UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

LICENCIATURA DE ENGENHARIA INFORMÁTICA E DE COMPUTADORES

PROJECTO DE FUNDAMENTOS DE PROGRAMAÇÃO 2005 / 2006

DOCUMENTAÇÃO TÉCNICA

APL – A Programming Language

21 de Dezembro de 2005

André Matias nº 56907

Pedro Pinto nº 57006

Índice

Introdução

O programa "APL – "A Programming Language" é a implementação, em Scheme, de uma variante da linguagem APL que, embora seguindo a sintaxe e semântica da linguagem Scheme, permite assim a utilização de um vasto conjunto de operações de APL.

APL caracteriza-se por ser uma linguagem para processamento de *arrays*. Em termos matemáticos, um *array* é denominado tensor e possui como casos particulares, os escalares, os vectores e as matrizes. Através da utilização de *arrays*, é possível operar sobre múltiplos valores em simultâneo, dispensando as estruturas de controlo tradicionais (selecção, iteração e recursão) que são necessárias nas linguagens que apenas conseguem lidar com um valor de cada vez. O facto de a linguagem operar sobre vários valores em simultâneo permite também tirar partido da inerente paralelização dessas operações, podendo ser extraordinariamente eficiente em modernas arquitecturas vectorizadas.

Para isso, conforme enunciado do projecto, será necessário definir em Scheme:

1. Um tipo de dados capaz de representar tensores, bem como os seus casos particulares escalares, vectores e matrizes. Apenas é necessário que os tensores consigam aglomerar números ou booleanos (i.e., os inteiros 0 e 1).
2. Um conjunto de procedimentos que, operando sobre o tipo de dados definido no ponto anterior, implementem as operações correspondentes existentes em APL.

Assim iremos abordar neste relatório, os pontos fulcrais no desenvolvimento do programa para que fique totalmente explicito de uma forma a poder introduzir-se mais operações ou ser facilmente alterado por outro programador.

Começamos por identificar e descrever detalhadamente o TAI (tipo abstracto de informação). O TAI implementado é o "Tensor".

Seguidamente apresentamos os principais algoritmos que sustentam o programa, algoritmos estes que permitem realizar as tarefas sobre APL. Estes algoritmos estão divididos em dois:

- Funções

- Operadores

Após estes algoritmos apresentamos alguns exemplos de interacção ao programa, que demonstram as suas funcionalidades e capacidades, bem como algumas limitações. Depois dos testes apresentamos as conclusões que tirámos da realização do trabalho, onde incluímos também uma análise das limitações do nosso programa e as dificuldades surgidas ao longo da realização deste.

Tipos de Dados Implementados

Neste ponto vamos definir o tipo abstracto de informação utilizado na implementação do projecto. O TAI utilizado é o tipo "Tensor".

Tipo "Tensor"

Operações Básicas:

Construtores

- *v : universal*$^k \mapsto$ *vector*

  v(obj$_1$*,…,obj*$_k$*)* devolve um vector com *k* elementos, preenchido com os elementos *obj*$_1$*… obj*$_k$.

- *cria-vector : inteiro* × *universal* $\mapsto$ *vector*

  cria-tensor(k,obj) devolve um vector com as dimensões referidas no inteiro *k*, preenchido com o elemento *obj*.

- insere-*vector : universal* × *vector* $\mapsto$ *vector*

  insere-vector(obj,k,v) devolve um vector que resulta de inserir o elemento *obj* na primeira posição do vector *v*.

- *cria-tensor : vector* × *universal* $\mapsto$ *tensor*

  cria-tensor(v,obj) devolve um tensor com as dimensões referidas no inteiro *v*, preenchido com o elemento *obj*.

- *junta-tensor : tensor* × *tensor* $\mapsto$ *tensor*

  junta-tensor(t1,t2) no caso de os dois argumentos serem escalares, devolve um vector contendo esses argumentos, no caso de os dois argumentos serem tensores, devolve um tensor que junta o tensor *t1* e tensor *t2* ao longo da sua ultima dimensão.

Selectores

- *acede-tensor : tensor* × *inteiro* $\mapsto$ *elemento*

acede-tensor(t,k) tem como valor o elemento na posição *k* da primeira da primeira dimensão do tensor *t*. Se o tensor *t* é um escalar, ou *k* for inferior a um ou superior ao número de elementos da primeira dimensão do tensor *t*, o valor da operação é indefinido.

- *resto-tensor : tensor* × *inteiro* ↦ *tensor*
  resto-tensor(t,k) tem como valor o tensor que resulta de remover o elemento que se encontra na posição *k* da primeira dimensão do tensor *t*. Se o tensor *t* é um escalar, ou *k* for inferior a um ou superior ao número de elementos da primeira dimensão do tensor *t*, o valor da operação é indefinido.

- remove-*tensor : tensor* × *inteiro* ↦ *inteiro*
  remove-tensor (t,k) devolve um tensor onde foram removidos *k* elementos do início da primeira dimensão (se $k > 0$), ou fim (se $k < 0$).

- *comp-tensor : tensor* ↦ *inteiro*
  comp-tensor(t) tem como valor o número de elementos da primeira dimensão do tensor *t*. Se o tensor *t* for um escalar o valor desta operação é indefinido.

- *forma-tensor : tensor* ↦ *vector*
  forma-tensor(t) tem como valor um vector cujos elementos correspondem ao comprimento de cada dimensão do tensor *t*. Se o tensor *t* for um escalar, devolve um vector vazio.

Modificadores

- *altera-tensor! : tensor* × *tensor* ↦ *tensor*
  altera-tensor!(t1,t2) muda os elementos do tensor *t1* e devolve o tensor *t1*. Os elementos são obtidos através da enumeração dos elementos do tensor *t2* segundo argumento, repetindo essa enumeração as vezes que forem necessárias para preencher o tensor *t1*.

Reconhecedores

– *escalar? : universal* $\mapsto$ *lógico*

escalar?(obj) tem o valor *verdadeiro*, se *obj* é um escalar, e tem valor *falso* caso contrário.

– *vector? : universal* $\mapsto$ *lógico*

vector?(obj) tem o valor *verdadeiro*, se *obj* é um vector, e tem valor *falso* caso contrário.

