Programação Vetorial

Tipos primitivos de dados e funções

Prof. Edson Alves

Faculdade UnB Gama

Sumário

Expressões em APL

- APL pode ser vista como uma notação matemática que também é executável por máquina

Expressões em APL

- APL pode ser vista como uma notação matemática que também é executável por máquina
- A linguagem é composta por funções, operadores, *arrays* e atribuições

Expressões em APL

- APL pode ser vista como uma notação matemática que também é executável por máquina
- A linguagem é composta por funções, operadores, *arrays* e atribuições
- Qualquer código que pode ser aplicado a dados é chamado função

Expressões em APL

- APL pode ser vista como uma notação matemática que também é executável por máquina
- A linguagem é composta por funções, operadores, *arrays* e atribuições
- Qualquer código que pode ser aplicado a dados é chamado função
- Dois exemplos de funções seriam a adição (+) e subtração (-)

Expressões em APL

- APL pode ser vista como uma notação matemática que também é executável por máquina
- A linguagem é composta por funções, operadores, *arrays* e atribuições
- Qualquer código que pode ser aplicado a dados é chamado função
- Dois exemplos de funções seriam a adição (+) e subtração (-)
- As funções de APL podem ser aplicadas monadicamente (prefixada, um operando) ou diadicamente (infixada, dois operando, um à esquerda e outro à direita)

Expressões em APL

- APL pode ser vista como uma notação matemática que também é executável por máquina
- A linguagem é composta por funções, operadores, *arrays* e atribuições
- Qualquer código que pode ser aplicado a dados é chamado função
- Dois exemplos de funções seriam a adição (+) e subtração (-)
- As funções de APL podem ser aplicadas monadicamente (prefixada, um operando) ou diadicamente (infixada, dois operando, um à esquerda e outro à direita)
- O tipo de dados mais elementar é o escalar (*array* de dimensão zero)

Inteiros

- Números são tratados internamente pela APL quanto ao tamanho e tipo e podem ser misturados sem problemas

Inteiros

- Números são tratados internamente pela APL quanto ao tamanho e tipo e podem ser misturados sem problemas
- Em APL os números podem ser inteiros, reais (em ponto flutuante) e números complexos

Inteiros

- Números são tratados internamente pela APL quanto ao tamanho e tipo e podem ser misturados sem problemas
- Em APL os números podem ser inteiros, reais (em ponto flutuante) e números complexos
- Um escalar inteiro pode ser grafado usando a notação decimal padrão:

```
    2 + 3
5
    2 × 3        ⍝ a multiplição é realizada pela função ×
6
```

Inteiros

- Números são tratados internamente pela APL quanto ao tamanho e tipo e podem ser misturados sem problemas
- Em APL os números podem ser inteiros, reais (em ponto flutuante) e números complexos
- Um escalar inteiro pode ser grafado usando a notação decimal padrão:

```
    2 + 3
5
    2 × 3        ⍝ a multiplição é realizada pela função ×
6
```

- Comentários são precedidos pelo símbolo ⍝

Inteiros

- Números são tratados internamente pela APL quanto ao tamanho e tipo e podem ser misturados sem problemas
- Em APL os números podem ser inteiros, reais (em ponto flutuante) e números complexos
- Um escalar inteiro pode ser grafado usando a notação decimal padrão:

```
    2 + 3
5
    2 × 3       ⍝ a multiplição é realizada pela função ×
6
```

- Comentários são precedidos pelo símbolo `⍝`
- Números negativos são precedidos pelo símbolo ¯ (*macron*)

```
    2 - 3
¯1
```

Novo símbolo

Símbolo	Aridade	Descrição
+ (*plus*)	diádico	Adição escalar

Unicode	TAB	APL
U+002B	-	-

Novo símbolo

Símbolo	Aridade	Descrição
− (*minus*)	diádico	Subtração escalar

Unicode	TAB	APL
U+002D	-	-

Novo símbolo

Símbolo	Aridade	Descrição
× (*times*)	diádico	Multiplicação escalar

Unicode	TAB	APL
U+00D7	x x <tab>	APL + -

Novo símbolo

Símbolo	Aridade	Descrição
¯ (*macron*)	monádico	Antecede um número negativo
Unicode	**TAB**	**APL**
U+00AF	- - <tab>	APL + 2

Novo símbolo

Símbolo	Aridade	Descrição
⍝ (*comment*)	monádico	Inicia um comentário. Tudo que o sucede até o fim da linha será considerado comentário
Unicode	**TAB**	**APL**
`U+235D`	`o n <tab>`	`APL + ,`

Números reais

- Em escalares reais, a parte inteira é separada das casas decimais por meio do ponto final

```
    0.2 ÷ 3.5
0.05714285714
```

Números reais

- Em escalares reais, a parte inteira é separada das casas decimais por meio do ponto final

```
    0.2 ÷ 3.5
0.05714285714
```

- APL também trata problemas de precisão de forma transparente ao usuário

```
    2÷3 ⋄ 6×2÷3
0.6666666667
4
```

Números reais

- Em escalares reais, a parte inteira é separada das casas decimais por meio do ponto final

```
    0.2 ÷ 3.5
0.05714285714
```

- APL também trata problemas de precisão de forma transparente ao usuário

```
    2÷3 ⋄ 6×2÷3
0.6666666667
4
```

- O símbolo ⋄ (*diamond*) separa duas expressões em uma mesma linha

Números reais

- Em escalares reais, a parte inteira é separada das casas decimais por meio do ponto final

```
    0.2 ÷ 3.5
0.05714285714
```

