MC-202
Aplicações de Pilhas

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

2º semestre/2018

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

- dizemos que estão balanceados

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

- dizemos que estão balanceados
- Exemplos corretos:

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

- dizemos que estão balanceados
- Exemplos corretos:
  - $(a + b)$

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

- dizemos que estão balanceados
- Exemplos corretos:
  - $(a + b)$
  - $(a \cdot b) + (c/(d - e))$

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

- dizemos que estão balanceados
- Exemplos corretos:
  - $(a + b)$
  - $(a \cdot b) + (c/(d - e))$
- Exemplos incorretos:

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

- dizemos que estão balanceados
- Exemplos corretos:
  - $(a + b)$
  - $(a \cdot b) + (c/(d - e))$
- Exemplos incorretos:
  - $(a + b$

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

- dizemos que estão balanceados
- Exemplos corretos:
  - $(a + b)$
  - $(a \cdot b) + (c/(d - e))$
- Exemplos incorretos:
  - $(a + b$
  - $(a \cdot b) + (c/d - e))$

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

- dizemos que estão balanceados
- Exemplos corretos:
  - $(a + b)$
  - $(a \cdot b) + (c/(d - e))$
- Exemplos incorretos:
  - $(a + b$
  - $(a \cdot b) + (c/d - e))$
  - $)a + b($

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

- dizemos que estão balanceados
- Exemplos corretos:
  - $(a + b)$
  - $(a \cdot b) + (c/(d - e))$
- Exemplos incorretos:
  - $(a + b$
  - $(a \cdot b) + (c/d - e))$
  - $)a + b($

Escreva uma função que dada uma sequência de parênteses, diz se ela é balanceada ou não

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

- dizemos que estão balanceados
- Exemplos corretos:
  - $(a + b)$
  - $(a \cdot b) + (c/(d - e))$
- Exemplos incorretos:
  - $(a + b$
  - $(a \cdot b) + (c/d - e))$
  - $)a + b($

Escreva uma função que dada uma sequência de parênteses, diz se ela é balanceada ou não

- Vamos ignorar operandos e operadores

Exercício

Temos uma expressão aritmética e queremos saber se os parênteses estão corretos

- dizemos que estão balanceados
- Exemplos corretos:
  - $(a + b)$
  - $(a \cdot b) + (c/(d - e))$
- Exemplos incorretos:
  - $(a + b$
  - $(a \cdot b) + (c/d - e))$
  - $)a + b($

Escreva uma função que dada uma sequência de parênteses, diz se ela é balanceada ou não

- Vamos ignorar operandos e operadores
- $()(())$

Balanceamento de parênteses

Uma sequência de parênteses é balanceada se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Balanceamento de parênteses

Uma sequência de parênteses é balanceada se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Exemplos:

Balanceada	Não Balanceada
	([]
[[ ]]	[[ ))
([( ) ([ ])])	([ )]

Balanceamento de parênteses

Uma sequência de parênteses é balanceada se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Exemplos:

Balanceada	Não Balanceada
	([]
[[ ]]	[[ ))
([( ) ([ ])])	([ )]

Para testar, leia cada símbolo e se:

Balanceamento de parênteses

Uma sequência de parênteses é balanceada se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Exemplos:

Balanceada	Não Balanceada
	([]
[[ ]]	[[ ))
([( ) ([ ])])	([ )]

Para testar, leia cada símbolo e se:

1. leu (ou [: empilha o símbolo lido

Balanceamento de parênteses

Uma sequência de parênteses é balanceada se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Exemplos:

Balanceada	Não Balanceada
	([]
[[ ]]	[[ ))
([( ) ([ ])])	([ )]

Para testar, leia cada símbolo e se:

1. leu (ou [: empilha o símbolo lido
2. leu]: desempilha [

Balanceamento de parênteses

Uma sequência de parênteses é balanceada se é:

vazia ou [*sequência válida*] ou (*sequência válida*)
ou a concatenação de duas sequências válidas

Exemplos:

Balanceada	Não Balanceada
	([]
[[ ]]	[[ ))
([( ) ([ ])])	([ )]