– *vector-vazio? : vector* $\mapsto$ *lógico*

vector?(v) tem o valor *verdadeiro*, se o vector *v* é um vector vazio, e tem valor *falso* caso contrário.

– *matriz? : universal* $\mapsto$ *lógico*

matriz?(obj) tem o valor *verdadeiro*, se *obj* é um matriz, e tem valor *falso* caso contrário.

– *tensor? : universal* $\mapsto$ *lógico*

tensor?(obj) tem o valor *verdadeiro*, se *obj* é um tensor, e tem valor *falso* caso contrário.

Testes

– *tensor=? : tensor* $\times$ *tensor* $\mapsto$ *lógico*

tensor=?(t1,t2) tem o valor *verdadeiro* se o tensor *t1* é igual ao tensor *t2*, e tem valor falso caso contrario.

– *membro-vector? : universal* $\times$ *tensor* $\mapsto$ *lógico*

membro-vector?(obj,v) tem o valor *verdadeiro* se o *obj* existe no vector, e tem valor falso caso contrario.

Transformadores

– *mapa-tensor : procedimento* $\times$ *tensor*^{k} $\mapsto$ *tensor*

mapa-tensor(proc,t1,…,t_k) devolve um tensor com as mesmas dimensões dos seus argumentos, se for dado apenas um procedimento e um tensor como argumento, aplica o procedimento a cada um dos elementos da última dimensão do tensor, se for dado mais que um tensor como argumento, aplica o procedimento a cada elemento das últimas dimensões dos tensores na mesma posição. A ordem em que o procedimento é aplicado aos elementos não é especificada. Se os tensores argumento não tiverem as mesmas dimensões, o resultado é indefinido.

- *mapa-dim : procedimento* $\times$ *tensor*$^k \mapsto$ *tensor*

  mapa-dim(proc,t1,…,t_k) devolve um tensor com as mesmas dimensões do seus argumentos, se for dado apenas um procedimento e um tensor como argumento, aplica o procedimento a cada um dos elementos da primeira dimensão do tensor, se for dado mais que um tensor como argumento, aplica o procedimento a cada elemento das primeiras dimensões dos tensores na mesma posição. A ordem em que o procedimento é aplicado aos elementos não é especificada. Se os tensores argumento não tiverem as mesmas dimensões, o resultado é indefinido.

- *funcao-tensor : procedimento* $\times$ *tensor* $\times$ *tensor* $\mapsto$ *tensor*

  funcao-tensor(proc,t1,t2) devolve um tensor que resulta de aplicar o procedimento aos elementos dos tensores argumentos. Se os argumentos forem tensores do mesmo tamanho e forma, o tensor resultado tem o mesmo tamanho e forma dos argumentos e tem, como elementos, o resultado de aplicar o procedimento aos elementos correspondentes dos tensores argumentos. Se um dos argumentos for um escalar, o tensor resultado tem o mesmo tamanho e forma que o outro e tem, como elementos, o resultado de aplicar o procedimento do argumento escalar com os elementos do outro argumento. Em qualquer outro caso, o resultado é um erro.

- *escreve-tensor : tensor* $\mapsto$ *void*

  escreve-tensor(t) escreve o tensor *t*.

– *tensor->vector : tensor ↦ vector*

tensor->vector(t) transforma o tensor *t* num vector.

Representação do Tipo:

Uma vez definidas as operações básicas para o tipo "Tensor", pensámos numa representação interna. Tendo em atenção que:

- Se o tensor tem zero dimensões, então é um escalar
- Se o tensor tem uma dimensão, então é um vector
- Se o tensor tem duas dimensões, então é uma matriz
- No caso geral, dizemos apenas que é um tensor com um dado número de dimensões.

Assim representaremos do seguinte modo tensores:

1. Um escalar é um real.
2. Um vector vazio é representado pelo objecto computacional ( ).
3. Um vector é uma lista, em que cada um dos elementos é um escalar.
4. Uma matriz é uma lista, em que cada um dos elementos é um vector. Todos os vectores têm dimensões iguais.
5. Para o caso geral, um tensor é uma lista, em que cada um dos seus elementos é outra lista, e por sua vez cada um desses elementos é uma lista, etc., até que cada um dos elementos dessas listas seja um escalar. Todos os subtensores têm dimensões iguais.

De forma a simplificar a representação gráfica das listas, cada elemento de uma lista apresenta o comportamento de uma caixa com um ponteiro a apontar para ele. Sendo um elemento vazio representado uma caixa com uma diagonal a cheio.

Nesta figura, e nas seguintes, mostram-se exemplos de tensores (consideramos X_1, X_2, …, X_n, como escalares)

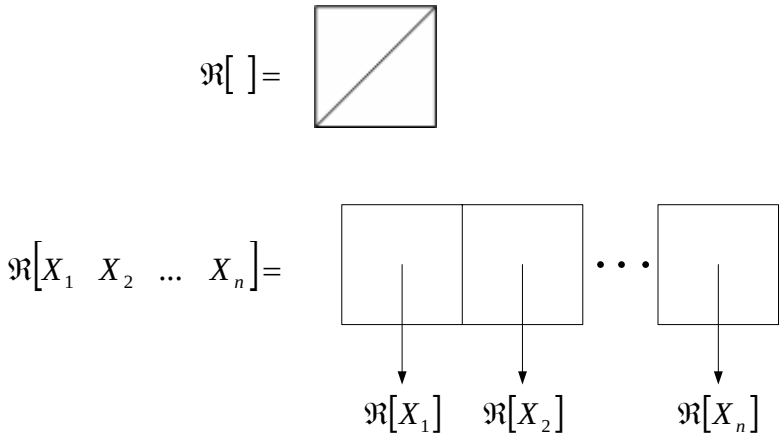

Figura 1: Representação de vectores.