- APL também trata problemas de precisão de forma transparente ao usuário

```
    2÷3 ⋄ 6×2÷3
0.6666666667
4
```

- O símbolo ⋄ (*diamond*) separa duas expressões em uma mesma linha
- Notação científica pode representar números muito pequenos ou grandes

```
    2E¯3 ⋄ 5e7      ⍝ O E pode ser maiúsculo ou minúsculo
0.002
50000000
```

Novo símbolo

Símbolo	Aridade	Descrição
÷ (*divide*)	diádico	Divisão escalar. Divisão por zero resulta em um erro

Unicode	TAB	APL
U+00F7	: - <tab>	APL + =

Novo símbolo

Símbolo	Aridade	Descrição
⋄ (*diamond*)	diádico	Separador de expressões

Unicode	TAB	APL
`U+22C4`	`< > <tab>`	`APL + '`

Constantes booleanas

- Em APL: falso é igual a `0` (zero) e verdadeiro é igual a `1` (um)

```
    2 = 3
0
    5 = 5.0
1
```

Constantes booleanas

- Em APL: falso é igual a `0` (zero) e verdadeiro é igual a `1` (um)

```
    2 = 3
0
    5 = 5.0
1
```

- Os operadores relacionais retornam valores booleanos

```
    2 ≠ 3
0
    5 < 7
1
    11 > 13
0
    17 ≤ 19 ⋄ 23 ≥ 27
1
0
```

Novo símbolo

Símbolo	Aridade	Descrição
= (*equal*)	diádico	Igual a

Unicode	TAB	APL
`U+003D`	`-`	`APL + 5`

Novo símbolo

Símbolo	Aridade	Descrição
≠ (*not equal*)	diádico	Diferente de

Unicode	TAB	APL
`U+2260`	`= / <tab>`	`APL + 8`

Novo símbolo

Símbolo	Aridade	Descrição
< (*less than*)	diádico	Menor que

Unicode	TAB	APL
`U+003C`	`-`	`APL + 3`

Novo símbolo

Símbolo	Aridade	Descrição
> (*greater than*)	diádico	Maior que

Unicode	TAB	APL
`U+003E`	`-`	`APL + 7`

Novo símbolo

Símbolo	Aridade	Descrição
≤ (*less than or equal to*)	diádico	Menor ou igual a

Unicode	TAB	APL
U+2264	< = <tab>	APL + 4

Novo símbolo

Símbolo	Aridade	Descrição
≥ (*greater than or equal to*)	diádico	Maior ou igual a

Unicode	TAB	APL
`U+2265`	`> = <tab>`	`APL + 6`

Números complexos

- O caractere 'J' separa a parte real da parte imaginária em números complexos

```
    2J3 × 5j¯7          ⍝ O J também pode ser minúsculo
31J1
```

Números complexos

- O caractere 'J' separa a parte real da parte imaginária em números complexos

```
    2J3 × 5j¯7              ⍝ O J também pode ser minúsculo
31J1
```

- Lembre-se de que o argumento à direita de uma função diádica é o resultado de toda a expressão à direita do símbolo

```
    2 × 3 + 5               ⍝ equivale a 2 × (3 + 5)
16
    2 - 3 - 5 - 7 - 11      ⍝ 2 - (3 - (5 - (7 - 11)))
8
    2 ÷ 3 ÷ 5               ⍝ 10 ÷ 3
3.333333333
    2 × 0J1 * 3             ⍝ 2 × (j elevado a 3)
0J¯2
```

Novo símbolo

Símbolo	Aridade	Descrição
* (*power*)	diádico	Eleva o argumento à esquerda a potência indicada no argumento à direita

Unicode	TAB	APL
`U+002A`	`-`	`APL + p`

Caracteres e strings

- Em APL, strings são vetores de caracteres

Caracteres e strings

- Em APL, strings são vetores de caracteres
- Tanto caracteres quanto strings são delimitadas por aspas simples

```
    'c'                     ⍝ um caractere
c
    'uma string'
uma string
```

Caracteres e strings

- Em APL, strings são vetores de caracteres
- Tanto caracteres quanto strings são delimitadas por aspas simples

```
      'c'                 ⍝ um caractere
c
      'uma string'
uma string
```

- Atribuições podem ser feitas por meio do símbolo ←

```
      s ← 'abacate'
      'a' = s             ⍝ compara 'a' a cada caractere de s
1 0 1 0 1 0 0
      s ≠ 'abacaxi'       ⍝ compara caracteres em posições correspondentes
0 0 0 0 0 1 1
```

Novo símbolo

Símbolo	Aridade	Descrição
← (*assign*)	diádico	Atribui o argumento à direta ao argumento à esquerda

Unicode	TAB	APL
U+2190	< - <tab>	APL + '

Funções aritméticas monádicas

- As funções aritméticas apresentadas até o momento tem versões monádicas

```
      +2J3                  ⍝ conjugado complexo
2J¯3
      -¯2                   ⍝ simétrico aditivo
2
      ×2J3                  ⍝ vetor unitário na direção do complexo
0.5547001962J0.832050294
      ÷2                    ⍝ inverso multiplicativo
0.5
      *2                    ⍝ função exponencial
7.389056099
```

Funções aritméticas monádicas

- As funções aritméticas apresentadas até o momento tem versões monádicas

```
      +2J3                    ⍝ conjugado complexo
2J¯3
      -¯2                     ⍝ simétrico aditivo
2
      ×2J3                    ⍝ vetor unitário na direção do complexo
0.5547001962J0.8320502943
      ÷2                      ⍝ inverso multiplicativo
0.5
      *2                      ⍝ função exponencial
7.389056099
```