Para testar, leia cada símbolo e se:

1. leu (ou [: empilha o símbolo lido
2. leu]: desempilha [
3. leu): desempilha (

Implementação em C

```
1 int eh_balanceada(char *str) {
```

Implementação em C

```
1 int eh_balanceada(char *str) {
2   p_pilha pilha;
```


Implementação em C

```
1 int eh_balanceada(char *str) {
2   p_pilha pilha;
3   int i, ok = 1;
4   char par;
```


Implementação em C

```
1 int eh_balanceada(char *str) {
2   p_pilha pilha;
3   int i, ok = 1;
4   char par;
5   p_pilha = criar_pilha();
6   for (i = 0; ok && str[i] != '\0'; i++)
```


Implementação em C

```
1 int eh_balanceada(char *str) {
2   p_pilha pilha;
3   int i, ok = 1;
4   char par;
5   p_pilha = criar_pilha();
6   for (i = 0; ok && str[i] != '\0'; i++)
7     if (str[i] == '['  || str[i] == '(')
```

Implementação em C

```
1  int eh_balanceada(char *str) {
2    p_pilha pilha;
3    int i, ok = 1;
4    char par;
5    p_pilha = criar_pilha();
6    for (i = 0; ok && str[i] != '\0'; i++)
7      if (str[i] == '['  || str[i] == '(')
8        empilhar(pilha, str[i]);
```


Implementação em C

```
1  int eh_balanceada(char *str) {
2    p_pilha pilha;
3    int i, ok = 1;
4    char par;
5    p_pilha = criar_pilha();
6    for (i = 0; ok && str[i] != '\0'; i++)
7      if (str[i] == '['  || str[i] == '(')
8        empilhar(pilha, str[i]);
9      else if (eh_vazia(pilha))
10       ok = 0;
11     else {
```

4

Implementação em C

```
1  int eh_balanceada(char *str) {
2    p_pilha pilha;
3    int i, ok = 1;
4    char par;
5    p_pilha = criar_pilha();
6    for (i = 0; ok && str[i] != '\0'; i++)
7      if (str[i] == '['  || str[i] == '(')
8        empilhar(pilha, str[i]);
9      else if (eh_vazia(pilha))
10       ok = 0;
11     else {
12       par = desempilhar(pilha);
```

4

Implementação em C

```
1  int eh_balanceada(char *str) {
2    p_pilha pilha;
3    int i, ok = 1;
4    char par;
5    p_pilha = criar_pilha();
6    for (i = 0; ok && str[i] != '\0'; i++)
7      if (str[i] == '['  || str[i] == '(')
8        empilhar(pilha, str[i]);
9      else if (eh_vazia(pilha))
10       ok = 0;
11     else {
12       par = desempilhar(pilha);
13       if (str[i] == ']' && par != '[')
14         ok = 0;
```

4

Implementação em C

```c
int eh_balanceada(char *str) {
  p_pilha pilha;
  int i, ok = 1;
  char par;
  p_pilha = criar_pilha();
  for (i = 0; ok && str[i] != '\0'; i++)
    if (str[i] == '['  || str[i] == '(')
      empilhar(pilha, str[i]);
    else if (eh_vazia(pilha))
      ok = 0;
    else {
      par = desempilhar(pilha);
      if (str[i] == ']' && par != '[')
        ok = 0;
      if (str[i] == ')' && par != '(')
        ok = 0;
```


Implementação em C

```c
int eh_balanceada(char *str) {
  p_pilha pilha;
  int i, ok = 1;
  char par;
  p_pilha = criar_pilha();
  for (i = 0; ok && str[i] != '\0'; i++)
    if (str[i] == '['  || str[i] == '(')
      empilhar(pilha, str[i]);
    else if (eh_vazia(pilha))
      ok = 0;
    else {
      par = desempilhar(pilha);
      if (str[i] == ']' && par != '[')
        ok = 0;
      if (str[i] == ')' && par != '(')
        ok = 0;
    }
```