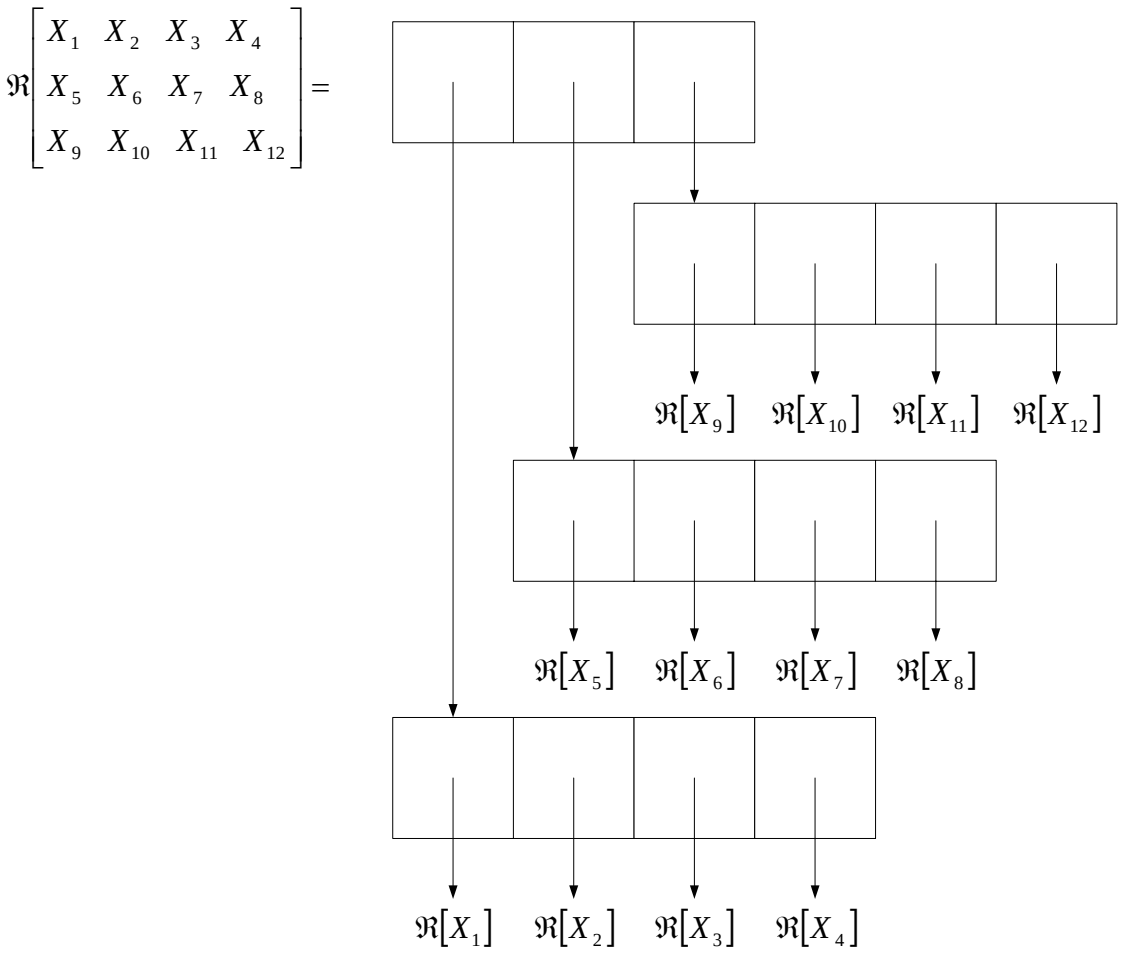

Figura 2: Representação de uma matriz 3 por 4.

Para representar o tipo “Tensor” decidimos usar listas, no entanto poderíamos ter feito o mesmo usando vectores. Optamos pelas listas por estarem optimizadas para mudanças de dimensão. Isto não acontece com os vectores, que estão optimizados para o acesso e procura, e como tal perderíamos eficiência ao usa-los em comparação com as listas.

Principais Algoritmos Usados no Programa

Neste ponto serão descritos os principais algoritmos usados no projecto, nomeadamente os seguintes:

Funções

Funções Monádicas

- *symmetrical : tensor* $\mapsto$ *tensor*

  symmetrical(t) produz um tensor cujos elementos são o simétrico dos elementos correspondentes do tensor argumento.

Após a compreensão do *symmetrical* como outras operações monádicas (*inverse, !, sin, cos, ~*), facilmente se chega à conclusão que existe um denominador comum, a operação básica mapa-tensor. Esta operação irá aplicar uma função a cada um dos elementos da última dimensão do tensor, e devolve o tensor resultante dessa operação. Logo as operações monádicas em acima referidas podem ser definidas à custa da seguinte forma:

- *chama o procedimento mapa-tensor, passa como argumento a função a a aplicar a cada um dos elementos do tensor, e o tensor, e devolve o tensor com os elementos após ter sido aplicada a função.*

No entanto existem outros tipos de funções monádicas, como por exemplo o interval. Neste caso em vez de receber um tensor, o interval irá receber um inteiro e vai construir um vector.

- *interval : inteiro* $\mapsto$ *vector*

  interval(k) produz um vector cujos elementos correspondem a uma enumeração dos inteiros desde 1 até ao escalar argumento.

O interval vai receber um escalar e cria um vector vazio, e vai inserindo elementos enumerados dos inteiros desde 1 até ao escalar argumento.

Funções Diádicas

– *+ : tensor* × *tensor* ↦ *tensor*

+(t1,t2) produz um tensor com a soma dos elementos dos tensores argumentos. Se os argumentos forem tensores com o mesmo tamanho e forma, o tensor resultado tem o mesmo tamanho e forma dos argumentos e tem, como elementos, a soma dos elementos correspondentes dos tensores argumentos. Se um dos argumentos for um escalar, o tensor resultado tem o mesmo tamanho e forma que o outro argumento e tem, como elementos, a soma do argumento escalar com os elementos do outro argumento. Em qualquer outro caso, a soma é um erro.

Após a compreensão do + como outras operações diádicas (-, *, /, *quotient*, r*emainder*), facilmente se chega à conclusão que existe um denominador comum, a operação básica funcao-tensor. Esta operação é similar ao mapa-tensor, no entanto verifica se a função for aplicada com um escalar e um tensor não escalar, se isto acontecer irá aplicar a função ao escalar com cada um dos elementos do tensor, caso contrário funciona de forma igual ao mapa-tensor, ou seja, aplica a função a cada um dos elementos dos tensores correspondentes.