- Quando aplicada a números reais, a função monádica × corresponde à função `signum()` de muitas linguagens

Novo símbolo

Símbolo	Aridade	Descrição
+ (*conjugate*)	monádico	Conjugado complexo

Unicode	TAB	APL
`U+002B`	-	-

Novo símbolo

Símbolo	Aridade	Descrição
− (*negate*)	monádico	Simétrico aditivo

Unicode	TAB	APL
U+002D	-	-

Novo símbolo

Símbolo	Aridade	Descrição
× (*direction*)	monádico	Vetor unitário na direção do número

Unicode	TAB	APL
U+00D7	x x <tab>	APL + -

Novo símbolo

Símbolo	Aridade	Descrição
÷ (*reciprocal*)	monádico	Inverso multiplicativo

Unicode	TAB	APL
`U+00F7`	`: - <tab>`	`APL + =`

Novo símbolo

Símbolo	Aridade	Descrição
* (*exponential*)	monádico	e elevado ao argumento à direita

Unicode	TAB	APL
`U+002A`	`-`	`APL + p`

Arrays

- Em APL, a estrutura de dados fundamental é o *array*, e todos os dados estão contidos em *arrays*

Arrays

- Em APL, a estrutura de dados fundamental é o *array*, e todos os dados estão contidos em *arrays*
- Um *array* é uma coleção retangular de números, caracteres e *arrays*, arranjados ao longo de um ou mais eixos

Arrays

- Em APL, a estrutura de dados fundamental é o *array*, e todos os dados estão contidos em *arrays*
- Um *array* é uma coleção retangular de números, caracteres e *arrays*, arranjados ao longo de um ou mais eixos
- Os elementos de um *array* podem ter tipos distintos

Arrays

- Em APL, a estrutura de dados fundamental é o *array*, e todos os dados estão contidos em *arrays*
- Um *array* é uma coleção retangular de números, caracteres e *arrays*, arranjados ao longo de um ou mais eixos
- Os elementos de um *array* podem ter tipos distintos
- *Arrays* especiais:
  (a) **escalar**: um único número, dimensão zero
  (b) **vetor**: um *array* unidimensional
  (c) **matriz**: um *array* bidimensional

Declaração de *arrays*

- *Arrays* são declarados separando seus elementos por espaços

```
      2 3 5 7 11
2 3 5 7 11
      'string' 2.0 3J¯5 'c' 7
string 2.0 3J¯5 c 7
```

Declaração de *arrays*

- *Arrays* são declarados separando seus elementos por espaços

```
      2 3 5 7 11
2 3 5 7 11
      'string' 2.0 3J¯5 'c' 7
string 2.0 3J¯5 c 7
```

- Parêntesis podem ser utilizados para agrupar vetores

```
      (2 3 5) (7 11) (13) (17 19 23)
2 3 5   7 11   13   17 19 23
      ((2 3 5) (7 11)) ((13))
2 3 5   7 11   13
      ⍳10                         ⍝ gera os 10 primeiros naturais
1 2 3 4 5 6 7 8 9 10
```

Novo símbolo

Símbolo	Aridade	Descrição
⍳ (*iota*)	monádico	Gera os primeiros n naturais

Unicode	TAB	APL
`U+2373`	`i i <tab>`	`APL + i`

Profundidade

- A profundidade (*depth*) de um *array* corresponde a o seu nível de profundidade/recursão

Profundidade

- A profundidade (*depth*) de um *array* corresponde a o seu nível de profundidade/recursão
- um vetor de escalares tem profundidade igual a 1

Profundidade

- A profundidade (*depth*) de um *array* corresponde a o seu nível de profundidade/recursão
- um vetor de escalares tem profundidade igual a 1
- um vetor cujos elementos são vetores de profundidade 1 tem profundidade igual a 2

Profundidade

- A profundidade (*depth*) de um *array* corresponde a o seu nível de profundidade/recursão
- um vetor de escalares tem profundidade igual a 1
- um vetor cujos elementos são vetores de profundidade 1 tem profundidade igual a 2
- um escalar tem profundidade zero

Profundidade

- A profundidade (*depth*) de um *array* corresponde a o seu nível de profundidade/recursão
- um vetor de escalares tem profundidade igual a 1
- um vetor cujos elementos são vetores de profundidade 1 tem profundidade igual a 2
- um escalar tem profundidade zero
- APL atribuí a um vetor que mistura escalares e vetores uma profundidade negativa

Profundidade

- A profundidade de um *array* pode ser obtida por meio da função ≡

```
      ≡ 2
0
      ≡ 'string'          ⍝ 'string' = 's' 't' 'r' 'i' 'n' 'g'
1
      ≡ ((2 3) (5 7 11)) ('um' 'dois' 'três')
3
      ≡ 2 'três'
¯2
```

Profundidade

- A profundidade de um *array* pode ser obtida por meio da função ≡

```
      ≡ 2
0
      ≡ 'string'          ⍝ 'string' = 's' 't' 'r' 'i' 'n' 'g'
1
      ≡ ((2 3) (5 7 11)) ('um' 'dois' 'três')
3
      ≡ 2 'três'
¯2
```

- Strings vazias são representadas por ''

```
      ≡ ''
1
```

Novo símbolo

Símbolo	Aridade	Descrição
≡ (*depth*)	monádico	Retorna a profundidade do *array*

Unicode	TAB	APL
`U+2261`	`= = <tab>`	`APL + Shift + ç`

Rank

- O *rank* é definido como o número de dimensões de um *array*

Rank

- O *rank* é definido como o número de dimensões de um *array*
- Escalares tem *rank* igual a zero

