funções de ordem superior são maneiras de se definir abstrações para a realização de certas operações sem se preocupar como essas ações são efetivamente feitas, uma vez que já estão definidas e podem ser usadas a qualquer momento.

Como exemplo, considere uma função, com toque genérico, chamada `somatorio` que fará uso de alguma outra função passada a ela como argumento. Desta forma, observe o código da função a seguir.

```
>>> (defun somatorio (I F FUNCAO)
  (if (> I F)
    0
    (+ (funcall FUNCAO I) (somatorio (+ I 1) F FUNCAO))))
```

A função somatorio recebe para sua operação três argumentos em sua chamada, representados pelas variáveis I (índice), F (fim) e FUNCAO (uma função de ordem superior a ser fornecida). Enquanto o valor da variável I não for maior que o valor da variável F ocorrerá o processamento da soma por recursão dos resultados aplicados pela função passada como argumento em (funcall FUNCAO I), onde FUNCAO representa a função passada no argumento de somatorio e funcall estabelece a execução da função passada como argumento.

Para verificar a ação da função somatorio execute a instrução seguinte.

```
>>> (somatorio 1 5 (function sqrt))
8.382332
```

Observe que para aplicar uma função de ordem superior como argumento de outra função é necessário defini-la com o uso do comando function entre parênteses, como feito na função somatorio a partir de (function sqrt).

A função somatorio poderá ser usada para diversas ações sobre certa sequência de valores. Por exemplo, para apresentar o somatório dos quadrados dos valores de 1 a 5 será definida uma função simples que será usada como função de ordem superior chamada quadrado.

```
>>> (defun quadrado (NUM)
  (expt NUM 2))
```

Agora execute a instrução.

```
>>> (somatorio 1 5 (function quadrado))
55
```

Note que a função somatorio pode ser usada para a obtenção da soma de diversas operações baseadas em funções de ordem superior que utilizem um argumento.

Como comentado o paradigma de programação funcional considera sua computação a partir do uso de estruturas de conjuntos, onde conjunto é em essência uma coleção de elementos que possuem características comuns (contexto matemático).

Do ponto de vista matemático um conjunto é delimitado entre chaves, tendo seus elementos separados por vírgulas e associados a uma letra maiúscula. Em CL o conjunto será representado por uma coleção de valores delimitados entre parênteses, separados por espaço em branco na forma de listas que poderão ou não estar associados a uma variável.

As operações previstas pela teoria dos conjuntos são: pertinência, inclusão, união, intersecção, diferença e igualdade.

A pertinência indica se determinado elemento pertence ou não a certo conjunto. Essa ação pode ser realizada com o uso da função `find`, já conhecida, que retorna a indicação do valor se este existir na lista ou retorna o valor `NIL` quando o elemento não está contido.

Veja os exemplos.

```
>>> (find 3 '(1 2 3 4 5)) ; o elemento 3 pertence ao conjunto  
3
```

```
>>> (find 6 '(1 2 3 4 5)) ; o elemento 6 não pertence ao conjunto  
NIL
```

A inclusão ocorre quando certo conjunto está contido em outro conjunto (ou seja, quando um conjunto com outro conjunto) e se certo conjunto não está contido em outro conjunto. Essa ação é realizada com o uso da função `subsetp` que verifica se o primeiro conjunto encontra-se contido no segundo conjunto retornando `T` se a operação for verdadeira ou `NIL` se a operação for falsa.

Veja os exemplos.

```
>>> (subsetp '(1 2) '(1 2 3)) ; conjunto 1 contido no conjunto 2  
T
```

```
>>> (subsetp '(1 2 3) '(1 2)) ; conjunto 1 não contido no conjunto 2  
NIL
```

O resultado da união corresponde a junção dos elementos de dois conjuntos. Essa ação é realizada com o uso da função `union`.

Veja os exemplos no programa CLISP.

```
[n]> (union '(1 2 3) '(4 5 6))  
(1 2 3 4 5 6)
```

```
[n]> (union '(1 2 2 3 4) '(4 5 5 6))  
(1 2 2 3 4 5 5 6)
```

Veja os exemplos no programa SBCL.

```
* (union '(1 2 3) '(4 5 6))  
(3 2 1 4 5 6)
```

```
* (union '(1 2 2 3 4) '(4 5 5 6))  
(3 2 2 1 4 5 5 6)
```

O resultado da intersecção entre dois conjuntos corresponde aos elementos comuns aos conjuntos. Essa ação é realizada com o uso da função `intersection`.