As funções diádicas que operam com tensores de booleanos funcionam de forma análoga ao +, no entanto tem 0 e 1 em vez de reais. Essas funções são o <, >, <=, >=, =, | |, &&.

Operadores

Operadores Monádicos

Definimos três operadores monádicos, o *fold*, o *scan* e o *outer-product*, o *fold* recebe uma função como argumento e devolve outra função, que dado um vector avalia o resultado a inserir a função entre cada dois elementos do vector.

O scan, irá usar o operador *fold*, logo está dependente deste, e irá aplica-lo ao vector constituído pelo primeiro elemento do vector passado como argumento, em seguida irá repetir o mesmo para o vector constituído pelos dois primeiros elementos usando o *fold*, irá continuar a faze-lo até aplicar o *fold* ao

vector constituído por todos os elementos do vector passado como argumento, e colocará o resultado na última posição do vector devolvido.

Outro operador monádico é o *outer-product*, que irá aplicar a função argumento a cada uma das combinações de valores do primeiro e segundo tensores. Para isso aplica um *mapa-tensor* ao primeiro tensor, que aplicará o procedimento *mapa-tensor* ao segundo tensor, que aplicará a função passada como argumento ao elemento do primeiro tensor e a cada um dos elementos do segundo tensor, fazendo assim todas as combinações possíveis.

Operadores Diádicos

No projecto apenas foi definido um operador diádico que é o *inner-product*, o *inner-product* recebe duas funções como argumentos e devolve uma função que, dados dois tensores, produz um novo tensor de acordo com a regra do produto interno algébrico mas substituindo a soma e a multiplicação ai empregues pela primeira e segunda funções, respectivamente.

Ao aplicar função devolvida pelo *inner-product*, essa função verificará se algum dos dois tensores é um escalar, e se isto acontecer aplica a segunda função de forma análoga ao procedimento +, no entanto em vez da soma será a segunda função.

Se isto não acontecer, verificar se são tensores com as mesmas dimensões, e se assim efectua o produto interno algébrico, se não verificar se são matrizes, ou um vector e uma matriz, e verifica se é possível efectuar o produto interno algébrico, se sim efectua. Se não o resultado é indeterminado.

Exemplos de Interacções

Como exemplo de código que usa o tipo de dados pedido no enunciado, apresentamos de seguida uma função que, dado um tensor qualquer devolve um tensor cujos elementos são o factorial dos elementos correspondentes do tensor argumento.

```
(define (! t)
  (letrec ((factorial (lambda (n)
                (if (igual n 0)
                    1
                    (multiplica n (factorial (subtrai n 1)))))))
    (mapa-tensor factorial t)))
```

Este procedimento produz, por exemplo, os seguintes resultados:

```
> (display-tensor (! (reshape (v 2 3) (interval 10))))
1 2 6
24 120 720
> (display-tensor (! 20))
2432902008176640000
```

Outro exemplo de código que usa o tipo de dados pedido no enunciado, é um procedimento *tally*, que dado um tensor qualquer devolve o número de elementos desse tensor.

Exemplo:

```
> (display-tensor (tally (reshape (v 3 3 2) (interval 5))))
18
```

Segue-se uma outra função denominada *rank* que usa o tipo de dados em questão, função esta que recebe um tensor e devolve um escalar com o número de dimensões do tensor argumento.

Exemplo:

```
> (display-tensor (rank (reshape (v 4 5 2) (interval 5))))
3
```

Como exemplo do código que usa o tipo de dados usado no programa, implementámos ainda uma outra função denominada *primes* que, dado um escalar, devolve um vector com todos os números primos desde 2 até esse escalar, inclusive.

Exemplo:

> (display-tensor (primes 50))

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

Conclusão

Para conseguirmos elaborar este projecto começámos por definir implementar o TAI definido. Foi necessário ter sempre em mente a separação por camadas de forma a não quebrarmos barreiras de abstracção.

Inicialmente o projecto era para ser feito apenas com conceitos subjacentes às linguagens funcionais, contudo, por razões de optimização, usámos alguns conceitos imperativos, nomeadamente o *set!* de forma a guardar valores já calculados.

Os maiores problemas encontrados estiveram directamente relacionados com a especificação dos requisitos, continha algumas falhas e alguns exemplos que poderiam ser muito melhores.

Apesar de ter sido pedida a entrega do código num único ficheiro, não foi esta a metodologia adoptada por nós no desenvolvimento. Criámos um ficheiro para o TAI e ainda outro contendo procedimentos meramente utilitários.

Aconselhamos o uso desta organização, uma vez que se torna mais fácil dividir e consolidar o trabalho entre os membros da equipa de desenvolvimento, além que facilita a gestão de ficheiros caso a equipa de desenvolvimento opte pela utilização de controlo de versões (não foi o caso).

Este programa apesar de se chamar “APL – A Programming Language”, apenas tem aplicação para algumas funções e operadores existentes na linguagem APL. Uma melhoria a fazer seria a inclusão de funções niládicas, assim como outros procedimentos existentes em APL.