Rank

- O *rank* é definido como o número de dimensões de um *array*
- Escalares tem *rank* igual a zero
- Vetores tem *rank* igual a 1

Rank

- O *rank* é definido como o número de dimensões de um *array*
- Escalares tem *rank* igual a zero
- Vetores tem *rank* igual a 1
- Matrizes tem *rank* igual a 2

Rank

- O *rank* é definido como o número de dimensões de um *array*
- Escalares tem *rank* igual a zero
- Vetores tem *rank* igual a 1
- Matrizes tem *rank* igual a 2
- Em APL os *arrays* são retangulares: cada linha de uma matriz deve ter o mesmo número de colunas

Rank

- O *rank* é definido como o número de dimensões de um *array*
- Escalares tem *rank* igual a zero
- Vetores tem *rank* igual a 1
- Matrizes tem *rank* igual a 2
- Em APL os *arrays* são retangulares: cada linha de uma matriz deve ter o mesmo número de colunas
- Para criar *arrays* com rank maior do que 1 é preciso usar a função ⍴ (*reshape*), que recebe como argumento à esquerda um vetor dos comprimentos das dimensões e os dados como argumento à direita

Novo símbolo

Símbolo	Aridade	Descrição
⍴ (*reshape*)	diádico	Retorna um *array* com as dimensões e dados indicados

Unicode	TAB	APL
`U+2374`	`r r <tab>`	`APL + r`

Declarando *arrays* multidimensionais

- A função ⍴ retorna *arrays* multidimensionais

```
    2 2 ⍴ 1 0 0 1
1 0
0 1
```

Declarando *arrays* multidimensionais

- A função ⍴ retorna *arrays* multidimensionais

```
    2 2 ⍴ 1 0 0 1
1 0
0 1
```

- Se há dados em excesso o que sobra é ignorado

```
    2 3 ⍴ 'ABCDEFGHIJ'
ABC
DEF
```

Declarando *arrays* multidimensionais

- A função ρ retorna *arrays* multidimensionais

```
      2 2 ρ 1 0 0 1
1 0
0 1
```

- Se há dados em excesso o que sobra é ignorado

```
      2 3 ρ 'ABCDEFGHIJ'
ABC
DEF
```

- Se faltam dados a função ρ retorna ciclicamente ao início dos dados indicados

```
      2 3 ρ 5 7
5 7 5
7 5 7
      ⍝ Arrays também podem ser aninhandos, contendo outros arrays
      2 3 ρ 'string' (5.7 11) ((13 17) (19)) 'a'
```

Forma de um *array*

- Em sua versão monádica, a função ⍴ retorna os comprimentos das dimensões (forma) do *array*

```
    ⍴ 'string'
6
    ⍴ ⍬                          ⍝ ⍬ é o vetor númerico vazio
0
    ⍴ 2                          ⍝ Escalares não tem forma
```

Forma de um *array*

- Em sua versão monádica, a função ⍴ retorna os comprimentos das dimensões (forma) do *array*

```
      ⍴ 'string'
6
      ⍴ ⍬                          ⍝ ⍬ é o vetor númerico vazio
0
      ⍴ 2                          ⍝ Escalares não tem forma
```

- Matrizes com uma única linha e vetores são distintos

```
      ⍴ 2 3 5 7 11                 ⍝ Vetor
5
      ⍴ (1 5 ⍴ 2 3 5 7 11)         ⍝ Matriz com uma única linha
1 5
```

Forma de um *array*

- Em sua versão monádica, a função ⍴ retorna os comprimentos das dimensões (forma) do *array*

```
    ⍴ 'string'
6
    ⍴ ⍬                         ⍝ ⍬ é o vetor númerico vazio
0
    ⍴ 2                         ⍝ Escalares não tem forma
```

- Matrizes com uma única linha e vetores são distintos

```
    ⍴ 2 3 5 7 11                ⍝ Vetor
5
    ⍴ (1 5 ⍴ 2 3 5 7 11)        ⍝ Matriz com uma única linha
1 5
```

- Vale a identidade `v ≡ ⍴ (v ⍴ A)`, onde `v` é um vetor e `A` um *array* qualquer

Novo símbolo

Símbolo	Aridade	Descrição
⍴ (*shape*)	monádico	Retorna a forma (comprimento das dimensões) de um *array*

Unicode	TAB	APL
`U+2374`	`r r <tab>`	`APL + r`

Novo símbolo

Símbolo	Aridade	Descrição
⍬ (*empty numeric vector*)	-	Vetor numérico vazio

Unicode	TAB	APL
`U+236C`	`0 - <tab>`	`APL + Shift + [`

Novo símbolo

Símbolo	Aridade	Descrição
≡ (*match*)	diádico	Retorna verdadeiro se ambos argumentos são idênticos (conteúdo e forma)

Unicode	TAB	APL
U+2361	= = <tab>	APL + Shift + ç

Cálculo do *rank*

- O *rank* de um vetor é igual ao comprimento da sua forma

Cálculo do *rank*

- O *rank* de um vetor é igual ao comprimento da sua forma
- Assim, o *rank* de um *array* pode ser computado por meio da dupla aplicação da função ⍴

```
      ⍴⍴ 2                        ⍝ Escalares tem rank zero
0
      ⍴⍴ 2 3 5 7 11
1
      ⍴⍴ 2 3 ⍴ ⍳5
2
```

