UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Fundamentos da Programação
Enunciado de Projecto 2005–2006
APL – A Programming Language

1 Introdução

A linguagem APL (**A P**rogramming **L**anguage) é uma das mais antigas linguagens de programação, tendo sido inventada por Kenneth Iverson em 1957. A primeira implementação da linguagem ficou pronta em 1964.

À semelhança da linguagem Lisp (antecessora de Scheme), APL foi idealizada como uma linguagem matemática sem qualquer preocupação com a arquitectura dos computadores onde viria a ser executada. Esta característica que, a princípio, jogava contra a linguagem, permitiu-lhe sobreviver facilmente à evolução do hardware que ocorreu nas últimas quatro décadas existindo hoje várias linguagens (APL2, A+, APL2000, Dyalog APL, Sharp APL, MicroAPL, J, K, Nial, Glee, Nesl e outras) que se auto-proclamam descendentes directos da linguagem APL de Iverson.

APL caracteriza-se por ser uma linguagem para processamento de *arrays*. Um *array* é simplesmente um aglomerado de valores de um mesmo tipo, dispostos numa estrutura rectangular. 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 controle 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.

Tal como Lisp, APL é uma linguagem interactiva, no sentido de disponibilizar uma forma de interação com o programador que se baseia na avaliação de expressões que o programador vai fornecendo. A cada pedido do programador, APL responde escrevendo o resultado da avaliação do pedido.

Sendo uma linguagem orientada à manipulação de *arrays*, APL disponibiliza um conjunto muito grande de operações sobre *arrays*. Para além das tradicionais operações matemáticas sobre escalar, vectores e matrizes (soma, produto, produto interno, produto externo, transposta, etc) existem ainda inúmeras operações para criação e adaptação de *arrays* (alterar a forma, mudar o número de dimensões, pesquisar, indexar e remover elementos, etc). É de notar que, na linguagem original, a grande maioria destas operações emprega uma simbologia matemática (ρ, ι, $\times$, $\square$, etc).

Ao manipularem simultaneamente grandes aglomerados de valores, estas operações conferem a APL um enorme poder expressivo. Isto, conjuntamente com o uso de símbolos matemáticos, permite que os programas APL possam ser extremamente sucintos. A título de exemplo, eis um programa APL **completo** que encontra todos os números primos até um certo limite R:[1]

$$\sim R \in R \circ . \times R) / R \leftarrow 1 \downarrow \iota R$$

Apesar de ser uma linguagem extremamente poderosa, quando comparada com linguagens *mainstream* como Java ou C++, APL tem um conjunto muito pequeno de seguidores. Isto deve-se, por um lado, ao facto de a aprendizagem da linguagem ser um desafio intelectual que nem todos os programadores estão dispostos a fazer e, por outro, ao facto de a linguagem original depender de um conjunto de símbolos matemáticos que não era fácil de conseguir escrever nos teclados tradicionais (e que continua a não ser facil de escrever nos teclados modernos).

Para ultrapassar este último problema, algumas versões da linguagem APL permitem escrever os símbolos matemáticos por extenso (i.e., empregam `rho` no lugar de ρ, `iota` no lugar de ι, etc) ou utilizam nomes mais aproximados ao verdadeiro sentido da operação (por exemplo, `drop` no lugar de $\downarrow$ e `interval` no lugar de ι).

Nas secções seguintes vamos fazer uma introdução à linguagem e apresentar as suas operações com mais detalhe.[2]

2 Tutorial de APL

A linguagem APL é, à semelhança de Scheme, uma linguagem onde é fácil experimentar pequenos programas. Nas próximas secções vamos demonstrar a utilização dos operadores de APL através de sessões com o interpretador.

Antes, porém, vamos explicar alguns conceitos importantes.

[1] Não tente perceber o programa já. Depois de conhecer a sintaxe e semântica da linguagem APL ser-lhe-á mais fácil compreender o funcionamento do programa.

[2] A versão da linguagem que vamos apresentar é uma simplificação da versão original com vista a tornar mais fácil a sua compreensão.

2.1 Sintaxe e Semântica

APL lida com valores simples ou agrupados no que se denominam *arrays* (ou *tensores*). Estes valores são manipulados usando funções (o que inclui as tradicionais operações matemáticas de soma, subtracção, etc) e operadores, que são funções que manipulam outras funções.