Listagem do Código do Programa

```
1   ;
2   ;
3   ; Instituto Superior Tecnico
4   ; Licenciatura de Engenharia Informatica e de Computadores
5   ;
6   ; Projecto de Fundamentos de Programacao 2005/2006
7   ;
8   ; no 56907 - Andre dos Santos Matias
9   ; no 57006 - Pedro Miguel da Rocha Pinto
10
11  ; TAI Tensor
12
13  ;;;;;;;;;;;;;;;
14  ; CONSTRUTORES
15  ;;;;;;;;;;;;;;;
16
17  ; v : universalk -> vector
18  ;
19  ; v(obj1,...,objk) devolve um vector com k elementos, preenchido com os elementos
20  obj1...objk
21  ;
22  (define v list)
23
24  ; cria-vector : inteiro x universal -> vector
25  ;
26  ; cria-vector(k,obj) devolve um vector com as dimensoes referidas no inteiro k,
27  preenchido com o
28  ;                    elemento obj
29  ;
30  (define (cria-vector k obj)
31    (if (menor k 1)
32        ()
33        (cons obj (cria-vector (subtrai k 1) obj))))
34
35  ; insere-vector : universal x vector -> vector
36  ;
37  ; insere-vector(obj,k,v) devolve um vector que resulta de inserir o elemento obj na
38  primeira
39  ;                        posicao do vector v.
40  ;
41  (define (insere-vector obj v)
42    (cons obj v))
43
44  ; cria-tensor : vector x universal -> tensor
45  ;
46  ; cria-tensor(v,obj) devolve um tensor com as dimensões referidas no vector v, preenchido
47  com o
48  ;                    elemento obj
```

```
49  ;
50  (define (cria-tensor v obj)
51    (define (cria-tensor-aux v obj)
52      (cond ((null? v) obj)
53            ((null? (cdr v)) (cria-vector (car v) obj))
54            ((menor (car v) 1) ())
55            (else (cons (cria-tensor (cdr v) obj)
56                        (cria-tensor (cons (subtrai (car v) 1)
57                                           (cdr v)) obj)))))
58    (if (vector? v)
59        (cria-tensor-aux v obj)
60        (error "cria-tensor: o primeiro argumento nao e um vector")))
61
62  ; junta-tensor : tensor x tensor -> tensor
63  ;
64  ; junta-tensor(t1,t2) no caso de os dois argumentos serem escalares, devolve um vector
65  contendo
66  ;                     contendo esses argumentos, no caso de os dois argumentos serem
67  tensores,
68  ;                     devolve um tensor que junta o tensor t1 e tensor t2 ao longo da sua
69  ultima
70  ;                     dimensao
71  ;
72  (define (junta-tensor t1 t2)
73    (cond ((escalar? t1) (v t1 t2))
74          ((null? t1) t2)
75          ((vector? t1) (cons (car t1)
76                              (junta-tensor (cdr t1) t2)))
77          ((tensor=? (forma-tensor t1) (forma-tensor t2)) (map junta-tensor t1 t2))
78          (else (error "junta-tensor: os argumentos devem ter o mesmo tamanho"))))
79
80  ;;;;;;;;;;;;
81  ; SELECTORES
82  ;;;;;;;;;;;;
83
84  ; acede-tensor : tensor x inteiro -> elemento
85  ;
86  ; acede-tensor(t,k) tem como valor o elemento na posicao k da primeira dimensao do tensor
87  t. se o
88  ;                   tensor t é um escalar, ou k for inferior a um ou superior ao numero
89  de elementos
90  ;                   da primeira dimensao do tensor t, o valor da operacao e indefinido
91  ;
92  (define (acede-tensor t k)
93    (cond ((null? t) (error "acede-tensor: posicao fora da lista"))
94          ((igual k 1) (car t))
95          (else (acede-tensor (cdr t) (subtrai k 1)))))
96
97  ; resto-tensor : tensor x inteiro -> tensor
98  ;
99  ; resto-tensor(t,k) tem como valor o tensor que resulta de remover o elemento que se
```

```
100     encontra na
101     ;                   posicao k da primeira dimensao do tensor t. se o tensor t é um
102     escalar, ou k
103     ;                   for inferior a um ou superior ao numero de elementos da primeira
104     dimensao do
105     ;                   tensor t, o valor da operacao e indefinido
106     ;
107     (define (resto-tensor t k)
108       (cond ((null? t) (error "resto-tensor: posicao fora do tensor"))
109             ((igual k 1) (cdr t))
110             (else (cons (car t)
111                         (resto-tensor (cdr t) (subtrai k 1))))))
112
113     ; remove-tensor : tensor x inteiro -> tensor
114     ;
115     ; remove-tensor(t,k) devolve um tensor onde foram removidos k elementos do inicio da
116     primeira dimensao
117     ;                    (se k > 0), ou fim (se k < 0)
118     ;
119     (define (remove-tensor t n)
120       (let ((comp (length t)))
121         (cond ((and (maior n 0) (menor-igual n comp))
122                (list-tail t n))
123               ((and (menor n 0) (maior-igual n (subtrai comp)))
124                (reverse (list-tail (reverse t) (multiplica n -1))))
125               (else (error "remove-tensor: posicao fora do tensor")))))
126
127     ; comp-tensor : tensor -> inteiro
128     ;
129     ; comp-tensor(t) tem como valor o numero de elementos na primeira dimensao do tensor t.
130     se o tensor t
131     ;                for um é escalar o valor desta operacao e indefinido
132     ;
133     (define (comp-tensor t)
134       (if (null? t)
135           0
136           (soma 1 (comp-tensor (cdr t)))))
137
138     ; forma-tensor : tensor -> vector
139     ;
140     ; forma-tensor(t) tem como valor um vector cujos elementos correspondem ao comprimento de
141     cada
142     ;                 dimensao do tensor t. se o tensor t for um escalar, devolve um vector
143     vazio
144     ;
145     (define (forma-tensor t)
146         (cond ((number? t) ())
147               ((vector? t) (v (comp-tensor t)))
148               (else (junta-tensor (v (comp-tensor t)) (forma-tensor (car t))))))
149
150     ;;;;;;;;;;;;;;;
```

```
151     ; MODIFICADORES
152     ;;;;;;;;;;;;;;;
153
154     ; altera-tensor! : tensor x tensor -> tensor
155     ;
156     ; altera-tensor!(t1,t2) muda os elementos do tensor t1 e devolve o tensor t1. os
157     elementos sao
158     ;                       obtidos atraves da enumeracao dos elementos do tensor t2 segundo
159     argumento,
160     ;                       repetindo essa enumeracao as vezes que forem necessarias para
161     preencher o
162     ;                       tensor t1
163     ;
164     (define (altera-tensor! t1 t2)
165       (let* ((vect (tensor->vector t2))
166              (comp (comp-tensor vect))
167              (i 0))
168         (mapa-tensor (lambda (x)