Implementação em C

```c
int eh_balanceada(char *str) {
  p_pilha pilha;
  int i, ok = 1;
  char par;
  p_pilha = criar_pilha();
  for (i = 0; ok && str[i] != '\0'; i++)
    if (str[i] == '['  || str[i] == '(')
      empilhar(pilha, str[i]);
    else if (eh_vazia(pilha))
      ok = 0;
    else {
      par = desempilhar(pilha);
      if (str[i] == ']' && par != '[')
        ok = 0;
      if (str[i] == ')' && par != '(')
        ok = 0;
    }
  if (!eh_vazia(pilha))
    ok = 0;
```


Implementação em C

```c
int eh_balanceada(char *str) {
  p_pilha pilha;
  int i, ok = 1;
  char par;
  p_pilha = criar_pilha();
  for (i = 0; ok && str[i] != '\0'; i++)
    if (str[i] == '['  || str[i] == '(')
      empilhar(pilha, str[i]);
    else if (eh_vazia(pilha))
      ok = 0;
    else {
      par = desempilhar(pilha);
      if (str[i] == ']' && par != '[')
        ok = 0;
      if (str[i] == ')' && par != '(')
        ok = 0;
    }
  if (!eh_vazia(pilha))
    ok = 0;
  destruir_pilha(pilha);
```

Implementação em C

```c
int eh_balanceada(char *str) {
  p_pilha pilha;
  int i, ok = 1;
  char par;
  p_pilha = criar_pilha();
  for (i = 0; ok && str[i] != '\0'; i++)
    if (str[i] == '['  || str[i] == '(')
      empilhar(pilha, str[i]);
    else if (eh_vazia(pilha))
      ok = 0;
    else {
      par = desempilhar(pilha);
      if (str[i] == ']' && par != '[')
        ok = 0;
      if (str[i] == ')' && par != '(')
        ok = 0;
    }
  if (!eh_vazia(pilha))
    ok = 0;
  destruir_pilha(pilha);
  return ok;
}
```


Implementação em C

```c
int eh_balanceada(char *str) {
  p_pilha pilha;
  int i, ok = 1;
  char par;
  p_pilha = criar_pilha();
  for (i = 0; ok && str[i] != '\0'; i++)
    if (str[i] == '['  || str[i] == '(')
      empilhar(pilha, str[i]);
    else if (eh_vazia(pilha))
      ok = 0;
    else {
      par = desempilhar(pilha);
      if (str[i] == ']' && par != '[')
        ok = 0;
      if (str[i] == ')' && par != '(')
        ok = 0;
    }
  if (!eh_vazia(pilha))
    ok = 0;
  destruir_pilha(pilha);
  return ok;
}
```

- E se usássemos `return 0` dentro do `for`?

Notação de expressões

Notação de expressões

Notação de expressões

Exemplo 1:

Notação de expressões

Exemplo 1:

- Infixa: `a + b`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Pré-fixa: é a notação polonesa do lógico Jan Lukasiewicz

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Pré-fixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Pré-fixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Pré-fixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos
3. Pós-fixa: é notação polonesa reversa (RPN), das calculadoras HP, ...

Notação de expressões

Exemplo 1:

- Infixa: `a + b`
- Pré-fixa: `+ a b`
- Pós-fixa: `a b +`

Exemplo 2:

- Infixa: `5 * ( (9 + 8) * 4 * 6 + 7)`
- Pré-fixa: `* 5 + * + 9 8 * 4 6 7`
- Pós-fixa: `5 9 8 + 4 6 * * 7 + *`

Notação de expressões aritméticas:

1. Infixa: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. Pré-fixa: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos
3. Pós-fixa: é notação polonesa reversa (RPN), das calculadoras HP, ...
   - Operador sucede operandos

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

```
2 * ( (2 + 1) * 4 + 1)
```


Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

```
2 * ( 3 * 4 + 1)
```


Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

```
2 * ( 12 + 1)
```


Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

```
2 * 13
```

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

2

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

2 2

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

`2 2 1`

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

`2 2 1 +`

6

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

`2 3`

6

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

`2 3 4`

6

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

`2 3 4 *`

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

`2 12`

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

`2 12 1`

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

`2 12 1 +`

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

`2 13`

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

`2 13 *`

Exemplo de cálculo de expressão

- Infixa: `2 * ( (2 + 1) * 4 + 1)` = 26
- Pós-fixa: `2 2 1 + 4 * 1 + *`

Resolvendo com notação infixa:

26

Resolvendo com notação pós-fixa:

26

Calculando expressões pós-fixas

Algoritmo:

Calculando expressões pós-fixas

Algoritmo:

1. Para cada elemento lido:

Calculando expressões pós-fixas

Algoritmo:

1. Para cada elemento lido:
   - Se for número n:

Calculando expressões pós-fixas

Algoritmo:

1. Para cada elemento lido:
   - Se for número n:
     - empilha n

Calculando expressões pós-fixas

Algoritmo:

1. Para cada elemento lido:
   - Se for número n:
     - empilha n
   - Se for operador $\oplus$:

Calculando expressões pós-fixas

Algoritmo:

1. Para cada elemento lido:
   - Se for número n:
     - empilha n
   - Se for operador $\oplus$:
     - desempilha $operando_1$

Calculando expressões pós-fixas

Algoritmo:

1. Para cada elemento lido:
   - Se for número n:
     - empilha n
   - Se for operador $\oplus$:
     - desempilha $operando_1$
     - desempilha $operando_2$

7

Calculando expressões pós-fixas

Algoritmo:

1. Para cada elemento lido:
   - Se for número n:
     - empilha n
   - Se for operador $\oplus$:
     - desempilha $operando_1$
     - desempilha $operando_2$
     - empilha $operando_2 \oplus operando_1$

7

Calculando expressões pós-fixas

Algoritmo:

1. Para cada elemento lido:
   - Se for número n:
     - empilha n
   - Se for operador $\oplus$:
     - desempilha $operando_1$
     - desempilha $operando_2$
     - empilha $operando_2 \oplus operando_1$
2. Desempilha (único) valor da pilha e retorna

7

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparece um operador na entrada

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparece um operador na entrada
  - enquanto o operador no topo tem precedência maior ou igual ao operador na entrada

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparece um operador na entrada
  - enquanto o operador no topo tem precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparece um operador na entrada
  - enquanto o operador no topo tem precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparece um operador na entrada
  - enquanto o operador no topo tem precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo
- No final, desempilhamos todos os elementos da pilha, copiando para a saída

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparece um operador na entrada
  - enquanto o operador no topo tem precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo
- No final, desempilhamos todos os elementos da pilha, copiando para a saída

Observações:

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparece um operador na entrada
  - enquanto o operador no topo tem precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo
- No final, desempilhamos todos os elementos da pilha, copiando para a saída

Observações:

- Usamos uma nova operação: "olhar topo da pilha".

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparece um operador na entrada
  - enquanto o operador no topo tem precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo
- No final, desempilhamos todos os elementos da pilha, copiando para a saída

Observações:

- Usamos uma nova operação: "olhar topo da pilha".
  - É possível fazer sem?

Convertendo de infixa para pós-fixa

Objetivo:

`1 + 2 * 3 / 4 * 5 - 6` ⇒ `1 2 3 * 4 / 5 * + 6 -`

Ideia:

- Copiamos os números diretamente na saída
- Quando aparece um operador na entrada
  - enquanto o operador no topo tem precedência maior ou igual ao operador na entrada
    - desempilhamos e copiamos na saída
  - empilhamos o operador novo
- No final, desempilhamos todos os elementos da pilha, copiando para a saída

Observações:

- Usamos uma nova operação: "olhar topo da pilha".
  - É possível fazer sem?
- Como generalizar para o caso em que a expressão tem parênteses?

Pilhas e recursão

Pergunta: Qual a relação entre *pilhas* e *recursão*?

Pilhas e recursão

Pergunta: Qual a relação entre *pilhas* e *recursão*?

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```


Pilhas e recursão

Pergunta: Qual a relação entre *pilhas* e *recursão*?