Em APL, as expressões são avaliadas **da direita para a esquerda** e **todas as funções têm a mesma precedência** mas **os operadores têm mais precedência que as funções**. A utilização de parêntesis permite alterar a ordem de avaliação. Esta caraterística faz com que a expressão `5 * 6 - 2` tenha o valor `20` pois é avaliada como se fosse `5 * (6 - 2)`. Do mesmo modo, `5 - 7 - 2` tem o valor `0` (e não `-4`). Esta ordem de avaliação pode ser reformulada dizendo que o argumento à direita de uma função é tudo o que estiver à direita dele.[3]

Em APL, as funções (que incluem os tradicionais operadores matemáticos) podem ser de três tipos:

Niládicas São funções que não recebem argumentos. Não serão discutidas neste trabalho.

Monádicas São funções que recebem apenas um argumento. A sintaxe de invocação é da forma `função argumento`. Por exemplo, o factorial do número `5` escreve-se `! 5`. No caso de operadores, a sintaxe de invocação é da forma `argumento operador`. Por exemplo, na expressão `+ fold`, o operador é `fold` e o seu argumento é `+`.

Diádicas São funções que recebem dois argumentos. A sintaxe de invocação é da forma `argumento1 função argumento2`. Por exemplo, a soma dos números `2` e `3` escreve-se `2 + 3`. Os operadores seguem a mesma notação.

Em termos de tipos de dados elementares, APL caracteriza-se por ter apenas booleanos, caracteres, inteiros e números de vírgula flutuante. No entanto, estes tipos de dados podem estar estruturados em *tensores*, sendo que estes podem possuir qualquer número de dimensões.

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

[3] A razão de ser de se usar esta regra de avaliação (e não outra baseada em precedências, como é tradicional em matemática e noutras linguagens de programação) assenta no facto de APL possuir uma centena de operadores e ser impossível para um programador decorar precedências envolvendo um tão grande número de operadores.

Note-se que, em qualquer caso, todos os elementos de um tensor são de um único tipo. Isto quer dizer que não é possível misturar no mesmo tensor elementos de tipos diferentes, como seja criar um vector que contenha simultaneamente caracteres e inteiros.

As funções disponíveis na linguagem podem ser caracterizadas nos seguintes tipos:

Funções escalares São funções que recebem tensores como argumentos e operam sobre todos os elementos dos tensores, mas sem alterarem o tamanho (i.e., as dimensões) ou a forma (i.e., o número de dimensões) dos tensores.

Funções de reestruturação São funções que alteram o tamanho ou a forma dos tensores.

Funções mistas São funções que combinam os dois efeitos anteriores.

Operadores São funções de ordem superior que, a partir de funções, produzem outras funções.

No caso das funções escalares, elas operam quer sobre escalares, i.e., um único valor, quer sobre todos os elementos de um tensor, i.e., vários valores em simultâneo. Na realidade, estas funções operam apenas sobre tensores, sendo que um valor escalar é tratado como um caso particular de um tensor com zero dimensões.

2.2 Operadores Aritméticos

A título de exemplo, consideremos a função `+`. Eis um exemplo da interacção com o interpretador de APL:

```
APL> 1 + 3
4
```

O texto `APL>` representa a *prompt* do interpretador, indicando que está à espera de receber uma expressão para avaliar.

Eis mais alguns exemplos:

```
APL> 6 / 3
2
APL> 5 / 2
5/2
APL> 5.0 / 2
2.5
APL> 5 - 2
3
APL> 5 * 2
10
```

Uma das características mais interessantes de APL está no facto de lidar de forma igualmente fácil com escalares ou com vectores e matrizes. No primeiro exemplo, apenas se somaram os escalares `1` e `3` para produzir o escalar `4`. No entanto, era perfeitamente possível somar os vectores $[1,2,3]+[4,5,6]$ através da interacção;

```
APL> 1 2 3 + 4 5 6
5 7 9
```

É também possível realizar operações entre escalares e vectores ou matrizes.

```
APL> 1 + 1 2 3 4 5
2 3 4 5 6
APL> 10 9 8 7 6 5 - 5
5 4 3 2 1 0
```