169                        (if (igual i comp)
170                            (set! i 0))
171                        (set! i (soma i 1))
172                        (acede-tensor vect i))
173                      t1)))
174
175     ;;;;;;;;;;;;;;;;
176     ; RECONHECEDORES
177     ;;;;;;;;;;;;;;;;
178
179     ; escalar? : universal -> logico
180     ;
181     ; escalar?(obj) tem o valor verdadeiro, se obj é um escalar, e tem o valor falso caso
182     contrario
183     ;
184     (define escalar? number?)
185
186     ; vector? : universal -> logico
187     ;
188     ; vector?(obj) tem o valor verdadeiro, se obj é um vector, e tem o valor falso caso
189     contrario
190     ;
191     (define (vector? obj)
192       (or (null? obj)
193           (and (list? obj)
194                (number? (car obj))
195                (vector? (cdr obj)))))
196
197     ; vector-vazio? : vector -> logico
198     ;
199     ; vector-vazio?(v) tem o valor verdadeiro, se o vector v é vector vazio, e tem o valor
200     falso caso
201     ;                   contrario
```

```
202     ;
203     (define vector-vazio? null?)
204
205     ; matriz? : universal -> logico
206     ;
207     ; matriz?(obj) tem o valor verdadeiro, se obj é uma matriz, e tem o valor falso caso
208     contrario
209     ;
210     (define (matriz? obj)
211       (define (matriz?-aux obj forma)
212         (if (not (null? obj))
213             (and (vector? (car obj))
214                  (equal? forma (forma-tensor (car obj)))
215                  (matriz?-aux (cdr obj) forma))
216             #t))
217       (if (and (list? obj)
218                (not (null? obj))
219                (vector? (car obj)))
220           (matriz?-aux (cdr obj) (forma-tensor (car obj)))
221           #f))
222
223     ; tensor? : universal -> logico
224     ;
225     ; tensor?(obj) tem o valor verdadeiro, se o obj é um tensor, e tem o valor falso caso
226     contrario
227     ;
228     (define (tensor? obj)
229       (define (tensor?-aux obj forma)
230         (if (not (null? obj))
231             (and (list (car obj))
232                  (equal? forma (forma-tensor (car obj)))
233                  (tensor?-aux (cdr obj) forma))
234             #t))
235       (or (number? obj)
236           (vector? obj)
237           (and (list? obj)
238                (if (vector? (car obj))
239                    (matriz? obj)
240                    (and (tensor? (car obj))
241                         (tensor?-aux (cdr obj) (forma-tensor (car obj))))))))
242
243     ;;;;;;;;
244     ; TESTES
245     ;;;;;;;;
246
247     ; tensor=? : tensor x tensor -> logico
248     ;
249     ; tensor=?(t1,t2) tem o valor verdadeiro se o tensor t1 é igual ao tensor t2, e tem valor
250     falso
251     ;                 caso contrario
252     ;
```

```
253  (define (tensor=? t1 t2)
254    (and (equal? (forma-tensor t1) (forma-tensor t2))
255         (not (membro-vector? 0 (tensor->vector (mapa-tensor (logico->escalar equal?) t1
256  t2))))))
257
258  ; membro-vector? : universal x vector -> logico
259  ;
260  ; membro-vector?(obj,v) tem o valor verdadeiro se o obj existe no vector, e tem valor
261  falso caso
262  ;                       contrario
263  ;
264  (define (membro-vector? obj v)
265    (if (membro obj v)
266       #t
267       #f))
268
269  ;;;;;;;;;;;;;;;;
270  ; TRANSFORMADORES
271  ;;;;;;;;;;;;;;;;
272
273  ; mapa-tensor : procedimento x tensork -> tensor
274  ;
275  ; mapa-tensor(proc,t1,...,tk) devolve um tensor com as mesmas dimensoes dos seus
276  argumentos, se
277  ;                             for dado apenas um procedimento e um tensor como argumento,
278  aplica
279  ;                             o procedimento a cada um dos elementos da ultima dimensao
280  do
281  ;                             tensor, se for dado mais que um tensor como argumento,
282  aplica o
283  ;                             procedimento a cada elemento das ultimas dimensoes dos
284  tensores
285  ;                             na mesma posicao. a ordem em que o procedimento é aplicado
286  aos
287  ;                             elementos não é especificada. se os tensores argumento nao
288  tiverem
289  ;                             as mesmas dimensoes, o resultado e indefinido
290  ;
291  (define (mapa-tensor proc . t)
292    (define (mapa-tensor-aux t)
293      (cond ((or (null? t) (null? (car t))) ())
294            ((vector? t) (apply proc t))
295            (else (cons (mapa-tensor-aux (map car t))
296                        (mapa-tensor-aux (map cdr t))))))
297    (cond ((or (number? t) (number? (car t))) (apply proc t))
298          ((null? (car t)) ())
299          ((vector? (car t)) (cons (apply proc (map car t))
300                                   (apply mapa-tensor (cons proc
301                                                            (map cdr t)))))
302          (else (mapa-tensor-aux t))))
303
```

```
304     ; mapa-dim : procedimento x tensork -> tensor
305     ;
306     ; mapa-dim(proc,t1,...,tk) devolve um tensor com as mesmas dimensoes dos seus argumentos,
307     se
308     ;                          for dado apenas um procedimento e um tensor como argumento,
309     aplica
310     ;                          o procedimento a cada um dos elementos da primeira dimensao do
311     ;                          tensor, se for dado mais que um tensor como argumento,  aplica
312     o
313     ;                          procedimento a cada elemento das primeiras dimensoes dos
314     tensores
315     ;                          na mesma posicao. a ordem em que o procedimento é aplicado aos
316     ;                          elementos não é especificada. se os tensores argumento nao
317     tiverem
318     ;                          as mesmas dimensoes, o resultado e indefinido
319     ;
320     (define mapa-dim map)
321
322     ; funcao-tensor : procedimento x tensor x tensor -> tensor
323     ;
324     ; funcao-tensor(proc,t1,t2) devolve um tensor que resulta de aplicar o procedimento aos
325     elementos dos
326     ;                           tensores argumentos. se os argumentos forem tensores do mesmo
327     tamanho e
328     ;                           forma, o tensor resultado tem o mesmo tamanho e forma dos