Veja os exemplos no programa CLISP.

```
[n]> (intersection '(1 2 3 4 5) '(3 4 5 6))  
(3 4 5)
```

Veja os exemplos no programa SBCL.

```
* (intersection '(1 2 3 4 5) '(3 4 5 6))  
(5 4 3)
```

O resultado da diferença entre conjuntos é formado pelos elementos do primeiro conjunto que não estão presentes no segundo conjunto. Essa ação é realizada com o uso da função `set-difference`.

Veja os exemplos no programa CLISP.

```
[n]> (set-difference '(1 2 3 4) '(1 2))  
(3 4)
```

Veja os exemplos no programa SBCL.

```
* (set-difference '(1 2 3 4) '(1 2))  
(4 3)
```

A igualdade entre conjuntos ocorre quando os conjuntos possuem independentemente da ordem de disposição os mesmos elementos.

A linguagem CL não possui uma função específica para esta ação. Esta operação pode ser realizada a partir do uso adicional de outras funções de apoio, como o uso das funções `null` e `set-exclusive-or`.

A função `null` retorna T se a lista estiver vazia, caso contrário o retorno será NIL, sua ação assemelha-se a função `not` e a função `set-exclusive-or` possui funcionamento operacional contrário ao funcionamento da função `intersection`. Desta forma, esta função retorna como resposta uma lista dos elementos que não se repetem nas duas listas indicadas. Se as listas possuírem os mesmos elementos independentemente da ordem que estejam definidos retornará o valor NIL.

Veja os exemplos.

```
>>> (null (set-exclusive-or '(1 2 3) '(1 2 3)))  
T  
  
>>> (null (set-exclusive-or '(3 2 1) '(1 3 2)))  
T  
  
>>> (null (set-exclusive-or '(1 2 3) '(1 2 9)))  
NIL
```

A partir da visão sobre manipulação de listas e sua relação com a teoria de conjuntos incluindo-se as operações de pertinência, inclusão, união, interseção, diferença e igualdade é possível expandir essas operações com ações de filtragem, redução, mapeamento e transposição aplicados sobre conjuntos a partir do uso de funções, de um ponto de vista matemático.

Computacionalmente uma função, pode ser entendida como sendo uma rotina de programa ou a definição de uma expressão aritmética ou lógica que a partir de um argumento de entrada fornece sempre um valor de saída como resposta de sua ação.

Matematicamente uma função é uma maneira de expressar dois valores existentes em diferentes conjuntos. Desta forma, todos os elementos do primeiro conjunto, chamado domínio, devem ser relacionados aos valores do segundo conjunto, contradomínio. No entanto, pode acontecer de existir no segundo conjunto valores que não estejam relacionados com os elementos do primeiro conjunto.

Algebricamente uma função pode ser definida como $f: A \rightarrow B$, onde "efe" representa a relação existente entre (seta para à direita) os elementos do primeiro conjunto "A" com os elementos do segundo conjunto "B". A partir dessa relação tem-se a definição da operação representada por $y = f(x)$ que será computacionalmente indicada como $f(x) = y$.

A partir da definição $f: A \rightarrow B$ tem-se como domínio o conjunto "A" e contradomínio o conjunto "B". O conjunto domínio refere-se aos valores independentes que determinam a partir da regra algébrica $y = f(x)$ os valores dependentes do conjunto contradomínio.

Poderá existir no conjunto contradomínio algum valor que não tenha relação com um valor do conjunto domínio, ou seja, não pertencente ao conjunto imagem definido a partir da função existente entre os conjuntos. Assim sendo, o conjunto imagem é por sua natureza um subconjunto do conjunto contradomínio possuindo os valores que correspondem diretamente aos valores do conjunto domínio.

A ação conhecida por **mapeamento** visa aplicar uma função a todos os elementos existentes em um conjunto domínio gerando a partir dessa ação um conjunto contradomínio com o resultado da operação definida como função. Essa ação é realizada com o uso da função mapcar.