2.3 Atribuição

Como é natural, é complicado estar a fazer contas sem poder guardar resultados intermédios. Para este fim, APL permite associar valores a nomes através do operador de atribuição `=:`. Eis alguns exemplos:

```
APL> pi =: 3.14159
3.14159
APL> raio =: 5
5
APL> p =: 2 * pi * raio
31.4159
APL> p =: 20 30 40 50
20 30 40 50
APL> p
20 30 40 50
APL> 1 + (n =: 99)
100
APL> n
99
```

2.4 Operadores Relacionais

Para além dos operadores aritméticos, APL permite ainda a utilização de operadores relacionais. No entanto, convém ter em conta que os valores lógicos falso e verdade são, respectivamente, `0` e `1` e que, para efeitos de operações lógicas, qualquer valor diferente de zero é tomado como verdadeiro. Assim, temos:

```
APL> 2 > 3
```

```
0
APL> 3 < 5
1
APL> x =: 1 2 3 4 5 6 7
1 2 3 4 5 6 7
APL> x > 3
0 0 0 1 1 1 1
APL> 5 > x
1 1 1 1 0 0 0
```

2.5 Operações de Ordem superior

APL também permite operações que recebem outras operações como argumento. Um dos operadores de ordem superior mais importante é `fold`, denominado *inserção* ou *redução*. Este operador insere a operação à sua esquerda entre cada dois elementos do argumento à sua direita. Eis alguns exemplos:

```
APL> + fold 1 2 3 4
10
APL> 1 + 2 + 3 + 4
10
APL> * fold 1 2 3 4
24
APL> 1 * 2 * 3 * 4
24
```

Na realidade, os operadores são funções especiais que modificam o comportamento de outras funções. Os operadores recebem funções como argumentos e produzem funções como resultados. Assim, na verdade, a expressão `+ fold 1 2 3 4` é reconhecido pelo APL como se tratasse de expressão `(+ fold) 1 2 3 4`.

O operador `scan` funciona de modo semelhante ao operador de *inserção* `fold`. Enquanto que `+ fold` calcula a soma de um vector de números, `+ scan` calcula as somas parciais de um vector, i.e.:

```
APL> + scan 1 2 3 4
1 3 6 10
```

Este operador pode ser visto como a aplicação do operador `fold` a fragmentos sucessivamente maiores do vector, como se pode comprovar pelo seguinte exemplo:

```
APL> + fold 1
1
APL> + fold 1 2
3
```

```
APL> + fold 1 2 3
6
APL> + fold 1 2 3 4
10
```

O facto de os valores lógicos serem `0` e `1` permite utilizar as tradicionais operações aritméticas para realizar operações lógicas. Por exemplo, dado um vector `x` de números, podemos perguntar se todos os números são negativos escrevendo:

```
APL> * fold x < 0
```

Outro exemplo é saber quantos dos seus elementos são superiores a um dado número, e.g., `5`:

```
APL> + fold x > 5
```

Finalmente, quantos números existem em `x`?[4]

```
APL> + fold x = x
```

2.6 Operações de Reestruturação

Consideremos agora duas das funções de reestruturação mais úteis. Originalmente denominadas ι e ρ vamos, por motivos de mais fácil utilização num computador, designá-las por `interval` e `reshape`.

```
APL> interval 7
1 2 3 4 5 6 7
APL> 2 3 reshape 1 2 3 4
1 2 3
4 1 2
APL> 2 3 reshape interval 4
1 2 3
4 1 2
APL> 2 2 reshape 1
1 1
1 1
APL> 2 2 2 reshape 1 2 3 4 5
1 2
3 4

5 1
2 3
```

[4]Para este último caso, podemos também usar a operação pré-definida `tally`.

A função monádica `interval` aplica-se a um número inteiro e produz um vector com todos os inteiros desde `1` até esse escalar (inclusive). A função diádica `reshape` cria um tensor com as dimensões indicadas no primeiro argumento (o vector à esquerda) e preenche-o com os elementos do segundo argumento (o vector à direita), repetindo-os as vezes que forem necessárias para preencher o tensor.

Note-se, no último exemplo, que a notação para escrever um tensor tri-dimensional é através de uma sequência de matrizes bi-dimensionais (denominadas *planos* do tensor). Cada plano é separado do plano seguinte por um número de mudanças de linha igual ao número de dimensões do tensor menos um.