329     argumentos e tem,
330     ;                           como elementos, o resultado de aplicar o procedimento aos
331     elementos
332     ;                           correspondentes dos tensores argumentos. se um dos argumentos
333     for um ;                           escalar, o tensor resultado tem o mesmo tamanho e
334     forma que o outro e tem,
335     ;                           como elementos, o resultado de aplicar o procedimento do
336     argumento escalar
337     ;                           com os elementos do outro argumento. em qualquer outro caso,
338     o resultado e
339     ;                           um erro
340     ;
341     (define (funcao-tensor proc t1 t2)
342       (cond ((escalar? t1) (mapa-tensor (lambda (x) (proc t1 x)) t2))
343             ((escalar? t2) (mapa-tensor (lambda (x) (proc x t2)) t1))
344             ((tensor=? (forma-tensor t1) (forma-tensor t2)) (mapa-tensor proc t1 t2))
345             (else (error "funcao-tensor: os argumentos nao tem a mesma forma"))))
346
347
348     ; escreve-tensor : tensor -> void
349     ;
350     ; escreve-tensor(t) escreve o tensor t
351     ;
352     (define (escreve-tensor t)
353       (define (escreve-vector t)
354         (define (escreve-vector-aux t)
```

```
355           (if (not (null? (cdr t)))
356               (begin (display (car t))
357                      (display " ")
358                      (escreve-vector-aux (cdr t)))
359               (display (car t))))
360         (if (not (null? t))
361             (escreve-vector-aux t)))
362       (define (escreve-tensor-aux t s)
363         (cond ((number? t) (display t))
364               ((vector? t) (escreve-vector t))
365               (else (if (not (null? (cdr t)))
366                         (if (not (null? (cdr s)))
367                             (begin (escreve-tensor-aux (car t) (cdr s))
368                                    (escreve-n (subtrai (comp-tensor s) 2))
369                                    (escreve-tensor-aux (cdr t) s)))
370                         (escreve-tensor-aux (car t) (cdr s))))))
371       (define (escreve-n n)
372         (if (or (igual n 0) (menor n 0))
373             (newline)
374             (begin (newline)
375                    (escreve-n (subtrai n 1)))))
376       (escreve-tensor-aux t (forma-tensor t)))
377
378     ; tensor->vector : tensor -> vector
379     ;
380     ; tensor->vector(t) tranforma o tensor t num vector
381     ;
382     (define (tensor->vector t)
383       (cond ((number? t) (v t))
384             ((vector? t) t)
385             (else (junta-tensor (tensor->vector (car t))
386                                 (tensor->vector (cdr t))))))
387
388     (define igual =)
389     (define soma +)
390     (define subtrai -)
391     (define multiplica *)
392     (define divide /)
393     (define div-inteira quotient)
394     (define div-resto remainder)
395     (define raizq sqrt)
396     (define seno sin)
397     (define coseno cos)
398     (define menor <)
399     (define maior >)
400     (define menor-igual <=)
401     (define maior-igual >=)
402     (define membro member)
403
404     ; logico->escalar
405     ;
```

```
406     (define (logico->escalar proc)
407       (lambda (x y)
408         (if (proc x y)
409             1
410             0)))
411
412     ;; Funções Monádicas
413
414     ; display-tensor
415     ;
416     (define (display-tensor t)
417       (escreve-tensor t))
418
419
420     ; symmetrical
421     ;
422     (define (symmetrical t)
423       (mapa-tensor (lambda (x) (multiplica x -1)) t))
424
425     ; inverse
426     ;
427     (define (inverse t)
428       (mapa-tensor (lambda (x) (expt x -1)) t))
429
430     ; !
431     ;
432     (define (! t)
433       (letrec ((factorial (lambda (n)
434                             (if (igual n 0)
435                                 1
436                                 (multiplica n (factorial (subtrai n 1)))))))
437         (mapa-tensor factorial t)))
438
439     ; sqrt
440     ;
441     (define (sqrt t)
442       (mapa-tensor raizq t))
443
444     ; sin
445     ;
446     (define (sin t)
447       (mapa-tensor seno t))
448
449     ; cos
450     ;
451     (define (cos t)
452       (mapa-tensor coseno t))
453
454     ; ~
455     ;
456     (define (~ t)
```

```
457       (let ((negacao (lambda (n)
458                        (if (igual n 0)
459                            1
460                            0))))
461         (mapa-tensor negacao t)))
462
463     ; shape
464     ;
465     (define (shape t)
466       (forma-tensor t))
467
468     ; interval
469     ;
470     (define (interval n)
471       (define (interval-aux resto acc)
472         (if (igual resto 0)
473             (v)
474             (insere-vector acc (interval-aux (subtrai resto 1) (soma acc 1)))))
475       (interval-aux n 1))
476
477     ;; Funções Diádicas
478
479     ; +
480     ;
481     (define (+ t1 t2)
482       (funcao-tensor soma t1 t2))
483
484     ; -
485     ;
486     (define (- t1 t2)
487       (funcao-tensor subtrai t1 t2))
488
489     ; *
490     ;
491     (define (* t1 t2)
492       (funcao-tensor multiplica t1 t2))
493
494     ; /
495     ;
496     (define (/ t1 t2)
497       (funcao-tensor divide t1 t2))
498
499     ; quotient
500     ;
501     (define (quotient t1 t2)
502       (funcao-tensor div-inteira t1 t2))
503
504     ; reminder
505     ;
506     (define (remainder t1 t2)
507       (funcao-tensor div-resto t1 t2))
```

```
508
509     ; <
510     ;
511     (define (< t1 t2)
512       (funcao-tensor (logico->escalar menor) t1 t2))
513
514     ; >
515     ;
516     (define (> t1 t2)
517       (funcao-tensor (logico->escalar maior) t1 t2))
518
519     ; <=
520     ;
521     (define (<= t1 t2)
522       (funcao-tensor (logico->escalar menor-igual) t1 t2))
523
524     ; >=
525     ;
526     (define (>= t1 t2)
527       (funcao-tensor (logico->escalar maior-igual) t1 t2))
528
529     ; =
530     ;
531     (define (= t1 t2)
532       (funcao-tensor (logico->escalar igual) t1 t2))
533
534     ; ||
535     ;
536     (define (|| t1 t2)
537       (funcao-tensor (logico->escalar (lambda (x y) (not (and (zero? x) (zero? y))))) t1 t2))
538
539     ; &&