Cálculo do *rank*

- O *rank* de um vetor é igual ao comprimento da sua forma
- Assim, o *rank* de um *array* pode ser computado por meio da dupla aplicação da função ⍴

```
      ⍴⍴ 2                          ⍝ Escalares tem rank zero
0
      ⍴⍴ 2 3 5 7 11
1
      ⍴⍴ 2 3 ⍴ ⍳5
2
```

- A função ⍴ pode ser usada para conversões entre um escalar x e um vetor v com um único componente igual a x:

```
      ⍴⍴ 1 ⍴ x                      ⍝ de x para v
1
      ⍴⍴ ⍬ ⍴ 1 ⍴ x                  ⍝ de v para x
0
```

Funções escalares monádicas

- Em APL uma função pode ser aplicada monadicamente (um argumento) ou diadicamente (dois argumentos)

Funções escalares monádicas

- Em APL uma função pode ser aplicada monadicamente (um argumento) ou diadicamente (dois argumentos)
- Há dois tipos de funções: escalares e mistas

Funções escalares monádicas

- Em APL uma função pode ser aplicada monadicamente (um argumento) ou diadicamente (dois argumentos)
- Há dois tipos de funções: escalares e mistas
- Funções escalares monádicas navegam nos diferentes níveis dos *arrays* até localizar e operar nos escalares

Funções escalares monádicas

- Em APL uma função pode ser aplicada monadicamente (um argumento) ou diadicamente (dois argumentos)
- Há dois tipos de funções: escalares e mistas
- Funções escalares monádicas navegam nos diferentes níveis dos *arrays* até localizar e operar nos escalares
- A estrutura se mantem e apenas o conteúdo é alterado:

```
    ! 2 3 5 7.11           ⍝ fatorial
2 6 120 5040 6296.086347
    | 2 ¯3 5J¯7 ¯11.13     ⍝ valor absoluto (norma)
2 3 8.602325267 11.13
    ÷ (2 3) 5 (7 (11.13 17))
0.5 0.3333333333   0.2   0.1428571429    0.08984725966  0.05882352941
```

Novo símbolo

Símbolo	Aridade	Descrição
! (*factorial*)	monádico	computa o fatorial (função gama) de x

Unicode	TAB	APL
`U+0021`	`-`	`APL + Shift + -`

Novo símbolo

Símbolo	Aridade	Descrição
| (*magnitude*)	monádico	computa o valor absoluto (norma) de x

Unicode	TAB	APL
`U+007C`	`-`	`APL + m`

Funções escalares diádicas

- Funções escalares diádicas obtém seus operandos das localizações correspondentes de seus argumentos

```
    2 3 5 + 7 11 13
9 14 18
```

Funções escalares diádicas

- Funções escalares diádicas obtém seus operandos das localizações correspondentes de seus argumentos

```
      2 3 5 + 7 11 13
9 14 18
```

- Se as formas dos argumentos diferem ocorre um erro

```
      2 3 ÷ 5 7 11
LENGTH ERROR: Mismatched left and right argument shapes
```

Funções escalares diádicas

- Funções escalares diádicas obtém seus operandos das localizações correspondentes de seus argumentos

```
    2 3 5 + 7 11 13
9 14 18
```

- Se as formas dos argumentos diferem ocorre um erro

```
    2 3 ÷ 5 7 11
LENGTH ERROR: Mismatched left and right argument shapes
```

- Se um dos argumentos é escalar, ele é replicado para todos os escalares do outro argumento

Funções escalares diádicas

- Funções escalares diádicas obtém seus operandos das localizações correspondentes de seus argumentos

```
    2 3 5 + 7 11 13
9 14 18
```

- Se as formas dos argumentos diferem ocorre um erro

```
    2 3 ÷ 5 7 11
LENGTH ERROR: Mismatched left and right argument shapes
```

- Se um dos argumentos é escalar, ele é replicado para todos os escalares do outro argumento
- O mesmo vale para escalares dentro do argumento, após o pareamento

```
    2 × 3 5 7
6 10 14
    (1 1) 2 (3 5 8) + 13 (21 34) 55
14 14   23 36   58 60 63
```

Funções mistas e definidas pelo programador

- Funções mistas consideram seus argumentos na íntegra, ou suas subestruturas

Funções mistas e definidas pelo programador

- Funções mistas consideram seus argumentos na íntegra, ou suas subestruturas
- Por exemplo, a função ⍴ monádica considera todo seu argumento

```
    ⍴ ((2 3 5) 7 ((11 13) 17)) 19
2
```

Funções mistas e definidas pelo programador

- Funções mistas consideram seus argumentos na íntegra, ou suas subestruturas
- Por exemplo, a função ⍴ monádica considera todo seu argumento

```
    ⍴ ((2 3 5) 7 ((11 13) 17)) 19
2
```

- Há três tipos de funções definidas pelo programador: *dfns*, *tradfns* e funções tácitas (implícitas)

Funções mistas e definidas pelo programador

- Funções mistas consideram seus argumentos na íntegra, ou suas subestruturas
- Por exemplo, a função ⍴ monádica considera todo seu argumento

```
      ⍴ ((2 3 5) 7 ((11 13) 17)) 19
2
```

- Há três tipos de funções definidas pelo programador: *dfns*, *tradfns* e funções tácitas (implícitas)
- Desde 2010 os dialetos da APL baseados no Dyalog removeram as *tradfns* em favor das *dfns*

