Estruturas de Dados

aula 23: A pilha

1 Introdução

- *código e estrutura de dados*
- *pensamento e representação*

2 Expressões aritméticas

Uma estrutura de dados que a gente costuma utilizar para realizar raciocínios matemáticos são as expressões aritméticas

$$\Big(3 \;+\; \big(\big(2 \;\times\; \big((15 \;+\; 1) \;/\; (13 \;-\; 9)\big)\big) \;+\; 5\big)\Big)$$

Daí, se alguém pede para você calcular essa expressão, é possível que você comece com a operação

$$15 \;+\; 1$$

Porque?

Ora, porque a gente aprendeu que deve realizar as operações dentro dos parênteses primeiro.

E essa operação se encontra dentro de um par de parênteses mais interno — (*i.e., não existem parênteses dentro desse par de parênteses*)

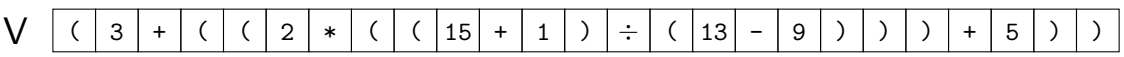
Logo, a gente pode começar por ali.

Certo.

Mas agora a gente pergunta

- *Como é que um computador pode calcular o valor dessa expressão ?*

Bom, a primeira coisa é que a expressão vai estar armazenada em uma lista

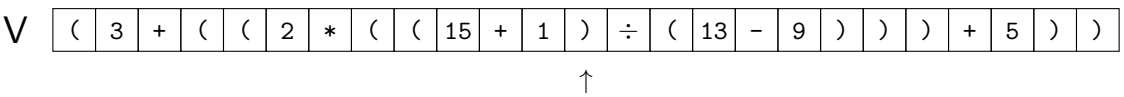


Mas, e agora?

Como é que a gente localiza um par de parênteses mais interno?

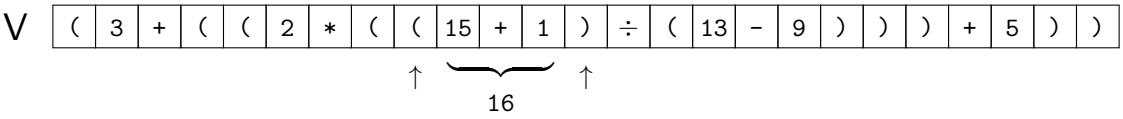
Bom, aqui uma pequena esperteza resolve o nosso problema.

Quer dizer, basta localizar o primeiro `fecha-parênteses`



Daí, andando para trás a gente encontra o `abre-parênteses` correspondente.

E agora, a gente pode fazer essa conta



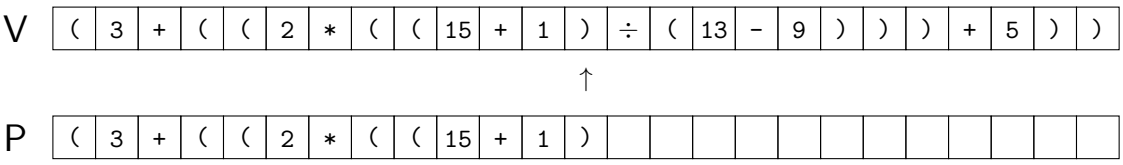
Mas, e agora?

Quer dizer, onde é que a gente armazena o `16`*?*

Porque o `16` é apenas um número, mas existem 5 posições que vão ficar vazias.

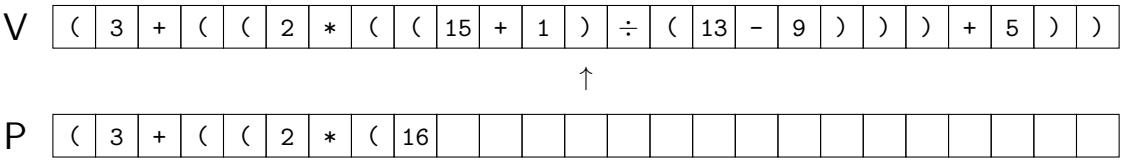
Bom, aqui é a hora para mais uma esperteza.

Quer dizer, a ideia é ir copiando a expressão para uma lista auxiliar, à medida que a gente procura o `fecha-parênteses`

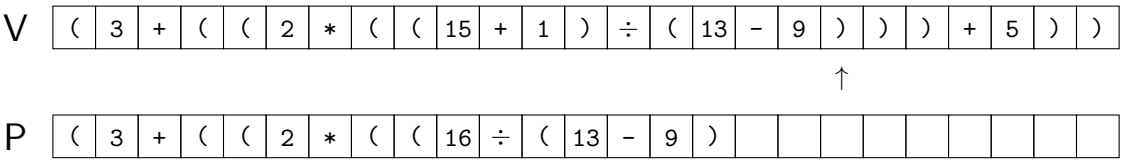


Daí, ao invés de fazer essa conta na lista original, a gente faz a conta na lista auxiliar.