Considerando um conjunto domínio com os elementos **1, 2, 3, 4 e 5** será definido o conjunto contradomínio com o triplo de cada elemento do domínio a partir da função $f(x) = x3$. Veja a seguir a instrução que permite obter o conjunto imagem **3, 6, 9, 12 e 15**.

```
>>> (mapcar #'(lambda (x) (* x 3)) '(1 2 3 4 5))
(3 6 9 12 15)
```

A função mapcar é usada basicamente a partir de dois argumentos, sendo o primeiro a definição de uma função (geralmente do tipo lambda) e o segundo argumento a definição de uma lista de valores. Observe mais alguns exemplos.

```
>>> (mapcar #'(lambda (x) (expt x 2)) '(1 2 3 4 5))
(1 4 9 16 25)
```

```
>>> (mapcar #'(lambda (x) (* x -1)) '(1 2 3 4 5))
(-1 -2 -3 -4 -5)
```

Dada a função $f: \{-2, 0, 2, 4, 6\} \rightarrow \{8, 6, 4, 2, 0, -2, -4, -6, -8\}$ apresentar seu conjunto imagem.

Observe a instrução executada em CLISP.

```
[n]> (intersection '(8 6 4 2 0 -2 -4 -6 -8)
  (mapcar #'(lambda (x) (- x (* 3 x))) '(-2 0 2 4 6)))
(4 0 -4 -8)
```

Observe a instrução executada em SBCL.

```
* (intersection '(8 6 4 2 0 -2 -4 -6 -8)
  (mapcar #'(lambda (x) (- x (* 3 x))) '(-2 0 2 4 6)))
(-8 -4 0 4)
```

A ação de ***filtragem*** visa aplicar uma função a todos os elementos existentes em um conjunto domínio gerando a partir dessa ação um conjunto contradomínio com o resultado da operação definida como função apenas para os resultados retornados verdadeiros. A linguagem CL não possui função específica, devendo então ser codificada uma função para esta operação. Assim sendo, considere o seguinte código.

```
>>> (defun filter (FUNCAO LISTA)
  (cond
    ((null LISTA) '())
    ((funcall FUNCAO (first LISTA))
     (cons (first LISTA) (filter FUNCAO (rest LISTA))))
    (t (filter FUNCAO (rest LISTA)))))
```

A função filter recebe para operação os argumentos FUNCAO que será a indicação da função a ser aplicado sobre os elementos de uma lista representada pelo argumento LISTA. O trecho ((null LISTA) '()) em cond verifica se a lista é vazia (null LISTA) e se a condição for verdadeira retorna NIL a partir da indicação '().

O trecho ((funcall FUNCAO (first LISTA)) verifica se o primeiro elemento existente na lista satisfaz como condição a função indicada em FUNCAO, sendo a condição verdadeira será criada uma lista com o elemento validado a partir do trecho

(cons (first LISTA) (filter FUNCAO (rest LISTA))), caso contrário será executado o trecho (t (filter FUNCAO (rest LISTA))) que recursivamente pega o restante da lista e o reaplica para novo teste.

Considerando um conjunto domínio com os elementos **1, 2, 3** e **4** será definido o conjunto contradomínio formado apenas pelos valores ímpares do domínio a partir da função $f(x) = x - 2 \angle x / 2 _ | x \sim= 0$. Veja a seguir a instrução que permite obter o conjunto imagem **1** e **3**.

```
>>> (filter #'oddp '(1 2 3 4))
(1 3)
```

Tomando por base o conjunto domínio $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$ para obter o conjunto imagem a partir da filtragem formada pelos valores do domínio que sejam pares. Observe as seguintes instruções.

```
>>> (filter #'evenp '(1 2 3 4 5 6 7 8 9 0))
(2 4 6 8 0)
```

A ação conhecida por **redução** visa aplicar uma função a todos os elementos existentes em um conjunto domínio gerando a partir dessa ação um único valor como resposta. Essa ação é realizada com o uso da função `reduce`.

Considerando um conjunto domínio com os elementos **1, 2, 3** e **4** será definido o conjunto contradomínio de um elemento formado pela soma dos valores do domínio a partir da função $f(x) = \sum x$. Veja a seguir a instrução que permite obter o conjunto imagem **10**.

```
>>> (reduce #'+ '(1 2 3 4))
10
```

A função `reduce` aceita o uso de qualquer função que opere com dois argumentos. Veja os próximos exemplos.

```
>>> (reduce #'* '(1 2 3 4))
24
```

```
>>> (reduce #'- '(1 2 3 4))
-8
```

```
>>> (float (reduce #'/ '(1 2 3 4)))  
0.041666668
```

A ação conhecida por **transposição** visa criar um conjunto formado com os elementos intercalados de outros conjuntos. A transposição ocorrerá com a junção dos primeiros elementos dos conjuntos informados e assim por diante até que acabe os elementos dos conjuntos. Se os conjuntos a serem transpostos forem de tamanho diferentes a transposição usará sempre como base o conjunto com menor número de elementos. Essa ação é feita com o uso das funções `mapcar` e `list`.