Funções mistas e definidas pelo programador

- Funções mistas consideram seus argumentos na íntegra, ou suas subestruturas
- Por exemplo, a função ⍴ monádica considera todo seu argumento

```
    ⍴ ((2 3 5) 7 ((11 13) 17)) 19
2
```

- Há três tipos de funções definidas pelo programador: *dfns*, *tradfns* e funções tácitas (implícitas)
- Desde 2010 os dialetos da APL baseados no Dyalog removeram as *tradfns* em favor das *dfns*
- Uma função definida pelo usuário se comporta como as funções primitivas: no máximo dois argumentos e são chamadas monadicamente (prefixadas) ou diadicamente (pós-fixadas)

dfns

- Uma *dfn* (anteriormente denominada *dynamic function*) é delimitada por chaves e seus argumentos à esquerda e à direita são representados pelas letras gregas alpha (⍺) e omega (⍵), respecivamente

```
plus ← {⍺+⍵}
    2 plus 3
5
```

dfns

- Uma *dfn* (anteriormente denominada *dynamic function*) é delimitada por chaves e seus argumentos à esquerda e à direita são representados pelas letras gregas alpha (⍺) e omega (⍵), respecivamente

```
plus ← {⍺+⍵}
    2 plus 3
5
```

- Na versão monádica, apenas o ⍵ é utilizado

```
    cube ← {⍵*3}
    cube 2
8
```

dfns

- Uma *dfn* (anteriormente denominada *dynamic function*) é delimitada por chaves e seus argumentos à esquerda e à direita são representados pelas letras gregas alpha (⍺) e omega (⍵), respecivamente

```
plus ← {⍺+⍵}
    2 plus 3
5
```

- Na versão monádica, apenas o ⍵ é utilizado

```
    cube ← {⍵*3}
    cube 2
8
```

- *Dfns* são funções anônimas (lambdas)

```
    2 {⍺×⍵} 3
6
```

Novo símbolo

Símbolo	Aridade	Descrição
⍺ (*alpha*)	-	Argumento à esquerda de uma *dfn*

Unicode	TAB	APL
`U+237A`	`a a <tab>`	`APL + a`

Novo símbolo

Símbolo	Aridade	Descrição
⍵ (*omega*)	-	Argumento à direita de uma *dfn*

Unicode	TAB	APL
`U+2375`	`w w <tab>`	`APL + w`

`if-then-else`

- É possível replicar o construto `if-then-else` de outras linguagens por meio de uma *dfn*

`if-then-else`

- É possível replicar o construto `if-then-else` de outras linguagens por meio de uma *dfn*
- A sintaxe é

```
{ a : b ⋄ c }
```

if-then-else

- É possível replicar o construto `if-then-else` de outras linguagens por meio de uma *dfn*
- A sintaxe é

```
{ a : b ⋄ c }
```

- Esta *dfn* equivale a

```
if a then b else c
```

onde a tem que ser uma expressão booleana

if-then-else

- É possível replicar o construto `if-then-else` de outras linguagens por meio de uma *dfn*
- A sintaxe é

```
{ a : b ⋄ c }
```

- Esta *dfn* equivale a

```
if a then b else c
```

onde a tem que ser uma expressão booleana

- Exemplo de uso:

```
max ← {α > ω : α ⋄ ω }
    2 max 3
3
```

Recursão

- Mesmo sendo anônimas, é possível implementar funções recursivas usando *dfns*

Recursão

- Mesmo sendo anônimas, é possível implementar funções recursivas usando *dfns*
- Uma maneira é nomeando a *dfns* por meio de uma atribuição

```
    gcd ← {⍵ > 0 : ⍵ gcd (⍵|⍺) ⋄ ⍺}
    20 gcd 12
4
```

Recursão

- Mesmo sendo anônimas, é possível implementar funções recursivas usando *dfns*
- Uma maneira é nomeando a *dfns* por meio de uma atribuição

```
    gcd ← {ω > 0 : ω gcd (ω|α) ⋄ α}
    20 gcd 12
4
```

- A segunda maneira é utilizar o símbolo ∇, que idenfica a função anônima e permite a chamada recursiva

```
    {ω = 1 : 1 ⋄ ω + ∇ ω - 1} 10        ⍝ soma dos n primeiros positivos
55
```

Novo símbolo

Símbolo	Aridade	Descrição
| (*residue*)	diádico	Computa o resto da divisão euclidiana do argumento à direita pelo argumento à esquerda

Unicode	TAB	APL
`U+007C`	`-`	`APL + m`

Novo símbolo

Símbolo	Aridade	Descrição
∇ (*recursion*)	-	Representa uma *dfn* em uma chamada recursiva

Unicode	TAB	APL
`U+2207`	`V V <tab>`	`APL + g`

Funções tácitas

- Funções tácitas (*tacit*, implícitas) são expressões sem referências aos argumentos

Funções tácitas

- Funções tácitas (*tacit*, implícitas) são expressões sem referências aos argumentos
- Códigos que usam apenas funções tácitas são ditos livre de pontos

Funções tácitas

- Funções tácitas (*tacit*, implícitas) são expressões sem referências aos argumentos
- Códigos que usam apenas funções tácitas são ditos livre de pontos
- A omissão dos parâmetros remete à conversão η do cálculo lambda

Funções tácitas

- Funções tácitas (*tacit*, implícitas) são expressões sem referências aos argumentos
- Códigos que usam apenas funções tácitas são ditos livre de pontos
- A omissão dos parâmetros remete à conversão η do cálculo lambda
- Uma única função é sempre uma função tácita

```
÷

f ← ×        ⍝ Atribuições podem nomear funções tácitas
```