O que deixa a coisa assim



Agora, é só continuar a leitura até encontrar o próximo `fecha-parênteses`



E daí, a gente continua fazendo a mesma coisa até o fim.

Em resumo, a ideia é a seguinte

1. a gente vai lendo os elementos de V e copiando para P, até encontrar um `fecha-parênteses`
2. daí, a gente faz a conta dentro do parênteses em P
3. e continua fazendo isso até o fim

E o algoritmo fica assim

```
V = { (, 3, +, (, 2, *, (, ...}
P = {}

i = 1                                          // coloca o dedo no início
j = 0                                          // não há nada em P

Enquanto ( i <= N )

    j++;  P[j] = V[i]                          // copia termo para P

    Se ( V[i] == fecha-parênteses )

       j--;    N2 = P[j]
       j--;    op = P[j]
       j--;    N1 = P[j]

       res = Calcula ( N1, op, N2 )

       j--;    P[j] = res

    i++

Imprime ( "o resultado é:", P[1] )
```

Legal, não é?

3 Trabalhando sem parênteses

Sim, mas o problema é que as pessoas não fazem assim.

Quer dizer, as pessoas gostam de escrever as coisa assim

$$3 \ + \ 2 \ \times \ 5$$

E daí, elas sabem que a multiplicação deve ser feita antes da soma.

E agora?

Como é que um computador pode fazer isso?

Por exemplo, imagine que a expressão é a seguinte

V	3	+	2	*	5	+	6	*	3	*	8	+	5	+	1

Bom, daí a gente faz a mesma coisa outra vez.

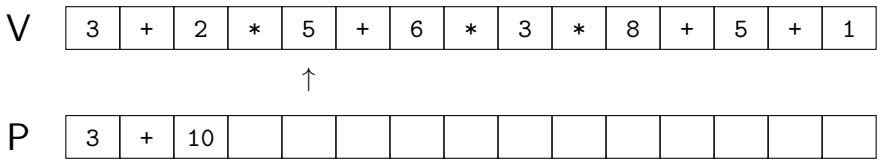
Quer dizer, a gente vai lendo a expressão, e copiando para a lista auxiliar P, até encontrar uma multiplicação

V	3	+	2	*	5	+	6	*	3	*	8	+	5	+	1
				↑											
P	3	+	2	*											

Nesse ponto, a gente encontra uma operação que já pode realizar.

Só que, para isso, ainda é preciso ler o próximo número.

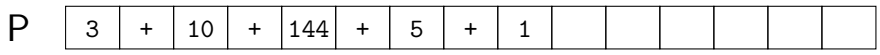
Fazendo isso, a gente chega no seguinte



E agora, é só continuar fazendo a mesma coisa até o fim.

Só que, dessa vez, a gente não obtém o resultado da expressão.

Quer dizer, no final do processo, a lista P contém o seguinte



E agora, é preciso realizar as operações de soma.

Só que isso é fácil.

Quer dizer, basta

1. ler os 3 últimos termos (removendo da lista)
2. fazer a conta
3. colocar o resultado na lista
4. e continuar fazendo isso até o fim

Abaixo, nós temos o algoritmo completo

```
V = { 3, +, 2, *, 5, +, 6, *, 3, *, 8, +, 5, +, 1 }
P = {}

// 1a ETAPA: calcular as multiplicações

i = 1                                          // coloca o dedo no início
j = 0                                          // não há nada em P

Enquanto ( i <= N )

    j++;  P[j] = V[i]                          // copia termo para P

    Se ( V[i] == multiplicação )

       i++;    N2 = V[i]
               op = P[j]
       j--;    N1 = P[j]

       res = Calcula ( N1, op, N2 )
```

```
            P[j] = res

        i++

    // 2a ETAPA: calcular as somas

    Enquanto ( j > 1 )

                N2 = P[j]
        j--;    op = P[j]
        j--;    N1 = P[j]

        res = Calcula ( N1, op, N2 )

        P[j] = res

    Imprime ( "o resultado é:", P[1] )
```

Legal, não é?

4 Trabalhando com e sem parênteses

Sim, mas o problema é que as pessoas não fazem assim.

Quer dizer, as pessoas gostam de escrever as coisas assim

$$3 \ + \ 2 \ \times \ \big(5 \ + \ 8 \ \times \ 2\big)$$

E daí, elas sabem que

1. as coisas dentro dos parênteses são feitas primeiro
2. e quando não há parênteses, as multiplicações são feitas primeiro

E agora?