Outra das funções de reestruturação mais utilizadas é a que remove os n primeiros (ou últimos, no caso de n ser negativo) elementos da de um tensor: `drop`. Eis uns exemplos:

```
APL> 2 drop interval 10
3 4 5 6 7 8 9 10
APL> -2 drop interval 10
1 2 3 4 5 6 7 8
```

Note-se que o operador `drop` pode remover elementos das várias dimensões de um tensor através da utilização de um vector como primeiro argumento:

```
APL> 1 drop 2 3 reshape interval 4
4 1 2
APL> 1 1 drop 2 3 reshape interval 4
1 2
APL> -1 drop 2 2 2 reshape 1 2 3 4 5
1 2
3 4
```

Na realidade, quando o primeiro argumento da função `drop` é um escalar, ele é tratado com um vector contendo apenas esse valor.

2.7 Outras Operações

Finalmente, apresentamos duas das operações mais úteis: o produto interno `inner-product` e o produto externo `outer-product`.

O operador `inner-product` recebe duas funções como argumentos e produz como resultado uma terceira função que, dadas duas matrizes, calcula o produto interno dessas duas matrizes mas, ao invés de empregar a soma dos produtos dos elementos (tal como em álgebra), emprega as funções argumentos. Eis alguns exemplos:

```
APL> m1 =: 2 2 reshape 10 20 30 40
```

```
10 20
30 40
APL> m2 =: 2 3 reshape 1 2 3 4 5 6
1 2 3
4 5 6
APL> m1 + inner-product * m2
90 120 150
190 260 330
APL> m1 * inner-product + m2
264 300 338
1364 1440 1518
APL> m1 + inner-product + m2
35 37 39
75 77 79
```

O operador `outer-product` recebe uma função como argumento e produz como resultado outra função que, dados dois tensores, produz um terceiro tensor que, para cada combinação com um elemento de cada um dos tensores, produz um tensor com o resultado de aplicar a função a esses dois elementos. Eis um exemplo:

```
APL> (interval 10) * outer-product interval 10
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

3 Objectivo do Projecto

O objectivo deste projecto é a implementação, em Scheme, de uma variante da linguagem APL que, embora seguindo a sintaxe e semântica da linguagem Scheme, permite a utilização de um vasto conjunto de operações de APL.

Para isso, terá de 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 exis-

tentes em APL. A subsecção seguinte especifica a sintaxe e semântica das operações a implementar.

Para além disto, terá de propor soluções para os exercícios de APL apresentados na próxima secção. As suas soluções para esses exercícios deverão empregar unicamente os procedimentos APL implementados e não poderão empregar quaisquer estruturas de controle (em particular, selecção, recursão ou iteração), para além das que já são usadas na implementação dos procedimentos APL que empregar.

3.1 Criação de Vectores

Uma das operações mais básicas que o seu projecto deverá disponibilizar é a que permite construir um vector a partir de um número arbitrário de elementos.

Essa operação denomina-se `v` e usa-se da seguinte forma:[5]

```
> (display-tensor (v 1 2 3 4 5 6))
1 2 3 4 5 6
```

3.2 Funções

O seu projecto deverá também implementar as funções que seguidamente descrevemos.

3.3 Funções Monádicas

display-tensor Recebe um tensor como argumento e escreve o conteúdo do tensor no ecran. A escrita do conteúdo deve obedecer às seguintes regras:

- Se o tensor é um escalar, escreve simplesmente o escalar.
- Se o tensor é um vector, escreve os elementos do vector todos na mesma linha e separados por um espaço em branco.
- Se o tensor é uma matriz, escreve as linhas da matriz como se estive a escrever vectores e muda de linha entre cada linha escrita.
- Se o tensor não é nenhum dos casos anteriores, então para cada sub-tensor da primeira dimensão, escreve o sub-tensor separado por um número de mudanças de linha igual ao número de dimensões do tensor menos um.

symmetrical Produz um tensor cujos elementos são o simétrico dos elementos correspondentes do tensor argumento.

inverse O mesmo comportamento que a anterior, mas empregando o inverso.

[5] A função `display-tensor` será apresentada adiante.

! O mesmo comportamento que a anterior, mas empregando o factorial.

sin O mesmo comportamento que a anterior, mas empregando o seno.

cos O mesmo comportamento que a anterior, mas empregando o cosseno.

~ O mesmo comportamento que a anterior, mas empregando a negação. O tensor resultante terá, como elementos, os inteiros `0` ou `1` consoante o elemento correspondente do tensor argumento é diferente de zero ou igual a zero, respectivamente.

Eis um exemplo da utilização das funções anteriores:[6]