Trens

- Funções tácitas com dois ou mais símbolos são chamadas **trens**, onde cada vagão é uma função ou vetor

Trens

- Funções tácitas com dois ou mais símbolos são chamadas **trens**, onde cada vagão é uma função ou vetor
- Trens podem ser monádicos ou diádicos

Trens

- Funções tácitas com dois ou mais símbolos são chamadas **trens**, onde cada vagão é uma função ou vetor
- Trens podem ser monádicos ou diádicos
- Um trem com dois carros monádicos é chamado *atop* (sobre, em cima)

Trens

- Funções tácitas com dois ou mais símbolos são chamadas **trens**, onde cada vagão é uma função ou vetor
- Trens podem ser monádicos ou diádicos
- Um trem com dois carros monádicos é chamado *atop* (sobre, em cima)
- `(f g) X` é equivalente a `f (g X)`, onde `f` e `g` são funções monádicas

```
    f ← +- ⋄ f 2J3      ⍝ Conjugado do simétrico
¯2J3
```

Trens

- Funções tácitas com dois ou mais símbolos são chamadas **trens**, onde cada vagão é uma função ou vetor
- Trens podem ser monádicos ou diádicos
- Um trem com dois carros monádicos é chamado *atop* (sobre, em cima)
- `(f g) X` é equivalente a `f (g X)`, onde `f` e `g` são funções monádicas

```
    f ← +- ⍤ f 2J3       ⍝ Conjugado do simétrico
¯2J3
```

- Isto quer dize que `f` é avaliada "*atop*" (sobre) o resultado de `g`

Trens

- Funções tácitas com dois ou mais símbolos são chamadas **trens**, onde cada vagão é uma função ou vetor
- Trens podem ser monádicos ou diádicos
- Um trem com dois carros monádicos é chamado *atop* (sobre, em cima)
- `(f g) X` é equivalente a `f (g X)`, onde `f` e `g` são funções monádicas

```
    f ← +- ⋄ f 2J3      ⍝ Conjugado do simétrico
¯2J3
```

- Isto quer dize que `f` é avaliada "*atop*" (sobre) o resultado de `g`
- Este trem corresponde a composição de funções em outras linguagens

Trens

- Funções tácitas com dois ou mais símbolos são chamadas **trens**, onde cada vagão é uma função ou vetor
- Trens podem ser monádicos ou diádicos
- Um trem com dois carros monádicos é chamado *atop* (sobre, em cima)
- `(f g) X` é equivalente a `f (g X)`, onde `f` e `g` são funções monádicas

```
    f ← +- ⋄ f 2J3      ⍝ Conjugado do simétrico
¯2J3
```

- Isto quer dize que `f` é avaliada "*atop*" (sobre) o resultado de `g`
- Este trem corresponde a composição de funções em outras linguagens
- Um trem não nomeado deve ser delimitado entre parêntesis

```
    (*÷) 2              ⍝ e elevado ao inverso de x
1.648721271
```

forks

- Um trem com três carros monádico é um *fork* (garfo, bifurcação)

forks

- Um trem com três carros monádico é um *fork* (garfo, bifurcação)
- Há duas variantes de *forks*:
  (a) `(f g h) X` equivale a `(f X) g (h X)`

```
      pm ← +,-
      pm 2
2 ¯2
```

forks

- Um trem com três carros monádico é um *fork* (garfo, bifurcação)
- Há duas variantes de *forks*:
  (a) `(f g h) X` equivale a `(f X) g (h X)`

```
      pm ← +,-
      pm 2
2 ¯2
```

  (b) `(A g h) X` equivale a `A g (h X)` onde `A` é um *array*

```
      areaCircle ← ((○1)×*∘2)   ⍝ ○1 é igual a Π, o operador bind (∘, jot) permite a
      areaCircle 2              ⍝ aplicação parcial de uma função diádica
12.56637061
```

forks

- Um trem com três carros monádico é um *fork* (garfo, bifurcação)
- Há duas variantes de *forks*:
  (a) `(f g h) X` equivale a `(f X) g (h X)`

```
    pm ← +,-
    pm 2
2 ¯2
```

  (b) `(A g h) X` equivale a `A g (h X)` onde `A` é um *array*

```
    areaCircle ← ((○1)×*∘2)   ⍝ ○1 é igual a Π, o operador bind (∘, jot) permite a
    areaCircle 2              ⍝ aplicação parcial de uma função diádica
12.56637061
```

- Em ambos casos, `g` deve ser diádica e `f` e `h` monádicas

Novo símbolo

Símbolo	Aridade	Descrição
, (*catenate*,*laminate*)	diádico	Concatena ambos argumentos

Unicode	TAB	APL
`U+002C`	-	-

Novo símbolo

Símbolo	Aridade	Descrição
○ (*pi times*)	monádico	Multiplica o argumento por π

Unicode	TAB	APL
`U+25CB`	`O O <tab>`	`APL + o`

Novo símbolo

Símbolo	Aridade	Descrição
∘ (*bind*)	diádico	Aplica parcialmente um argumento, à esquerda ou à direita, de uma função diádica

Unicode	TAB	APL
`U+2218`	`o o <tab>`	`APL + j`

Trens diádicos

- Os trens diádicos estão relacionados aos monádicos

Trens diádicos

- Os trens diádicos estão relacionados aos monádicos
  (a) `X (f g) Y` equivale a `f (X g Y)`, com `f` monádica, `g` diádica