Como é que um computador pode fazer isso?

Bom, aqui é a hora para mais uma esperteza.

Quer dizer, a gente começa observando que dentro dos parênteses não há parênteses

$$3 \ + \ 2 \ \times \ \big(\underbrace{5 \ + \ 8 \ \times \ 2}\big)$$

E nós já temos um algoritmo que calcula expressões sem parênteses.

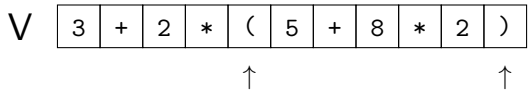
Então, a gente pode pensar que esse algoritmo é uma função.

E que a gente pode chamar essa função para calcular o valor da expressão entre parênteses.

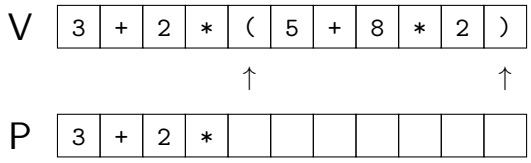
Para a coisa funcionar, é preciso fazer algumas adaptações no algoritmo.

Porque agora, ele vai começar a leitura no `abre-parênteses`.

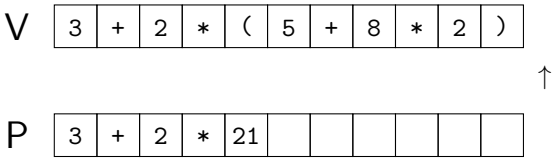
E vai terminar a leitura no `fecha-parênteses`



Nesse ponto, a lista auxiliar P tem a parte anterior da expressão



E a ideia é que após a execução da função, o resultado da expressão entre parênteses já está na lista P



Abaixo, nós temos o código da função

```
func CESP ( V, i, P, j )

    i++                                          // pula o abre-parênteses
    zero = j                                     // imagina que P está vazia

    // 1a ETAPA: calcular as multiplicações

    Enquanto ( V[i] != fecha-parênteses )

        j++;  P[j] = V[i]                        // copia termo para P

        Se ( V[i] == multiplicação )

           i++;    N2 = V[i]
                   op = P[j]
           j--;    N1 = P[j]

           res = Calcula ( N1, op, N2 )

           P[j] = res

        i++

    // 2a ETAPA: calcular as somas

    Enquanto ( j > zero + 1 )

                N2 = P[j]
```

```
        j--;    op = P[j]
        j--;    N1 = P[j]

        res = Calcula ( N1, op, N2 )

        P[j] = res

    i++                                          // pula o fecha-parênteses
```

Mas, já dá para advinhar que isso não resolve todos os nossos problemas.

Porque dentro dos parênteses pode haver outro par de parênteses

$$3 \ + \ 2 \ \times \ \big(5 \ + \ 8 \ \times \ (2 \ + \ 3)\big)$$

E agora?

Bom, agora é só fazer a função chamar ela mesma quando ela encontra um par de parênteses.

A coisa fica assim

```
func CECSP ( V, i, P, j )

    i++                                          // pula o abre-parênteses
    zero = j                                     // imagina que P está vazia

    // 1a ETAPA: calcular as multiplicações

    Enquanto ( V[i] != fecha-parênteses )

        Se ( V[i] == abre-parênteses )    CECSP ( V, i, P, j )

        Senão                             j++;  P[j] = V[i]

        Se ( V[i] == multiplicação )

           Se ( V[i] == abre-parênteses )    CECSP ( V, i, P, j )

           Senão                             j++;  P[j] = V[i]
                                             i++


                   N2 = V[i]
           j==;    op = P[j]
           j--;    N1 = P[j]

           res = Calcula ( N1, op, N2 )

           P[j] = res

        i++
```

```
// 2a ETAPA: calcular as somas

Enquanto ( j > zero + 1 )

            N2 = P[j]
    j--;    op = P[j]
    j--;    N1 = P[j]

    res = Calcula ( N1, op, N2 )

    P[j] = res

i++                                          // pula o fecha-parênteses
```

5 A pilha

A esperteza que nós acabamos de ver é uma estratégia de computação que é usada em um monte de situações diferentes.

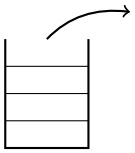
Quer dizer, essa é a situação em que nós temos várias coisas para fazer, mas nem todas podem ser feitas imediatamente — (*algumas precisam aguardar que outras sejam feitas primeiro*)

E daí, a esperteza consiste em utilizar uma pilha de coisas que ainda precisam ser feitas — (*ou uma pilha de tarefas*)

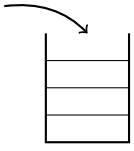
Foi por isso que nós demos o nome de P para a nossa lista auxiliar.