```
> (display-tensor (reshape (v 2 2 2) (v 1 2 3 4 5)))
1 2
3 4

5 1
2 3
> (display-tensor (reshape (v 2 2 2 2) (v 1 2 3 4 5)))
1 2
3 4

5 1
2 3



4 5
1 2

3 4
5 1
> (display-tensor (symmetrical (! (v 4 3 2 1 0))))
-24 -6 -2 -1 -1
```

É também necessário implementar as funções que descrevemos em seguida.

shape Produz um vector cujos elementos correspondem ao comprimento de cada dimensão do tensor argumento.

interval Produz um vector cujos elementos correspondem a uma enumeração dos inteiros desde 1 até ao escalar argumento.

Eis alguns exemplos:

```
> (display-tensor (shape (v 1 2 3)))
```

[6] A função `reshape` tem a mesma semântica que a função homónima da linguagem APL e será especificada mais adiante.

```
3
> (display-tensor (shape (reshape (v 2 3) (v 1 2 3 4 5 6))))
2 3
> (display-tensor (shape (shape (reshape (v 2 3) (v 1 2 3 4 5 6)))))
2
> (display-tensor (interval 6))
1 2 3 4 5 6
> (display-tensor (reshape (v 3 3) (interval 6)))
1 2 3
4 5 6
1 2 3
```

3.3.1 Funções Diádicas

+ 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.

- O mesmo comportamento que a anterior, mas empregando subtracção.

***** O mesmo comportamento que a anterior, mas empregando multiplicação.

/ O mesmo comportamento que a anterior, mas empregando divisão.

quotient O mesmo comportamento que a anterior, mas empregando divisão inteira.

remainder O mesmo comportamento que a anterior, mas empregando o resto da divisão inteira.

< O mesmo comportamento que a anterior, mas empregando a relação "menor que." O tensor resultante terá, como elementos, os inteiros `0` ou `1` consoante a relação é falsa ou verdadeira.

> O mesmo comportamento que a anterior, mas empregando a relação "maior que."

<= O mesmo comportamento que a anterior, mas empregando a relação "menor ou igual que."

>= O mesmo comportamento que a anterior, mas empregando a relação "maior ou igual que."

= O mesmo comportamento que a anterior, mas empregando a relação "igual a."

|| O mesmo comportamento que a anterior, mas empregando a disjunção lógica.

&& O mesmo comportamento que a anterior, mas empregando a conjunção lógica.

Eis alguns exemplos:

```
> (display-tensor (+ (v 1 2 3) (v 4 5 6)))
5 7 9
> (display-tensor (+ 1 (v 4 5 6)))
5 6 7
> (display-tensor (> 5 (v 3 2 6 1 4 7 5)))
1 1 0 1 1 0 0
> (display-tensor (< (v 3 1 2) (v 2 3 1)))
0 1 0
> (let ((v (v 1 2 3 4 5 6 7 8 9)))
    (display-tensor (&& (< 3 v) (< v 5))))
0 0 0 1 0 0 0 0 0
```

Eis outro conjunto de funções a implementar:

drop Recebe um escalar n_1 ou vector (de elementos n_i) e um tensor não escalar e devolve um tensor onde foram removidos n elementos no início (se $n > 0$) ou fim (se $n < 0$) da dimensão i do tensor.

reshape Produz um tensor com as dimensões referidas no vector primeiro argumento e cujos elementos são obtidos através da enumeração dos elementos do tensor segundo argumento, repetindo essa enumeração as vezes que forem necessárias para preencher o tensor resultado.

catenate No caso de os dois argumentos serem escalares, produz um vector contendo esses argumentos. No caso de os dois argumentos serem tensores, produz um novo tensor que junta os outros dois ao longo da sua última dimensão.

member Recebe dois tensores como argumento e devolve um tensor de booleanos com a mesma forma e dimensão do primeiro argumento contendo um booleano `1` para cada elemento na posição correspondente do primeiro tensor que ocorra algures no segundo tensor e `0` em caso contrário.

select Recebe um vector de booleanos (i.e., contendo apenas os elementos `0` ou `1`) e um tensor e devolve um tensor contendo apenas os elementos da última dimensão do tensor argumento que possuem o valor `1` na posição correspondente do primeiro vector.

Eis alguns exemplos:

```
> (display-tensor (reshape (v 2 2 2) (v 1 2 3)))
1 2
3 1
```

```
2 3
1 2
> (display-tensor (catenate (v 1 2) (v 3 4 5)))
1 2 3 4 5
> (display-tensor (drop 2 (interval 10)))
3 4 5 6 7 8 9 10
> (display-tensor (drop -2 (interval 10)))
1 2 3 4 5 6 7 8
> (display-tensor (drop (v 1 1) (reshape (v 3 3) (interval 9))))
5 6
8 9
> (display-tensor (member (reshape (v 3 3) (interval 4)) (v 1 2)))
1 1 0
0 1 1
0 0 1
> (let ((v (v 1 6 2 7 3 0 5 4)))
    (display-tensor (select (> v 3) v)))
6 7 5 4
> (display-tensor
    (select (v 1 0 1)
            (reshape (v 2 3) (interval 6))))
1 3
4 6
```

3.4 Operadores

Como foi referido atrás, existe em APL uma categoria especial de funções denominadas *operadores*. Um *operador* é uma função que recebe outras operações como argumentos e devolve funções como resultados.

Vamos agora descrever os operadores que é necessário implementar no seu projecto.

3.4.1 Operadores Monádicos

`fold` O operador de *redução* `fold` recebe uma função como argumento e devolve outra função que, dado um vector, avalia o resultado de inserir a função entre cada dois elementos do vector.

`scan` O operador `scan` funciona de modo semelhante ao operador `fold` mas empregando conjuntos sucessivamente maiores de elementos, desde um conjunto com apenas o primeiro elemento até ao conjunto com todos os elementos.

`outer-product` Recebe uma função como argumento e devolve uma função que, dados dois tensores, produz um novo tensor com o resultado de aplicar a função argumento a cada uma das combinações de valores do primeiro e segundo tensores.

Eis alguns exemplos:

```
> (display-tensor ((fold +) (v 1 2 3 4)))
10
> (display-tensor ((fold *) (v 1 2 3 4)))
24
> (display-tensor ((scan +) (v 1 2 3 4)))
1 3 6 10
> (display-tensor ((scan *) (v 1 2 3 4)))
1 2 6 24
> (display-tensor
    ((outer-product =) (interval 4) (interval 4)))
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
> (display-tensor
    ((outer-product =) (v 4 7) (reshape (v 3 4) (interval 12))))
0 0 0 1
0 0 0 0
0 0 0 0

0 0 0 0
0 0 1 0
0 0 0 0
```

3.4.2 Operadores Diádicos

`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 substituíndo a soma e a multiplicação aí empregues pela primeira e segunda funções, respectivamente.

Exemplificando, temos:

```
> (display-tensor
    ((inner-product + *)
     (reshape (v 2 2) (v 10 20 30 40))
     (reshape (v 2 3) (v 1 2 3 4 5 6))))
90 120 150
190 260 330
> (display-tensor
    ((inner-product * +)
     (reshape (v 2 2) (v 10 20 30 40))
     (reshape (v 2 3) (v 1 2 3 4 5 6))))
264 300 338
1364 1440 1518
```

4 Exercícios

Para além da implementação das operações pedidas, deverá propor soluções para os exercícios apresentados nesta secção. As suas soluções deverão empregar, em exclusivo, os procedimentos definidos nos pontos anteriores. Note que as suas soluções para os exercícios não poderão empregar quaisquer estruturas de controle (em particular, selecção, recursão ou iteração).

1. Defina um procedimento `tally` que, dado um tensor, devolve o número de elementos desse tensor.

   Exemplo:

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

2. Defina um procedimento `rank` que, dado um tensor, devolve um escalar com o número de dimensões do tensor.

   Exemplo:

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

3. Defina um procedimento `average` que, dado um vector de números, devolve um escalar representando a média dos números do vector.

   Exemplo:

   ```
   > (display-tensor (average (interval 5)))
   3
   ```

4. Defina um procedimento `within` que, dado um vector de números *v* e dois números *n1* e *n2*, devolve um vector que apenas contém os elementos de *v* cujo valor é maior ou igual a *n1* e menor ou igual a *n2*.

   Exemplo:

   ```
   > (display-tensor (within (v 2 7 3 1 9 8 4 6 5) 5 8))
   7 8 6 5
   ```

5. Defina um procedimento `substitute<` que, dado um vector de números *v* e dois números *n1* e *n2*, devolve um vector em que todos os elementos de *v* que sejam inferiores a *n1* estão substituídos por *n2*.

   Exemplo:

   ```
   > (display-tensor (substitute< (v 2 7 3 1 9 8 4 6 5) 4 0))
   0 7 0 0 9 8 4 6 5
   ```

6. Defina o procedimento `ravel` que, dado um tensor, devolve um vector contendo todos os elementos do tensor.

   Exemplo:

```
> (display-tensor (ravel (reshape (v 2 3 4) (interval 10))))
1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10 1 2 3 4
```

7. Defina um procedimento `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
```

5 Classificação

A nota do projecto será baseada nos seguintes aspectos:

- Execução correcta (40%)
- Facilidade de leitura, abstracção procedimental, abstracção de dados, nomes escolhidos, paragrafação, qualidade dos comentários (25%)
- Documentação de acordo com o especificado nas aulas teóricas (30%)
- Estilo de programação (5%)

6 Condições de realização e prazos

O projecto deve ser realizado em grupos de 2 alunos.

A inscrição do grupo deve ser realizada pelos alunos e será disponibilizada na página da cadeira.

Os projectos devem ser entregues até às 15:00H do dia 21 de Dezembro de 2005. Projectos entregues até esta data terão uma bonificação de 0.5 valores. Os alunos que não conseguirem entregar o projecto a 21 de Dezembro, poderão fazer a sua entrega no dia 4 de Janeiro de 2006 até às 15:00, não tendo qualquer bonificação nem penalização. Projectos em atraso serão aceites durante a semana de 4 de Janeiro, sendo penalizados com meio valor por cada dia de atraso (a partir das 15:00 do dia 6 de Janeiro de 2006 não se aceitam projectos, seja qual for o pretexto).

O relatório do projecto deverá constar do seguinte:

- Documentação do programa de acordo com o especificado nas aulas teóricas, seguindo o modelo que será disponibilizado na página da cadeira;
- Uma listagem do programa contendo comentários, paragrafação e nomes adequados.

Atenção: A forma como o relatório é apresentado (sua relevância e clareza) é importante.

O relatório do projecto deve ser entregue dentro de uma capa ou encadernado, apresentando visivelmente o número do grupo, os números e os nomes dos seus autores. Projectos que não sejam entregues nestas condições serão penalizados com três valores.

Para além disto, a entrega do código por via electrónica é obrigatória e deverá ser feita até às 24:00H do dia em que é feita a entrega do relatório do projecto. Este código não pode conter diferenças significativas em relação à listagem que foi entregue com o relatório do projecto (apenas correcção de pequenos erros).

O código do projecto deve estar contido num único ficheiro. Se durante o desenvolvimento forem usados vários ficheiros, estes devem, antes de enviar o projecto, ser aglomerados num único ficheiro. Este ficheiro deve conter, no início, um comentário com o número do grupo, os números e os nomes dos autores.

A entrega electrónica deve ser feita de acordo com as instruções indicadas na página da cadeira.

Pode haver uma discussão oral do projecto e/ou uma demonstração do funcionamento do programa, o que será decidido caso a caso.

Projectos iguais, ou muito semelhantes, serão considerados cópias. As cópias que forem detectadas penalizam igualmente tanto o grupo que copia como o grupo que foi copiado e serão classificadas com 0 (zero) valores. O corpo docente da cadeira será o único juiz do que se considera ou não copiar no projecto.

Durante o desenvolvimento do projecto é importante não esquecer a Lei de Murphy:

- Todos os problemas são mais difíceis do que parecem;
- Tudo demora mais tempo do que nós pensamos;
- Se alguma coisa puder correr mal, ela vai correr mal, na pior altura possível.