Trens diádicos

- Os trens diádicos estão relacionados aos monádicos
  (a) `X (f g) Y` equivale a `f (X g Y)`, com `f` monádica, `g` diádica
  (b) `X (f g h) Y` equivale a `(X f Y) g (X h Y)`, todas diádicas

Trens diádicos

- Os trens diádicos estão relacionados aos monádicos
  (a) `X (f g) Y` equivale a `f (X g Y)`, com `f` monádica, `g` diádica
  (b) `X (f g h) Y` equivale a `(X f Y) g (X h Y)`, todas diádicas
  (c) `X (A g h) Y` equivale a `A g (X h Y)`, ambas diádicas

Trens diádicos

- Os trens diádicos estão relacionados aos monádicos
  (a) `X (f g) Y` equivale a `f (X g Y)`, com `f` monádica, `g` diádica
  (b) `X (f g h) Y` equivale a `(X f Y) g (X h Y)`, todas diádicas
  (c) `X (A g h) Y` equivale a `A g (X h Y)`, ambas diádicas
- Dentro de um trem diádicos os símbolos `⊣` e `⊢` retornam os argumentos à esquerda e a direita, respectivamente

```
      3 (×-) 5            ⍝ (a), sinal da diferença
¯1
      imc ← ÷÷⊢           ⍝ (b), IMC
      75 imc 1.79
23.40750913
      2 (0.5×+) 3         ⍝ (c), média aritmética
2.5
```

Trens diádicos

- Os trens diádicos estão relacionados aos monádicos
  (a) `X (f g) Y` equivale a `f (X g Y)`, com `f` monádica, `g` diádica
  (b) `X (f g h) Y` equivale a `(X f Y) g (X h Y)`, todas diádicas
  (c) `X (A g h) Y` equivale a `A g (X h Y)`, ambas diádicas
- Dentro de um trem diádicos os símbolos `⊣` e `⊢` retornam os argumentos à esquerda e a direita, respectivamente

```
      3 (×-) 5           ⍝ (a), sinal da diferença
¯1
      imc ← ÷÷⊢          ⍝ (b), IMC
      75 imc 1.79
23.40750913
      2 (0.5×+) 3        ⍝ (c), média aritmética
2.5
```

- Em trens monádicos ambos retornam sempre o argumento à direita

Novo símbolo

Símbolo	Aridade	Descrição
⊣ (*left*)	monádico/diádico	Em um trem, retorna o argumento à esquerda (ou a direita, no caso monádico)

Unicode	TAB	APL
`U+22A3`	`- | <tab>`	`APL + Shift + |`

Novo símbolo

Símbolo	Aridade	Descrição
⊢ (*right*)	monádico/diádico	Em um trem, retorna o argumento à direita

Unicode	TAB	APL
`U+22A2`	`| - <tab>`	`APL + |`

Trens de tamanho 4 ou maior

- Para trens de tamanho quatro ou maior, a regra é simples: os últimos 3 símbolos se tornam um trem de tamanho 3 e são tratados como uma única função

Trens de tamanho 4 ou maior

- Para trens de tamanho quatro ou maior, a regra é simples: os últimos 3 símbolos se tornam um trem de tamanho 3 e são tratados como uma única função
- O que resta será um *atop*, um *fork* ou é preciso repetir a operação

Trens de tamanho 4 ou maior

- Para trens de tamanho quatro ou maior, a regra é simples: os últimos 3 símbolos se tornam um trem de tamanho 3 e são tratados como uma única função
- O que resta será um *atop*, um *fork* ou é preciso repetir a operação
- Por exemplo, `(p q r s) X` equivale a

```
(p (q r s)) X = p ((q r s) X) = p ((q X) r (s X))
```

Trens de tamanho 4 ou maior

- Para trens de tamanho quatro ou maior, a regra é simples: os últimos 3 símbolos se tornam um trem de tamanho 3 e são tratados como uma única função
- O que resta será um *atop*, um *fork* ou é preciso repetir a operação
- Por exemplo, `(p q r s) X` equivale a

```
(p (q r s)) X = p ((q r s) X) = p ((q X) r (s X))
```

- Outro exemplo:

```
X (p q r s t) Y = (X p Y) q ((X r Y) s (X t Y))
```

Trens de tamanho 4 ou maior

- Para trens de tamanho quatro ou maior, a regra é simples: os últimos 3 símbolos se tornam um trem de tamanho 3 e são tratados como uma única função
- O que resta será um *atop*, um *fork* ou é preciso repetir a operação
- Por exemplo, `(p q r s) X` equivale a

```
(p (q r s)) X = p ((q r s) X) = p ((q X) r (s X))
```

- Outro exemplo:

```
X (p q r s t) Y = (X p Y) q ((X r Y) s (X t Y))
```

- Trens podem ser usados para simplificar expressões do tipo `((cond1 x) and (cond2 x) and ... and (condN x))` para `cond1 ∧ cond2 ∧ ... ∧ condN`

Novo símbolo

Símbolo	Aridade	Descrição
∧ (*and*)	diádico	Conjunção (e) lógica escalar

Unicode	TAB	APL
`U+2227`	`^^ <tab>`	`APL + 0`

Referências

1. APL Wiki. Defined function (traditional, acesso em 27/09/2021.
2. Dyalog. Try APL – Interactive lessons, acesso em 23/09/2021.
3. **IVERSON**, Kenneth E. *A Programming Language*, John Wiley and Sons, 1962.
4. Unicode Character Table. Página principal, acesso em 27/09/2021.
5. Xah Lee. Unicode APL Symbols, acesso em 23/09/2021.