Considerando a definição de dois conjuntos, um contendo os elementos **1, 2 e 3** e outro contendo os elementos **4, 5 e 8** será definido o conjunto resultando formados pelos valores **(1 4) (2 5) e (3 6)**. Veja a seguir a instrução que permite obter a transposição dos conjuntos mencionados.

```
>>> (mapcar #'list '(1 2 3) '(4 5 6))  
((1 4) (2 5) (3 6))
```

Note que a operação de transposição faz a distribuição intercalada dos elementos um a um combinando-os em um conjunto resultante.

Veja alguns outros exemplos de transposição.

```
>>> (mapcar #'list '(1 2 3) '(4 6))  
((1 4) (2 6))
```

```
>>> (mapcar #'list '(1 2 3) '(4 6 5 7))  
((1 4) (2 6) (3 5))
```

Os exemplos anteriores mostram os resultados que são obtidos quando se opera com conjuntos de tamanho diferentes.



Referências bibliográficas

ABELINO, T. **História do Lisp: abra os olhos para programação funcional**.

iMasters.com, 2018. Disponível em: <<https://imasters.com.br/desenvolvimento/historia-lisp-abra-os-olhos-para-programacao-funcional>>. Acesso em: 17 mar. 2019.

BARSKI, C. **Land of LISP: Learn to program in Lisp, one game at a time**. San Francisco: No Starch Press, 2010.

BERKELEY, E. C. & BOBROW, D. G. **The programming language LISP: Its operation and applications**. 3th. Massachusetts: MIT Press, 1974.

GRAHAM, P. **ANSI Common Lisp**. New Jersey: Prentice-Hall, 1995.

_____. **OnLisp: Advanced techniques for COMMON LISP**. New Jersey: Prentice-Hall, 1993.

KEENE, S. E. **Object-oriented programming in Common LISP: A programmer's guide to CLOS**. New York: Addison-Wesley, 1989.

LÉVÉNEZ, É. **Computer Languages History**. Paris: VIERLING, 2018. Disponível em: <<https://www.levenez.com/lang/>>. Acesso em: 17 mar. 2019.

LIPSCHUTZ, S.; LIPSON, M. **Matemática discreta**. 3. ed. Porto Alegre: Bookman, 2013.

McCARTHY, John. **LISP 1.5 Programmer's Manual**. Massachusetts: MIT Press, 1962.

McJONES, P. **The LISP 2 Project**. Washington: IEEE Computer Society, 2017.

MIRANKER, D. P. **TREAT: A new and efficient match algorithm for AI production systems**. California: Morgan Kaufmann Publishers, Inc. 1990.

MOON, D. A. **MacLISP reference manual**. Massachusetts: MIT Press, 1974.

- NORVIG, P. **Paradigms of artificial intelligence programming: Case studies in COMMON LISP.** California: Morgan Kaufmann Publishers, Inc. 1992.
- QUARESMA, P. **ANSI and GNU Common Lisp Document.** Coimbra: DEMUC - Departamento Matemática Universidade de Coimbra, 2018. Disponível em: <http://www.mat.uc.pt/~pedro/cientificos/funcional/lisp/gcl_toc.html>. Acesso em: 17 mar. 2019.
- SEIBEL, P. **Practical Common LISP.** New York: Apress, 2005.
- SHAPIRO, S. C. **Common LISP: An Interactive Approach (Principles of Computer Science Series).** New Yors: W H Freeman & Co, 1991.
- STEELE Jr. G. L. **Common LISP: The language.** Massachusetts: Digital Press, 1990.
- SYMBOLICS. **Symbolics Common Lisp: Language Concepts.** Massachusetts: Symbolics, Inc., 1986.
- TEITELMAN, W. **InterLISP reference manual.** California: XEROX, 1974.
- TIERNEY, L. **LISP-STAT: An object-oriented environment for statistical computing and dynamic graphics.** New York: John Wiley & Sons, 1990.
- TOURETZKY, D. S. **Common LISP: A gentle introduction to symbolic computation.** California: The Benjamin/Cummings, 2013.
- WEITZ, E. **Common LISP Recipes: A problem-solution approach.** New York: Apress, 2016.
- YUASA, T. & HAGIYA, M. **Introduction to Common LISP.** California: Morgan Kaufmann. 1987.