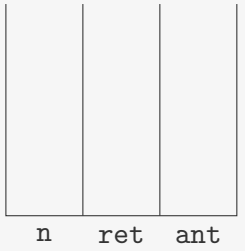
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

Vamos tentar descobrir simulando uma chamada: `fat(4)`

Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

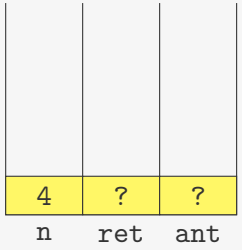
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

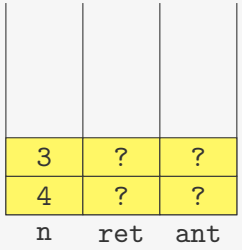
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

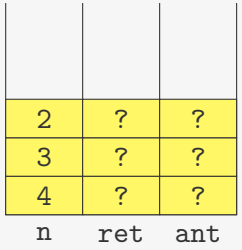
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

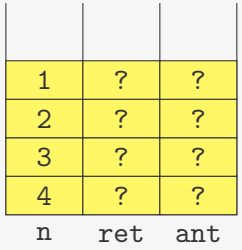
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

n	ret	ant
0	?	?
1	?	?
2	?	?
3	?	?
4	?	?

Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

n	ret	ant
0	1	?
1	?	?
2	?	?
3	?	?
4	?	?

Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

n	ret	ant
0	1	?
1	?	?
2	?	?
3	?	?
4	?	?

Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

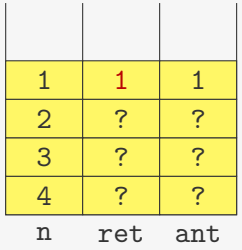
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```

n	ret	ant
1	?	1
2	?	?
3	?	?
4	?	?

Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

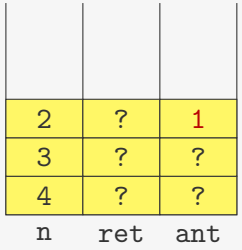
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

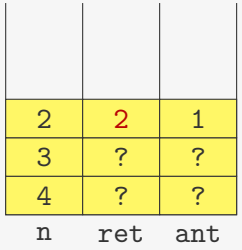
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

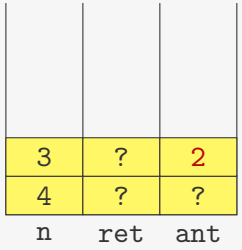
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

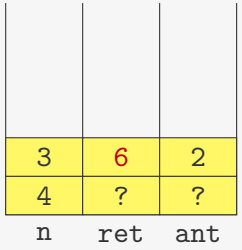
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

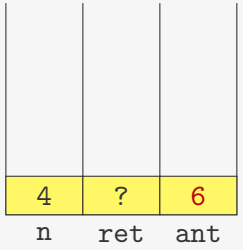
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

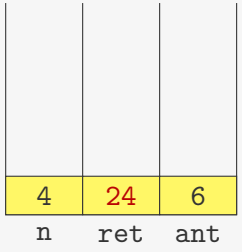
```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilha de Chamadas

Estado da “pilha” de chamadas para `fat(4)`:

```
1  int fat(int n) {
2    int ret, ant;
3    if (n == 0)
4      ret = 1;
5    else {
6      ant = fat(n-1);
7      ret = n * ant;
8    }
9    return ret;
10 }
```



Pilhas e recursão (continuando)

Quando empilhamos:

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais
- Restabelecemos os valores das variáveis locais para o valor que tinham antes da chamada

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais
- Restabelecemos os valores das variáveis locais para o valor que tinham antes da chamada

O conjunto de variáveis locais formam um elemento da pilha

Pilhas e recursão (continuando)

Quando empilhamos:

- Alocamos espaço para as variáveis locais (`n`, `ret`, `ant`)

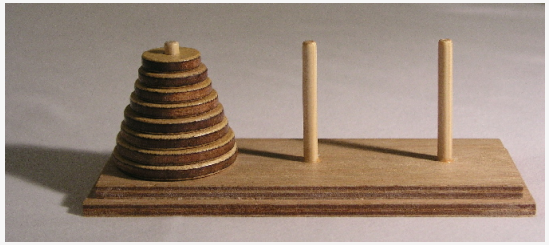
Quando desempilhamos:

- Quando a chamada de `fat(n)` retorna, apagamos o espaço para as variáveis locais
- Restabelecemos os valores das variáveis locais para o valor que tinham antes da chamada

O conjunto de variáveis locais formam um elemento da pilha

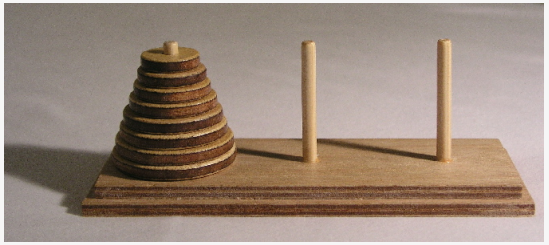
Isto é, a recursão pode ser simulada usando uma pilha de suas variáveis locais

Um exemplo mais complexo: Torres de Hanói



```
1 void hanoi(int n, char orig, char dest, char aux) {
2   if (n > 0) {
3     hanoi(n-1, orig, aux, dest);
4     printf("move de %c para %c\n", orig, dest);
5     hanoi(n-1, aux, dest, orig);
6   }
7 }
```


Um exemplo mais complexo: Torres de Hanói



```
1 void hanoi(int n, char orig, char dest, char aux) {
2   if (n > 0) {
3     hanoi(n-1, orig, aux, dest);
4     printf("move de %c para %c\n", orig, dest);
5     hanoi(n-1, aux, dest, orig);
6   }
7 }
```

Precisamos além de empilhar a variáveis locais, armazenar em qual linha devemos voltar a execução do código

Pilhas e recursão

O registro de ativação de uma função é o conjunto formado por:

Pilhas e recursão

O registro de ativação de uma função é o conjunto formado por:

1. Variáveis locais

Pilhas e recursão

O registro de ativação de uma função é o conjunto formado por:

1. Variáveis locais
2. Endereço de retorno após a chamada

Pilhas e recursão

O registro de ativação de uma função é o conjunto formado por:

1. Variáveis locais
2. Endereço de retorno após a chamada

A pilha de execução (ou pilha de chamadas) é a pilha dos registros de ativação das várias chamadas em execução em um programa

Buscando um elemento em uma lista ligada

Versão recursiva:

```
1 p_no busca_rec(p_no lista, int x) {
2   if (lista == NULL || lista->dado == x)
3     return lista;
4   else
5     return busca_rec(lista->prox, x);
6 }
```


Buscando um elemento em uma lista ligada

Versão recursiva:

```
1 p_no busca_rec(p_no lista, int x) {
2   if (lista == NULL || lista->dado == x)
3     return lista;
4   else
5     return busca_rec(lista->prox, x);
6 }
```

Note que:

Buscando um elemento em uma lista ligada

Versão recursiva:

```
1 p_no busca_rec(p_no lista, int x) {
2   if (lista == NULL || lista->dado == x)
3     return lista;
4   else
5     return busca_rec(lista->prox, x);
6 }
```

Note que:

- a recursão é a última coisa antes do retorno da função

Buscando um elemento em uma lista ligada

Versão recursiva:

```
1 p_no busca_rec(p_no lista, int x) {
2   if (lista == NULL || lista->dado == x)
3     return lista;
4   else
5     return busca_rec(lista->prox, x);
6 }
```

Note que:

- a recursão é a última coisa antes do retorno da função
- i.e., apenas retornamos o valor de `busca_rec(lista->prox, x)` sem manipulá-lo

Buscando um elemento em uma lista ligada

Versão recursiva:

```
1 p_no busca_rec(p_no lista, int x) {
2   if (lista == NULL || lista->dado == x)
3     return lista;
4   else
5     return busca_rec(lista->prox, x);
6 }
```

Note que:

- a recursão é a última coisa antes do retorno da função
- i.e., apenas retornamos o valor de `busca_rec(lista->prox, x)` sem manipulá-lo
- exceto na base, o retorno não depende do valor das variáveis locais

Buscando um elemento em uma lista ligada

Versão recursiva:

```
1 p_no busca_rec(p_no lista, int x) {
2   if (lista == NULL || lista->dado == x)
3     return lista;
4   else
5     return busca_rec(lista->prox, x);
6 }
```

Note que:

- a recursão é a última coisa antes do retorno da função
- i.e., apenas retornamos o valor de `busca_rec(lista->prox, x)` sem manipulá-lo
- exceto na base, o retorno não depende do valor das variáveis locais
  - Depende apenas do valor da chamada recursiva

14

Eliminação de Recursão

Podemos eliminar o uso de recursão na nossa função

Eliminação de Recursão

Podemos eliminar o uso de recursão na nossa função

Versão recursiva:

```
1 p_no busca_rec(p_no lista, int x) {
2   if (lista == NULL || lista->dado == x)
3     return lista;
4   else
5     return busca_rec(lista->prox, x);
6 }
```

15

Eliminação de Recursão

Podemos eliminar o uso de recursão na nossa função

Versão recursiva:

```
1 p_no busca_rec(p_no lista, int x) {
2   if (lista == NULL || lista->dado == x)
3     return lista;
4   else
5     return busca_rec(lista->prox, x);
6 }
```

Eliminando a recursão:

```
1 p_no busca_iterativa(p_no lista, int x) {
2   while(lista != NULL && lista->dado != x)
3     lista = lista->prox;
4   return lista;
5 }
```


Recursão de Cauda

Se o último passo de uma função `f(x)` é o retorno de `f(y)` então

Recursão de Cauda

Se o último passo de uma função `f(x)` é o retorno de `f(y)` então

- podemos reiterar a função `f(x)` usando `x = y`

Recursão de Cauda

Se o último passo de uma função `f(x)` é o retorno de `f(y)` então

- podemos reiterar a função `f(x)` usando `x = y`
- usando um `while`

Recursão de Cauda

Se o último passo de uma função `f(x)` é o retorno de `f(y)` então

- podemos reiterar a função `f(x)` usando `x = y`
- usando um `while`
- até chegar em uma das bases da recursão

Recursão de Cauda

Se o último passo de uma função `f(x)` é o retorno de `f(y)` então

- podemos reiterar a função `f(x)` usando `x = y`
- usando um `while`
- até chegar em uma das bases da recursão

Chamamos esse tipo de recursão de recursão de cauda

16

Recursão geral

Note que `hanoi` e `fat` não têm recursão de cauda

Recursão geral

Note que `hanoi` e `fat` não têm recursão de cauda

17

Recursão geral

Note que `hanoi` e `fat` não têm recursão de cauda

Recursão geral

Note que `hanoi` e `fat` não têm recursão de cauda


```
1 void hanoi(int n, char orig, char dest,
      char aux) {
2   if (n > 0) {
3     hanoi(n-1, orig, aux, dest);
4     printf("move de %c para %c\n", orig,
      dest);
5     hanoi(n-1, aux, dest, orig);
6   }
7 }
```


Recursão geral

Note que `hanoi` e `fat` não têm recursão de cauda


```
1 void hanoi(int n, char orig, char dest,
      char aux) {
2   if (n > 0) {
3     hanoi(n-1, orig, aux, dest);
4     printf("move de %c para %c\n", orig,
      dest);
5     hanoi(n-1, aux, dest, orig);
6   }
7 }
```

```
1 int fat(int n) {
2   int ret, ant;
3   if (n == 0)
4     ret = 1;
5   else {
6     ant = fat(n-1);
7     ret = n * ant;
8   }
9   return ret;
10 }
```


Recursão geral

Note que `hanoi` e `fat` não têm recursão de cauda


```
1 void hanoi(int n, char orig, char dest,
      char aux) {
2   if (n > 0) {
3     hanoi(n-1, orig, aux, dest);
4     printf("move de %c para %c\n", orig,
      dest);
5     hanoi(n-1, aux, dest, orig);
6   }
7 }
```