Agora, duas características importantes de uma pilha de tarefas são

1. a gente sempre pega a tarefa no topo da pilha para realizar em seguida



2. a gente sempre coloca a uma nova tarefa no topo da pilha



Essas operações são realizadas tão frequentemente, que é útil ter funções que as implementam

```
func Push ( elem, P )
     topo++
     P[topo] = elem
```

```
func Pop ( P )
     elem = P[topo]
     topo--
     Retorna ( elem )
```

Além disso, também é útil ter uma função que apenas inspeciona o elemento do topo, sem removê-lo

```
func Peep ( P )
     elem = P[topo]
     Retorna ( elem )
```

E é útil ter uma função que nos diz qual é o tamanho da pilha

```
func Size ( P )
     Retorna ( topo )
```

Certo.

Agora que nós temos essa nova estrutura de dados, é uma boa ideia reescrever o código do nosso avaliador de expressões

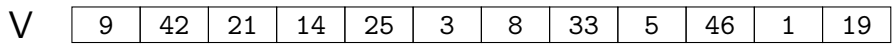
```
func CECSP ( V, i, P )

    i++                                          // pula o abre-parênteses
    zero = Size(P)                               // imagina que P está vazia

    // 1a ETAPA: calcular as multiplicações

    Enquanto ( V[i] != fecha-parênteses )

        Se ( V[i] == abre-parênteses )    CECSP ( V, i, P )

        Senão                             Push ( V[i], P )

        Se ( V[i] == multiplicação )

           i++

           Se ( V[i] == abre-parênteses )    CECSP ( V, i, P )

           Senão                             Push ( V[i], P )
                                             i++

           N2 = Pop ( P )
           op = Pop ( P )
           N1 = Pop ( P )

           res = Calcula ( N1, op, N2 )

           Push ( res, P )

        Senão

           i++
```

```
// 2a ETAPA: calcular as somas

Enquanto ( j > zero + 1 )

    N2 = Pop ( P )
    op = Pop ( P )
    N1 = Pop ( P )

    res = Calcula ( N1, op, N2 )

    Push ( res, P )

i++                                          // pula o fecha-parênteses
```

6 O algoritmo quicksort

Imagine que você tem uma lista desordenada.

Por exemplo,

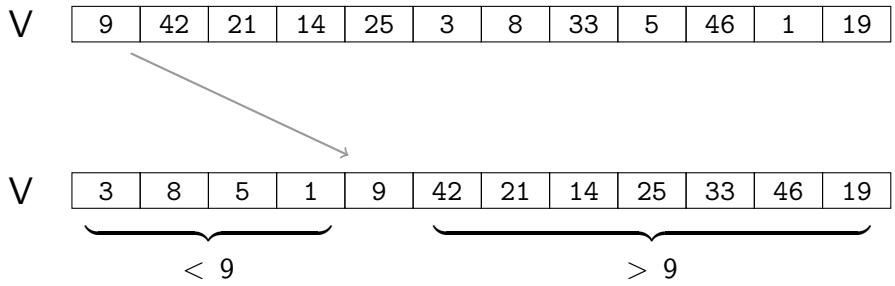


Como é que a gente coloca essa lista em ordem?

Bom, um dos algoritmos mais rápidos para fazer isso é o *Quicksort*.

E a sua ideia é bem simples.

Quer dizer, ele começa aplicando o procedimento de partição, utilizando o primeiro elemento como pivô



E agora, é preciso ordenar duas faixas da lista: `L[1..4]` e `L[6..12]`.

A ideia, claro, é fazer a mesma coisa outra vez.

E nós vamos utilizar uma pilha para organizar essa computação.

Quer dizer,

1. No início, a gente coloca na pilha a tarefa de ordenar a lista inteira
2. Daí, a cada passo, a gente
   - remove uma tarefa da pilha

- realiza a partição
- e coloca as duas tarefas resultantes na pilha

— (*a menos que a tarefa seja trivial*)

3. E continua fazendo isso até que a pilha esteja vazia

A coisa fica assim

```
func Quicksort ( L )

    Push ( (1..N), P )                      //  Coloca a lista inteira na pilha

    Enquanto ( P não está vazia )

       (i..f)  =  Pop ( P )

       Se ( i < f )                         //  Se existe mais de um elemento na faixa

          k  =  Partição ( L, i, f )        //  Particiona a faixa

          Push ( (i..k-1), P )              //
                                            //  e empilha as duas faixas resultantes
          Push ( (k+1..f), P )              //
```

7 Como os computadores fazem as coisas

(. . .)

- *a notação polonesa*