540     ;
541     (define (&& t1 t2)
542       (funcao-tensor (logico->escalar (lambda (x y) (not (or (zero? x) (zero? y))))) t1 t2))
543
544     ; drop
545     ;
546     (define (drop t1 t2)
547       (define (drop-aux t1 t2)
548         (cond ((escalar? t1) (remove-tensor t2 t1))
549               ((vector-vazio? (resto-tensor t1 1)) (remove-tensor t2 (acede-tensor t1 1)))
550               (else (mapa-dim (lambda (x)
551                                 (drop-aux (resto-tensor t1 1) x))
552                               (remove-tensor t2 (acede-tensor t1 1))))))
553       (if (escalar? t2)
554           (error "drop: o segundo argumento nao pode ser um escalar")
555           (drop-aux t1 t2)))
556     ; reshape
557     ;
558     (define (reshape v t)
```

```
559       (let ((t-final (cria-tensor v 0))
560             (t-aux (tensor->vector t)))
561         (altera-tensor! t-final t-aux)))
562
563     ; catenate
564     ;
565     (define (catenate t1 t2)
566       (junta-tensor t1 t2))
567
568     ; member
569     ;
570     (define (member t1 t2)
571       (let ((v (tensor->vector t2)))
572         (mapa-tensor (lambda (x)
573                        ((logico->escalar membro-vector?) x v))
574                      t1)))
575
576     ; select
577     ;
578     (define (select v t)
579       (letrec ((i 0)
580                (filtro-vector
581                 (lambda (proc? v1)
582                   (cond ((vector-vazio? v1) v1)
583                         ((proc? (acede-tensor v1 1))
584                          (insere-vector (acede-tensor v1 1)
585                                         (filtro-vector proc? (resto-tensor v1 1))))
586                         (else (filtro-vector proc? (resto-tensor v1 1))))))
587                (testa?
588                 (lambda (x)
589                   (if (igual i (comp-tensor v))
590                       (set! i 0))
591                   (set! i (soma i 1))
592                   (if (igual 0 (acede-tensor v i))
593                       #f
594                       #t))))
595         (cond ((escalar? t) ())
596               ((vector? t) (filtro-vector testa? t))
597               (else (mapa-dim (lambda (x)
598                                 (select v x))
599                               t)))))
600
601     ;; Operadores Monádicos
602
603     ; fold
604     ;
605     (define (fold proc)
606       (let* ((fold-aux (lambda (x)
607                          (if (vector-vazio? (resto-tensor x 1))
608                              (acede-tensor x 1)
609                              (proc (acede-tensor x 1) ((fold proc) (resto-tensor x 1)))))))
```

```
610         (lambda (x)
611           (if (vector-vazio? x)
612               (v)
613               (fold-aux x)))))
614
615     ; scan
616     ;
617     (define (scan proc)
618       (lambda (x)
619         (if (vector-vazio? x)
620             (v)
621             (junta-tensor ((scan proc) (resto-tensor x (comp-tensor x)))
622                           (v ((fold proc) x))))))
623
624     ; outer-product
625     ;
626     (define (outer-product proc)
627       (lambda (t1 t2)
628         (mapa-tensor (lambda (x)
629                (mapa-tensor (lambda (y)
630                       (proc x y))
631                     t2))
632              t1)))
633
634     ;; Operadores Diadicos
635
636     ; inner-product
637     ;
638     (define (inner-product proc1 proc2)
639       (define (transpose m)
640         (cond ((not (matriz? m)) (v))
641               ((vector-vazio? (resto-tensor (acede-tensor m 1) 1))
642                (insere-vector (mapa-dim (lambda (x) (acede-tensor x 1)) m)
643                               (v)))
644               (else (insere-vector (mapa-dim (lambda (x) (acede-tensor x 1)) m)
645                                    (transpose (mapa-dim (lambda (x) (resto-tensor x 1))
646     m))))))
647       (lambda (x y)
648         (cond ((or (escalar? x) (escalar? y)) (proc2 x y))
649               ((and (vector? x)
650                     (vector? y)
651                     (tensor=? (shape x) (shape y)))
652                (if (not (vector-vazio? x))
653                    ((fold proc1) (mapa-tensor proc2 x y))))
654               (else (mapa-dim (lambda (x)
655                                 (mapa-dim (lambda (y)
656                                             ((inner-product proc1 proc2) x y)) (transpose
657     y))) x)))))
658
659     ;; 4 Exercicios
660
```

```
661     ; 1 tally
662     ;
663     ; dado um tensor, devolve o numero de elementos desse tensor
664     ;
665     (define (tally t)
666       ((fold *) (shape (v t))))
667
668     ; 2 rank
669     ;
670     ; dado um tensor, devolve um escalar com o numero de dimensoes do tensor
671     ;
672     (define (rank t)
673       (reshape () (shape (shape t))))
674
675     ; 3 average
676     ;
677     ; dado um vector de numeros, devolve um escalar representado a media dos numeros do
678     vector
679     ;
680     (define (average v)
681       (/ ((fold +) v)  (reshape () (shape v))))
682
683     ; 4 within
684     ;
685     ; dado um vector de números e dois números, devolve um vector que apenas contém os
686     elementos
687     ; do vector cujo valor é maior ou igual a n1 e menor ou igual a n2
688     (define (within v n1 n2)
689       (let ((v2 (select (<= v n2) v)))
690         (select (>= v2 n1) v2)))
691
692     ; 5 substitute<
693     ;
694     ; dado um vector de numeros e dois numeros, devolve um vector em que todos os elementos
695     do vector
696     ; que sejam inferiores ao primeiro numero sao substituidos pelo segundo numero
697     ;
698     (define (substitute< v n1 n2)
699       (+ (* (>= v n1) v) (* (< v n1) n2)))
700
701     ; 6 ravel
702     ;
703     ; dado um tensor, devolve um vector contendo todos os elementos do tensor
704     ;
705     (define (ravel t)
706       (reshape (v ((fold *) (shape (v t)))) t))
707
708     ; 7 primes
709     ;
710     ; dado um escalar, devolve um vector contendo todos os numeros primos desde 2 ate esse
711     escalar,
```

```
712     ; inclusive
713     ;
714     (define (primes n)
715       (let ((x (drop 1 (interval n))))
716         (select (~ (member x ((outer-product *) x x))) x)))
717
```