```
1 int fat(int n) {
2   int ret, ant;
3   if (n == 0)
4     ret = 1;
5   else {
6     ant = fat(n-1);
7     ret = n * ant;
8   }
9   return ret;
10 }
```

Recursões que não são de cauda também podem ser eliminadas

Recursão geral

Note que `hanoi` e `fat` não têm recursão de cauda

```
1 void hanoi(int n, char orig, char dest,
      char aux) {
2   if (n > 0) {
3     hanoi(n-1, orig, aux, dest);
4     printf("move de %c para %c\n", orig,
      dest);
5     hanoi(n-1, aux, dest, orig);
6   }
7 }
```

```
1 int fat(int n) {
2   int ret, ant;
3   if (n == 0)
4     ret = 1;
5   else {
6     ant = fat(n-1);
7     ret = n * ant;
8   }
9   return ret;
10 }
```

Recursões que não são de cauda também podem ser eliminadas

- Porém é necessário utilizar uma pilha

Recursão geral

Note que `hanoi` e `fat` não têm recursão de cauda

```
1 void hanoi(int n, char orig, char dest,
      char aux) {
2   if (n > 0) {
3     hanoi(n-1, orig, aux, dest);
4     printf("move de %c para %c\n", orig,
      dest);
5     hanoi(n-1, aux, dest, orig);
6   }
7 }
```

```
1 int fat(int n) {
2   int ret, ant;
3   if (n == 0)
4     ret = 1;
5   else {
6     ant = fat(n-1);
7     ret = n * ant;
8   }
9   return ret;
10 }
```

Recursões que não são de cauda também podem ser eliminadas

- Porém é necessário utilizar uma pilha
- E o processo é mais trabalhoso

Recursão vs. Iteração

Algoritmos recursivos:

Recursão vs. Iteração

Algoritmos recursivos:

- mais fáceis de entender e de criar

Recursão vs. Iteração

Algoritmos recursivos:

- mais fáceis de entender e de criar
- mais elegantes

Recursão vs. Iteração

Algoritmos recursivos:

- mais fáceis de entender e de criar
- mais elegantes

Algoritmos iterativos:

Recursão vs. Iteração

Algoritmos recursivos:

- mais fáceis de entender e de criar
- mais elegantes

Algoritmos iterativos:

- Normalmente mais rápidos do que os recursivos

Recursão vs. Iteração

Algoritmos recursivos:

- mais fáceis de entender e de criar
- mais elegantes

Algoritmos iterativos:

- Normalmente mais rápidos do que os recursivos
- Não precisamos empilhar registros a cada iteração

Recursão vs. Iteração

Algoritmos recursivos:

- mais fáceis de entender e de criar
- mais elegantes

Algoritmos iterativos:

- Normalmente mais rápidos do que os recursivos
- Não precisamos empilhar registros a cada iteração

Eliminação de recursão de cauda é uma ótima forma de otimização

Recursão vs. Iteração

Algoritmos recursivos:

- mais fáceis de entender e de criar
- mais elegantes

Algoritmos iterativos:

- Normalmente mais rápidos do que os recursivos
- Não precisamos empilhar registros a cada iteração

Eliminação de recursão de cauda é uma ótima forma de otimização

- E é feita automaticamente por alguns compiladores

Recursão vs. Iteração

Algoritmos recursivos:

- mais fáceis de entender e de criar
- mais elegantes

Algoritmos iterativos:

- Normalmente mais rápidos do que os recursivos
- Não precisamos empilhar registros a cada iteração

Eliminação de recursão de cauda é uma ótima forma de otimização

- E é feita automaticamente por alguns compiladores

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"

Donald E. Knuth

Exercício

Elimine a recursão da busca binária:

```
1  int busca_binaria(int *dados, int l, int r, int x) {
2    int m = (l+r)/2;
3    if (l > r)
4      return -1;
5    if (dados[m] == x)
6      return m;
7    else if (dados[m] < x)
8      return busca_binaria(dados, m + 1, r, x);
9    else
10     return busca_binaria(dados, l, m - 1, x);
11 }
```